PCubeS Type Architecture and IT Programming Language

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Muhammad Nur Yanhaona

May

2017

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Nuo Yan

AUTHOR

The dissertation has been read and approved by the examining committee:

Andrew Grimshaw

Advisor

Kevin Skadron

Jack Davidson

Westley Weimer

Mircea Stan

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

May

2017

Thesis advisor: Professor Andrew Grimshaw Muhammad Nur Yanhaona

PCubeS Type Architecture and IT Programming Language

Abstract

As the rate of improvement of uniprocessors slows, parallel programming has become an issue for more programmers now, not just for high performance scientific computing developers. Multicore processors and accelerators are now ubiquitous in both small and large scale computing devices. The potential for performance improvement through parallelism is significant in contemporary hardware.

Despite the heightened interest towards parallel computing in recent years; writing efficient, portable parallel applications remains a challenge, in particular, in the presence of increasingly heterogeneous architectures. The predominant way of parallel computing is to extend a sequential language with low-level parallelization primitives. These primitives are platform specific, difficult to combine, and often error-prone. Recent high-level language-based alternatives to these tools struggle to perform well and lack in portability also.

This research proposes an alternative parallel programming paradigm to strike a balance between existing platform specific and hardware agnostic approaches. The paradigm lets a program's logic be expressed over a generic machine abstraction, called *PCubeS*, to enable portability. The language counterpart, called *IT*, enables reasoning about hardware features in all aspects of programming but emphasizes on separation of concerns and uses a declarative syntax so that program efficiency can be achieved productively.

Three *IT* compilers have been developed for three architecture types (multicore CPUs, distributed memory machines, and hybrid supercomputers having both CPUs and NVIDIA GPUs as nodes) as part of this research. Early performance experiment results on representative hardware platforms for five well-known applications suggest that the proposed paradigm is a viable future option for portable and efficient parallel computing.

Contents

Listing of figures

To the three individuals I wanted to please the most by doing my study

my beloved Mom and Dad,

and my wonderful advisor, Andrew Grimshaw.

Acknowledgments

I thank, first and foremost, my advisor Professor Andrew Grimshaw. I often wonder if he ever spent more time and energy on any student than he has spent on me.

My good friend Chris Koeritz who is a source of encouragement for me and is always ready to help even before I ask him for it.

My fellow student, like a younger brother in the US, Anindya Prodhan, whom I never even bother to give thanks for what he does for me.

My fellow student Swaroopa Dola for doing most of the experiments. I hope she will enjoy continuing the project after I am gone.

Scott Ruffner, who have helped me tremendously by making software and hardware resources available whenever I needed them.

My respected committee members for their support, guidance, and feedback.

My present and previous friends among the graduate students and office staff. I have learned a lot from them, their feedback have made valuable contributions to my work, and their company have made my staying in the US bearable.

Both the University of Virginia and Indiana University for my use of their computing resources.

Finally, I thank this wonderful country for giving me the opportunity to do the study.

1

Introduction

1.1 Motivation

The history of parallel machines can be traced back to the early days of computers. For example, researchers at IBM proposed SOLOMON, a SIMD machine with 1024 1-bit processing elements, back in 1958. Development of CMU Multiminiprocessor Computer (C.mmp) started as early as 1970 [20]. This attention toward parallelism from the very beginning is, more or less, driven by two factors. First, many applications of interest in both

science and industry have significant inherent parallelisms. Second, and more importantly, the computation power demanded by the applications always surpassed the capacity available in the machines.

Consequently, there is a continuing effort for increasing the power of parallel supercomputers. For example, while an early parallel supercomputer, ZMOB, started with only 256 processors[78]; present day supercomputers have millions of cores. There is a great diversity in these machines' architecture in terms of the nature of processing elements, their coordination, interconnection network, memory layout, cache management, etc. Such diversity indicates that there is no single agreed upon ideal vision of how best to build a parallel computer.

This diversity in parallel architectures comes as no surprise. Parallel supercomputers are often built to target specific classes of applications, and different applications have widely different computation and communication requirements, necessitating different hardware organizations. Nonetheless, as large scale data analysis and massively parallel computations have become more mainstream in recent years, new parallel architectures tend to be more general purpose. In the absence of an all-serving, ideal vision; the current trend in parallel supercomputer development is to incorporate heterogeneity within a single machine to serve a multitude of application classes. For example, two recent parallel supercomputers, Titan[tit] and Stampede[sta], are hybrid architectures comprising multi-core CPUs, accelerators, and large memory nodes. Furthermore, often there are heterogeneity in the organization of these nodes and hierarchy in their interconnection network's architecture.

With the end of rapid uniprocessor performance improvement, parallel programming – that used to be a concern for high performance scientific computing – has also become an issue for everyone else. Multicore processors and accelerators are now ubiquitous. They are in tablets, desktop PCs, workstation clusters, supercomputers, and many other small and big computing devices. There is an enormous potential of performance improvement through parallelism for applications running in these hardware platforms.

An accelerated demand for better performance on compute intensive problems compounded by a consistent trend of increasing hardware complexity leads to a heightened interest toward parallel computing in recent years. A panel of multidisciplinary Berkeley researchers[16] thus have concluded that "developing programming models that productively enable development of highly efficient implementations of parallel applications is the biggest challenge facing the deployment of future many-core systems."

Unfortunately, parallel computing remains hard in general, despite decades of research and development of a plethora of programming tools, techniques, and languages. The predominant way of programming in parallel architectures remains to be extending a sequential language with low-level parallelization primitives as in MPI[91], Pthreads[25], OpenMP[36], and CUDA[71]. These tools are platform specific, difficult to combine, and often error-prone. Recent high-level language based alternatives to these tools such as Chapel[30] and X10[32] are struggling to perform well. Further, so far their application has been limited to specific architecture types.

This is regrettable, as we cannot require an average computer user to be an expert in multiple modes of parallel programming. Parallel programming is no more restricted to costly computation intensive sciences or to expensive industry funded data analysis; rather it has spread in many branches of academic, industrial, and personal computing. We need a common framework for exploiting parallelism in computations in an easy way. That is the objective of my research: *to provide a portable, efficient, and productive paradigm for parallel computing*.

First, I emphasize portability as there is a long-standing opposition against accepting a new programming paradigm in any community that is already accustomed to some other paradigm. One of the reasons, I believe, techniques like MPI, Pthreads, OpenMP, or CUDA are so successful is that they built on existing knowledge of widely used languages like C and Fortran. Even though these techniques make writing and debugging pro-

grams difficult, once one masters a technique, those issues become much less of a barrier. Rather our current problem is that a typical programmer is exposed to a variety of parallel architectures and his/her experience from programming on a hardware is often not useful on the next hardware because existing techniques are tied to particular execution platforms. Hence, new paradigms should address this problem as a central issue.

Second, given high performance is the primary concern in parallel computing, we cannot offer portability at the expense of efficiency. Performance of programs written using the new paradigms should match the performance of equivalent programs written using existing techniques. This is the only reasonable goal given that a programmer can use platform specific low-level tools to achieve the same effects in a program that a new paradigm automatically does for him/her.

Finally, I advise for a systematic, easy to understand, and human-centric approach of program development for the new paradigms as productivity problems have been identified as a significant barrier to the application of parallel computing to the sciences and other disciplines[6,5,16]. Nevertheless, I believe productivity should never compromise efficiency given the latter's importance. Rather, new paradigms should offer features and programming style that enable high performance in a productive way.

1.2 Problems in Current Solutions

Andrews in *Concurrent Programming: Principles and Practice*[15] expresses that "concurrent programs are to sequential programs what chess is to checkers." That developing a parallel algorithm requires a more creative involvement in general than an equivalent sequential algorithm is undeniable. The difficulty of devising the algorithm, however, is not commonly the core problem with parallel programming. Rather, most problems

occur in the process of translating the parallel algorithm into a program, as deftly described by Snyder in his seminal work "Type Architectures, Shared Memory, and the Corollary of Modest Potential[82]."

Correctly implementing interactions between parallel pieces of a program can be difficult. So much so that sometimes the original algorithm is lost in the plethora of interaction primitives. The biggest problem, however, is extracting good performance from a parallel machine. Features like memory organization, cache hierarchies, interconnection network structure, speed and bandwidth of memory access and data transfers that are orthogonal to the algorithm by and large determine the performance of its program implementation.

Predominant parallel programming tools such as MPI, Pthreads, and CUDA give a programmer the ultimate control over an execution environment. In doing so, these tools enable the programmer to write a code optimized for the target platform. For example in MPI, he/she can send messages anywhere he/she wants and in any shape or form, synchronize pieces of computations as he/she intended, and combine sub-programs whatever way it suits him/her – but all that at the expense of programmer's productivity and lack of program's portability.

Furthermore, these tools and primitives may be perfect for their designated original environments, but with the advent of hierarchical and hybrid architectures they fall short of utilizing many important hardware features. For example, neither MPI nor Pthreads addresses cache hierarchies. Combining these primitives in a program for hybrid platforms – though feasible – is often difficult for an average programmer. Lack of portability often leads to vendor locked in also. For example, CUDA programming paradigm works quite well for NVIDIA GPGPUs but inapplicable in other accelerators and co-processors.

The strategy taken by recent high-level programming language initiatives such as Co-array Fortran[73], UPC[42], X10[32], or Chapel[30] is to fix a machine model of the target execution platform – in these cases the partitioned

global address space (PGAS) model[37] – and provide primitives to use features of the model that are implemented with appropriate low-level primitives available in the target platform. Consequently the programmer is relieved from the specific details of a particular hardware. For example, PGAS exposes a model of a single globally addressable memory composed of the local memories of a collection of sequential processors. Thus the programmer can assume a shared memory environment when writing a PGAS program even if the underlying hardware is a distributed memory machine.

Heterogeneity of hardware platforms has been a major barrier for efficient implementation of the building block features of these languages across the board. Further, the simplicity of the machine model often makes it too restrictive. For example, PGAS model does not expose cache hierarchy and cannot portray hybrid architectures. In addition, most high-level languages still allow writing programs in a hardware oblivious manner. This strategy enables easy expression of parallelism but at the expense of loss of efficiency. Furthermore, there is no longer a clear correlation between runtime behavior of a program and its source code that hurts debugging performance. Given the paramount importance of performance in parallel programming, productivity at the expense of efficiency may not be an attractive proposition. No wonder none of the PGAS languages has been widely accepted by the parallel computing community despite a decade of development.

I argue that parallel programming should neither be hardware specific nor be hardware agnostic. Rather, the right approach is to adopt a middle ground: a *hardware cognizant* mode of programming.

1.3 My Proposition

The process of writing a parallel program involves designing an algorithm exhibiting adequate parallelism, determining how independent pieces of the program should communicate with each other, the granularity of those pieces, and finally their mapping to physical processing units (the four-step process, known as the Foster's methodology of parallel programming [46]). Good decisions in all these steps depend on knowing the characteristics of the physical platform on which the program will eventually run. Therefore, hardware concerns should be highlighted in all aspects of parallel programming. Nevertheless, instead of expressing the program's logic over the bare hardware features, we should express it over an abstraction of the machine that exposes its key underlying features along with their associated costs. This abstraction, called a *type architecture* [82], enables program portability without sacrificing performance. This is the central idea behind hardware cognizant parallel programming.

For this idea to work, the type architecture abstraction has to be generic enough to be able to describe most contemporary parallel hardware, and the languages that will serve as the mediums of expression have to expose and utilize that abstraction throughout the programming process effectively. My research touches both aspects of the type architecture based parallel programming. I propose a new type architecture that can describe the salient features of most parallel hardware of present time and a programming language for high performance parallel computing that operates on top of my type architecture abstraction.

The type architecture is called the *Partitioned Parallel Processing Spaces (PCubeS). PCubeS* describes a parallel execution environment as a hierarchy of parallel processing spaces. Each space is further partitioned into identical, independent processing units having a defined processing power and memory capacity. A space is fur-

ther characterized by its aggregate memory capacity and inter-component communication capability. *PCubeS* handles heterogeneity of target execution environments through its hierarchical breakdown of a platform and by supporting multiple models per hardware in unusual cases.

The programming language is called *IT*. *IT* is a language for high performance and mostly data parallel programming [a]. In an *IT* program, computations take place in a hierarchy of logical processing spaces each of which may impose a different partitioning for a data structure. The programmer has to explicitly map the logical spaces to the physical spaces of *PCubeS* to generate the executable. The compiler decides the proper implementation of computation and communication based on the mapping. The implementation of computation and communication varies further with the change of the target hardware architecture *PCubeS* describes. *IT* emphasizes separation of programming concerns and avoidance of compiler introduced non-deterministic runtime overhead. The expectation is that an *IT* programmer should be able to predict, debug, and analyze the suitability of a program and its runtime performance on a specific platform just by comparing the source code and the *PCubeS* description of the hardware.

These two components, *PCubeS* and *IT*, are tied together to substantiate the broader vision, the claim of my research:

> *A high-level parallel programming paradigm based on type architecture will enable portability, simplify learning and performance debugging, and approximate the efficiency of contemporary low-level programming techniques.*

[a] *IT* supports other forms of parallelism but the focus is on data parallelism

1.4 Research Contributions

The *PCubeS* type architecture, the *IT* programming language, and the implementation of the *IT* compilers are the major contributions of this research. The combination of the first two forms my programming paradigm that is referred as the *PCubeS* + *IT* paradigm in the rest of this writing. To show that *PCubeS* + *IT* paradigm is a practicality – not just a research proposition – I have developed three *IT* compilers for three different hardware architectures.

These architectures are multicore CPUs, distributed memory supercomputers or compute clusters, and hybrid supercomputers having both multicore CPUs and NVIDIA GPGPUs as nodes. These architectures are referred as *multicore*, *segmented-memory*, and *hybrid* respectively in the rest of the writing. The compilers generate C++ programs parallelized with MPI, Pthreads, and/or CUDA constructs from *IT* source codes as appropriate for respective target hardware.

I then took *IT* implementations of five representative application kernels (matrix-matrix multiplication, LU factorization, conjugate gradient, finite difference approximation in a regular grid, and Monte Carlo area estimation), compiled them with each of the compilers, ran the executables on representative machines, and examined the results with respect to portability, productivity, and performance. Regarding performance, I compared *IT* executables against handwritten versions, and conducted strong and weak scalability analysis.

The results are promising. With respect to portability, the exact same code was executed on each architecture. With respect to productivity, the IT programs are consistently shorter, less cluttered, and easier to comprehend than their hand-written counterparts. With respect to performance, the strong and weak scaling results are convincing and what one would expect for the respective application kernels.

With respect to the absolute performance compared to the hand-written counterparts, the early unoptimized results are encouraging. In both multicore and segmented-memory architectures, *IT* executables' performance is consistent with what a reasonably good programmer can achieve. These are just early results and significant optimization remains. Hence there are reasons to be optimistic about the paradigm's success.

Currently *IT* executables' Performance in the hybrid architecture is considerably lagging their hand-written counterparts. I am having difficulties building a performance model of the parts of the code that execute inside the GPUs. Not having a reliable performance model makes good code generation difficult. Furthermore, there are redundancies and inefficiencies in the CPU and GPU interaction that I have to mitigate. I am already working on these issues.

Two technical reports have been published on the language[96] and the type architecture[94]. There is a paper on the overall paradigm also[95]. Another paper discussing the performance results with sample applications on three target hardware architectures is in the planning phase.

1.5 Structure of the Dissertation

Chapter 2 discusses parallel programming tools, techniques, and languages relevant to my research. In particular, the chapter provides a comparison of the *PCubeS* + *IT* paradigm with other contemporary parallel programming research initiatives. Type architecture based programming is not a new concept. The chapter also presents previous research in this direction and analyzes their lack of success.

Chapter 3 presents a detailed investigation on the *PCubeS* type architecture, its design rationale, and its applicability in contemporary parallel machines. Furthermore, the chapter provides examples of *PCubeS* de-

scriptions of some modern hardware.

Chapter 4 gives an overview of the *IT* language. Although the chapter talks about features and syntax, much of the conversation is focused on the philosophical underpinning of the language and its behavioral characteristics. This is done to avoid diverting readers' attention from the core research objectives.

Chapter 5 illustrates how the concepts of the language are mapped to the features of the target hardware, and the consequence of that mapping on the compilation process and the runtime behavior of an executable. The three compilers being developed as parts of this research implement a common runtime environment. The chapter describes that runtime environment also.

Chapter 6 discusses the three compilers. The chapter gives a broad overview of the compilers and does not delve into too much internal detail to keep the discussion short and centered at the research objectives. Nevertheless, implementations of critical runtime elements are described for individual compilers so that the readers understand how the *PCubeS* + *IT* paradigm becomes a reality across hardware architectures.

Chapter 7 presents and analyzes experimental results with *IT* implementations of five well-known building block applications on three different target platforms. The lessons and the consequent directions for future research learned from these experiments are also discussed along the way.

Finally, Chapter 8 concludes with a reflection on the research findings, experience conducting the research, and what should be the short and long term plans for the future of this work.

2

Related Work

In this chapter I discuss prominent work on parallel computing tools, techniques, and approaches of the past and present. The tools, techniques, and approaches are presented in a manner to form a historical perspective of parallel computing. Along the way, I provide an assessment of their relative success and failure and my expectation about the future of parallel computing that leads to my proposed programming paradigm. At the end, I contrast my proposal with other contemporary research on parallel computing to clarify the differences

in their philosophical underpinning.

Many notable parallel computing projects – in particular, those that put less emphasis on high-performance – are not discussed here as the goal is not to provide a survey of the field; rather to establish the context for my work.

2.1 Approaches to Parallelism

Development of a parallel program – as mentioned in Foster's influential work[46] – can be divided into four distinct activities: partitioning data structures and the computation into independent units, determining communication requirements among the units, agglomerating units into tasks of appropriate grain, and finally mapping those tasks to processors. An intuitive understanding of these activities was there long before the advent of Foster's methodology of parallel computing. It was well understood that parallel programming involves much more than mere translating an algorithm into an executable code, but researchers greatly differ in segregating the compiler's and programmer's responsibilities the activities parallel programming entails. Consequently, a wealth of research has been done that expresses these differences of opinion. Parallel computing literature is replete with research on fully parallelizing compilers, parallel extensions to sequential languages, parallel languages built out of sequential cores, and finally built-from-scratch parallel languages.

2.1.1 Parallelizing Compilers

Research on parallelizing compilers had the vision that a programmer's involvement in writing a parallel program should be minimal. The central assumption here is that a compiler should be able to identify and group

parallelisms inherent in a sequential program and map them appropriately to processors. Most works that being done on this area such as Fortran D[55], Parascope[33], Polaris[22], and SUIF[50] focused on parallelizing programs written in Fortran and C, languages that were already popular within scientific communities. These compilers used sophisticated data dependence and control flow analysis, in both intra and inter-procedural levels, to transform a sequential code into parallel node programs.

Unfortunately, the amount of parallelism that can be eked out of a sequential program is often quite low. In addition, often times, significant scopes of parallelism escape compiler analysis due to the presence of false dependencies and aliases. Therefore, these compilers had much less success in parallelizing than what they envisioned.

On the other hand, often the amount of parallelism discovered in a sequential program is high but so fine-grain that its efficient exploitation becomes difficult. This latter problem is the source to the principal reason behind parallelizing compilers' lack of adoption: compilers' failure to effectively group and map node programs into processors. With the rise of distributed memory multicomputers, where the cost of communication can become several times higher than that of computation, there is so much to lose due to inefficiency in program placement. Consequently, initial enthusiasms for parallelizing compilers were soon all gone.

2.1.2 Parallel Extensions to Sequential Languages

Significant research has been done to extend sequential languages with mechanism for parallelism. Supports for parallelism may be incorporated as language features or as libraries that a programmer can use. Viena FORTRAN[31], High Performance Fortran[62], pC++[70] are bygone examples of the first approach. These languages extend a sequential FORTRAN or C core by introducing distributed arrays and constructs like parallel for

loops. Note that the idea of distributed arrays was already in Fortran D[54], but the difference now is that the programmer has more control regarding allocation of the pieces of a data structure in processors. These languages, exposing a shared memory model of the execution environment, were ported in both shared and distributed memory machines; and saw limited success in terms of performance and adoption.

Recent partitioned global address space (PGAS) languages such as Co-array Fortran[73], Titanium[97], or UPC[42] bear the same spirit of maintaining a shared memory view of the environment. Here the shared memory is, however, visibly partitioned among available processors. The programmer distinguishes between local and nonlocal memory references and responsible for explicit synchronizations. A problem with both past and present extensions of sequential languages is that they are built on top of an already feature-heavy core. One can argue that mixing of data distribution, synchronization, and similar other parallel constructs with the features of the core language makes the program harder to get right and less readable.

The library based approach of extending sequential languages to support parallelism, however, has achieved widespread success. The message passing interface (MPI)[91] is the standard for parallel programming for over two decades. The OpenMP[36] directive based parallelization is also popular for easy parallelization in shared memory environments. Efforts have also been taken to combine the two to improve MPI codes' performance in hardware platforms having distributed shared memory multi-processors[26], but the results are inconclusive. A central reason for MPI and OpenMP's success is that they require minimal learning over a programmer's existing knowledge of C or FORTRAN. To parallelize a code using OpenMP, the programmer just adds few pragma directives on top of loops. Although one can do only so much with pragmas, often-times that is all what he/she needs. On the other hand, MPI, that extends C and FORTRAN with explicit communication facilities, provides the programmer with absolute control regarding exploitation of parallelism. It is difficult

to beat a hand-tuned MPI code with code generated by other means due to MPI's expressive power. On the flip side, MPI programs are often accused of being difficult to get right, difficult to debug, and less readable. Freedom of expression in MPI comes at the expense of lack of structures in the program and a preponderance of communication tricks that often overshadows the underlying algorithm. These issues often discourage new programmers from using MPI despite its ubiquity.

Similar to Von Neumann style imperative languages discussed above, considerable expectations were there about prospects of functional languages in parallel computing. Functional languages are known for their brevity, mathematical rigor, expressiveness, and elegance[17 88]. They are widely used by domain scientists[53 60] for analytics and simulations, particularly when infinite list manipulation is required. The most attractive feature of functional languages regarding parallel programming is that they are inherently parallel: nothing more is added to the language to support parallelism. If the underlying algorithm has it, any inherent parallelism in a program can be easily discovered through compiler analysis. Therefore, theoretically, only a runtime support is needed that will dispatch independent pieces of computations to available processors. Unfortunately, functional languages in their unmodified form faced the same peril that imperative parallelizing compilers faced with the advent of distributed memory machines, and for the same reason too: a lack of success in automatic task agglomeration and mapping[93]. For functional languages the inefficiency problem was more intense as parallelisms exposed by a functional code are usually too fine-grain to be exploited effectively.

Consequently, functional languages are extended with different forms of explicit or semi-explicit parallelism supports that can be implemented more or less efficiently. Parallelism is introduced via programmer's annotations, explicit task creations, parallel skeleton functions, and data-parallel arrays under various names in different languages[51 52 28]. In our opinion, most of such innovations are in essence against the spirit of functional

programming as a program may no longer be viewed solely as a mathematical object being manipulated by repeated applications of functions – it is cluttered with parallel directives. Parallelism supports in the popular Haskell programming language is a good example in that regard,[86]. Presently Haskell embodies almost all forms of parallelism ever proposed in any functional language, including annotations that dispatch computation exclusively in accelerators! The proponents of Haskell believe that a comprehensive set of parallelism constructs should bring the language in the forefront of parallel computing, but we would argue that their proposal of 'learn only the parts you need' of a feature-rich language is in discord with the principals of good programming languages design that promote a lean language core[56].

2.1.3 Built-from Scratch Parallel Languages

Dataflow Languages

A close companion of functional programming, data-flow programming has among the first parallel programming languages that are built with the objective of efficient execution in parallel architectures[39,89] in mind. As opposed to in an imperative program where an explicit flow of control manipulates computations over data, in a data-flow program there is no control flow; rather the computation flows along the pieces of data as they become available. Data-flow languages such as VAL[68] and Sisal[45] present a syntax that is similar to that of a typical imperative language, but operations are restricted to single assignment per variable to facilitate flow analysis. Like functional programs, data-flow programs expose fine-grained irregular parallelisms. Researchers of data-flow languages, however, had the optimism that efficient data-flow supercomputers can be built. Out of that optimism, hardware like the Manchester data-flow machine[48] had indeed been built, but they could not

meet the efficiency demand. Overhead of token-space management and associative memory matching were the principal reasons behind their inefficiency.

Nonetheless, as an idea, the view of a computation as a flow governed by availability of data retains its attractiveness. Presently, coarse-grained data-flows or work-flows have widespread use in scientific communities. Work-flows like DAGuE[23] and Pegasus[38] have limited expressive powers compared to a typical data-flow language. Here a computation can only be modeled as a directed acyclic graph, which is too restrictive for most conventional parallel programs. Accordingly, work-flows are developed as tools or engines for managing inter dependent tasks, never as language alternatives.

Imperative Languages

Work on 80's and 90's such as Occam[59], Linda[27], PLITS[44], and Concurrent C[Gehani & Roome] are examples of languages that are built with explicit emphasis on parallelism. Most, if not all, of these languages are variants of Hoare's communicating sequential processes (CSP)[57]. They model the computation of a parallel program as decomposed among a set of processes that interact with one another by exchanging messages. Their differences lie mainly in how they expose communication channels used by the processes to the programmer. Such an interpretation of a program is, however, too coarse-grained for most cases of high performance parallel computing, as mentioned in[51]. To elaborate, it is easy to view a client-server application as communicating sequential processes, but the same is not true for a system of linear equations solver. Unfortunately, the latter – not the former – represents the kind of problems that are dominant in parallel computers. Consequently, CSP languages, although quite successful in operating system and distributed system programming, did not become main-stream in the arena of parallel programming. Interestingly enough, an MPI program is hardly

anything more than a CSP.

Parallel programming languages that came out in recent years are, in the contrary to their predecessors, directed towards high productivity[65]. Chapel[30], X10[32], and Fortress[83] – the three DARPA funded languages – have manifold features for expressing, grouping, and mapping parallelisms. Fortress even has all its expressions treated to be parallel by default. Benchmark programs written in these languages often run on par with equivalent MPI programs, but the languages' future wide adoption is still uncertain.

A wide assortment of features can make efficient compiler development for these languages difficult, and so far they have limited success in that regard. Further, a language that allows you to express all forms of parallelism is not necessarily the language that guides you to choose the appropriate form for a particular problem, and in our opinion that is precisely the problem with contemporary parallel programming language initiatives. The desire to support everything may also lead to poor performance. Fortress is a glaring example of this problem. The project itself has been terminated due to lack of efficient implementations of proposed features.

Among the few parallel programming languages that ever emphasized learn-ability, NESL[21] is most notable. NESL asks a parallel algorithm to be understood as characterized by two parameters: work and depth. The work is the total amount of computation involved and the depth is the number of steps. The running time of a program on a fixed number of processors can be estimated by analyzing these two parameters of the underlying algorithm. The programmer is expected to write a code that strikes the optimal balance between its work and depth. A mechanism introduced in NESL called nested data parallelism has also received a wide acclaim. Programs exhibiting nested data parallelisms can be transformed by a compiler into flat data-parallel programs that run efficiently in most parallel computers. Many interesting programs are, regrettably, not nested data parallel. So NESL has a limited applicability.

2.1.4 Domain Specific Parallel Languages

Finally, developing domain specific languages (DSL) and toolkits is an emerging trend in high performance parallel programming[66]. A DSL grows from a general purpose language and provides efficient application specific abstractions for common data structures and problem patterns. There are frameworks to develop DSL over low-level primitives such as MPI and threads[74], and the performance of DSL programs are often competitive to general purpose language implementations. Nevertheless, as their nature suggests, they cannot be the solution for high performance parallel computing in general.

2.2 Hardware Modeling in Parallelism Approaches

Earlier attempts on introducing parallelism, regardless of the specific form they took, were much more concerned about expressing different forms of parallelism than efficient realization of those expressions. Given performance is the central concern, many projects failed due to the lack of it originating from the absence of hardware support for efficient implementations of parallel primitives or incorrect modeling of the hardware itself. Snyder in his seminal work 'Type Architectures, shared memory, and the corollary of a modest potential'[82] explicates this problem at length.

Snyder posits that a language should expose a type-architecture of the environment where a program will run. The type architecture working as an intermediary between the hardware and the language should expose key hardware facilities available with their associated costs so that the programmer and the compiler both can perform appropriate optimizations in the process of translating a parallel algorithm into an efficient executable.

2.2.1 *Parallel Type Architectures*

Although the notion of type-architecture remains attractive since its proposal, few parallel type architectures have been proposed so far and there is none that has been universally accepted. Some are not even formally presented as type architectures.

PRAM or paracomputer was proposed in 1980's by Schwartz[79] and implicitly accepted by the theory community as a type architecture. PRAM's view of a parallel execution environment is of an arbitrarily large number of identical processors sharing a common memory where any number of processors can read and write simultaneously at a unit cost. The problem with paracomputer is that it is unrealizable. Although it makes designing parallel algorithms easy, Snyder deftly explains how its impracticality can easily fool someone into unwittingly choosing sub-optimal algorithms[82].

SIMD or Single Instruction Multiple Data type architecture[81] was temporarily popular in 1970s and 1980s during the time of machines like CM-1, CM-2, ILLIAC IV, SIMDA, etc. A SIMD machine is described as a computer system that has a controller unit and a fixed number of processing elements that are connected by some interconnection network. The controller unit broadcasts an instruction to the processing elements and the processing elements that are active at that time all execute that instruction on different pieces of data. Present day accelerators' Single Program Multiple Data (SPMD) execution model can be tracked back to a SIMD origin, but pure SIMD is no longer a candidate for type architecture for modern machines.

CTA is offered by Snyder as an alternative to PRAM as the first Candidate Type Architecture, hence the name. CTA describes a parallel hardware as a finite set of sequential computers connected by a fixed, bounded degree graph, with a global controller[82]. CTA may be a good choice for describing purely distributed memory systems

of last decades, for present day hybrid machines, it is plainly inadequate.

`Systolic Architecture` gained some momentum in late 1980s and early 1990s[63]. CMU's Warp machines are good examples of this architecture. In a systolic system, data flows out from the computer memory in a regular order, passes through a series of processors, and then returns back to the memory. Multiple independent series of processors, aka assembly lines, work simultaneously to provide parallelism. Systolic systems offer a data-driven programming paradigm that is fundamentally different from the instruction-driven paradigm of Von Neumann architecture that is at the heart of present day parallel machines. Since multiple operations can be done on a single piece of data, systolic systems are better able to balance computation with IO. Regardless, systolic architecture is suitable only for highly specialized systems – not for general purpose parallel computing – as it is difficult to map most algorithms to its restrictive model.

`LogP and LogGP` type architectures are proposed for distributed memory machines[35,12]; and unlike the previous two examples, they take a parametric approach to type architecture description. LogP describes a machine in terms of four parameters: communication latency (L), communication overhead (o), minimum gap in successive communications (g), and finally processor count (P). Later the proponents added another parameter (G) for capturing long message communication bandwidth to form LogGP. It is possible to gauge the expected performance of an algorithm to a great precision using LogP or LogGP parameters, but one might find it difficult to imagine how a programming language's features can embody those parameters.

`PMH` or Parallel Memory Hierarchy describes a parallel computer as a tree of modules that hold and communicate data with only leaf modules being able to do computations[13]. Each level in the hierarchy has four attributes: the size of memory blocks, memory capacity in terms of blocks, communication delay in transferring a block between a parent and a child, and child count. Although PMH is better suited than the previous

examples in capturing the hierarchical nature of recent parallel architectures, its lack of concern for processing capacity and elimination of direct module-to-module communication within a level make it too conservative a type architecture for many high performance computing problems.

PGAS is proposed more as a programming model than as a type architecture[Stitt], but essentially it serves both purposes. PGAS or Partitioned Global Address Space describes a shared memory execution environment composed of sequential processors. The shared memory is, however, not a conventional single unit; rather it is the collective sum of the local memories of individual processors. Each processor has access to the entire memory but access time varies depending on the locality of the reference.

OpenCL or Open Computing Language offers a programming model and a hardware abstraction both at the same time[85]. Its type architecture description is of a host CPU and any number of attached OpenCL 'devices.' Each such device contains one or more 'compute units' each of which holds one or more SIMD 'processing elements' to execute instructions in lockstep. There are four types of memory: a global device memory, a small low-latency read-only memory, a per unit shared memory, and finally a per element private memory. One or more of these memories can be missing in a particular platform. This is quite a rigid description to be general purpose. Hence OpenCL is restricted to hardware accelerators and to some cell processors and multicore CPUs only.

2.2.2 Type-Architecture Based Languages

Early attempts on type-architecture based languages are few and far between. Most of those languages suffer due to the inadequacy of the type-architecture as well as the lack of important parallel features in the languages. Snyder's own Poker[72] programming language suffers from difficulties in mapping many algorithms to the CTA

type-architecture. His subsequent type-architecture base language ZPL uses the same type architecture[82] and found to perform well on Cray machines though. Type architectures such as LogP[35] and LogGP[12] serve as more of an analysis tool and, to the best of our knowledge, never beget any language. Space Limited Procedures[14] and Sequoia[43,18] are two languages developed with PMH as the underlying type-architecture. These languages are restrictive in the sense that they provide no support for communication between parallel task units that rules out many common computational problems.

In recent years, there is a renewed interest for type-architecture based languages among which PGAS languages are most prominent. Co-array Fortran[73], Titanium[97], UPC[42], Chapel[30], and X10[32] are all PGAS languages. The first three are extensions to sequential base languages and the last two are built-from-scratch parallel languages. These languages make a tradeoff between hardware modeling and programming features enrichment that is different from that of their earlier type-architecture based counterparts. In the contemporary languages, the primary focus is still on enabling diverse forms of parallelism, but mechanisms have been provided to make those features sensitive to the PGAS model of the target execution platform. In particular, all these languages allow programmatic control over computation and data locality.

Although to a lesser extent, similar development can be observed in parallel functional languages. For example, Legion[19] is a functional language follow-on of Sequoia[18] that uses PMH. Surprisingly, the type architecture aspect is not highlighted in Legion publications, making it hard to gauge PMH's impact on the language. Nevertheless, Legion supports hierarchical partitioning and programmer controlled mapping of those partitions to different layers of a hardware. At the time of this writing, two domain specific languages have been built on top of the Legion programming model and Legion has a compiler that generates executable for multicore-CPU + GPGPU hybrid platforms.

Coprocessor Languages

Coprocessor languages such as CUDA and OpenCL also fall into the type-architecture based language category as their efficient usage depends on the proper exploitation of hardware features exposed through the respective language paradigms.

CUDA[71] is the programming tool used for NVIDIA GPGPUs. Initially it was deemed to be difficult, but GPGPUs' popularity in high performance computing gave it a rapid surge in acceptance. CUDA follows the footsteps of earlier SIMD/SPMD languages (C*, pC++, C**, DataParallel C) with additional features to manipulate memory unique to NVIDIA architectures. A CUDA program is a C or Fortran program with functions to be offloaded to the accelerators and additional instructions for data transfers between a CPU host and accompanying accelerators. The programmer needs to understand the inner working of the accelerator threads and details of memory access to make his/her offloading functions to behave correctly and efficiently.

The OpenCL[85] standard has a strong CUDA heritage and was originally targeted for accelerators only. Recently, it has become more of a standard for parallel programming for multicore CPUs and GPUs alike. It is, however, unlikely to be successful beyond single work-station machines as the OpenCL type architecture is inapplicable to distributed systems.

2.3 The Current Landscape of Parallel Computing

After several decades of research on parallel computing, the dominant mode of parallelism is still low-level sequential languages extended with parallelism support. As mentioned earlier, MPI is the standard for parallel computing in distributed memory supercomputers and workstation clusters. OpenMP and Pthreads[25] both

are immensely popular in shared memory systems. CUDA has become the standard of parallel computing in all NVIDIA GPGPU platforms. All these tools perform excellently in their respective target platforms. Although producing optimized code using these tools is an engaging endeavor, they have a low learning curve and writing a moderately efficient program is often quite easy.

The problem arises with the advent of hybrid architectures. Some of the most powerful supercomputers of the present such as Tianhe-2 [top], Titan [tit], Stampede [sta], and Mira [1] are all hybrid machines. Hybrid architectures are becoming the norm even in the personal computing domain as currently most desktop and laptop computers come with a multicore CPU and an accelerator. These trends portend trouble for low-level programming approaches as combining those tools in a single program to get the best out of a hybrid platform is considerably difficult. Furthermore, the paradigm differences in MPI, CUDA, and OpenMP make it a daunting task for an average programmer to correctly combine them.

This productivity problem is the principal motivation behind X10 and Chapel, two prominent high-level parallel programming languages under development. Both projects were initiated in the beginning of the new millennium through DARPA HPC initiative, have ongoing industry support, and dedicated development team. Both languages have many features allowing expression of a wide variety of parallelisms. On the other hand, their low-level building block elements regarding computation and data locality allow a programmer to some extent control the runtime behavior of the high-level parallel constructs.

Unfortunately, even after a decade of development, both languages have limited success in generating efficient executable for large-scale system. At the time of this writing, Chapel has its compiler for multicore CPUs generating efficient binaries, and according to Chapel developers, the binaries generated for distributed shared-memory supercomputers exhibit mixed performance due to communication inefficiencies [29]. Support-

ing hybrid supercomputers having both multicore CPUs and accelerators is still a future concern. X10 has a java compiler for multicore CPU platforms. The distributed shared-memory X10 compiler, however, works for IBM's blue-gene Q systems only.

It seems to us, these two languages will continue to have significant challenges in generating efficient code for large scale PGAS machines due to having features (e.g., X10's arbitrary nesting of dynamic parallelism) that are difficult to implement efficiently at scale. Furthermore, the failure of the PGAS model itself in accurately describing present-day deeply hierarchical and heterogeneous hardware architectures should continue to be a significant barrier for Chapel, X10, and other PGAS languages' good performance.

2.4 PCubeS + IT Paradigm against Contemporary Parallel Solutions

2.4.1 Differences in Philosophical Foundations

The *PCubeS + IT* programming paradigm is based on the observation that writing an efficient parallel program requires reasoning about hardware features in all four component activities identified by Foster [46]. So hardware concern should not be an afterthought that applies only at runtime mapping of a finished program's features to processing units as done in X10, Chapel, or Legion; rather the entire program development process should be cognizant of the hardware features and their costs.

Despite of its hardware-centric program development approach, the paradigm has to be portable across diverse parallel platforms to be a practical alternative to existing tools that are intended and tuned for specific platforms. This is a major problem in languages like X10 or Chapel that their applicability is limited beyond the PGAS type hardware platforms. On top of that, those platforms already support efficient low-level pro-

gramming alternatives to the emerging languages.

To elaborate on that note, the type architecture in the proposed paradigm serves as a common grammar to describe diverse types of hardware architectures and aids in efficient program development as opposed to as a template description of a specific architecture that rules out many others. This is a very important distinction *PCubeS* has with other type architectures, and the main reason *PCubeS* describes machines in-terms of their programming capabilities and allows multiple machine models for the same hardware.

Further, *PCubeS* + *IT* paradigm strives to make writing highly efficient programs simple instead of just enabling them. In that respect, the paradigm surpasses even tools like MPI and OpenMP. For example; it is true that highly efficient, cache-sensitive code can be written using OpenMP; but it is much easier to achieve the same feat using *IT*. This behavior is part of the paradigm's broader philosophy 'the medium is the message' which suggests that an effective programming paradigm should serve not only as the tool but also as the guide towards developing efficient programs.

Finally, *PCubeS* + *IT* paradigm emphasizes that the execution-time behavior of a program should be obvious from its source code description and the machine model of the target platform. This principle leads to the breakdown of responsibilities where the programmer decides everything about his/her program behavior and the compiler only efficiently implements those decisions. Further, this principle requires *PCubeS* + *IT* to have a heightened interest in readability and performance debugging compared to existing low and high-level parallel programming alternatives.

2.4.2 Differences in the Language Development Approach

Underlying philosophical differences, as expected, make *IT* a very different kind of language than others. The distinction will be apparent in subsequent sections. Each language or tool brings in some unique flavor and style in the art of programming. It is difficult to make a qualitative comparison of programming tools and techniques in terms of their features set. Therefore, that comparison is skipped here.

Nevertheless, it is important to draw attention to an important difference between *IT* and other ongoing parallel programming research projects regarding the nature of the language development. From the onset of the project, the decision was to keep an extremely lean language definition and add, remove, or change features based on their feasibility and performance across target platforms. This strategy has the direct consequence that *IT* already has three compilers for three different platforms in only three years while X10 and Chapel, bogged down with their extensive feature sets, are yet to support more than two platforms even after a decade of development.

This may raise the question if the proposed paradigm is just an academic endeavor without a vision for practical usage in the future as real-world applications widely vary in feature requirements. The answer is NO but with a caveat. The project at the current stage is an ongoing investigation on productive, high-performance, parallel programming in a portable fashion for present and future parallel systems. As the language continues to mature during the investigation, plans on making it available for the community will be made.

Regarding future adoption, *IT* is likely to face fewer obstructions than other recent proposals because *IT* does not try to replace MPI/Pthreads/CUDA – rather *IT* compilers are built over those tools and will continue to benefit from their improvement – *IT* only argues that those tools should be part of the compiler infrastruc-

ture as opposed to details that a programmer has to deal with. This reliance on existing technologies will make it much easier to port *IT* across platforms than other recent languages that strive to replace those technologies.

3

PCubeS Type Architecture

This chapter presents the type architecture foundation of my proposed programming paradigm. This chapter starts with a brief discussion on the current trends on parallel architectures to set up the context for my type architecture. As the notion of type architectures may be unknown to many, a short introduction to type architecture succeeds the discussion on trends. That is followed by a discussion on my *PCubeS* type architecture, its elements, and design principles. To provide some practical examples of how present day parallel architectures

can be described using *PCubeS*, detailed case studies of two supercomputers' *PCubeS* modeling are presented at the end.

This chapter is an abridged version of our technical report on *PCubeS* available here[94]. I also encourage interested readers to look at Snyder's seminal work on Type Architecture[82] that I refer to many times in this chapter.

3.1 Current Architectural Trends in Parallel Hardware

One of the most significant differences of present day both large and small scale parallel architectures from architectures of just a decade ago is that they tend to be constructed out of building blocks that are parallel computing units themselves. For large scale parallel systems, this trend means that purely shared memory such as SGI's Blacklight or purely distributed memory such as IBM's Blue Gene L systems of yesteryears are now extreme rarities. Machines like Blue Gene P and Q systems, next generations of L and dating only a few years back (2007 and 2012 respectively), or the Ranger supercomputer in TACC (2008) were already offering a hybrid computing environment connecting multicore processor nodes in a distributed manner. This trend is further intensified as recent machines such as Stampede and Titan have not only multicore CPUs but also hardware accelerators within nodes to offload computations from the former.

This push toward hybrid architectures is not for any conceptual clarity or programmatic simplicity; on the contrary, hybrid architectures make it more difficult to achieve those objectives. The push is due to memory and power wall barriers that have become[16] major concerns in computer architecture since the beginning of this millennium. As these concerns are unlikely to go away in near future, proliferation of hybrid architec-

tures should continue, and proper attention must be given to their deep processing and memory hierarchies in programs for effective utilization of available capacities.

Let us briefly examine the component and overall architectures of typical, present day large scale parallel computing platforms to better comprehend their hierarchical nature.

3.1.1 Node Architectures

A typical supercomputer of current time has multicore CPUs, hardware accelerators, or one or more of both as a node. The number of processor cores per CPU varies significantly from supercomputer to supercomputer. For examples, an IBM Blue Gene Q system uses 18-core PowerPC, Stampede 8-core Intel Sandy Bridge, and Titan 16-core AMD Opteron as their multicore CPU nodes. Sometimes the individual cores are simultaneously multithreaded such as the 4-way multithreading of PowerPC cores. That further multiplies the parallel processing capacity of the CPU.

These cores are general purpose execution units and typically have their own L1 and L2 caches and share an L3 cache among them. A memory controller connects an external RAM to the CPU to be equally accessible to all the cores. Figure 3.1 shows the block diagram of a 16-core AMD Opteron CPU as an illustrative example. We already see a rich hierarchy in the architecture of a single CPU that should be taken into consideration while writing a high-performing program.

At the same time, hardware accelerators that were initially popularized by the gaming industry and computer graphics are becoming increasingly common in supercomputers. For example, Stampede has an Intel Xeon Phi co-processor alongside 2 Sandy Bridge CPUs and Titan has an NVIDIA Kepler K20X alongside the AMD Opteron CPU in each compute node. Consider an NVIDIA GPU, the difference between a multicore

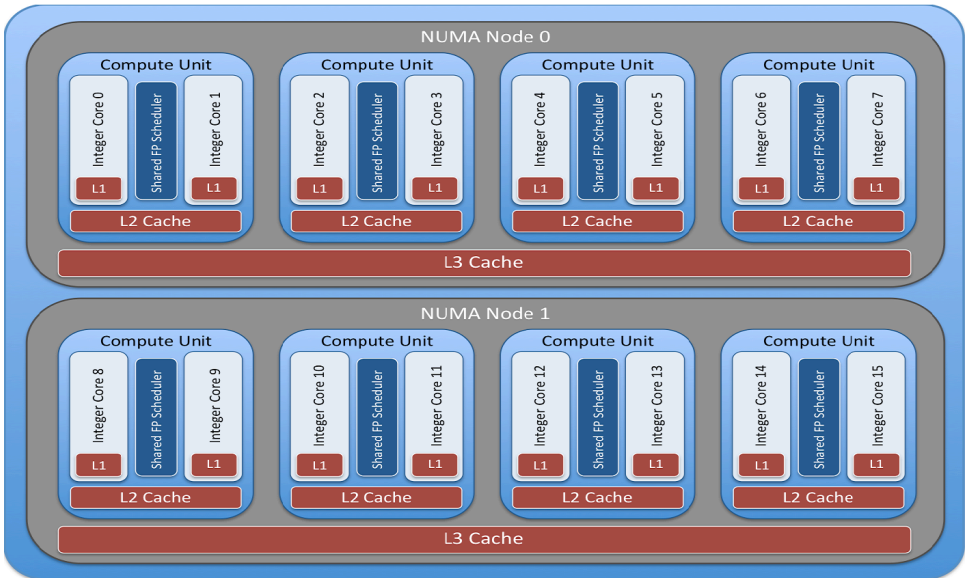


Figure 3.1: A 16-Core AMD Opteron CPU (Source: AMD Corporation)

CPU and the GPU is that the latter has a large number of streamlined, simple cores running in parallel in a SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) manner as opposed to the independent processing model supported in the heavy, general purpose cores constituting the former. A large number of scientific computations have significant regularity and data parallelisms that these GPUs are particularly suitable for, explaining their popularity.

Just like a contemporary microprocessor, accelerators have a hierarchy in the organization of their processing elements, and it is critical to give proper attention to that for efficient utilization of these hardware. For example, if we look at the construction of the NVIDIA Kepler K20X GPU that is been used in Titan, we see its scalar cores are grouped under 14 symmetric multiprocessors that work independently of each other but share the on board DRAM and a small L2 cache. Figure 3.2 depicts the block diagram of the GPU [a]. Each symmetric

[a] K20X GPUs used in Titan have one SM disabled [ana]. So the number of active SMs is 14 – not 15.

Figure 3.2: Block diagram of a NVIDIA Kepler GPU having 15 SMs (Source: NVIDIA Website)

multiprocessor (SM) can run upto 2048 threads by multiplexing them to its 192 scalar cores [a]. An SM has a small, 64 kB, memory to be shared by the cores. Furthermore, these threads do not work independently – they get dispatched as groups of 32 lockstep threads that are called warps in NVIDIA terminology. Shared memory access alignment within the threads of a warp, and global memory access alignment among successive transactions made from an SM are critical for good performance in this environment.

A similar, albeit different, hierarchical decomposition can be discovered in Intel's Xeon Phi co-processor [b]; the only other accelerator currently been extensively used in high performance computing.

To summarize our discussion on node architecture, we see two points coming out as important. First, there is significant processing power and memory capacity available within a single node; and second, it is inappro-

[a] There are 2688 scalar cores in total.

[b] Each Intel Xeon Phi core has a vector processing unit that can execute 16 single precision or 8 double precision operations per cycle.

priate to view a node as a flat collection of processing units. Apart from in programming large-scale machines, this trend has significance in the domain of personal computing too. As recent personal computers and workstations are regularly equipped with powerful multicore CPUs and one or more accelerators, it is possible to view them as good candidates for small and medium scale scientific computing. Consequently, programming frameworks that can simultaneously support both domains have the potentials for bringing great benefits.

3.1.2 Supercomputer Architectures

Turning our focus from individual nodes to entire supercomputers, the first noticeable thing is their sheer scale. Titan has 18,688; Stampede has 6,400; and Mira, a Blue Gene Q system, has 49,152 compute nodes. Given that the demand for more computation power always outstripped the capacity available, this trend towards larger and larger machines is unlikely to stop.

It is unthinkable to wire such massive numbers of nodes directly to one another. At the same time, connecting them in a uniform fashion through a sparse network does not give good communication performance either. Therefore, a typical supercomputer has multistage interconnection network or networks that divide the nodes into densely connected subgroups that are further connected by high-in-bandwidth communication channels of progressively sparser topologies [a]. Evidently, the interconnection network of a supercomputer has been subjected to a hierarchical breakdown too. Nature of this breakdown can make a particular application more or less efficient in a particular supercomputer.

For example, the nodes of Stampede are connected by a two-level fat tree Infiniband interconnect [sta]. Figure 3.3 illustrates the topology. It is obvious from the figure that all else being equal, a communication between a

[a] As an alternative, some supercomputers use multi-dimensional torus networks.

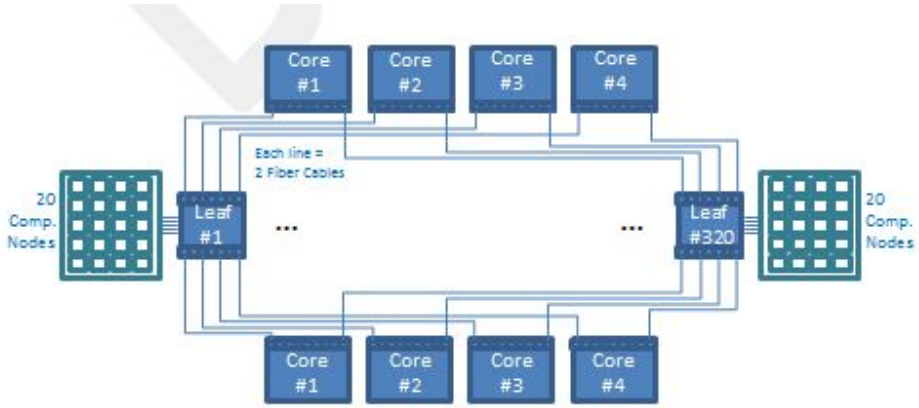


Figure 3.3: Stampede's Interconnection Structure (Source: Texas Advanced Computing Center)

pair of nodes within a single leaf should complete significantly faster than that between a pair of nodes belonging to different leaves.

In Mira, each compute rack or cabinet hosts 32 compute drawers with 16 drawers forming a mid-plane. A single mid-plane hosts 512 compute nodes that are electrically connected into a 5D torus topology, and beyond the mid-plane level there are only optical connections[1]. Again the differential nature of communication becomes evident.

From the programming perspective, this suggests that treating nodes differently based on their relative positions in the interconnection network may provide performance advantage. Some recent research, e.g. one done on Blue Waters system on topology sensitive MPI codes, bolsters that assumption[75] – here we need to have some reservation [a].

[a] Although it has been shown that if the nature of communications within a program exactly matches the interconnection topology of its execution platform then the potential for performance advantage is great, that is unlikely to happen for most programs[46]. Furthermore, graph mapping itself is an NP-complete problem; therefore, programming effort on such mapping even when it exists is prohibitive. Rather, we believe, considering positional difference at the group level such as intra-leaf and intra-plane as oppose to across-leaves and across-planes within programs is both manageable and beneficial.

3.2 Type Architecture Fundamentals

The huge capacities of present day massively parallel architectures may appear impressive, but historically their computational power was never quite enough to satisfy the demand made by their contemporary applications. This is because most interesting scientific applications are quadratic or above in their runtime complexity. Therefore, only a modest improvement in problem size and running time can be achieved through a linear increase of processors that parallelism offers. The exponential growth of data in large scale data analytics while each technological advance merely doubling available machine capacities, has made this demand and supply disparity even more intense in recent years.

It is not that we were unaware of this problem before. It has been identified as early as in 80's. Snyder in his 1986's seminal article 'Type Architectures, shared memory, and corollary of modest potentials[82]' points out that it is crucial to translate all of the capabilities of a parallel execution platform into useful computation, rather than losing much of that in implementation heat, to even modestly keep up with the growing demand of the applications. He argues for a hardware sensitive programming paradigm for writing parallel applications, and in so doing introduces the notion of a Type Architecture.

Snyder describes the endeavor of writing a program as a two steps translation process: from algorithm to program then from program to executable. The programmer is responsible for the first transformation and the compiler for the second. Overheads should be scrupulously avoided in both steps for the sake of efficiency. Given that the programmer did his/her best in writing the program, unwanted overheads may still arise due to limitations of the abstract machine model exposed by the programming language that works as the medium of communication between him and the compiler. Depending on how the language defines it, the abstraction

can be prohibitively expensive so that he/she has to work against it to get the program to run efficiently in a particular environment or it can be so low-level that he/she can write programs for only a particular class of hardware.

There is a rift between high and low-level programming techniques regarding the role their underlying abstract machine models play. Most high-level languages present a simple abstract machine model of the execution environment to simplify coding and enhance portability of written codes, but that model provides close to no guidance for efficiently exploiting the features of the environment. Thereby, it is often difficult to get good performance in high-level languages. On the other hand, low-level programming techniques make the programmer deal with minute details of the execution environment. Their abstract machine model is effectively the bare hardware. Hence efficiency can be attained, but with considerably greater programming effort and at the expense of program portability.

To resolve this tension between high and low-level programming techniques and combine the best of both worlds, Snyder proposes to adopt an idealized machine model that should serve as the standard hardware-programming language interface, and thereby provide the foundation for the machine abstraction of any parallel programming language. This interface is called the Type Architecture. In other words, the type architecture is a description of the facilities of hardware. To be effective, the type architecture description should bear the following two characteristics.

1. A type architecture must expose the salient architectural features of a hardware
2. And, it must accurately reflect the costs of those features

It is important to understand the distinction between a type architecture and the programming model, formally known as the abstract machine model, of a programming language. For example, the Von Neumann

architecture can be considered a type architecture for sequential machines. FORTRAN[69] and Lisp[67] present two different machine models on top of that. FORTRAN offers a programming style that fosters generic array operations; in contrast, Lisp offers a programming style that relies on recursive list manipulations. The type architecture here tells how the operations and primitives of these languages will be translated and how well they should perform in an execution platform. A successful programmer internalizes the cost of model-to-architecture translation and chooses his/her primitives and operations accordingly to write an efficient program.

To understand the different roles an abstract machine model and a type architecture play in programming from an alternative angle, consider the pseudo code of an algorithm. A type architecture description tells the programmer how to assess the efficacy of the algorithm on a target execution platform. The abstract machine model, on the other hand, tells him/her how to implement the pseudo code in a particular programming paradigm supported by that platform.

The further the abstract machine model goes away from the actual type architecture the more difficult it becomes for the programmer to make a correct assessment about the performance of his/her program. Consequently, the algorithm he/she uses may be inappropriate for the underlying execution platform. This is not a problem when performance is not a major concern, and that is often the case for sequential programming paradigms. That is, however, definitely not the case for parallel computing.

3.3 Partitioned Parallel Processing Spaces

The formal description of our proposed type architecture, The *Partitioned Parallel Processing Spaces (PCubeS)*, is as follows.

> *PCubeS* is a finite hierarchy of parallel processing spaces each having fixed, possibly zero, compute and memory capacities and containing a finite set of uniform, independent sub-spaces that can exchange information with one another and move data to and from their parent.

Before we define parallel processing spaces in more detail, we have two things to add on Snyder's notion of type architecture that are fundamental to our proposal and essential to understand the elements of *PCubeS*.

`Programmability:` Snyder uses the term 'structural feature' and 'facility' interchangeably. This is because he is more concerned about describing hardware features than about how they can be put to meaningful use in a program. We believe this approach of – describe first then derive programming models – is not right as the type architecture is intended for standardizing the hardware-language interface, and an interface design has to take into account the demands of both sides. Thus, in our opinion, a type architecture description should rather focus on the programmatic usage of hardware's structural features than on their actual working principal. To make the idea concrete through an example, the description should not bother if a vector is implemented as a pipeline or a SIMD lane. That the described hardware has a vector should be the primary concern. Furthermore, if the hardware is good for multiple modes of programming – as many contemporary supercomputers are – there may be multiple type architecture instances for it describing its features from different programming perspectives.

`Parameterization:` we think parameterization is needed for any type architecture description to enable

a programmer to make accurate estimation of the performance of his/her algorithm for a particular input set. To understand why it is important, imagine a CTA description of a parallel hardware where processors are connected through a complete binary tree network configuration. From the description, a programmer can deduce that on an average communication between a pair of processors should take steps logarithmic to the number of processors. To determine how frequently a processor should communicate and what should be the individual message sizes to balance computation with communication, however, he/she needs to know the actual latency and bandwidth of the network. Lack of such information hurts CTA's applicability. On the other hand, a type architecture description in terms of such values such as in the case of LogP should be avoided as that obstructs programmability. We believe a PMH[13] like description that has a core model parameterized by actual values is the proper middle ground. The programmer can use the model to design the algorithm and then the parameter values to assess a program's runtime performance.

3.3.1 Elements of PCubeS

Let us now characterize the elements of *PCubeS* to better understand how this type architecture can be used to describe present day parallel machines.

`Parallel Processing Space (PPS):` the notion PPS is used to describe any part of a parallel hardware or the entire hardware itself where a computation can take place and/or data can reside. For example, in case of a multicore CPU, both the CPU and its individual cores are spaces, the latter lying within the former as sub-spaces. From the perspective of the former, the latter are its *Parallel Processing Units (PPU)*. As part of a computation, a space can perform two fundamental operations: floating point arithmetic and data movement.

`Capacity of a space:` parallel processing and memory access capacities of a space are defined as the number of corresponding fundamental operations that can be done in parallel. For example, a SIMD thread group within an NVIDIA GPU's symmetric multiprocessor has 32 operations per clock cycle as its parallel processing capacity as 32 threads run in lock-step within each group. Note that the actual hardware implementation of a parallelization feature is not important; rather its programmatic manifestation is. This allows us to treat cores, SIMD lanes, vector pipelines all in the same way. We believe the speed of instruction execution is enough to expose their efficiency differences.

The memory, if it exists in the space, is characterized by its size and the number of transactions to and from that can be done on it in parallel. We use the term *transaction* to represent a single load/store operation or a communication. A transaction is further characterized by the volume of data it carries and its latency. For example, if a read/write operation by a thread within a dual hyper-threaded CPU core involves 8 bytes of data and takes 15 cycles to complete, and operations from both threads can take place simultaneously then the core's parallel memory access capacity is 2 transactions of width 8 bytes and latency 15 cycles.

`Uniformity and Independence of Sub-spaces:` given that an execution platform is viewed as a hierarchy of spaces, it is important that there is a guiding principal for breaking larger hardware components into smaller components to form sub-spaces – otherwise, any hardware can be described as a flat *PCubeS* hierarchy having a single space only. In that regard, we use uniformity and independence as the defining factors. Uniformity requires that not only all sub-spaces of a particular space have the same processing and memory capacities but also their information exchange with one another and data movement to and from their parents have the same average values for transactional attributes. Meanwhile, independence requires that operations done by different sub-spaces are independent. Whenever both of these requirements are met for a hardware

component then we divide its capacities into sub-spaces; otherwise we do not.

To understand how this rule works in practice, consider a supercomputer node having two 8-core CPUs. We cannot view a node as a 2-space hierarchy where the node is the higher space containing 16 sub-spaces, one for each core. This violates the uniformity requirement due to difference in intra-CPU and inter-CPU information exchanges among cores. There should be another space in-between that represents the individual CPUs to bring uniformity into the hierarchy. Similarly, the 32 lock-step threads of an NVIDIA GPU thread block cannot form sub-spaces despite being uniform. This is because their operations are not independent.

`Information Exchange:` it is important that we characterize interactions between sibling spaces as information exchanges instead of as communication or other platform specific term. This allows us to treat shared memory and distributed memory systems in a uniform way. Furthermore, within the umbrella of distributed memory architecture, different execution platforms may have different implementations of communication mechanism. The use of an abstract term enables us to mitigate these differences. An information exchange is characterized solely by its latency. We believe that is enough to capture the efficiency differences among different modes of interaction.

For example, if we consider a shared memory environment of four CPU cores sharing an L3 cache. Then information exchange between a pair of cores is equivalent to the sending core writing data in the cache and receiving core subsequently reading it. The only difference between such an interaction and a read following a write by a single core is that there should be a mutex operation signaling the completion of writing by the former. So the latency of information exchange in this case is the latency of the mutex operation. For another example, in a distributed memory system supporting two-sided communication, the latency of information exchange would be that of handshaking added to the latency of actual data transfer. For a system with one-

sided communication, the handshaking cost is replaced by the cost of setting a flag in the receiver's memory.

`Data Movement:` unlike in the case of information exchange, in characterizing data movements between a parent space and its sub-spaces all three transactional attributes – latency, width, and concurrency – are needed. Transaction latency and width are used to understand the cost of a single data motion between the parent and one of its children, and transaction concurrency is used to determine how many data movement operations can take place in parallel. For example, each symmetric multiprocessor (SM) in an NVIDIA K-20 GPU can read and write global memory at a maximum chunk size of 128 bytes that takes around 100 to 300 clock cycles. Assuming all 15 SMs in the GPU can initiate a global memory operation at the same time, the data movement between the sub-spaces representing individual SMs and the space representing the entire GPU has transactional latency of 100 to 300 cycles, width 128 bytes, and concurrency 15 operations.

When the space under concern represents a distributed memory segment of the hardware, transactional attributes need to be determined from the capacity of the communication channel and the settings of the communication protocol. For example, if we consider the space represented by a leaf in the fat-tree network of Stampede as shown in Figure 3.3 then the 20 nodes are its sub-spaces. In this case, transaction concurrency for data movement is the number of concurrent flow that can go in and out of the leaf router, latency is self-explanatory, and transaction width is the maximum amount of data that each packet within a flow can carry.

3.3.2 Design Principles

In designing *PCubeS*, we adopt a few principles that have programmatic significance. We will like to draw readers' attentions to these principles now.

Caches as Memories

PCubeS treats hardware managed caches as if they were programmable memories. This treatment of caches is unusual but we believe it is important to capture and expose the efficiency of present day multi-level cache based memory designs. Contemporary multicore CPUs and accelerators have significant capacities in their caches, and often time effective utilization of these caches becomes the determining factor for good performance. For example, a single core of an Intel Xeon Phi Co-processor has 512 KB of L2 cache and the latency ratios between accessing that cache and accessing the on-board DRAM is about 1:10. Evidently, ignoring the L2 cache is not a good choice.

On the other hand, treating a cache as a memory may raise the concern that the type architecture is describing a feature in a way that differs from its true nature. We believe that this is not a problem for two reasons. First, a cache's size is interpreted as the memory capacity in its *PCubeS* interpretation. That suggests that programs have to be written with the assumption that at runtime the memory can hold up to the cache size amount of data. Although actual loading and unloading of data to and from the cache is beyond programmatic control, since the working set can fit into the cache, it will behave more or less like a memory. The compiler should be able to generate addresses so that cache conflict is minimized or eliminated altogether.

Second, replacing the caches by programmable memories, or allowing a hardware switch for software controlled management of caches is not difficult to implement in the hardware [a]. Such provisions can bring performance benefits. As a type architecture is a vehicle for not only describing existing architectures but also for guiding future architecture developments, emphasizing programmatic manipulation of the caches, we believe,

[a] Some recent AMD and Intel CPUs have primitive abilities to lock and unlock cache lines.

is the right approach to take.

Exposing caches as memories raises another question. Which memory should be exposed when a hardware feature has exclusive access to multiple caches and/or memories? This is an important concern given that a PPS can have only one memory. At the end, the person building the *PCubeS* description is responsible for making the correct choice in that regard. Nonetheless, our opinion on that can be expressed as two generic rules.

1. The largest cache/memory should be exposed as the memory of a space as it is generally more important to handle the largest capacity more efficiently.
2. Information exchange among sub-spaces within a space should take place through the closest/smallest memory as it generally represents the fastest path of communication.

If there is some definite advantage in exposing more than one memory then an alternative is to model the hardware feature as a linear hierarchy of PPSes with each PPS holding a different memory.

Contention Oblivious Modeling

It is noticeable that *PCubeS* does not model contention. We recognize that contention is an important issue in parallel hardware design. For example, tree saturation or hotspot contention problem[76] in interconnection networks is so significant that it has become a standard practice to have a separate network/channel in supercomputers for collective communication, a common source of hotspots. Memory bank conflict in multi-banked memories is another source of considerable performance loss in many programs.

One can argue that a CTA like description may be better suited in addressing hotspot contentions due to its emphasis on the specific structure of the interconnection network, or PMH is better suited for memory bank conflicts as it has the notion of memory blocks; but we argue otherwise because contentions are very application specific problems and just focusing on hardware features is not enough to deal with them. Therefore including

features like memory bank size, memory stride, or network topology in the type architecture description may not pay off. Rather there is a change that they will further confuse the programmer.

Does our approach introduce a potential gap between perceived and actual performance of a program written over the type architecture abstraction? The chance is there as it exists in all other type architectures, but we believe its probability can be reduced using standard programming techniques. For example, the use of primitives such as MPI reduction and scatter-gather are effective in avoiding most hotspot contentions. Similarly, a compiler can do a lot to reduce memory bank conflicts when it has significant control over memory management of a program. Given that we are arguing for type architecture based programming paradigms, we envision that language primitives and compilers will play bigger roles in dealing with problems like contentions.

Symmetrical Space Hierarchy

As we have mentioned earlier, an ideal *PCubeS* description requires that sub-spaces of a PPS are uniform. This rule implies that sibling PPSes at any level of the hierarchy have an equal number of sub-spaces. In other words, the hierarchy is symmetrical. We emphasize symmetry because, in our opinion, a symmetrical system is significantly easier to both program and design than an asymmetrical system.

Many large scale parallel architectures are inherently symmetrical, e.g., Mira Blue Gene Q supercomputer[1]. Furthermore, most accelerators and multicore CPUs that are been used in supercomputers have symmetrical organization of internal processing elements and memory modules. Therefore, *PCubeS*' restriction of sub-space uniformity readily applies.

Some hybrid systems, however, cannot be described as symmetrical hierarchies. For example, a compute node of Stampede supercomputer has two 8-core CPUs and a 61-core accelerator/co-processor. The accelerator

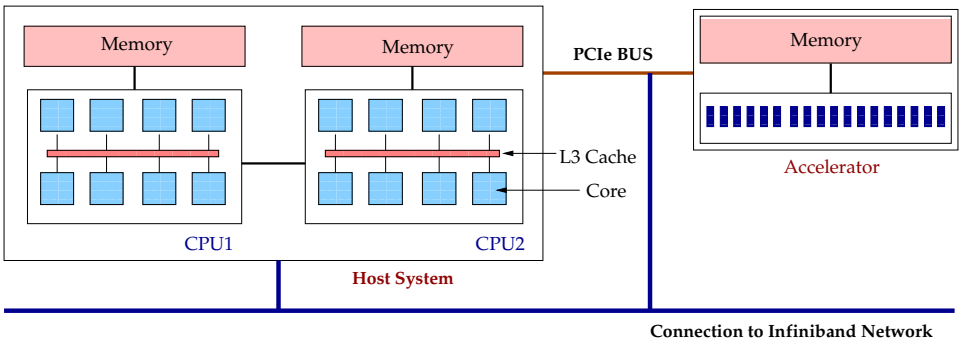


Figure 3.4: Block Diagram of a Stampede Node

cores can work just like the CPU cores or can be used to offload computations from the latter. Unfortunately, in both programming paradigms sub-space uniformity is directly inapplicable. The first case fails because accelerator cores are widely different in their capacities from the CPU cores. The second case fails because accelerator cores cannot be uniformly distributed under the two CPUs.

Note that Stampede supports CPU-only and accelerator-only programming models. Due to the symmetry of the interconnection network, the whole hardware can be described as a *PCubeS* instance for such usages. It is only when all resources of a node are used simultaneously a problem occurs. Therefore, *PCubeS* is not appropriate for all parallel architectures or their supported programming paradigms. Nevertheless, often times a conservative *PCubeS* estimate can be drawn by merging sibling spaces, dividing space capacities, and often discarding some spaces altogether.

To give an example of how this can be done, consider the CPU-to-accelerator offloading computation model for a Stampede node. A block diagram of that node is given in Figure 3.4.

In the offloading paradigm, individual cores can offload small pieces of computations to the accelerator from both CPUs, or the two CPUs can offload two relatively larger computations. This gives rise to two sub-cases,

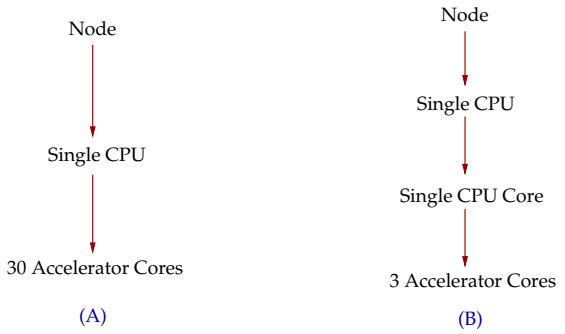


Figure 3.5: Two Alternative Symmetrical Breakdowns of a Stampede Node

and both can be conservatively described using *PCubeS* by ignoring some accelerator cores as shown in Figure 3.5(A) and (B). Note that in the second case, only 48 of the 61 accelerator cores can be used in a program.

Whether a conservative estimate is worth consideration or not depends on the hardware and its expected usage. In the case of a Stampede node, the second description seems to be quite wasteful; thereby should be avoided. Despite these kind of problems, we stick to the uniform sub-space requirement because of programming simplicity.

Furthermore, we would like to encourage parallel hardware manufacturers to adopt symmetrical designs through *PCubeS*. In our opinion, an asymmetric organization of components makes it difficult to reason about actual runtime behavior of a program. For example, all 61 cores of the Intel Xeon Phi accelerator/co-processor can be utilized by the 16 cores of the two Sandy Bridge CPUs in a Stampede node, but it is difficult to imagine any offloading scheduler for the co-processor other than a work-pool for such usage. A symmetrical decomposition with 48 or 64 co-processor cores provides more scheduling alternatives. Common wisdom suggests it should not be difficult to restrict the number of cores to 48 or increase it to 64 in the hardware.

3.4 Mapping Hardware Architectures to PCubeS

If we compare *PCubeS* with other previously proposed type architectures; we see many machines described by CTA, SIMD, PMH, and PGAS that can also be described using *PCubeS*. Most supercomputers of yesteryears such as purely distributed machines and purely shared memory machines are instances of *PCubeS*. Many clusters of workstations and even clusters of such clusters fall under *PCubeS* category. Most present day supercomputers that have only multicore CPUs as nodes such as Blue Gene P and Q systems are *PCubeS* instances. More complicated cases are the ones that have both multicore CPUs and accelerators within nodes. *PCubeS* can describe many of them too. To show how this can be done, we take the Titan Supercomputer[tit] of Oak Ridge Leadership Community Facility as a case study for *PCubeS*.

3.4.1 The Titan Supercomputer: a Case Study

Titan has 18,688 compute nodes. Each compute node contains a 16 core AMD Opteron CPU and an NVIDIA Tesla K20x GPU. Two nodes share a single Gemini interconnection router and the interconnection network connects these routers in a 3D torus topology. The underlying philosophy in Titan's design is to let the GPUs do the heavy lifting of a scientific application and use the CPUs mostly as coordinators of activities in the GPUs. Within that paradigm, there are two use cases to consider for a node.

1. A single activity offloaded to a GPU by its accompanying CPU consumes entire GPU capacity
2. Individual cores within a CPU offload relatively smaller, parallel computations to the GPU

Given that GPUs are not efficient in handling irregular parallelism, for some applications, doing the heavy lifting within the CPUs may be the right approach. This gives us the third programming paradigm for the node

3. CPU cores compute pieces of a parallel computation and coordinate their results

The network connecting the nodes is only a medium for communicating intermediate results of computations generated by individual nodes running under one of the above three paradigms. Note that *PCubeS* does not allow different programming paradigms in different nodes to coexist simultaneously nor does it allow a node to have multiple different sub-spaces. This is a limitation of our type architecture. If such behavior is expected, this limitation can be partially circumvented by dividing the capacity of the entire hardware into independent groups of nodes where each group runs a different programming paradigm.

What features of the GPU and CPU of a node the type architecture should expose and how they should manifest in the description depend on the choice of programming paradigm. Before we present the salient features of a node from the perspective of the supercomputer as a whole, however, let us derive the *PCubeS* descriptions of the multicore CPU and GPU when they are treated in isolation.

PCubeS Mapping of AMD Opteron CPU

If we refer back to the block diagram of a 16 core AMD Opteron CPU[61] given in Figure 1, we see that it has a non-uniform memory access (NUMA) model. Individual cores have their exclusive 16 KB L1 cache, a pair of cores share a 2 MB L2 cache, a group of 8 cores share a 8 MB L3 cache, then all cores have uniform access to a 32 GB main memory. CPU cores run at a clock frequency of 2.2 GHz and data paths are 64 bits wide.

Although there are 16 cores in total, two cores sharing an L2 cache share a single floating point execution unit. Since *PCubeS* measures compute capability in terms of floating point instruction density, there is no processing capacity in the lowest level of the hierarchy – there is just the memory capacity of an L1 cache per PPU. Rather, the processing capacity is assigned to the immediate higher level, Space-2, supporting 1 operation

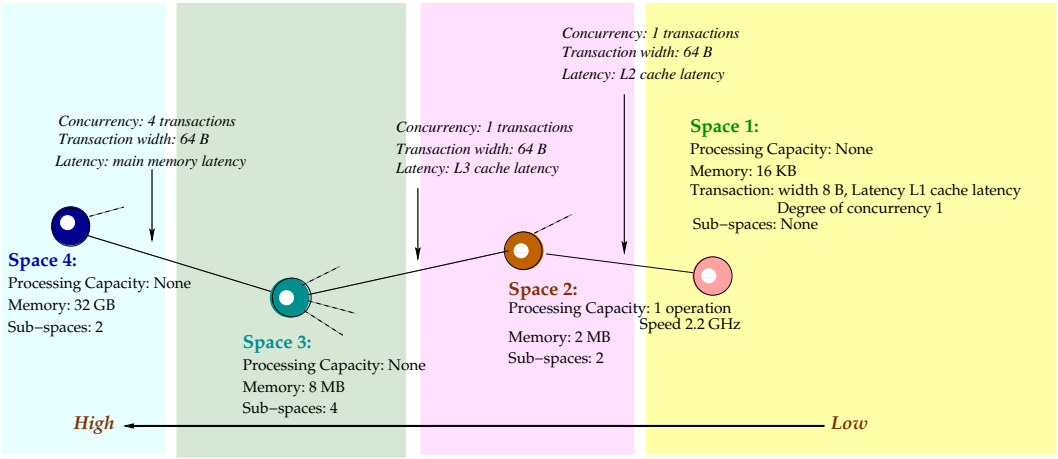


Figure 3.6: PCubeS Model of AMD Opteron Showing Only 1 PPU Per PPS

per clock cycle. A Space-2 PPU contains a pairs of Space-1 PPUs with a single 2 MB L2 cache shared among the pair. Each space in the next level combines four Space-2 PPUs and owns an 8 MB L3 segment. The final space represents the entire CPU comprising two Space-3 PPUs sharing the 32 GB main memory. A pictorial depiction of the *PCubeS* description is given below in Figure 3.6.

In the above model, the latency of information exchange (the cost of an interaction between two sibling PPUs of the same PPS) is not shown because we do not know the cost of doing atomics in the hardware.

Only one data movement operation at a time can take place between a parent and its sub-spaces up to level 3 as there is only one 64 bits wide data path. Meanwhile, 4 concurrent data movement operations can take place between Space-4 and Space-3 as the main memory is quad channeled. Finally, transaction width is 64 bytes along each arc as the cache lines are 64 bytes long.

The assignment of memory access and data movement capacities in Figure 6 requires particular attention. Note that the L1 cache attributes are described as transaction properties of the Space-1 memory; but, except for the size, the attributes of other caches and the main memory are described as the transaction properties for

data movements between adjacent spaces. This distinction is important as data need to flow into the L1 cache from upper level caches and the main memory for any computation to be done. In addition, this interpretation provides an indirect memory contention model for upper spaces through the transaction concurrency attribute.

PCubeS Mapping of NVIDIA Tesla K20X Accelerator

A Tesla K20X GPU[64], shown in Figure 3.2, has 2688 streaming cores running at 837 MHz clock speed and distributed within 14 streaming multiprocessors (SM). It has a 6GB on-board DDR2 RAM as the main memory unit. Shared memory per SM is 64 KB but only 48 KB of it is accessible to programs. The streaming cores run in lock step within each SM as a group of 32 threads known as warps and each SM can run up to 32 warps. Each shared memory load/store operation can process 16 32-bit words and a global memory load/store is twice that size. The *PCubeS* description of the hardware depicts a 3 levels space hierarchy. Figure 3.7 illustrates this description.

A warp represents a Space-1 unit. Parallel processing capacity of a Space-1 is 32 operations per cycle. The clock speed is, however, only 26.16 MHz instead of 837 MHz as warps are executed as pipeline instead of concurrently and there are 32 of them. This clock speed setting in the *PCubeS* description is in contrast to NVIDIA's advertised value as *PCubeS* is focused on actual performance characteristics of warps rather than mere numbers and structural details [a]. A warp has no memory. This indicates that the result of any computation needs to be moved to the closest space with a memory, which is in this case the parent Space-2.

[a] In the CUDA programming paradigm, it is possible to have just one warp running per SM. Theoretically the warp can run at 837 MHz clock speed then. In practice the effective performance of most CUDA kernels is dominated by the memory access latencies for such a configuration so it is discouraged.

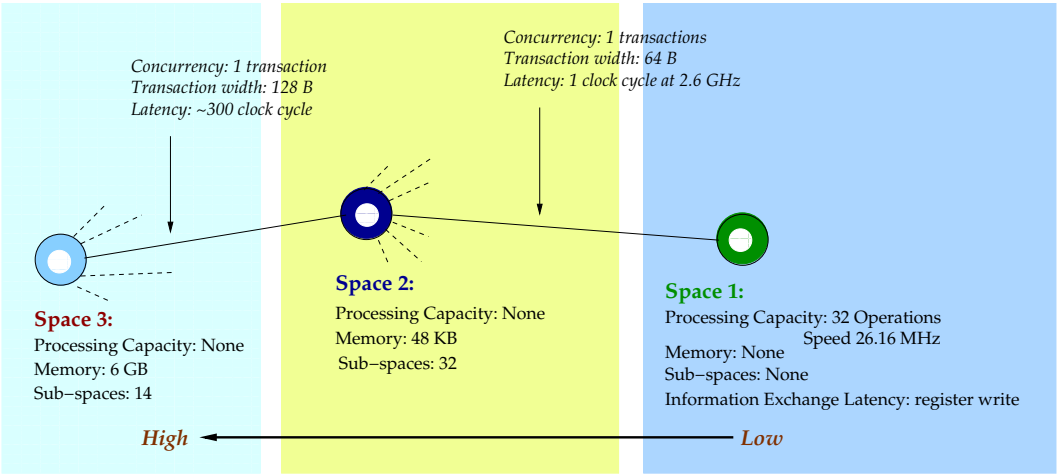


Figure 3.7: PCubeS Model of NVIDIA K20X Showing Only 1 PPU Per PPS

A Space-2 represents an SM and holds 32 Space-1 PPUs as sub-spaces. Only one Space-1 PPU can transfer data to its parent Space-2 at a time due to the pipelined nature of warp execution. So there is no concurrency, but the latency is minimal. A transaction carries 64 bytes of data and takes only one memory cycle to finish.

Finally, there is only one unit in Level-3 of the hierarchy. A single Space-3 PPU represents the entire GPU. It has 6 GB memory and holds 14 Space-2 PPUs as sub-spaces.

The *PCubeS* description of the accelerator resembles the hardware abstraction popularized by NVIDIA's CUDA programming model that also has 3 levels. Nonetheless, there are important differences to discover as we examine the parameters. We see that *PCubeS* makes the limitations of the hardware more explicit. For example, in CUDA, threads of a warp can diverge and execute different instructions. This gives a programmer more flexibility. In reality, however, divergent paths are executed sequentially. The *PCubeS* description makes divergence impossible by coupling the threads together. Such divergences in a program then need to be implemented as sequential streams of less degrees of parallelism. So the programmer is aware of his wastage of

processing capacity. For another example, data transfers between global and shared memory happen in chunks of 128 bytes, but CUDA threads can issue arbitrarily small and irregular requests to the global memory. In *PCubeS*, any data transfer between global and shared memory is mediated by the Space-2 unit under concern that makes such irregular requests impossible [a].

To summarize, *PCubeS* provides a more restrictive but accurate representation of the hardware than the CUDA machine model – with some exceptions [b]. As a type architecture description and an abstract machine model are not the same the comparison is not exact. Our point is that a programming paradigm whose machine model resembles the *PCubeS* description more than that of CUDA is more likely to guide the programmer in the right direction regarding efficient use of the GPU hardware.

PCubeS Mapping of Titan

Given that three different programming models are supported by the supercomputer, there should be three different *PCubeS* descriptions for it. In the easiest case, where only the multicore CPUs can be used, we get the description depicted in Figure 3.8.

On top of the node hierarchy presented in Figure 3.8, we have one additional level representing the pair of nodes sharing a Gemini interconnect router. There are 18,688/2 = 9,344 such Space-5 PPUs in the machine. Information exchange latency between a pair of Space-5 PPUs is the average latency of a packet transmission in the 3D torus interconnection network. Notice the un-rootedness of the hierarchy. This is expected as there is no global controller or common aggregation point for the nodes of the supercomputer.

[a] Again, a program can still have irregular data reads from the global memory but the programmer is cognizant about their inefficiency.

[b] Additional memories such as constant or texture memories that are visible in CUDA cannot be used in *PCubeS*.

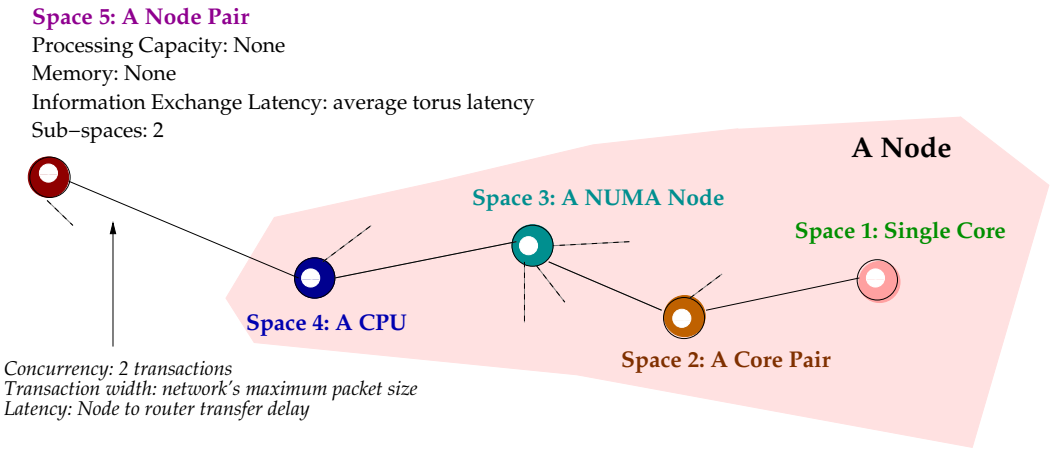


Figure 3.8: First PCubeS Description of the Titan Supercomputer

Let us now consider the case where the CPU within a node works as a coordinator of computations offloaded to its accompanying GPU and individual computations are large enough to consume the entire GPU capacity. In this scenario, only one core within the CPU does any useful work – the rest remain idle. So the spaces in upper three levels of the 4 levels hierarchy presented in Figure 3.8 become memory only spaces with branching factor 1 [a]. The GPU representation remains as it is. Figure 3.9 illustrates the *PCubeS* description for this scenario.

One important concern in describing this kind of hybrid environments is 'how to define the transaction attributes for data transfer between the CPU and accelerator.' Note that although Space-4 (a Core) is the parent of Space-3 (a GPU) in the *PCubeS* description, actual data transfer takes place from Space-7 to Space-3. When moving data out of the GPU to CPU core, data flows from Space-3 to Space-7 then from there to Space-4. As *PCubeS* is silent about the actual mechanism of data transfer this is not a problem, but one has to define the transaction attributes appropriately to derive a conservative estimate for the communication channel.

[a] As an alternative, the PPSes of the CPU can be merged together to form a single PPS if it is unimportant to expose the memory latencies.

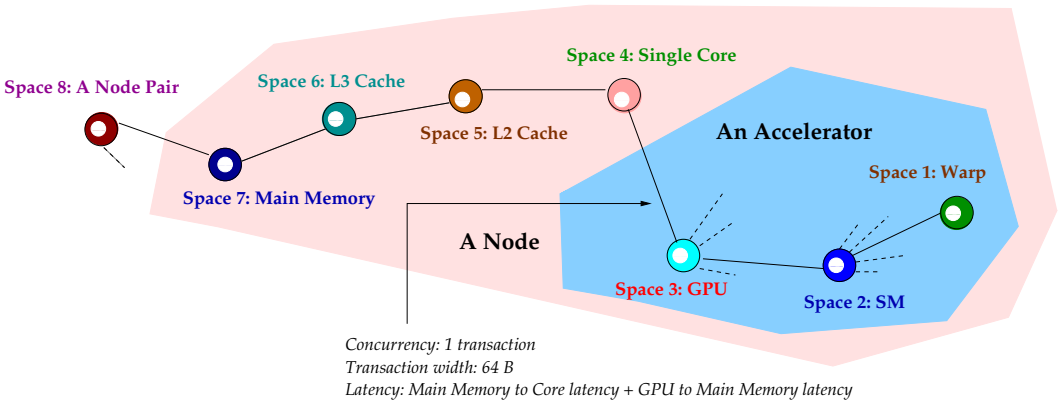


Figure 3.9: Second PCubeS Description of the Titan Supercomputer

The consistent rule for deriving a conservative estimate is: set the transaction width and concurrency to that of the least capacity link along the physical data transfer path, set the latency as the sum of the latencies of individual links, and if there is any shared link along the path then divide its capacity uniformly. For Titan, this gives us 1 concurrent transaction, 64 byte transaction width, and latency of main memory to L1 cache transfer augmented with main memory to GPU memory transfer latency for data movement between Space-4 and Space-3.

Finally, let us consider the case where individual cores of a CPU offload parallel pieces of computations to the GPU. In this scenario, we have an imbalance as there are 16 cores in the CPU but only 14 SMs in the GPU. Thus, we have to discard two cores and adjust the CPU hierarchy of Figure 3.8 to regain symmetry before assigning a segment of the GPU to individual cores.

One way to do this is to merge the two Space-3 PPUs and get rid of Space-2 PPUs altogether. Therefore, the cores share the L3 cache and own an exclusive L1 cache but cannot access the L2. Figure 3.10 shows the final *PCubeS* description of Titan after these modifications.

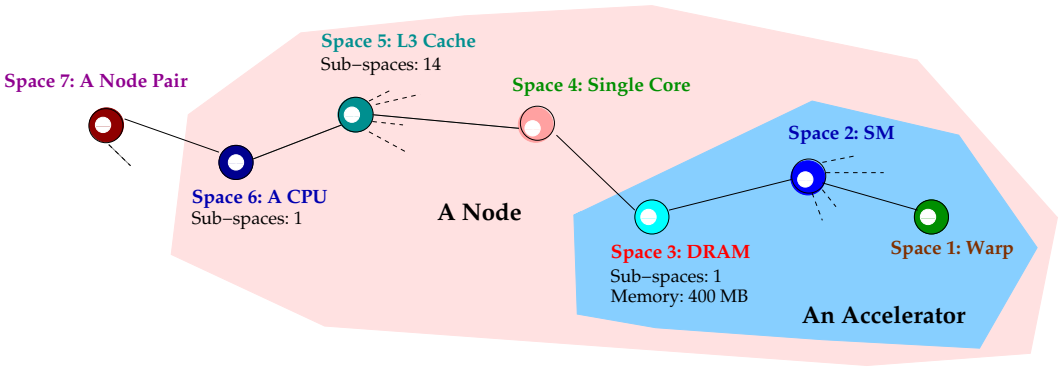


Figure 3.10: Third PCubeS Description of the Titan Supercomputer

Not shown in Figure 3.10, but transaction concurrencies and latencies of shared links need to be adjusted in the final description where appropriate.

3.4.2 The Mira Supercomputer: Second Case Study

The Mira Supercomputer[mir] in Argonne Leadership Computing Facility is a Blue Gene Q system. It has 48 compute racks hosting a total of 49,152 IBM PowerPC[92] A2 nodes. Nodes are connected in a 5D torus interconnection topology and each node has 18 cores running at 1.6 GHz clock speed. This is a symmetric system of homogeneous nodes. Nonetheless, there are important subtleties in the architecture that reveals a rich hierarchy in the *PCubeS* description, as illustrated in Figure 3.11.

Down at the bottom, each core can run up to 4 hyper-threads that have access to a SIMD instruction unit of 4 words wide. So a Space-1 of Mira is a hyper-thread with a processing capacity of 4 parallel operations and no memory. The clock speed for an operation is only 400 MHz, instead of 1.6 GHz, as there are 4 Space-1 PPUs.

In the next level, a single core with its 16 KB L1 cache represents a Space-2 PPU. As the data path is 64 bits and hyper-threads' data load/store happens in the L1 cache, transactions between Space-1 and Space-2 are 8 B

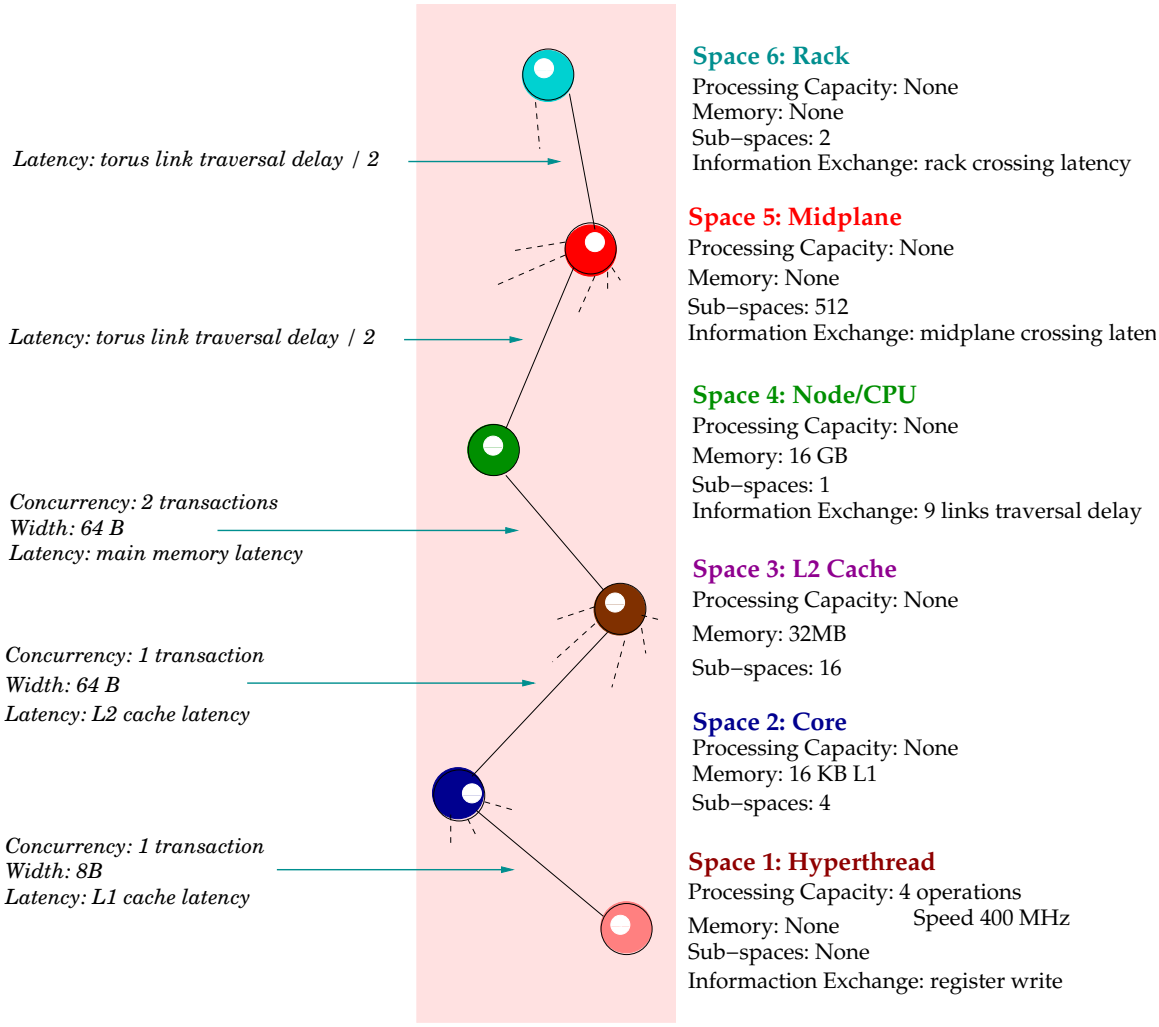


Figure 3.11: The PCubeS Description of the Mira Supercomputer

wide. Transaction concurrency is 1 as only one hyper-thread issues a load or store at a time.

The single Space-3 PPU in the next level represents the 32 MB L2 cache shared by all cores. The common cache line size is 64 B. So, that is the width for a transaction between a Space-3 and its Space-2 sub-spaces. A Space-3 has 16 sub-spaces as opposed to 18 because only 16 out of the 18 cores can be used in a program. Above the L2 cache a CPU or node constitutes a Space-4 with no processing and 16 GB memory capacities.

From Space-4, the 5D torus interconnection network becomes effective. Nonetheless, we get more levels in the hierarchy because of the non-uniform nature of node wiring. A single compute rack of Mira holds 1024 nodes. The rack is further divided into two mid-planes containing 512 nodes each. A mid-plane is the smallest full torus configuration where the average distance between a pair of nodes is 9 torus links. Average distance between a communicating pair lying in different mid-planes is thus 10 links.

Therefore, the *PCubeS* description has two additional levels on top of Space-4 for representing a mid-plane and a rack as Space-5 and Space-6 PPUs respectively. There are 48 racks in the system. Hence we have 48 Space-6 PPUs in total. Like Titan's, Mira's hierarchy is also un-rooted as the racks are equivalent.

Notice the Latencies of information exchange among sibling spaces and between a parent and child space starting from Space-4 and above. Space-4 siblings are the 512 nodes connected in a $4 \times 4 \times 4 \times 4 \times 2$ torus topology. So an information exchange between a pair of Space-4 PPUs has on average 9 torus links traversal delay. Mid-plane crossing adds one more link in the communication path of two nodes. Therefore, we divide the cost of traversing a single link into half and assign that as the cost of communication between a Space-4 and its parent Space-5 PPU. This augmented with any additional delay for mid-plane crossing – set as the information exchange latency in Space-5 – reflects the true average cost of communication between nodes of opposite mid-planes. The same logic applies for nodes' cross-rack communications in Space-6.

3.5 Summary

If we reflect on our discussion in previous sections, we realize that developing *PCubeS* descriptions of a machine is an involved procedure. The choice of spaces and their capabilities depends very much on the expected usage of the underlying features, which requires the developer to have a solid understanding of the expected usage to begin with.

PCubeS' focus on programmability over physical implementation of structural features also makes it difficult to judge a machine as non-*PCubeS* architecture. This is different from Snyder's original idea of type architecture as we see different machines are easily classified as CTA or non-CTA. In *PCubeS*, the efficiency of the description dictates whether the underlying hardware is a *PCubeS* instance or not.

For example, Sun Microsystems' UltraSPARK T1[58] microprocessors have 8 cores. Each core can handle 4 concurrent threads and has large caches. The system, however, has a single floating point unit to be shared among all cores. With a scheduler that gives round robin access to the floating point unit to all threads, one can give a *PCubeS* description of the system. The sheer inefficiency of the description – not the infeasibility of constructing one – shows that the architecture is not suitable for a *PCubeS* description.

4

IT Parallel Programming Language

This chapter provides an overview of the language component of my proposed dual-part parallel programming paradigm. After a brief introduction to *IT* programming, the chapter explores the language characteristics and its abstract machine model, then provides an overview of its current features. Afterwards a linear algebra *IT* program is explained in detail to serve as a practical example. A discussion on interoperability follows before the chapter concludes.

An in-depth investigation of *IT* language is avoided here lest that diverts our focus from the core research objectives. For interested readers, a detailed description of the language syntax and its features is available in our technical report on *IT* [96]. In addition, a concise presentation of the entire programming paradigm can be found in our earlier publication [95].

4.1 Introduction

The central concept behind the *PCubeS* + *IT* programming paradigm is to make reasoning about hardware features an integral part of all aspects of a parallel program's development. Despite the constant presence of hardware concerns; we want the paradigm to be portable, programming on it to be productive, and performance of its executable to be adequate.

On its own, the *PCubeS* type architecture only addresses the portability aspect of the problem. It provides the mechanism to describe different hardware in a uniform way without hiding their salient features. Effective utilization of those features in a program without overwhelming programmer effort, however, largely depends on the ability of the underlying language being used to cleanly expose hardware features through its abstract machine model and on its ability to encourage programming styles that are efficient.

The *IT* programming language is designed with those concerns in mind. As *PCubeS* describes a parallel architecture as a hierarchy of physical processing spaces, in an *IT* program, computations take place in a corresponding hierarchy of logical processing spaces each of which may impose a different partitioning for a data structure. To generate an executable the programmer needs to explicitly map these logical spaces to physical spaces of *PCubeS*. Efficient partitioning and mapping is critical for the good performance of a program. The im-

plementation of computation and communication that is determined by the compiler depends on the features of the physical spaces computations have been mapped to.

On the surface, *IT* may appear as another high-level programming language just like X10 or Chapel – and several *IT* constructs have semantic analogs in those languages – but a key difference lies in the treatment of these features. An *IT* program looks like a parallel pseudo-code with space markers controlling flow of data and computations and their inter-dependency, data partition specification remains separate from the algorithmic logic, and mapping is not even a part of the source program. This gives flexibility for tuning individual pieces of a program based on the *PCubeS* description of the target hardware. Separation of concerns makes it easy to write the program and enhances its readability. Finally, emphasis has been given to avoid any compiler introduced non-deterministic overhead in the program.

To summarize, *IT* forces the programmer to be cognizant of the features of target architectures when developing an algorithm as in low-level programming, but allows him/her to express the program over a broadly applicable and portable hardware abstraction as in the high-level programming, and encourages him/her to learn how to exploit hardware features by establishing a clear relationship between the source code and its runtime performance.

4.2 IT Programming Model

The paradigm asks a programmer to first break a program in a top-down manner into cooperating *tasks*. The execution order of tasks is determined by their data dependencies and availability of physical resources in the target platform. The execution platform and the data structures located in it constitute a program's environment.

The execution of a task changes the environment by updating existing data structures and adding new structures that may be needed at later time for subsequent tasks. By-products of tasks that are not finally written to external files are automatically garbage collected. An example of *IT* task breakdown is a conjugate gradient program where matrix-vector multiplication, vector dot-products and so on are the individual tasks. The granularity of the individual tasks is subjective, but the guideline is to restrict tasks to independent program units that use a particular arrangement of data.

Beneath the task level parallelism resides the data parallelism of *compute stages* within a task (computation stages are similar to procedures in conventional programming languages). A task specifies the logic of the parallel algorithm as a sequence of compute stages executing in one or more **`Logical Processing Spaces (LPS)`** within a *Computation Section*. The relationship between these logical spaces and how data structures are partitioned within each space are specified in an accompanying *Partition Section*. The partition specification dictates the number of groups of data structure pieces or partitions – we call them **`Logical Processing Units (LPU)`** – in each logical space. Each LPU runs independently in data parallel fashion. Inter-LPU interaction is governed by LPUs' data dependencies and the compiler takes care of the underlying synchronization and data transfer required for that. The programmer controls the degree of parallelism by specifying the mapping of logical spaces to physical spaces (PPS) and runtime arguments for the parameters of the Partition Section.

Finally, the calculation inside a compute stage is written using a declarative syntax supporting parallel loops, reductions, and conventional arithmetic and logical operators. When writing a compute stage, *the programmer assumes the locality of any data structure used within* and is encouraged to focus on identifying instruction level parallelisms through the declarative parallel constructs. Depending on the support available in the target

platform these parallel constructs may get translated into vectorized or SIMD instructions or just execute sequentially. Again, the nature of the translation, consequently its efficiency, depends on the mapping of LPSes to PPSes which is under programmer's control. Listing 4.1 presents an example *IT* single-task program as an illustration of the aforementioned concepts.

```
1 Program (args) {
2
3         // create an environment object for the matrix–matrix multiplication task
4         mmEnv = new TaskEnvironment(name: ``Block Matrix–Matrix Multiply")
5
6         // specify how external input files are associated with the environmental objects
7         bind_input(mmEnv, ``a", args.input_file_1)
8         bind_input(mmEnv, ``b", args.input_file_2)
9
10        // execute the task
11        execute(task: ``Block Matrix–Matrix Multiply"; environment: mmEnv; partition: args.k, args.l, args.q)
12
13        // specify where the output should be written to
14        bind_output(mmEnv, ``c", args.output_file)
15 }
16
17 Task ``Block Matrix–Matrix Multiply":
18        Define:
19                a, b, c: 2d Array of Real single–precision
20        Environment:
21                a, b: link
22                c: create
23        Initialize:
24                c.dimension1 = a.dimension1
25                c.dimension2 = b.dimension2
26        Stages:
27                // a single computation stage embodying the logic of the matrix–matrix multiplication
28                multiplyMatrices(x, y, z) {
29                        do { x[i][j] = x[i][j] + y[i][k] * z[k][j]
30                        } for i, j in x; k in y
31                }
32        Computation:
33                Space A {
34                        // the stage has to be repeated for each sub–partition of Space A to have a block implementation
35                        // as opposed to a traditional one
36                        Repeat foreach sub–partition {
37                                multiplyMatrices(c, a, b)
38                        }
39                }
40        Partition (k, l, q):
41                // 2D partitioning of space giving a block of c in each partition along with a chunk of rows of a
42                // and a chunk of columns of b
43                Space A <2d> {
44                        c: block_size(k, l)
45                        a: block_size(k), replicated
46                        b: replicated, block_size(l)
```

```
47                      // block–by–block flow of data inside a PPU is governed by the sub–partition specification
48                      Sub–partition <1d> <unordered> {
49                              a<dim2>, b<dim1>: block_size(q)
50                      }
51              }
```

Listing 4.1: An IT Block Matrix-Matrix Multiplication Program

Although the details will only become clear once we discuss the component features, the systematic structure of the *IT* program should be recognized even from a cursory inspection. The basic idea is a single space task *Block Matrix-Matrix Multiply*. The heart of the computation is the *multiplyMatrices* stage defined in the *Stages Section*. The code does exactly what you expect: a set of vector dot products. *multiplyMatrices* is called repeatedly and in parallel within Space-A in the Computation Section: once for each sub-partition of Space-A data structures. The sub-partitions are defined in the Partition Section.

More formerly, the computation flow (Line 32 to 39) specifies that within the confinement of Space-A partition the compute stage *multiplyMatrices* should execute in each of its sub-partitions one after another. The Partition Section (Line 40 to 51) dictates that each independent Space-A partition will have a $k \times l$ blocks of c, k rows of a, and l columns of b. Then within a partition, q columns of a and q rows of b will be operated at once for that k rows and l columns respectively.

For example, if the sole space of Listing 4.1 is mapped to the symmetric multiprocessor (SMs) of a GPGPU then the degree of parallelism will be equal to the number of SMs in the hardware and the Space-A LPUs will be multiplexed on them to be executed one after another. The threads of an SM will execute the compute stage in SIMD fashion over data loaded on SM's shared memory. The partition arguments k, l, and q should be chosen so that data for a sub-partition can be held within the limited shared memory.

On the other hand, if Space-A is mapped to the cores of a multicore CPU then the degree of parallelism will

be equal to the number of cores and each core will execute the instructions of *multiplyMatrices* stage sequentially. Then a different setting for the partition arguments, based on cache capacities, may be appropriate.

In short, *IT* asks a programmer to design programs in a declarative manner with flexible breakdowns of computations and associated data structures that can be efficiently tuned to the features of a specific execution platform if he/she knows how to exploit those features.

4.3 IT Design Rationale

Several elemental design choices make programming in *IT* quite different from programming in other contemporary parallel language offerings. These choices are made not to give *IT* a different look-and-feel for the sake of it; rather they contribute to the paradigm's overall objective of enabling portable and efficient hardware-cognizant parallel programming. This section elaborates on these design choices.

4.3.1 Separation of Concerns

An *IT* program is viewed as a set of tasks each of which has parts being executed in different spaces. An *IT* task is further partitioned into separate sections for, among other things, specifying the computation and the data partition. Therefore, the programmer addresses the issues of parallelism in the algorithm and cross component communication separately from partitioning and mapping.

This separation of concerns is there to simplify reasoning about the type architecture model of a target execution platform when programming. In addition, it provides a degree of clarity in a task and allows adapting the task for a new execution platform with minimal changes.

The systematic nature of *IT* programming not only affects how but also when different aspects of parallel programming become relevant. When breaking down a program into cooperating tasks, the programmer is concerned about high-level task parallelism among program units; when constructing a task, he/she is concerned about data parallelism in the algorithm; inside the compute stages of a task, he/she is concerned about instruction-level parallelism and vectorization. Operations such as creating a new task inside another task or invoking a compute stage from another stage are not permitted to concentrate programming effort on a particular aspect at a time.

4.3.2 Lean Language Core

The *IT* programming language is designed to be minimalist and has a small set of programming primitives. As the language is intended for high performance parallel computing, features that are not essential in that regard are avoided. Although the language is still in its preliminary stage and will absorb new features with time, a central theme will remain to provide a basic set of features over which other useful features can be built – instead of having a bloated core.

4.3.3 Declarative Syntax

IT uses declarative syntax for both the compute-stages and the flow definition in the Computation Section. The syntax is declarative as it instructs what computation to be done but not how to do it. For example, the *do . . . for* loop of the *multiplyMatrices* stage in Listing 4.1 iterates over three indices i, j, k. This should result in a triple-nested loop in conventional imperative programming languages. In *IT*, however, the ordering of three index traversals is left for the compiler to decide. For the flow definition, this philosophy extends fur-

ther at a coarser level where requirements for synchronization and communication in-between compute-stage transitions are determined based on the context in the flow.

This choice of declarative syntax is of paramount importance in enabling full exploitation of features of a target execution platform. Regarding the flexibility in the implementation choice for communication and synchronization, the need for a declarative syntax may be quite apparent; but the need is there, and no less intense, for translation of the compute-stages also.

Most modern architectures have vector or SIMD instruction units, memory banks, etc. whose effective usage is essential for good performance in the respective hardware. Indeed *PCubeS* exposes many of these features in terms of parallel transaction and computation capacities of the PPSes. Ordering instructions and memory operations to exploit these capacities, in particular across architectures, is difficult for an average programmer to master nevertheless. Furthermore, the architectural differences in different target platforms may render a highly efficient code in one platform to perform poorly in another when order-sensitive instructions and memory operations are used.

Thus *IT* adopts a declarative syntax to let the compiler for a specific platform to deal with the specifics of its target. The programmer only enables utmost compiler level optimizations by choosing size of data partitions properly and specifying the calculation over them in a declarative manner [a].

In addition, a declarative syntax simplifies static analysis of the source code, which an *IT* compiler heavily depends on for proper code generation. Moreover, in some situations, a declarative syntax makes it easy to express a parallel algorithm that would be difficult to express otherwise. For example, Listing 4.2 shows a com-

[a] Since the programmer explicitly deals with all non-deterministic aspects of a program, compiler optimizations should bring only deterministic improvements that the programmer can reason with.

pute stage from a finite difference stencil computation that uses Jacobi iterations to calculate the next state of a heated plate based on the current state.

```
1       refineEstimates(plate) {
2               localRows = plate.dimension1.range
3               localCols = plate.dimension2.range
4               do { plate[i][j] at (current)                                   \
5                       = 1/4 * (plate[i-1][j]                                  \
6                       + plate[i+1][j]                                         \
7                       + plate[i][j-1]                                         \
8                       + plate[i][j+1]) at (current - 1)
9               } for i, j in plate                                             \
10                and (i > localRows.min and i < localRows.max)                 \
11                and (j > localCols.min and j < localCols.max)
12      }
```

Listing 4.2: A Compute Stage for an Iterative Refinement Stencil Task (note the use of *at (current)* and *at (current-1)*)

Without the declarative syntax for calculating the current plate cell values from previous values of neighboring cells, this parallel code would be difficult to express cleanly. The problem with an imperative syntax is that sometimes the difficulty in expression leads to the choice of a sequential algorithm as opposed to a parallel one.

4.3.4 Programmer-Compiler Responsibility Breakdown

IT makes the programmer responsible for defining the nature and frequency of communication in a task through his/her algorithm and partition specification, but he/she is oblivious of the actual communication mechanism – though not the costs! The compiler decides about the communication mechanism based on what is available in the target platform and the runtime tries its best to optimize communication using computation-communication overlapping, message combining, etc.

Data synchronization takes place automatically at LPS boundaries and as needed basis. The compiler generates code that checks for possible modification of any data structure shared among multiple LPSes (or LPUs). As the computation for a space ends within an LPU, the runtime synchronizes data based on the result of

the check. If the updates are needed immediately, execution halts until synchronization happens. Otherwise, computation of subsequent LPUs can proceed while asynchronous data updates are going on for future use.

Furthermore, note that because of the declarative syntax, the runtime, not the programmer, is responsible for loading data in the appropriate memory and reading/writing that data efficiently from there. This raises the important question, 'where does *IT* draw the line between the programmer and the compiler's responsibilities in generating an efficient program?' The answer is anything that the compiler or runtime can always decide on optimally is left for automatic analysis and everything else is a programmer's responsibility.

This policy is adopted so that an *IT* programmer can predict, debug, and analyze the suitability of a program and its runtime performance on a specific platform just by comparing the source code with the *PCubeS* description of the hardware [a].

4.4 Abstract Machine Model of IT

As *IT* presents a machine model of a hierarchy of LPSes where tasks can execute, it is critical to understand the notion of LPSes and the nature of computations within them to make sense of subsequent discussions.

4.4.1 Logical Processing Spaces

A Logical Processing Space (LPS), as its name suggests, is an entity where computations can be done over data structures. It is not directly associated with the computations or the data structures; rather both are assigned to it. The best way to understand an LPS is how variables and instructions are treated in a traditional Von

[a] Note that this vision can be realized only if the *IT* runtime engine has adequate control over the features of the hardware exposed through its *PCubeS* description and utilizes that control properly.

Figure 4.1: A Von Neumann Space

Neumann programming language.

In a traditional Von Neumann machine, there is a memory and instructions. Instructions are fetched from the memory and executed. When an instruction executes, it may load and store variables in the memory. The entire memory is visible to each instruction - whether a variable is in the scope of the current programming context or not, or whether it is a local variable or a global. Figure 4.1 illustrates this concept. The address space of the Von Neumann computer is shown as a slate. Within the memory three variables *a*, *b*, and *c* have been defined. Instructions operating in the Von Neumann machine can load and store variables in the address space. In essence all instructions can access all memory, this makes the Von Neumann model a single-space model.

IT supports multiple spaces in a program where both variables and computations can be assigned (notice the distinction between definition and assignment). Figure 4.2 depicts a scenario with two spaces. Here the variables *average*, *median*, and *earning_list* are defined externally; the first two reside in Space-A and the second two in Space-B. Thus the *earning_list* is shared between the two spaces. The update done on the *earning_list* in Space-A is visible in Space-B, but the variable *average* is not accessible from the latter by any instruction.

There are further additions to the notion of a space in *IT*. First, an *IT* space or LPS has a dimensionality attribute. That is, an LPS is not a characterless vacuum; rather it is more like a geometric coordinate space. Second, an LPS can be partitioned into independent units called Logical Processing Units (LPU). The same

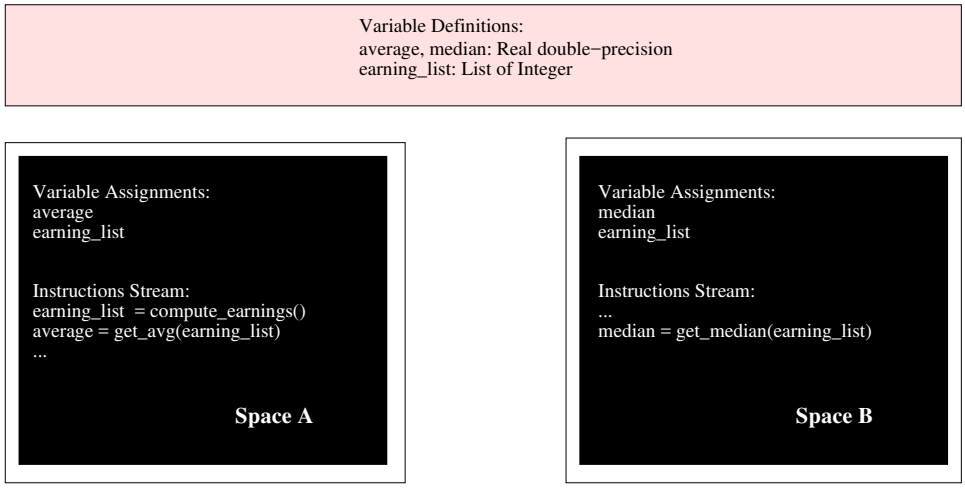


Figure 4.2: Demonstration of a Dual Space Model

instructions stream executes in all LPUs but on different parts of data structures. A group of LPUs may, however, share a particular part of a data structure. Then only one can update that part at a time and afterwards the remaining LPUs receive the update.

Execution in LPSes

In the absence of data dependencies on shared/overlapped parts instruction streams in individual LPUs execute independently. Figure 4.3 depicts the breakup of a single LPS for the matrix-matrix multiplication task of Listing 4.1.

Further, *IT* allows the data to be used inside an LPU to be loaded incrementally as opposed to all at once. This facility comes in handy when the computation for the LPU needs to be done in a memory constrained physical processor (PPU) such as an NVIDIA GPU symmetric multiprocessor. Incremental loading and unloading allows virtually limitless amount of data to be processed in a small PPU.

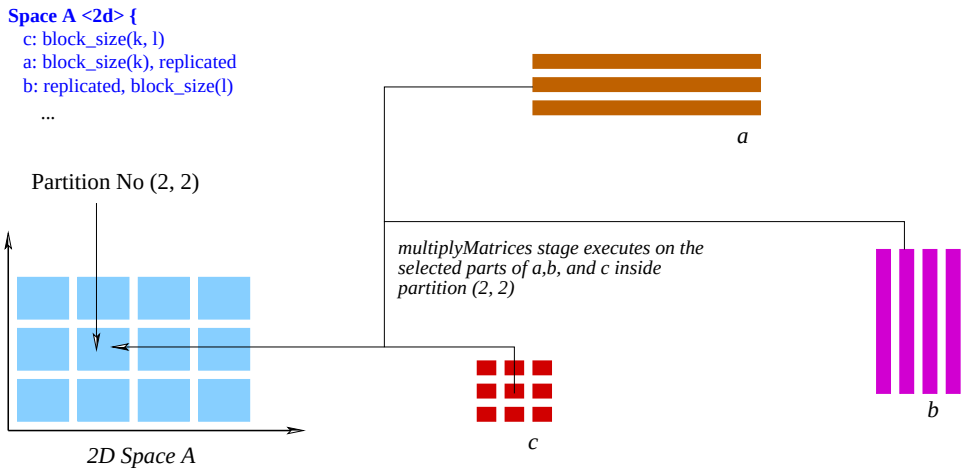


Figure 4.3: An Illustration of LPS Partitioning for a Small Matrix-Matrix Multiply Problem

For example, this facility can be used to convert a traditional matrix-matrix multiplication program into a block matrix-matrix multiplication program as done in Listing 4.1. Programmatic-ally this is achieved using the Sub-partition construct (Line 48 to 50 of Listing 4.1). Figure 4.4 below illustrates the effect of sub-partitioning for the discussed problem.

Finally, the power of LPS goes further in *IT* with the support of LPS hierarchies. In *IT* each LPU can be treated as an LPS and can be further divided into lower level LPUs. In fact, there may be multiple lower level LPSes for an upper level LPU that may differ in their dimensionality and partition counts. Figure 4.5 illustrates the idea of the LPS hierarchy for a Monte Carlo area estimation program.

When data sharing among multiple LPSes is involved, a relevant concern is the consistency model. *IT* adopts the strong consistency model. Regardless of how many LPSes share a single data structure and what the relationships among those LPSes are, *IT* ensures that the update in any data part is exclusive and all instruction streams operate over the most up-to-date data. It is the responsibility of the *IT* compiler to establish strong data consistency through whatever mechanism appropriate in the target execution platform. The programmer

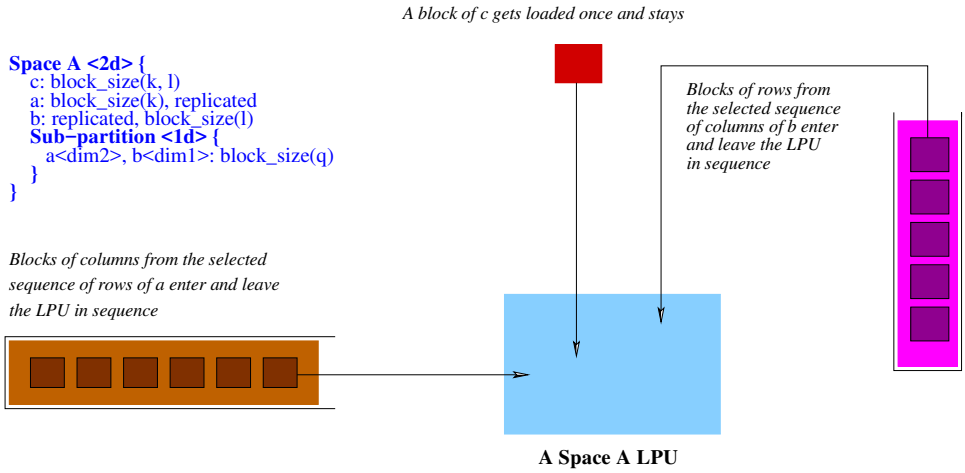


Figure 4.4: Effect of Sub-partitioning in the Matrix-Matrix Multiply Problem

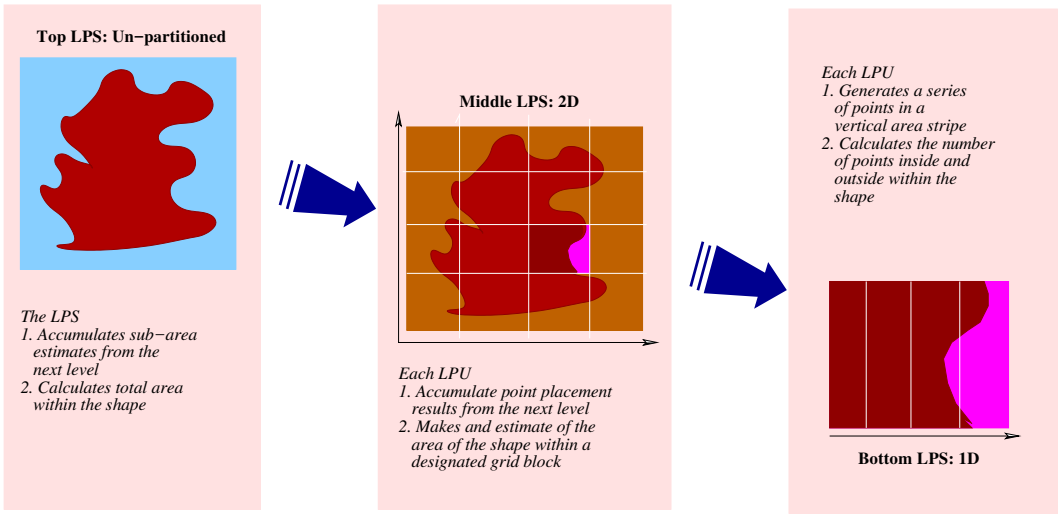


Figure 4.5: Three Space Breakdown of a Monte Carlo Area Estimation Problem

is only responsible for applying appropriate conditions in the code to ensure that only one LPU updates the shared data part at a time.

So at the end, *IT* abstract machine model is a partial hierarchy of LPSes, where each LPS consists of one or more LPUs that are capable of holding data and executing instruction streams over their data content, and each LPU has access to the most up-to-date version of the data before it starts any computation.

4.5 IT Features Overview

Data Structures

The basic primitive types in *IT* are signed and unsigned integers, single and double precision real numbers, characters, Boolean values, and fixed-length strings. The two collection types are list and array. There is no notion of pointers in *IT*. Hence multidimensional arrays are supported but not arrays of arrays.

User defined types are supported; which resemble C structures. A user defined data type can have other custom data types, primitive types, and fixed length arrays as properties, but not lists or dynamic arrays.

Both arrays and lists can hold objects of user defined types. A list in *IT* is a dimension-less sequence of elements of the same type. A list can hold arrays as elements and vice versa, but the elements of the nested collections cannot themselves be collections.

Parallel Instructions

IT supports instruction-level parallelism in terms of parallel loops and parallel reduction operations inside compute stages of a task. For example, the *do ...for* loop of Listing 4.2 code snippet is a parallel loop that can

iterate over each i, j index combination concurrently. A parallel reduction is a reduction statement placed inside a parallel loop. A reduction statement takes the form, $lvalue = reduce(OpCode, vector\_expr)$. This causes the vector expression being evaluated for each vector position then the result being reduced and assigned to the left-hand-side.

If sequential execution is intended then the programmer just changes the looping instruction from *do ...for* to *do in sequence ...for*. Sequential while loops can be used also.

Functions

IT functions are sequential and similar to C functions, but with two notable exceptions. First, *IT* functions are type polymorphic: the argument and return types are deduced from the calling context. Second, partitioned arrays cannot be used as function arguments. The second restriction is there to encourage the programmer to consider the possibility of parallel operations on arrays before settling for the sequential alternative. Although an entire array cannot be passed as an argument to a function, its individual elements can be passed as references. An '&' precedes the names of those function arguments to be passed as references as opposed to values.

Data Partitioning

Currently, *IT* supports the standard block-size, block-count, stride, and block-stride partition functions to divide arrays within an LPS to form the LPUs. The array partitions can be overlapped along any dimension. A special construct, *padding*, is provided to specify the overlapping regions among neighboring partitions. More commonly, an entire array or array dimension can be shared among LPUs (as done in the Partition Section of Listing 4.1 using the *replicated* keyword).

The long term goal is to allow programmer-defined partition functions. The mechanism implemented in the compiler to determine synchronization and communication requirements already accommodate arbitrary functions, but the interface to enable such feature is currently unavailable in the language.

4.6 An IT Programming Example: LU Factorization

This section examines a comprehensive *IT* sample program. An involved and long example is chosen to expose and discuss various aspects of *IT* programming. Except for understanding the code samples given in Appendix A, the content of this section is not needed for any subsequent discussion. Therefore, the reader can entirely skip this section or come back to it later.

4.6.1 Problem Description

A common approach to solve a system of linear equations of matrix form $Ax_i = b_i$ is to first divide the argument matrix A into two upper and lower triangular matrices L and U. Then solve two resulting triangular systems using simple Gaussian Elimination. The triangulation process is known as LU Factorization (or Decomposition). For numerical stability the decomposition is done using either rows or columns as pivots that results in another pivot matrix being generated as the third outcome.

The traditional LU Factorization algorithm suffers from bad memory/cache reuse characteristics. It proceeds diagonally by adding one more column in the lower and one more row in the upper triangular matrices respectively. At each step of the algorithm, however, the entire remaining portion of the upper triangular matrix needs to be updated based on the most recently calculated row and column. Updating the remaining part

of the upper triangular matrix is the most time consuming part of the algorithm and dominates its asymptotic running time. For larger matrices this step causes a lot of cache thrashing also.

The logic behind a better algorithm is that most rows of the remaining part of the upper triangular matrix are not needed immediately. Therefore, it is wasteful to update them in each step. Rather, only a portion of those rows should be updated for a certain number of iterations. Then the remaining stale part can be updated using a SAXPY ($c = \alpha c + \beta a \times b$) operation from the up-to-date rows. SAXPY has a better memory reuse potential. The asymptotic runtime of both algorithms are the same but the gain through memory reuse is massive in many architectures. This algorithm is called the Parallel Blocked *kji*-SAXPY LU Factorization algorithm[90].

4.6.2 Task Invocation and Coordination

In *IT* the modified algorithm can be easily represented using two tasks, one for the factorization part and another for the SAXPY part. A third task is used to initialize the upper and lower triangular matrices from the input argument matrix. A single *program coordinator function* – the entry point to any *IT* program – specifies how the tasks of the program are related and when they can be scheduled for execution. Listing 4.3 shows the coordinator function for the LU factorization program.

The sole argument of the program coordinator holds all command line arguments passed at the beginning. The command line arguments are passed as key, value pairs in the form 'key=value.'

```
1 Program (args) {
2         blockSize = args.block_size
3         initEnv = new TaskEnvironment(name: ``Initiate LU")
4         bind_input(initEnv, ``a", args.argument_matrix_file)
5
6         // execute a parallel initialization task that copies element from a to u and l
7         execute(task: ``Initiate LU"; environment: initEnv)
8
9         luEnv = new TaskEnvironment(name: ``LU Factorization")
10
11        luEnv.u = initEnv.u
12        luEnv.l = initEnv.l
13        rows = initEnv.a.dimension1.range
14        max1 = rows.max
15        max2 = initEnv.a.dimension2.range.max
16
17        do in sequence {
18                lastRow = k + blockSize - 1
19                if (lastRow > max1) { lastRow = max1 }
20                range = new Range(min: k, max: lastRow)
21
22                // execute a modified version of LU factorization that updates l properly in each step but
23                // updates only a section of rows and columns of u
24                execute(task: ``LU Factorization"; environment: luEnv; initialize: range)
25
26                if (lastRow < max1) {
27                        mMultEnv = new TaskEnvironment(name: ``SAXPY")
28                        mMultEnv.a = luEnv.u[(lastRow + 1)...max1][k...lastRow]
29                        mMultEnv.b = luEnv.l[k...lastRow][(lastRow + 1)...max2]
30                        mMultEnv.c = luEnv.u[(lastRow + 1)...max1][(lastRow + 1)...max2]
31
32                        // execute a saxpy operation of the form c = c - a * b to update the remaining
33                        // section of u that was left unmodified during LU factorization
34                        execute(task: ``SAXPY"; environment: mMultEnv; partition: args.k, args.l, args.q)
35                }
36        } for k in rows step blockSize
37
38        bind_output(luEnv, ``u", args.upper_matrix_file)
39        bind_output(luEnv, ``l", args.lower_matrix_file)
40        bind_output(luEnv, ``p", args.pivot_matrix_file)
41 }
```

Listing 4.3: The Program Coordinator Function for Block LU Factorization

First it executes the *Initiate LU* task (Line 7) to initiate upper and lower triangular matrices from elements of the argument matrix that is read from an external file (Line 4). Then the outputs of the first task are assigned to the environment of the second, *LU Factorization*, task (Line 11 and 12) that executes repeatedly inside a loop (Line 24). After the completion of one round of the *LU Factorization* task, a *SAXPY* task (Line 34) updates the stale portion of the upper triangular matrix using another portion of the same matrix and a portion of the

lower triangular matrix.

A task invocation in the program coordinator takes the following form:

```
1 execute(task: task–name;
2         environment: environment–reference;
3         initialize: comma separated initialization–parameters;
4         partition: comma separated integer partition parameters)
```

Here the Initialize and Partition parameters are optional but must be supplied for tasks needing them. Note that the execute statement in a program is expected to be non-blocking. Further, the invocation of the execute command does not necessarily launch the task immediately – rather it schedules the task for execution. The task can start only after all previous tasks, if any, manipulating its environmental data structures finish executing.

Since there are environmental dependencies between the *LU Factorization* and *SAXPY* tasks, the latter cannot start before the former finish. For the same reason, the next iteration of the loop cannot re-launch the former until the latter finishes for the previous iteration.

Relating tasks through their environments provide two primary benefits. First, tasks can share as many data structures as deemed appropriate for the logic of the program. Second – and more importantly – this allows the compiler to generate code to directly use the data distribution of preceding tasks in subsequent tasks whenever applicable. Even when data reordering is beneficial that can be done within individual tasks in parallel. There is no need for accumulating intermediate results in a central place.

Like computation, the principal mechanism for file I/O in *IT* is parallel. The coordinator program only specifies the external input/output files – the action is called *binding* – and the actual read/write happens in parallel inside the individual tasks. For example, the *bind_input* command at Line 4 only attaches appropriate instruction to the Initiate LU task that causes the argument matrix to be read when that task starts execution.

Similarly, the three *bind_output* commands of Line 38 to 40 results in the outputs of the *LU Factorization* task to be written to files in parallel at the end of the program. The *bind_input* and *bind_output* commands are sensitive to the type of the data structure and read/write data accordingly. The file format must be appropriate for *bind_input's* correct behavior though.

4.6.3 Task Definition

Listing 4.4 depicts the *Initiate LU* task. Every *IT* task follows the same structure of this simple task.

```
1 Task ''Initiate LU":
2       Define:
3             a, u, l: 2d Array of Real double-precision
4       Environment:
5             a: link
6             u, l: create
7       Initialize:
8             u.dimension1 = l.dimension1 = a.dimension2
9             u.dimension2 = l.dimension2 = a.dimension1
10      Stages:
11            prepare(a, u, l) {
12                  do { u[j][i] = a[i][j] } for i, j in a
13                  do { l[i][i] = 1 } for i in l
14            }
15      Computation:
16            Space A { prepare(a, u, l) }
17      Partition:
18            Space A <id> {
19                  a<dim2>, u<dim1>, l<dim1>: stride()
20            }
```

Listing 4.4: Definition of Initiate LU Task

A task has six distinct sections (highlighted in red): *Define, Environment, Initialize, Stages, Computation, and Partition*. The Define Section specifies all data structures along with their types. The Environment Section lists structures in the environment that this task manipulates. If the type of a structure is *link* then it must be available in the environment before the coordinator function can launch the task (similar to IN/OUT); if it is *create* then it will be added as a result in the environment after the task finishes execution (similar to OUT).

The remaining option is create-if-not-linked. Any non-environmental data structure is garbage collected at task completion.

The Initialize Section executes sequentially at the inception of the task. As the name suggests, it initializes data structures. In particular, it sets the dimension lengths of all arrays that are not linked so that parts of arrays can be distributed among LPUs properly.

The parallel core of the task is specified in the remaining three sections. The Stages Section lists all compute stages required to implement the logic of the task. A compute stage can be compared with a procedure in traditional programming language. The difference is that a stage embodies parallel instructions.

The Computation Section describes the parallel algorithm as a flow of compute stages to be executed in different LPSes. In this case, there is only one compute stage, *prepare*, to be executed in parallel on LPUs of Space-A.

The Partition Section specifies the configuration of the LPSes used in the computation flow, their relationships, and how data structure parts are distributed among LPUs of an LPS. In the *Initiate LU* task The Partition Section specifies a 1D partition for Space-A where the columns of a and rows of u and l are distributed using a unit stride to form the LPUs.

4.6.4 Specification of Computation

The Computation block of the *LU Factorization*, as shown in Listing 4.5, is a more illustrative example of specification of a parallel computation in an *IT* task.

```
1   Computation:
2       Space A {
3           Repeat for k in k_range {
4               Space B {
5                   Where k in u.local.dimension1.range { selectPivot(pivot, u, k) }
6               }
7               storePivot(p, k, pivot)
8               Space B {
9                   Where k != pivot {
10                      Epoch { interchangeRows(pivot, k, u, l) }
11                  }
12                  Where k in l.local.dimension1.range  { updateLower(l, k, l_row) }
13                  updateUpperRowsBlock(u, l_row, k, row_range)
14                  collectLColParts(l_column, l, k, row_range)
15              }
16              generatePivotColumn(p_column, l_column, row_range, k)
17              Space B {
18                  updateUpperColsBlock(u, p_column, k, row_range)
19              }
20          }
21      }
```

Listing 4.5: Algorithmic Logic of LU Factorization Task

The Compute Section describes the logic of the task as a flow of compute-stages in LPSes. In the computation of Listing 4.5, the flow of control operates entirely in the confinement of Space-A and repeats for a sequence of rows specified by *k_range*. Within a repeat loop, the control first enters Space-B to select the pivot element for the current iteration. Then it comes out of Space-B and records the pivot in a Space-A. Upon its re-entrance to Space-B; it executes stages of classic LU factorization such as *interchangeRows*, and *updateLower*. The difference with an *IT* implementation of the classic LU factorization and the current task is only a block or rows and columns of the upper triangular matrix are updated as opposed to all rows and columns. The update here happens in two steps using the *updaterUpperRowsBlock* and *updateUpperColsBlock* and an intermediate LPS transition to Space-A to generate a supporting data structure in the *generatePivotColumn* stage.

LU Factorization of Listing 4.5 involves several updates of shared variables inside compute-stages. For example, the *pivot* that is selected in the *selectPivot* stage (Line 5) is needed during execution of *storePivot* (Line 7) and *interchangeRows* (Line 10). The programmer ensures that only one LPU of Space-B updates the *pivot*

in an iteration of the repeat loop using the *Where* condition for the *selectPivot's* invocation. The *selectPivot* stage executes only on the LPU that has the associated condition true. Notice the use of the keyword *local* in the *Where* condition. It indicates that the associate range to be compared is for the part of the array an LPU has – not for the entire array. The compiler injects appropriate synchronization/communication in between the transition from *selectPivot* to subsequent stages to ensure that the up-to-date value of the *pivot* is available wherever needed.

The above discussion exposes another importance of breaking down the logic of the task into compute-stages. Once inside a compute stage, an LPU executes oblivious of other LPUs and any sharing of data part the former may have with the latter. The compiler ensures that data shared among LPUs being synchronized only at the compute-stage boundaries. In the absence of data dependencies, LPUs execute independently. Therefore, if the logic of a task supports it, different LPUs may be at completely different stages and iterations of the task's computation flow.

To lay out the flow of the task, three flow control instructions can be placed in the Compute Block. These are as follows.

```
1        Repeat Boolean–expression { nested sub–flow }
2        Where Boolean–expression { nested sub–flow }
3        Epoch { nested stages accessing version dependent data structures }
```

The *Repeat* instruction causes the nested sub-flow to iterate for the time-being the associated condition remains valid. The *Where* condition restricts the execution of the nested sub-flow in the LPUs the associated condition evaluates to true. The *Epoch* boundary is used to dictate when the version number of any version-dependent data structure should be updated.

4.6.5 Data Partitioning

Listing 4.6 illustrates the Partition Section for the *LU Factorization* task of Listing 4.5. Here, the upper-most LPS, Space-A, is un-partitioned and has been divided by 1-dimensional LPS Space-B.

```
1       Partition (b):
2               Space A <un–partitioned> { p, p_column, l_column }
3               Space B <1D> divides Space A partitions {
4                       u<dim1>, l<dim1>, l_column: block_stride(b)
5                       l_row, p_column: replicated
6               }
```

Listing 4.6: Data Partition Configuration for LU Factorization Task

The array *p* that keeps track of pivot row indexes from different iterations only exists in Space-A. So there will be a single undivided copy of it for the entire task and, hence, *storePivot* stage of Listing 4.5 (Line 7) that updates the array has been placed to execute in Space-A. The number of LPUs in Space-B, on the other hand, will depend on the number of strided blocks of *u* and *l* generated at task launch time. That is, the Space-B LPUs count depends on the input size and the partition parameter *b*. Some other data structures such as *l_row* will be shared among all those LPUs. Any of the replicated data structures can be updated by one Space-B LPU at a time only. Hence stages that update those structures are constrained with a *Where* condition in Listing 4.5.

4.7 Interoperability with other Languages

Since it is natural for programmers to have libraries written in other languages that they need to incorporate in an *IT* program, *IT* supports code snippets written in other languages to be included within compute stages of a task as *extern code blocks*. The syntax for an extern code block is illustrated in the following example.

```
@Extern {
        @Language ``Case sensitive name of the language"
        @Includes { comma separated list of header files needed to compile the code snippet }
        @Libraries { comma separated list of libraries to be linked to the executable for the snippet }
        ${
                The code snippet
        }$
}
```

In the above, the *@Include* and *@Libraries* sections are optional.

Any task-global scalar variable, user defined type, and non-partition-able data structures (such as a constant size array) are accessible within an extern block; so is any local variable declared within the compute stage the extern code block resides in. These variables are accessed using the same name within the extern code snippet as they are accessed in the *IT* code. In addition, the metadata of all task-global arrays and the runtime partition arguments are accessible within the snippet. In the letter case the programmer has to follow a different naming convention though.

Since the *IT* compiler does not interpret the content of an *extern* block, any change made within the extern block to a task-global variable or an element of a partitioned array is not traceable from *IT*. So the automatic dependency detection and subsequent synchronization and communication mechanism will fail to synchronize LPUs for such a change. Therefore, the programmer should rather do his/her update on a temporary variable inside the extern code snippet. Then after coming out of the extern block assign the value of the temporary variable to the variable/element he/she originally intended to update.

IT's interoperability support is geared toward accommodating compute-bound sequential code blocks and code blocks processing hardware information written using low-level sequential languages such as C or FORTRAN. Interaction with other parallel languages/tools which enables a bulk of computation to be done outside *IT* is never intended as that would encourage the old way of thinking. Consequently partitioned arrays

are kept unaccessible from extern code blocks. There would be a safety concern also if such interaction was allowed.

As *IT* arrays can be repeatedly partitioned along a task's LPS hierarchy where each LPS partition can potentially reorder the array indices, mapping an original index to its storage location and vice-versa is an intricate process. *IT* compilers take care of this transformation. If done incorrectly, the index transformation may easily lead to memory corruptions. Passing partitioned arrays to extern code blocks opens a floodgate of such opportunities [a].

4.8 Summary

Notice that despite emphasizing the importance of reasoning about hardware features in parallel programming, we avoided addressing the mapping of logical processing spaces of an *IT* program to the physical processing spaces of the *PCubeS* description of the target hardware. This is to show that an *IT* program should make perfect logical sense and be analytically accurate regardless of the target platform.

This is the very essence of hardware-cognizant as opposed to hardware-specific parallel programming. The *IT* abstract machine model of hierarchy of LPSes suggests that there must be a corresponding hierarchy of PPSes – equivalently a *PCubeS* description – for the program to execute, but a program is not written with a specific *PCubeS* architecture in mind. It is important to understand the subtle but important distinction between *PCubeS* + *IT* paradigm's treatment of the type architecture and that of Chapel, X10, or Legion here. The programmer can completely bypass the type architecture during both program construction and its runtime

[a] Currently the segmented memory and multicore compilers support C/C++ extern code blocks. Our future plan is to allow direct C++/CUDA codes inside compute stages of offloaded LPUs and support FORTRAN codes for all three back-ends architectures.

in those languages. In *IT*, the programmer has to construct the program on a *PCubeS* like model that he/she explicitly maps to the *PCubeS* description of the target hardware before the program can even compile.

A gamut of runtime behaviors can be achieved from the same *IT* source code using different mappings of LPSes to PPSes. In fact, if the program is sufficiently flexible, just a mapping variation can be enough to achieve good performance in diverse target architectures. The next chapter discusses the mapping process and related topics.

5

From Abstract Programming Model to Physical Hardware

This chapter discusses the mapping of LPSes of an *IT* program to PPSes of the target platform and the restrictions a particular mapping choice imposes on the *IT* program's runtime behavior regarding data placement and computation. This discussion will illustrate how the abstract machine model of *IT* directly corresponds

with the *PCubeS* description of the execution platform once mapping is done, which enables the programmer to assess and improve a program's performance.

Although *IT* offers considerable flexibility on how a compiler should implement LPU (and at the coarser level task) execution and coordination based on a mapping, all three compilers constructed as part of this research implement a common runtime model. This chapter also examines that runtime model and establishes the bridge between the theory and practice.

5.1 Mapping LPSes to PPSes

In terms of procedure, the mapping of a program's LPSes to PPSes of the target execution platform is straightforward. The programmer has to explicitly map each LPS of each task of the program separately to the PPS he/she deems appropriate. The only restriction in the LPS-PPS mapping is that the partial ordering of LPSes in a task's partition hierarchy must be respected when mapping them to PPSes. To elaborate, if Space-A is an ancestor of Space-B in a task's partition then the latter cannot be mapped to a PPS situated higher than that the former has been mapped to. Both LPSes can be mapped to the same PPS though. If the target architecture has multiple *PCubeS* descriptions, the programmer should specify the particular description being chosen using an optional *model* parameter.

The mapping is specified in a separate mapping configuration file and is an input to the compiler in addition to the *IT* source code. The grammar for the mapping configuration file is as follows.

```
1   mapping := task_mapping+
2   task_mapping := ``Task_Name" { model? lps_mapping+ }
3   model := Model : ``Model_Name"
4   lps_mapping := Space id : pps_no
5   id := [A-Z] | Root
6   pps_no := [0-9]+
```

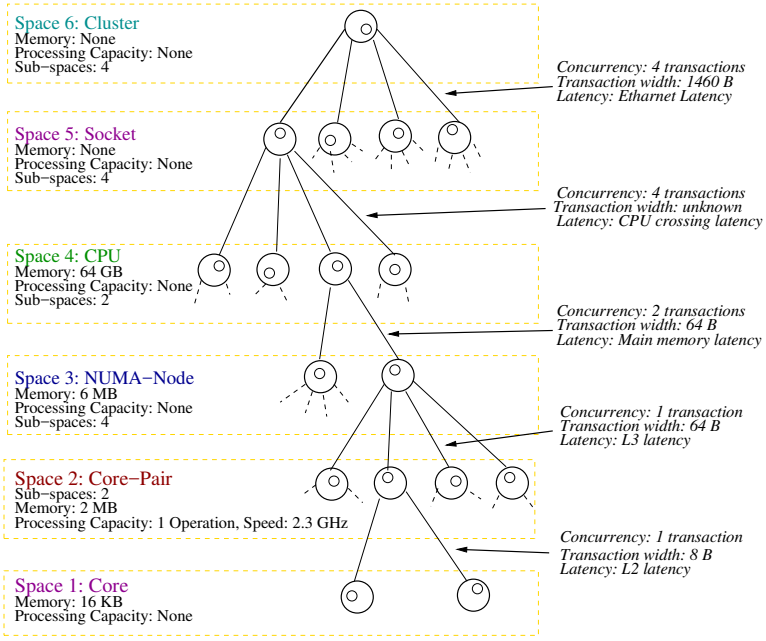


Figure 5.1: PCubeS Description of Hermes Cluster with One PPS Expanded at Each Level

To illustrate the mapping process with an example, assume an *IT* program has three tasks whose LPS hierarchies have the topologies of Figure 5.2 (the figure shows the invisible *Root* LPS for the entire program that tasks' LPS hierarchies descend from) and the *PCubeS* description of the target execution platform, called the Hermes Cluster [a], is as shown in Figure 5.1. Then a feasible mapping configuration for the program can be as shown in Listing 5.1.

[a] Hermes cluster is an actual cluster in Computer Science, UVA having 4 nodes each having 4 16-core AMD CPUs

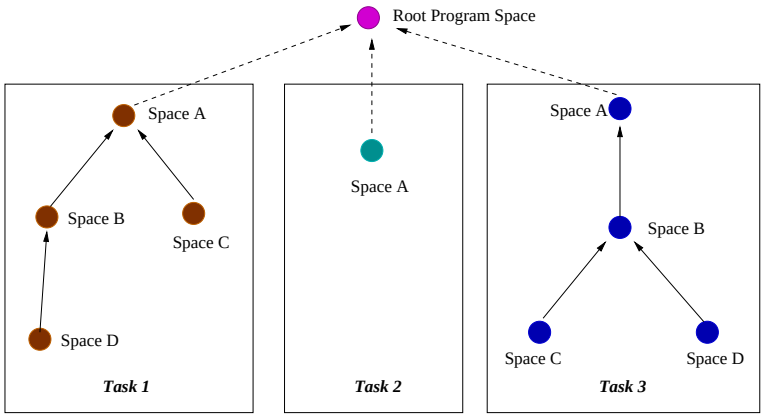


Figure 5.2: Pictorial Description of the LPS Hierarchies of an IT Program

```
1     ‘‘Task 1” {
2           Space A: 6
3           Space B: 3
4           Space C: 4
5           Space D: 1
6     }
7     ‘‘Task 2” {
8           Space A: 2
9     }
10    ‘‘Task 3” {
11          Space A: 4
12          Space B: 3
13          Space C: 2
14          Space D: 2
15    }
```

Listing 5.1: An LPS-PPS Mapping for the Program

Note that in Listing 5.1 above, the LPS-PPS mappings of individual tasks are independent of one another. For example, Space-B of Task 1 and 3 might have different data structures, or different partitioning for some common data structures, or even the exact same data partitions regardless of the LPSes being mapped to the same PPS in both cases. What happens to data organization as the program makes transition from one task to the other depends upon the actual runtime relationship between these tasks' partition hierarchies. The compiler is responsible for generating appropriate data movement instructions by comparing task partition

hierarchies and tasks' LPS-PPS mappings.

5.2 Consequence of an LPS-PPS Mapping

Despite the process of LPS-PPS mapping being straightforward, considerable discretion should be applied when it is done as the mapping efficiency alone can make a program perform good or bad on a particular execution platform (note that LPS-PPS mapping never affects a program's correctness). This is because the LPS-PPS mapping dictates the degree of parallelism at different phases of a program (and task), roles of different PPUs during program execution, computation and data locality, data communication and synchronization requirements, and even high-level memory management policies.

5.2.1 Degree of Parallelism in an IT Program

The degree of parallelism in an *IT* program varies with tasks and within a task in execution of compute stages assigned to different LPSes. When an LPS is mapped to a PPS of the target hardware, the LPUs correspond to that LPS are multiplexed to the PPUs of mapped PPS. So the degree of parallelism is dictated by the number of PPUs participating in LPU computation.

For example, in Listing 5.1 the sole LPS, Space-A, of Task-2 has been mapped to Space-2 of the Hermes cluster. An inspection of Figure 5.1 reveals that there are 128 Space-4 PPUs in the cluster (4 core-pairs per NUMA-node, 2 NUMA-nodes per CPU, 4 CPUs per socket/node, and 4 nodes in the cluster). The LPUs of Space-A will be distributed to these 128 PPUs and the compute stages of Space-A will execute 128-way parallel assuming that enough LPUs exist. Task-2 has only one LPS. Thus the degree of parallelism remains fixed for the time-being

of its execution. For tasks having multiple LPSes more interesting behavior can be introduced by mapping the LPSes to different PPSes.

For example, in Listing 5.1 Space-A of Task-1 has been mapped to Space-6 of the Hermes cluster. There is only one PPU in Space-6 in that machine, as apparent from the *PCubeS* description of Figure 5.1. Thus Task-1 compute stages that are assigned to Space-A will all execute in that single PPU. Space-B, on the other hand, has been mapped to Space-3 of Hermes cluster. There are total 32 Space-3 PPUs in the cluster (2 NUMA nodes for each CPU, 4 CPUs per node, and 4 nodes in the cluster). Therefore, Space-B compute stages will execute independently in parallel in 32 PPUs.

The multiplexing of LPUs to PPUs respects logical hierarchies of LPSes in the source task. For example, Space-D of Task-1 divides Space-B. According to the mapping specification, Space-B LPUs will be multiplexed to the NUMA-nodes. Then Space-D LPUs that divides a particular Space-B LPU assigned to a specific CPU will be multiplexed within the cores of that NUMA-node only.

So in the course of the execution of Task-1, the degree of parallelism varies from 1 to 256 according to the logic of the computation – and that is just Task-1. Task-2 and Task-3 exercise the PPSes quite differently from the way Task-1 does.

Note that Task-1 LPSes have been mapped to Space-6, 3, 4, and 1. According to Figure 5.1, none of those PPSes has any compute capacity. This raises the question how a code can execute in such a PPS. When an LPS is assigned to a PPS lacking compute/memory capacity, the PPUs of the latter execute the LPUs of the former using the compute/memory capacity of the closest ancestor/descendant PPS according to the following rule.

`PPS Substitution Rule:`

1. If an LPS is assigned to a PPS having no compute capacity then computation for all LPUs of that LPS

is done by a single PPU in the nearest lower/upper level PPS capable of computation.

2. If a PPU has insufficient memory to hold the data for LPUs it operates on then it will use the memory of the nearest ancestor PPU that can hold that data and stage data in/out as needed.

The programmer can confine the execution of a task within a smaller section of the target machine instead of consuming it entirely by specifying a mapping for the special Root LPS. Assume he/she wants to use the core-pairs of a single node as opposed to the entire cluster (as instructed by the mapping of Listing 5.1) to execute Space-A LPUs of Task-2. Then the LPS-PPS mapping for Task-2 should have the following configuration of Listing 5.2 instead.

```
1       ``Task 2" {
2               Space Root: 5
3               Space A: 2
4       }
```

Listing 5.2: an Alternative Mapping for Task 2

Space-Root is by-default un-partitioned and all LPSes of a task descends from it. Therefore according to the mapping of Listing 5.2, only one Hermes node will be utilized for this task and the overall degree of parallelism for the sole Space-A compute-stages will be 32.

This ability to vary the degree of parallelism can be a powerful tool to control a program's performance. For example, if the LPUs of an LPS operate on disjoint data parts that are large then a lesser degree of parallelism but more memory space per LPU might be a good choice; thus the LPS should be mapped to some higher level PPS. On the other hand, if there is considerable data sharing among LPUs and the data parts are small then emphasis might better be on more parallelism; thus the LPS should be mapped to some lower level PPS. Parts of a real world parallel application often have different computation and data usage characteristics. *IT*'s mapping feature enables the programmer to work in concert with those characteristic differences as opposed to find his/her way out of them.

5.2.2 A PPU's Role in a Program Execution

As shown in the mapping of Listing 5.1, the tasks of a program can have more or less LPSes than the PPSes available in the target hardware but LPS-PPS mapping can still be done smoothly by mapping multiple LPSes to the same PPS and/or skipping some PPSes altogether. Once mapping is done, however, the programmer should assume that the abstract LPS hierarchy of his/her program has materialized in the hierarchy of the PPSes being used in the mapping, and the cost of execution and coordination of LPUs of the program matches the capacities of the PPUs the former have been multiplexed into. If the programmer cannot assume that then the central theme of the paradigm, hardware-cognizant programming, is compromised.

This assumption imposes restrictions on how PPUs are realized at the execution time – not the choice of their implementation primitives (e.g. processes or threads) though. To understand the restrictions with an example, assume the partition configuration of Task-1 of Figure 5.2 is as shown in Listing 5.3 and the computation flow of the task has a section as shown in Listing 5.4. Further assume that all three stages in Listing 5.4 update *matrix_b*.

```
1 Partition(k, l, m):
2         Space A <un-partitioned> {
3                 matrix_a, matrix_b, vector
4         }
5         Space B <2d> divides Space A partitions {
6                 matrix_a: block_size(k, l)
7                 matrix_b<dim2, dim1>: replicated, block_size(1)
8         }
9         Space C <1d> divides Space A partitions {
10                vector, matrix_b<dim2>: block_stride(m)
11        }
12        Space D <1d> divides Space B partitions {
13                matrix_b<dim1>: stride()
14        }
```

Listing 5.3: Task 1 Partition Configuration

```
1  Space B {
2          Repeat for i in matrix_b.dimension1.range {
3                  Space D {
4                          stage1(matrix_b)
5                  }
6                  stage2(matrix_a, matrix_b)
7          }
8  }
9  Space C {
10         stage3(vector, matrix_b)
11 }
```

Listing 5.4: A Part of Task 1 Computation Flow

The flow of computation resides in Space-B from Line 1 to Line 8 in Listing 5.4. Given that Space-B has been mapped to NUMA-nodes (Listing 5.1), at runtime the execution of that part of the task is expected to happen inside the confinement of individual NUMA-nodes. This condition should hold even for any data movement needed to resolve the data dependency for *matrix_b* between Space-B and D LPUs. If parts of *matrix_b* have to escape the NUMA-node confinement for the sake of data dependency resolution in-between *stage1* and *stage2* transitions then that adds hidden runtime overhead that the programmer cannot estimate. During the transition from Space-B to Space-C at Line 9, however, data reshuffling should take place within the entire cluster. This is because despite Space-C has been mapped to CPUs, the NUMA-node PPUs of individual CPUs do not necessarily hold the data needed by the latter as Space B and C are unrelated in the partition hierarchy.

This example illustrates that the PPUs cannot be realized as one-dimensional entities responsible for executing instructions only. Rather, the ideal implementation of PPUs should involve donning execution, memory management, and data dependency resolution roles as dictated by the runtime context.

5.2.3 Data and Computation Locality

Since data structures are assigned to LPSes and not being owned by them, the LPU execution model to be realized in PPUs is not owner compute. This becomes obvious if one considers that LPUs may have shared data that they can individually modify or LPU data partitions may be overlapped. For example, the same *matrix_b* part is shared among all Space-B LPUs of Listing 5.3 that occupy the same position along the 2^{nd} LPS dimension. The sole restriction on the data modification is that there is no race condition. Thus a specific implementation can pass around a single data part among interested PPUs or have per-PPU versions of the same part that are updated as needed.

The LPU execution model required in *IT* should, rather, be called local compute: a PPU executes instruction streams on data made available locally for the LPUs being multiplexed to the PPU. Thus what is local and what is not is an important consideration for the compiler to implement the PPUs and for the programmer to understand the performance of programs running over those PPUs.

Consider again the mapping of Task-1 LPSes in Listing 5.1 in this regard. None of the LPSes has been mapped to the Core-Pair PPS: the only PPS capable of computation in the Hermes cluster. Nevertheless, all computation will take place in the core-pairs as dictated by the PPS Substitution Rule. When computing for a Space-D LPU (e.g. *stage1* of Listing 5.4), once loaded, associated data is expected to stay in the core memory (given the data fit into that memory) for the course of that LPU execution as Space-D has been mapped to cores. In other words, the boundary for data locality is the core memory when dealing with Space-D. Subsequently the same core-pair may compute for a Space-C LPU (e.g., *stage3* of Listing 5.4). Then the associated data can reside anywhere within the CPU, as Space-C has been mapped to the CPU PPS, and still be local to that core-pair.

On the surface this may appear to be a mere reiteration of the *PPS Substitution Rule*, but probing deeper, one realizes that the locality restriction provides a definitive memory management implementation guideline for the compiler and consequently enables the programmer to estimate the memory utilization efficiency of an executing program.

In the context of the Hermes cluster and Task-1 mapping example, this means any *IT* compiler implementation needs to ensure L1 cache data locality (which is the core memory) for Space-D computations and main memory data locality (which is the CPU memory) for Space-C computations. A clever compiler may further optimize a Space-C computation's memory access but that is optional and doing so will not violate any conservative estimate of memory efficiency the programmer may deduce based on the *PCubeS* description and mapping configuration.

5.2.4 Memory Management

The partition configuration of a task such as Task-1 partition in Listing 5.3 establishes the hierarchical relationship among LPSes and their data contents. The actual sizes of the data parts are, however, only known at runtime by applying the partitioning functions with the runtime arguments for partition parameters on the input data structures. The paradigm encourages the programmer to choose the partition arguments in a manner so that his/her LPU data parts are ideal for the PPUs that will do the execution. Hence memory consumption on overhead data structures that the programmer is unaware of has to be negligible. As any PPU implementation will have some memory footprint, one way to deal with this issue can be to reserve a small fraction of the hardware features' memory capacity for PPU implementation and expose the remaining capacity through the *PCubeS* description. Still, the reserved fraction has to be small to be true to the paradigm's objectives.

Furthermore, data parts should be present in a particular memory only for the time they are expected to according to the logic of the program and the mapping configuration, or their existence should not affect the performance of an ongoing LPU execution they are not part of. This is because the programmer is expected to reason about the system that way. Therefore, the timeliness of data movement in and out of memories is an important implementation concern along with the automatic garbage collection demanded by the programming model.

5.2.5 Concurrent Task Executions

According to the *IT* programming model described in Chapter 4, tasks in a program can execute concurrently as long as there is no data dependency among them and the hardware resources needed to execute them are available. Concurrent task executions, however, can introduce unexpected runtime behavior such as cache thrashing among LPUs of different tasks that runs counter to the objective of enabling proper performance assessment. So it is important to understand the notion of resource allocation in the context of *IT + PCubeS* paradigm to support concurrent task executions properly.

The resource allocation requirement can be summarized in a single sentence. *When a particular LPS-PPS mapping is chosen for an* IT *task, not only the computation resources but also the memory and communication resources the mapping entails are committed to the task for the time being of its execution.* Thus Task-1 of Figure 5.2 cannot run concurrently with the remaining two tasks given the mapping of Listing 5.1 even if many core pairs remain idle during former's execution. This is because the particular mapping of Task-1 commits the entire cluster to the task. Had the mapping configuration been as in Listing 5.5 then all three tasks could run

concurrently in three different Hermes nodes in the absence of any data dependencies [a].

```
1       ‘‘Task 1” {
2               Space A: 5
3               Space B: 3
4               Space C: 4
5               Space D: 1
6       }
7       ‘‘Task 2” {
8               Space Root: 5
9               Space A: 2
10      }
11      ‘‘Task 3” {
12              Space Root: 5
13              Space A: 4
14              Space B: 3
15              Space C: 2
16              Space D: 2
17      }
```

Listing 5.5: An Alternative Mapping for the Program in Figure 5.2

This restriction on concurrent task executions may appear wasteful but it is not. If the programmer wanted he/she could combine the independent tasks together to form a bigger task allowing LPUs of independent LPSes to execute in parallel in idle PPUs. The requirement is any nondeterministic behavior is introduced and accordingly understood by the programmer instead of by the compiler. Allowing opportunistic exploitation of parallelism by the compiler achieves nothing that cannot be achieved via other means; it only undermines the goal of predictable runtime behavior.

5.3 Designing and Debugging an IT Program

After the *PCubeS* type architecture, the *IT* language, and the mapping process; there is nothing more for a programmer to learn about the *PCubeS* + *IT* paradigm to program in it efficiently. So this is the proper place to reflect on earlier discussions and see how we envision the program construction process to be in this paradigm.

[a] Task 1 does not need an explicit Root LPS mapping as its own top-most LPS, Space-A, is un-partitioned (Listing 5.3) which effectively encloses the task's resource consumption to a single Hermes node.

It starts with an investigation of the nature of the specific problem at hand that the program attempts to solve. The programmer should focus on identifying different forms and granularities of parallelisms in the problem and consider different algorithmic approaches for exploiting those opportunities. For example, different trade-offs between space and time overhead is possible when solving a finite difference problem using the existing algorithms. Before making a specific choice of algorithm, however, he/she should consider the kind of *PCubeS* architecture the program will be targeted to and decide accordingly.

Some aspects of *IT* programming are no different from generic parallel computing principles. For example, if the chosen algorithm has independent or loosely connected components then breaking them into separate tasks is a good practice in general. The paradigm ensures that there is negligible overhead in having multiple tasks as opposed to a single task if the data arrangement remains the same. If the data need to be rearranged then that provides a stronger argument in favor of multiple tasks.

It is the construction of individual tasks that may confuse a rookie programmer. The big question is how to architect a portable task that can execute efficiently in different target platforms without or with little modifications. This concern may be bewildering given that most contemporary parallel machines are deeply hierarchical which may suggest the need of corresponding long hierarchy of LPSes in the task. The problem, however, is not as perplexing as it may appear in the beginning and the answer again lies in the nature of the algorithmic sub-part the task is intended for.

If the algorithm has interdependent phases having different computation and data access characteristics then those phases should be implemented as compute stages. Then the phase transitions and phases' hierarchical relationships should be encoded as a computation flow that makes transition between LPSes along the way it executes compute stages. If the opportunity of such breakdown is not inherent in the algorithm then no

attempt should be made to introduce it artificially. For example, a vector-vector addition can have one phase only and a blocked matrix-matrix multiplication task can hardly be improved beyond the definition given in Listing 4.1 of the previous chapter. An LU factorization on the other hand has several phases allowing a more interesting flow definition.

Data partition specification for the LPSes immediately follows the computation flow definition. In fact they should often be constructed together as the validity of the computation flow is tied with the specification of the data partitions. Even in the partitioning step, the specific *PCubeS* target can be largely ignored; instead the programmer can analyze the task's behavior assuming there is a one-to-one correspondence between the LPS hierarchy and the target machine's PPS hierarchy. Possibility of load imbalance and loss of parallelism can be discovered from analysis done on the assumed ideal machine and should be dealt with by changing the partition functions accordingly.

Details of the *PCubeS* description of the target become major concerns only during the mapping step. Mapping can be an involved process for a complex program with many tasks, but given LPS-PPS mapping does not affect program correctness, it can be improved gradually. An uncertain programmer may start with an LPS-PPS mapping that makes the program completely sequential, utmost parallel, or anything in-between. None of those initial choices may be appropriate, but the predictability of the runtime behavior enables the programmer to identify the sources of mapping inefficiencies and rectify them.

Finally, note that the program construction is not a linear process as described above. Initial inadequate performance may not be rectifiable by mapping variations alone. The programmer may need to refine tasks' computation flow and partition specifications, and how tasks interact in the program. Unlike some other programming paradigms though, the refinement can lead to a more flexible, consequently more portable, program

definition when carefully done.

5.4 Implemented IT Run-time Model

As long as the requirements of the language model and the mapping configuration are satisfied, different implementations are possible for the *IT* runtime engine (RTE) for task and PPU handling. In fact, it is better to choose an implementation model that can utilize the strength of the underlying low-level programming primitives an *IT* compiler uses to generate the executable in a specific *PCubeS* platform. Nonetheless, the three compilers we have developed as part of the research implement a common RTE. This section gives an overview of that RTE for illustrative purposes.

5.4.1 Run-time Process Model

The run-time process model for *IT* is straight-forward. On each compute node of the target execution environment a single process is started. The process executes the *IT* program that includes references to the *IT* run-time libraries described below. The main program starts up, initializes global variables, and starts the program controller.

Applications compiled with the multicore complier have just the single process running on the host. At runtime, parallelism is achieved by using Pthreads threads that play the role of PPUs. LPUs are assigned to PPUs, and PPUs execute a loop in which they execute the LPUs that have been assigned to them and synchronize as necessary.

Applications compiled for distributed memory machines use one MPI process per node with one corre-

sponding Unix process for each node. When the node has multiple cores (almost always the case today) then Pthreads threads are used for intra-node parallelism in much the same way as in the multicore compiler. Communication and synchronization within a node is achieved using shared memory and Pthreads barriers. Inter-node communication and synchronization is performed using MPI.

Applications that use accelerators, in particular GPUs, have a slightly different execution model. As above, each node executes a single Unix process. If the cores are to be used then there is one pthread per PPU specified by the user in the mapping file. The PPU controllers get the next LPU to execute from an internal data structure executing them as required. If a set of LPUs in an LPS are to be executed on the GPU, a single controller thread is started that manages interactions with the GPU. The GPU controller thread is responsible for copying LPU data structures onto the card, initiating the kernel calls on the GPU, and staging data back from the cards.

LPU execution on the GPU is managed by LPU management code executed on each GPU SM that stages data into the SM from card memory, executes the LPU, and copies data back to card memory. The SM LPU management code in the kernel continues executing until it has executed its entire batch of LPUs, at which point it terminates.

Once the GPU kernel calls have completed, the GPU controller thread on the host first copies data back from the GPU, then if there are more LPUs to execute, the GPU controller thread sends another batch of LPU's to the GPU. This process repeats until all of the LPUs have been executed.

5.4.2 Components of the RTE

The architecture diagram of the implemented RTE is depicted in Figure 5.3. The components of the RTE are grouped into three hierarchical categories based on their lifetime and role during the execution of an *IT*

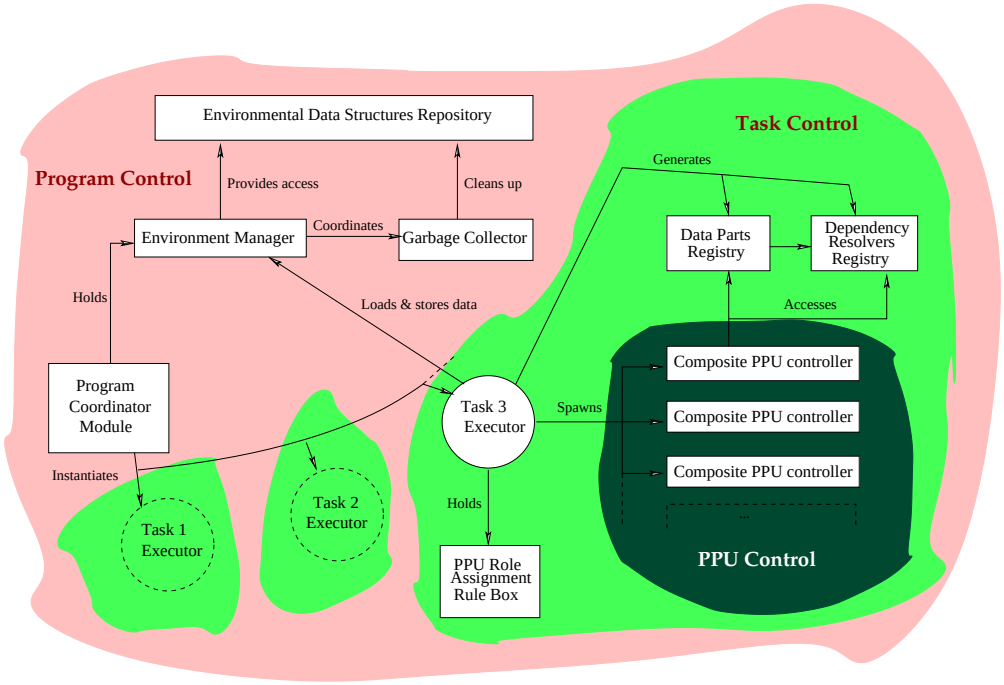


Figure 5.3: Architecture Diagram for IT Run-time Execution Engine

program.

The top level, *Program Control*, components aid in scheduling and coordination of the tasks of an *IT* program. These components remain active for the entire duration of a program execution. The components are as follows.

- **`Program Coordinator Module:`** executes the logic of the program coordinator function of the *IT* program and schedules tasks.
- **`Environmental Data Structures Repository:`** holds parts of environmental data structures created and updated by different tasks.
- **`Environment Manager:`** ensures freshness of data in the repository, provides access to data parts from different tasks, and governs garbage collection.
- **`Garbage Collector:`** removes environmental data structures that are no longer needed and reclaims their memory.

Currently, the RTE does not support concurrent task executions. Hence there is no inter-task dependency checker among the Program Control components.

The middle level, *Task Control*, components set up software and hardware resources for a single task execution. These resources remain alive during the lifetime of that particular task and are accessed by all PPUs that participate in the task execution. The components are as follows.

- **`Task Executor:`** configures and manages all resources for a task execution, interacts with the Environment Manager for data preparation for the task, and cleans all task specific resources before exit.
- **`Data Parts Registry:`** holds all data structures, both environmental and non-environmental, needed by the task.
- **`Dependency Resolvers Registry:`** holds synchronization and communication resources for data dependency resolutions among PPUs during task execution.
- **`PPU Role Assignment Rule Box:`** retains the hardware resource configuration logic that the Task Executor uses to run the PPUs.

The bottom level, *PPU Control*, components are the interacting parts of a single physical processor responsible for executing PPUs. Since a single physical processor in the target hardware may be responsible for executing code and managing data for multiple PPUs, the whole of the parts is called a *Composite PPU Controller*. Figure 5.4 shows the content of a Composite PPU Controller.

A Composite PPU Controller remains affixed to a particular physical processor during a task execution and has the following components.

- **`Flow Executor:`** executes the computation flow of the task on the LPUs desginated for the Composite PPU Controller.
- **`Role Assignments:`** answers if the querying Composite PPU Controller has the proper role to execute a particular part of the computation flow of interest.

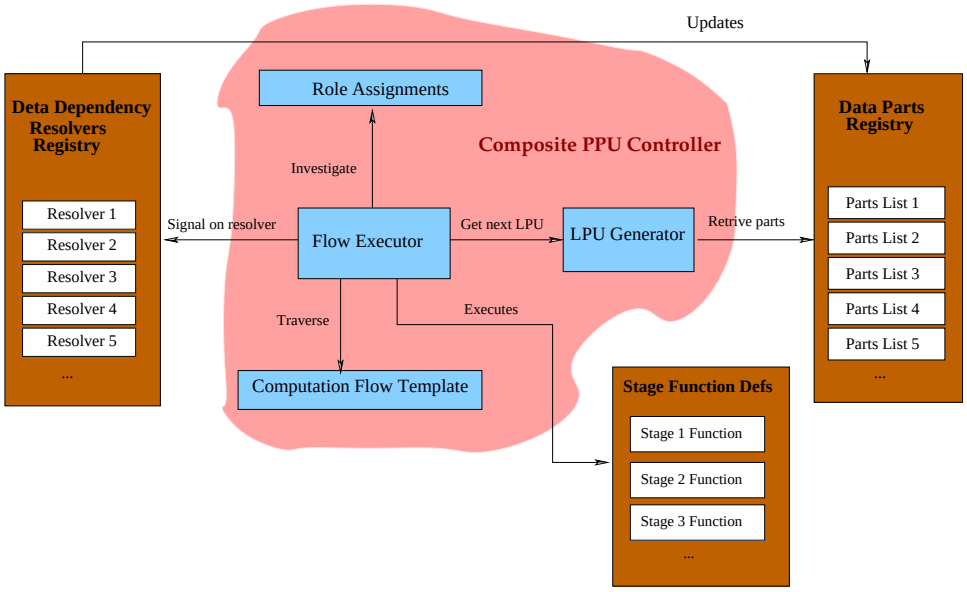


Figure 5.4: Expanded View of a Composite PPU Controller

- **`Computation Flow Template:`** is a graph representation of the computation flow of the task that the Flow Executor steps through.
- **`LPU Generator:`** generates LPUs on demand for the Flow Executor.

The components of the RTE are implemented in different ways for different back-end architectures. Even their interfaces are not the same across the board. Nevertheless, the services they provide and the manner they interact are uniform in all our implementations. Now we know the services; next we discuss how and when the RTE components interact to execute an *IT* program.

5.4.3 Interactions among RTE Components

For the sake of scalability, the RTE is implemented as a system of distributed processes. That is, as opposed to a centralized controller managing RTE components throughout the hardware; multiple instances of RTEs run independently across hardware units as processes. In the current implementation, a single process is created

for each largest swath of shared memory (e.g., one per CPU or cluster node) but other placements (e.g., one per NUMA-node) are possible and may be better performing. An RTE process encompasses the Program Control components and creates, governs, and destroys other components as needed for the tasks the process participates in execution.

Program Coordination and Tasks Scheduling

The Program Coordinator Module of a process schedules tasks one after another as they are encountered in the program coordinator function. Before scheduling the next task, the Program Coordinator Module interacts with the Environment Manager to ensure all environmental data structure parts needed for the task execution are locally available, up-to-date, and are in their proper format. The Environmental Data Structures Repository only holds data structure parts that are relevant to the current process. Therefore, if successive tasks partition a common data structure in different ways then data re-shuffling is needed in-between task transitions. Environment Managers of participating processes cooperatively perform the re-shuffling – there is no centralization of data.

Re-shuffling of a data structure parts consists of two steps. First, each process communicates to all other processes a description of its current content for the data structure of interest and another description for the content it needs to launch the upcoming task. Second, based on that information, a process determines what data it needs to send/receive to/from whom then acts accordingly. Upon receiving control back from the Environment Manager, the Program Coordinator Module instantiates a Task Executor of appropriate type to execute the task.

After each task completion, the Environment Manager investigates the Environmental Data Structures Repos-

itory and instructs the Garbage Collector to remove any data structure parts that are no longer needed.

Task Preparation

The Task Executor interacts with the Environment Manager to load/store environmental data structures. A task may use other non-environmental data structures too as auxiliary variables. All data and metadata needed for successful execution of the task are prepared and stored in the Data Parts Registry. Task Executors of different processes collectively determine and agree on the proper communication primitives for resolving the various data dependencies that will arise during the course of execution. Each stores local descriptions of those primitives in the Dependency Resolvers Registry.

Some data dependencies are confined within the PPUs of the individual processes. Each Task Executor prepares proper data movement and synchronization primitives for resolving those dependencies independently of others and places those primitives in the Dependency Resolvers Registry also.

The PPU Role Assignment Rule Box serves as a static template generated from the mapping configuration for assigning PPUs to the processing elements of the hardware. Once the data and resolver registries are ready, the Task Executor spawns a number of Composite PPU Controllers, configures them as dictated by the PPU Role Assignment Rule Box, and then allots them to proper processing elements to execute the LPUs. The rest of the task execution is collectively taken care of by these Composite PPU Controllers.

Each Composite PPU Controller reports its completion to the Task Executor. After receiving notifications from all of them, the Task Executor does the resource cleanup.

PPU Execution

Within a Composite PPU Controller, the Flow Executor executes the logic of the task. All Composite PPU Controllers are given the same Computation Flow Template [a] and independently traverse it. As a Flow Executor encounters a direction to execute any stage in the Computation Flow Template, it checks its Role Assignments to determine if the owner Composite PPU Controller has the specific PPU role required to execute that stage. If the role checking is successful then the Flow Executor instructs the LPU Generator to generate the LPUs multiplexed to the PPU one-by-one as the Flow Executor executes the concerned stage on each LPU.

LPU generation does not cause any memory allocation for data structure parts. The LPU Generator only gathers relevant data structure parts by querying the Data Parts Registry using the metadata supplied by the Flow Executor. Further, LPUs are generated and processed one at a time so that a single LPU instance can be updated with new data structure part references and there is no dynamic memory allocation for LPUs either.

If the stage execution requires some data dependency to be resolved first or creates new dependencies afterwards then the Flow Executor contacts the Data Dependency Resolvers Registry, retrieves the specific resolver for the dependency under concern, and issues appropriate *signal* on the resolver. Each resolver is already tailored by the Task Executor for the group of Composite PPU Controllers that participate in the dependency resolution. So upon receiving a signal, the resolver does the intended action, if applicable updates data parts in the Data Parts Registry, and then passes control back to the Flow Executor so that it can proceed. When the Flow Executor reaches the end of the Computation Flow Template, task execution is done for the owner Composite PPU Controller which then notifies the Task Executor.

[a] A Composite Flow Template is a translated form of computation flow from the *IT* task annotated with partition and data dependency information.

In the above, the Flow Executor does not know the inner mechanism of the dependency resolvers. Neither does it understand the specific implementation functions/procedures used for the compute stages. It only invokes those functions/procedures with proper LPU data. So these pieces can have different implementations in different platforms. Further, note that as LPUs execute independently in the language model except for data dependencies so do the Composite PPU Controllers in the actual implementation.

6

Compilation

This chapter discusses the three *IT* compilers developed as part of this research. These compilers generate executables for multicore CPUs, supercomputers and compute clusters constructed from multicore CPU nodes, and hybrid supercomputers having both multicore CPUs and NVIDIA GPUs as nodes respectively. These are all source-to-source compilers that generate C++ code parallelized with tools [b] appropriate for respective target environments. Although each new *PCubeS* back-end type brings its own challenges for code generation, the

[b] MPI, Pthreads, and/or CUDA

compilers are largely built on top of each other. The chapter also describes the general strategy for breaking down the compilation process that makes such stacked compiler implementations feasible.

The three architectures are chosen as targets to show the portability of the *PCubeS* + *IT* paradigm across machine types common in contemporary high performance parallel computing problems. Due to time constraints, the focus was never on making a single compiler perfect. This left all three compilers with significant optimization opportunities. The chapter concludes with a review of those optimization opportunities.

Discussions of this chapter frequently refer to the *IT* Runtime Environment (RTE) presented in Chapter 5 Section 5.4. So the reader should review that section before proceeding further.

6.1 The Broad Picture

One can view the act of programming as a three-step transformation process of the form

$$Problem \rightarrow^{\text{to}} Algorithm \rightarrow^{\text{to}} Program \rightarrow^{\text{to}} Executable$$

in which the programmer is responsible for the first two transformations and the compiler for the third. Then in the context of *PCubeS* + *IT*, the program is actually a combination of two things: the source code and the mapping configuration. The *IT* compiler is expected to understand the directives of the program and transform them to proper machine instructions based on an intimate knowledge of the underlying hardware. Since the program directives are expressed in terms of *PCubeS*, the compiler's knowledge of the hardware entails understanding its *PCubeS* description and the correspondence between elements of that description and the features of the actual hardware. Therefore in terms of input/output, the compilation process can be depicted as in Figure 6.1.

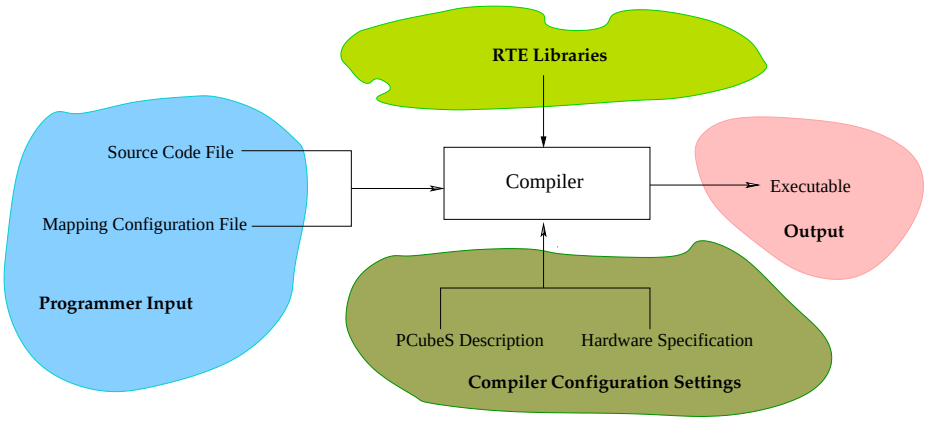


Figure 6.1: A Schematic Diagram of the Compilation Process in Terms of Input/Output

In Figure 6.1, the *PCubeS* description and the hardware specification are depicted as external to the compiler as opposed to its integral parts. This is intended as from the inception the plan was make *PCubeS* compilers rely on existing, matured, and widely used low-level parallel programming primitives available in different architecture types for the architecture specific aspects of the code generation process. This strategy allows a single compiler to work in all target hardware where a particular set of low-level programming primitives such as MPI or Pthreads are applicable. The compiler is responsible for ensuring that those primitives are applied accurately, efficiently, and in accordance to the *PCubeS* description of the hardware.

There are three steps in the compilation process as shown in Figure 6.2.

The first step takes the *IT* source code as the sole input and produces an intermediate code for a virtual machine specification [a]. The second stage takes the intermediate code and the mapping configuration, constrains the virtual machine specification with actual hardware characteristics, and then generates a low-level native program (a C++ program parallelized with MPI, Pthreads, and/or CUDA). Finally, the pieces of the na-

[a] The virtual machine specification has been described in Section 6.2.

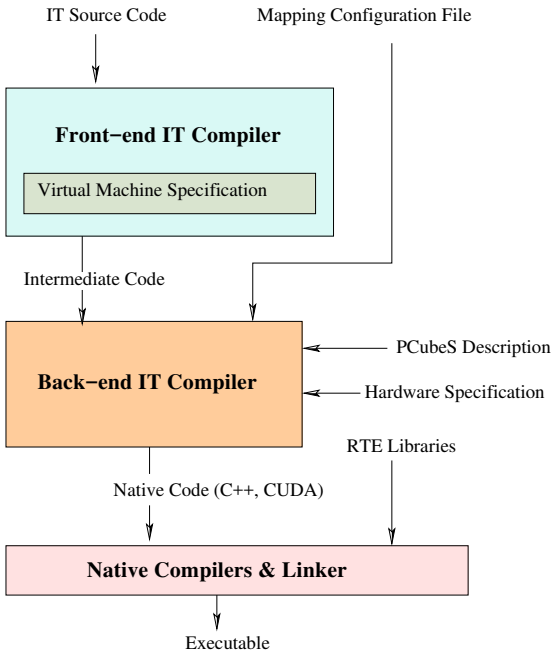


Figure 6.2: Generic Architecture of an IT Compiler

tive program are compiled with appropriate native compilers and linked together with RTE and user supplied libraries to form the executable.

The *IT* programming model makes the compiler responsible for all memory management, data synchronization, and communication for a program. From the discussion of Chapter 5, it is apparent that the programmer's choice of the LPS-PPS mapping and the features of the execution platform dictate the proper implementations of those aspects. Nevertheless, the policy for memory management, data synchronization, communication, and so on can largely be determined from the *IT* source code directly when viewed in light of a proper virtual machine specification. This is a result of the particular structuring and declarative syntax of the *IT* program, and it has a great consequence. The consequence is we can describe data allocation requirements, data access characteristics, and data dependencies within LPSes and LPUs of individual *IT* tasks in an abstract way that

hold true irrespective of the LPS-PPS mapping and the specific target execution platform. Hence we have the front-end compiler transforms the *IT* source code and augments additional details to the transformation as policy directives for the back-end compiler. The back-end compiler for a specific target further refines and then implements those directives based on the mapping configuration and machine characteristics.

This strategy enables us to keep the front-end compiler fixed and only vary the back-end compiler and run-time library implementation to generate executables for different architecture types. The challenge is in defining a proper virtual machine that supersedes any real *PCubeS* architecture and having an elaborate intermediate code that has all information needed to guide generation of a proper native program for any target execution platform.

Subsequent sections discuss the front-end compiler and the three back-end compilers we implemented. Before we dive into that, however, it is important to inform the reader about the syntax difference between the 1^{st} version of *IT* that the compilers currently parse and the 2^{nd} version that is being used throughout this writing to explain the language and is being published in our technical report[96].

6.1.1 Syntax Difference between Two Language Versions

The initial version of the *IT* language that we drafted in late 2013 remains the same in terms of the expressive power of the language, semantics of the features, and all other regards with one notable exception: the syntax of the language has undergone several changes as we discovered strangeness and irregularities in it. Most of these changes are syntactic sugar or minor tweaks that we believe have improved the readability of *IT* programs. The only major syntax change is the way the computation flow of a task is specified, and the change has a significant bearing on the front-end compilation process. Listing 6.1 and 6.2 illustrate the syntax change in the context of

the *Block Matrix-Matrix Multiplication* task discussed in Chapter 4.

```
1 Stages:
2         multiplyMatrices(x, y, z) {
3                 do { x[i][j] = x[i][j] + y[i][k] * z[k][j]
4                 } for i, j in x; k in y
5         }
6 Computation:
7         Space A {
8                 Repeat foreach sub-partition {
9                         multiplyMatrices(c, a, b)
10                }
11        }
```

Listing 6.1: New Syntax for Specifying a Task Computation

```
1 Compute:
2         ``multiplyMatrices" (Space A) {
3                 do { c[i][j] = c[i][j] + a[i][k] * b[k][j]
4                 } for i, j in c; k in a
5         }
6         Repeat: from ``multiplyMatrices" foreach Space A sub-partition
```

Listing 6.2: Old Syntax for Specifying a Task Computation

In the new syntax, the compute stages of a task are defined in the *Stages Section*. The stages have no LPS association of their own. The algorithmic logic of the task is then expressed as a flow of those stages through LPSes in the separate *Computation Section*. The letter section uses LPS markers (e.g., **Space A** of Line 7) and other control constructs (e.g., repeat statement of Line 8) along with the stage invocations to define the flow. In the old syntax, however, these two sections are fused together. Consequently compute stages are fixed to particular LPSes, directly access task-global variables, and bear LPS marker tags.

We switched to the new syntax, primarily, to be able to invoke the same compute stage with different arguments and from different LPSes, which would cause redundant stage definitions in the old syntax. In addition, we wanted to have a clearer separation of concerns between coarser-grain data parallelism realized through parallel stage executions in different LPUs from finer instruction parallelism within individual stages. From the compilation perspective the situation does not change much.

There is another consequence of this syntax change. The new syntax enables type-polymorphic stage definitions that were not even a concern in the old syntax. Since *IT* extensively uses type-inference, having type-polymorphic stages appears to be a natural choice.

Type polymorphism is strictly a front-end compiler concern. Therefore, we need to make change in one place only. Due to time constraints, however, we did not incorporate any changes ensued by the new language definition to the front-end compiler that has already been fully functional before those changes. So the subsequent discussion on compilers does not address the issue of type-polymorphism. Type polymorphism is a well-studied issue that should not be difficult to implement in *IT* compilers, as our draft assessment suggests.

6.2 The Front-End Compiler

We mentioned before that the front-end compiler assumes a virtual machine specification to generate an intermediate representation of the *IT* source code that can be used by any back-end compiler to generate an efficient native program. The virtual hardware has the following properties.

1. It is a fully distributed-memory *PCubeS* architecture with arbitrary number of PPSes each having a non-zero processing and memory capacity.
2. Each PPS in the hardware has more than one PPUs
3. Direct communication is possible between PPUs of a pair of PPSes regardless of the number of intermediate PPSes separating the two in the hierarchy.

In the absence of any mapping configuration at this step of compilation, it is further assumed that each LPS of a task has been mapped to a different PPS of the hardware.

The merit of choosing this particular machine specification is that it exposes all potential data dependencies in an *IT* task and the nature of those dependencies. A reflection on the three aforementioned properties shows why that is true.

First, the hardware has distributed memory in all PPS levels. That means any LPS transition in the task's computation flow will ensue a data movement requirement if some data common in the involved LPSes being updated in the former and read in the upcoming compute stage to be executed in the latter.

Second, there are multiple PPUs with disjoint memories in each PPS. Hence an update of a data structure having overlapped LPU partitions or being shared among LPUs will also create data dependencies within the PPUs of the single PPS executing those LPUs.

Finally, direct communication is possible between PPUs of any pair of PPSes. Given that LPSes are assumed to be mapped to different PPSes, this makes LPS and PPS mutually interchangeable and enables data dependencies to be characterized (such as all-to-all, scatter, gather, or point-to-point dependencies) based on the positions of the involved LPSes in the LPS hierarchy. The idea is to account for all possible dependencies in the first step then configure or remove specific dependencies based on the actual hardware features and mapping configuration during the back-end compilation step.

Another important responsibility of the front-end compiler is to process the data partitioning and alignment instructions of tasks' *Partition Section* and encode all those instructions in a format useful during the back-end compilation. Efficiency of this encoding is pivotal for successful back-end compilation as memory allocation, communication, compute stage translation, more-or-less all aspects of the back-end compilation are affected by it.

The front-end compiler also does type-inference, data version update tracking, and other analyses on the

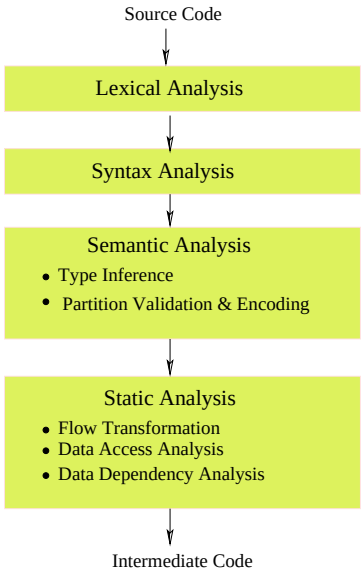


Figure 6.3: Phases of the Front-end Compiler

source code to generate the intermediate code. The phases of the front-end compiler are illustrated in Figure 6.3 with key sub-phases highlighted. There is no separate intermediate code generation phase in the Figure. This is because successive sub-phases of Static Analysis incrementally build up the intermediate code.

Lexical and Syntax Analysis and some parts of Semantic Analysis (e.g., scoping) are not particularly interesting. The interested reader can find how to do those in any compiler textbook such as[11]. Here we briefly discuss the highlighted sub-phases that are challenging and specific to *IT*.

6.2.1 Semantic Analysis

Type Inference

Since the programmer is required to provide data types of only the task-global variables in the *Define Section* of a task and in the properties of his/her custom data types, the compiler has to determine types of any other

variables introduced anywhere else. This is done through a recursive type inference process. At each pass of the recursion, the compiler takes input the variables and functions whose types are already known and tries to infer the types of variables and functions from those known types. If no new variable or function type is determined at a particular pass but there are still unknown types then the process has come to a fixed-point, the compiler halts and asks the programmer to specify the unknown types explicitly. Type validation takes place along with type inference. If some type errors are discovered in a particular pass then the process halts and error messages are displayed. The type inference process terminates successfully and hands over control to the next sub-phase when there are no more unknown types.

Partition Validation and Encoding

Although the grammar for the *Partition Section* provides a human-readable and intuitive mechanism to specify tasks' LPSes, their data content, and data partitions; the immediate parsed form is not directly usable in subsequent compilation phases. Take the partition configuration of Listing 5.3 of Chapter 5 for example, neither the LPS hierarchy, nor the array to LPS dimension alignments, nor can the recursive partitioning of arrays be deduced without going back and forth in the abstract syntax tree. Hence, in the semantic analysis phase that part of the abstract syntax tree is extracted and transformed into a tree – a tree of coordinate systems called the *Data Partition Tree* – where any information can be discovered through a direct lookup. Partition validation also takes place during this transformation process.

Each LPS is a node in the Data Partition Tree and forms a coordinate system with number of dimensions equal to the LPS dimensions. The dimensionless placeholder *Root* LPS forms the root of the tree. An array's dimension, the function being used to partition that dimension and associated partition parameters are deposited

as a token to the aligned dimension of the LPS node. A balanced coordinate system has an equal number of tokens along each dimension and a valid partition hierarchy has all LPS coordinate systems balanced. Once the validity of the partition hierarchy has been established, the array partition tokens in hierarchically related LPS nodes are linked together to construct hierarchical partition instructions for individual arrays.

The decision of transforming of a task's *Partition Section* into a tree of nested coordinate systems is not arbitrary. This conversion and subsequent generation of hierarchical partition instructions for individual arrays enable the back-end compiler to derive an accurate association between an array element index and its actual storage location for any memory allocation scheme. This capability is crucial to realize *IT*'s support for hierarchically dividing arrays using index reordering partition functions any number of times.

6.2.2 Static Analysis

Flow Transformation

We mentioned earlier that successive passes of static analysis incrementally generate the intermediate code. In that regard, the first pass transforms the abstract syntax trees for each task in the program into a directed and recursive *flow graph*. The nodes in the graph are compute stages and compiler introduced meta-stages for flow-control (such as conditional stage executions and repeat loops) and LPS transitions. Each node in the flow graph is linked to the appropriate node in the partition hierarchy to enable easy access to LPS and data structure partition information.

A crucial aspect of the flow transformation process is to augment data distribution and transfer control stages in the flow graph for LPSes having sub-partitions. These LPSes requires that data part allocation and

LPU distribution decisions are made on-the-fly during task execution – unlike in the case of normal LPSes where those decisions are made at task inception. To avoid managing parts of data structures that are not being used during a particular transition to such an LPS in the flow graph, the control stages are augmented to the flow graph after a data access analysis on the existing stages. Thus the flow transformation is actually a two-step process.

The directed, recursive flow graph representation for intermediate code has a significance. The *Computation Section* of an IT task, despite having the appearance of a pseudo-code, is a specification of a parallel algorithm. As described in Chapter 5, the PPUs of the target hardware are responsible to execute the algorithm concurrently and resolve any memory management issues and data dependencies that occur during its parallel execution. The chosen representation, once augmented with data access and dependency information, facilitates such PPU behavior. Whether a PPU should execute a compute stage independently, or update data versions in the memory, or interact with other PPUs to resolve a data dependency, etc. is uniquely determined by the particular context the PPU is in the flow graph at that instance. Only the implementations of those operations vary depending on the back-end architecture type and LPS-PPS mapping. Therefore, the intermediate flow graph can be directly translated into a native code with pluggable platform specific implementations of different PPU roles. This finding is the foundation of the multi-component breakdown of the Composite PPU Controller in our *IT* RTE.

Data Access Analysis

Data access analysis annotates stages of the flow graph with information regarding what task-global data has been accessed where, how many versions of those data being accessed, and the nature (read/write/reduce) of a

data access. This information is needed to derive data dependencies among the stages of the flow graph during later part of the front-end compilation, and to make memory management and LPU generation decisions during the back-end compilation.

For arrays, distinction has been made between accessing data and metadata such as querying a dimension length. Some final data access validations are also done during this step, e.g., whether accessing a particular data structure or its element in a particular stage is allowed by the partition configuration of that data structure. These validations do not cover for all possible incorrect data accesses at run-time.

Data Dependency Analysis

Once the flow graph generation is complete and data access information is available, data dependency analysis kicks off. First, the dependency relationships are implanted into the flow graph as arcs connecting compute stage nodes based on nodes' data access characteristics. Then source and/or destination of those arcs are shifted to encircling meta-stages, if appropriate, to avoid having any arc connecting nodes at different recursion levels. Afterwards, redundant arcs and arcs whose data dependency is indirectly resolved by some other arcs are removed.

The dependency arcs shifting is a semantic preserving transformation whose accuracy can be proven mathematically from the structure of the recursive flow graph and the nature of execution of that graph. It is logically impossible to make recursive boundary crossing transitions among nodes of the graph without violating the algorithmic structure of the source code. Consequently, any data dependency among two graph nodes at separate recursion boundaries need to be resolved only once at the location where their ancestor nodes are at the same recursion level.

Similarly, redundant and voided dependency arcs determination has a sound logical underpinning. The transitivity of data dependency relationships is utilized in that regard. Note that the programmer's choice of LPS-PPS mapping can make even more data dependencies redundant during the back-end compilation step. For example, if a pair of LPSes using the same partitioning for a data structure have been mapped to the same PPS then all dependencies among their compute stages for that data structure become void. The front-end compiler only eliminates dependencies that are logically redundant.

Once all valid dependencies are identified, individual dependency arcs are characterized further to reveal the nature of the dependencies. Some data dependencies are just ordering dependencies among involved stages. They are annotated likewise. Dependency arcs that will require some form of synchronization or data transfer are classified into five categories: point-to-point, data gather, data scatter, broadcast, and boundary sharing. This classification is done based on the relative positions of the LPSes of the compute stages that a dependency arc connects and the nature of the data partitioning in those LPSes.

The dependency arcs are classified in this manner so that the back-end compiler can apply the synchronization/communication primitive ideal for a particular type. The capability of implementing appropriate, type-specific synchronization and communication primitives is essential for generating a native code that can compete in performance with a hand-written low-level code. Any generic implementation of those features may unduly punish a particular *IT* program.

Final Note on the Intermediate Code

Figure 6.4 depicts a simplified graphical representation of the intermediate code, or the flow graph, that highlights the dependency and ordering relationships among nodes of a 3-space task for traditional LU Factorization

with Row Pivoting. The task's computation and data partition are presented in the old *IT* syntax in Listing 6.3.

```
1 Compute:
2       ``Prepare" (Space B) {
3               do { u[j][i] = a[i][j] } for i, j in a
4               do { l[i][i] = 1 } for i in l
5       }
6       ``Select Pivot" (Space B)
7       Activate if k in u.local.dimension1.range {
8               do {    pivot = reduce(``maxEntry", u[k][j])
9               } for j in u and j >= k
10      }
11      ``Store Pivot" (Space A) {
12              p[k] = pivot
13      }
14      ``Interchange Columns" (Space B) {
15              if (k != pivot) {
16                      do {    pivot_entry = u[i][k]
17                              u[i][k] = u[i][pivot]
18                              u[i][pivot] = pivot_entry
19                      } for i in u and i >= k
20                      do {    pivot_entry = l[i][k]
21                              l[i][k] = l[i][pivot]
22                              l[i][pivot] = pivot_entry
23                      } for i in l and i < k
24              }
25      }
26      ``Update Lower" (Space B)
27      Activate if k in l.local.dimension1.range {
28              do {    l[k][j] = u[k][j] / u[k][k]
29                      l_row[j] = l[k][j]
30              } for j in l and j > k
31      }
32      ``Update Upper" (Space C) {
33              do {
34                      u[i][j] = u[i][j] - l_row[j] * u[i][k]
35              } for i, j in u and i > k and j >= k
36      }
37      Repeat: from ``Select Pivot" for k in a.dimension1.range
38 Partition(b):
39      Space A <un-partitioned> { p }
40      Space B <id> divides Space A partitions {
41              a<dim2>, u<dim1>, l<dim1>: stride ()
42              l_row: replicated
43      }
44      Space C <id> divides Space B partitions {
45              u<dim2>, l_row: block_size(b)
46      }
```

Listing 6.3: LU Factorization with Row Pivoting Task Logic

Note that the dependency arcs inside the sub-graph of the Repeat Control node (Node 2) are classified as forward or reverse to separate dependencies between a pair of nodes in a single iteration of the repeat loop

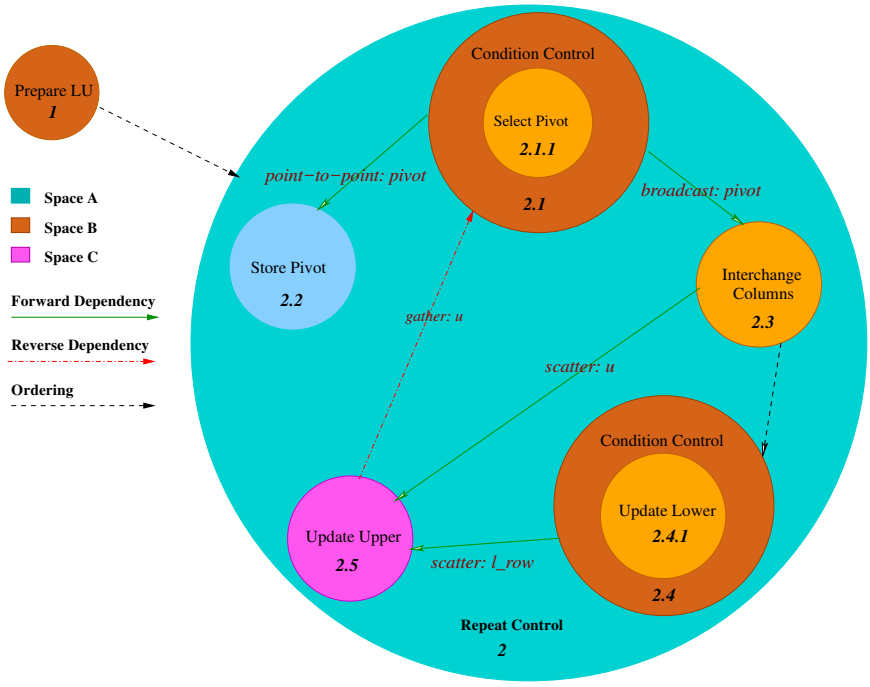


Figure 6.4: A Simplified Flow Diagram of LUF with Row Pivoting

(forward) from dependencies between a pair of nodes in adjacent iterations (reverse). The back-end compiler uses this information to avoid deadlock during the entrance and exit of a repeat loop occurring from waiting on non-existent dependencies.

A flow graph with dependency and ordering arcs as the intermediate form allows the back-end compiler to reorder the stage executions to optimize memory access, if applicable. Additionally, based on the source and destination of a data dependency arc, the compiler can overlap computation with communication to further improve run-time performance. None of those optimizations are tried in the existing back-end compilers. Regardless, we wanted to keep the front-end compiler's output the utmost flexible.

Finally, no attempt is made to combine the flow graphs of the various interacting tasks of a program during the front-end compilation as the nature of their actual interaction depends on the mapping configuration

which is unknown at this compilation step.

6.3 The Multicore Back-end Compiler

The first *IT* back-end compiler we developed generates executables that run in multicore CPUs. The *IT* compiler generates the native code as a collection of C++ source files parallelized with POSIX threads (or Pthreads) which the native C++ compiler of the target hardware compiles and links into the executable.

For the multicore back-end, our vision was to take advantage of matured shared-memory programming techniques as much as possible and then minimize the overhead of *IT* RTE implementation using carefully chosen data structures and algorithms. The expectation was that if the overhead cost of *IT* RTE operations is negligible then the runtime behavior of an *IT* executable should resemble that of an equivalent hand-written multi-threaded parallel program. We chose Pthreads as the mechanism for parallelism due to its wide applicability. Furthermore, Pthreads provides the utmost programmer control of a thread's behavior which was much needed for flexible and efficient implementations of *IT* RTE components.

6.3.1 RTE Implementation

Referring back to the run-time model of Chapter 5, the Task Executor in this case is a manager of threads. Each task execution creates a number of Pthreads as the Composite PPU Controllers and cleans them up when the task completes. The Pthreads are pinned to appropriate cores of the CPU during the lifetime of a task and share or exclusively access hardware caches according to the mapping configuration.

To elaborate, take for example a Hermes node of Figure 5.1 as the target hardware; if an LPS of a task has been

mapped to the NUMA-nodes then there will be a single thread pinned to the first core of each NUMA-node. Hence that thread will have exclusive access to all shared caches inside the NUMA-node. Threads of different NUMA-nodes, however, have to compete for the main memory access.

The flow graph from the front-compiler is converted to a C++ function and serves as the Computation Flow Template. The compute stages of the task are translated as void functions that take in an LPU and task-global scalar variables as reference parameters, and update them directly. Calls to those functions are implanted into appropriate places in the Computation Flow Template. The Flow Executor invokes a function or skips it based on the Composite PPU Controller's PPU role and availability of LPUs. Listing 6.4 shows the central part of the Computation Flow Template the compiler generates for the *Block Matrix-Matrix Multiply* task of Listing 4.1. This illustrates how the Flow Executor works.

```
1       int spaceA_SubLpuId = INVALID_ID;
2       int spaceA_SubIteration = 0;
3       SpaceA_Sub_LPU *spaceA_SubLpu = NULL;
4       LPU *lpu = NULL;
5       // iteration over LPUs
6       while((lpu = threadState->getNextLpu(Space_A_Sub, Space_Root,
7                       spaceA_SubLpuId)) != NULL) {
8               spaceA_SubLpu = (SpaceA_Sub_LPU*) lpu;
9               // role checking to decide whether to execute this stage
10              if (threadState->isValidPpu(Space_A_Sub)) {
11                      // executing user computation
12                      multiplyMatrices(spaceA_SubLpu, arrayMetadata,
13                                      taskGlobals, threadLocals, partition);
14              }
15              spaceA_SubLpuId = spaceA_SubLpu->id;
16              spaceA_SubIteration++;
17      }
```

Listing 6.4: The Main Section of the Computation Flow Template of a Block Matrix-Matrix Multiply Task

Let us now examine some important elements of the multicore implementation of the *IT* RTE.

6.3.2 Memory Model

The multicore run-time has the simplest memory model. It allocates only main memory and relies on the multicore hardware's cache control mechanism for data transfer to/from caches during LPU executions. The discussion in Chapter 5 makes it evident that programmatic control of the caches is ideal for any *IT* run-time to ensure data locality. The current strategy cannot guarantee that. Nonetheless, we adopt this strategy to save development time.

Data structures – in particular, arrays – are allocated as whole as opposed to separate parts for individual LPUs[a]. This policy greatly simplifies data structure management and inter-task interaction handling. Both the Environmental Data Structures Repository and Data Parts Registry of Figure 5.3 are associated maps that Task Executors and Composite PPU Controllers look up for data they will use. Since tasks execute one after another, the inter-task data dependency problem is gone altogether. This simplification of key *IT* RTE components goes a long way to reduce overhead computation cost in the *IT* executable.

How the notion of independent LPU execution is implemented and how data dependency resolution for a data (or data part) shared among LPUs is done under this model are critical concerns that we discuss next.

6.3.3 LPU Generation

An LPU in the multicore memory model is just a set of metadata descriptions that define what data structures and what regions of those data structures an LPU will use. Since arrays in the source program can be partitioned multiple times along the LPS hierarchy, LPU metadata descriptions for arrays are also hierarchical.

[a] Any multidimensional array in the source code is implemented as a single-dimensional array in the back-end.

For this LPU generation scheme, the LPU Generator of Figure 5.4 is implemented as a collection of library functions. This is done by investigating the coordinate space definition of each LPS in the partition hierarchy and generating functions that determine LPU counts, which part of an array should fall within a specific LPU, and so on. Based on its PPU and group IDs for the concerned LPS, a Composite PPU Controller gets the IDs of the subset of LPUs that it needs to execute. Another set of libraries is used to generate the LPU when passed an LPU ID and the total LPU count. As described in Chapter 5, no memory allocation is done for the LPU structure; rather the properties of the same instance are updated for subsequent LPUs. Thus, the LPU Generator does not need to look up a Data Parts Registry as shown in Figure 5.4. Affixing the data structure references to the sole LPU instance of an LPS during that instance's creation is sufficient.

LPU generation is a recursive process. Suppose a task has the following partition hierarchy: Space-A divides Space-B which divides Space-C. Then if a directive has been found in the Computation Flow Template to execute a compute stage in Space-A, the Flow Executor has to iterate over LPUs based on what ancestor LPS the transition has been made from to reach Space-A in that instance. If the transition has been made from Space-B then the Flow Executor should iterate only over the LPUs of Space-A. If, however, the transition has been made from Space-C then the Flow Executor should recursively go between the LPUs of Space-B to accumulate all Space-A LPUs. An efficient implementation of this recursive LPU generation process was essential for minimizing the *IT* RTE overhead. We solved this problem using a clever modification of the well-known *depth-first search (DFS)* tree traversal algorithm[34].

Since LPSes of a task form a rooted tree hierarchy in the *IT* source code, their LPUs form such a hierarchy also at run-time. It is as if the branches of the LPS tree hierarchy are replicated over and over in the LPU tree hierarchy based on the sizes of data structures and specific partition instructions applied at each tree level.

A Composite PPU Controller's LPU execution can be imagined as a process of traversing the nodes of the LPU tree, executing compute stages on the LPUs as it traverses them, and handling other management issues (e.g., dependency resolution) as needed during a transition from a tree level to the next. The difference from a straightforward DFS traversal is that a Composite PPU Controller traverses only dedicated tree nodes based on its PPU and group IDs, as mentioned before.

In our implementation, the LPU Generator is supplied with the Data Partition Tree generated by the front-end compiler that encodes both the LPS hierarchy and partition instructions in those LPSes. Each Composite PPU Controller gets its own instance of the LPU Generator that the former initializes with its PPU and group IDs and sets to a LPU traversal state at the root of the would be LPU tree hierarchy. With each LPU request from the Flow Executor of the Composite PPU Controller, the LPU Generator expands the LPU tree, adjusts the traversal state, and returns the next LPU. To minimize space overhead for this process, at all times the LPU Generator maintains a single path – not the entirety – of the LPU tree. In other words, LPU tree expansion and traversal happen simultaneously.

A check-pointing mechanism is introduced to be able to constraint the DFS traversal (and re-traversals) of the LPU tree within a sub-tree based on the context of the Computation Flow Template. A checkpoint informs the LPU Generator about the sub-tree root of the current DFS traversal. Then only LPUs within that sub-tree are returned with successive invocations of *getNextLpu*. If all those LPUs are exhausted, the LPU Generator reports that. Then the checkpoint needs to be moved to restart LPU generation from a different sub-tree. To traverse the same LPUs from the previous sub-tree, the checkpoint needs be reset but not moved.

Note that this entire logic of setting checkpoints and traversing LPUs is encapsulated within the LPU Generator. The Flow Executor just accesses the LPU Generator with appropriate parameters to specify the traversal

boundaries. Thus the *getNextLpu* invocation at Line 6 of Listing 6.4 has the second argument pointing to the LPS the transition has been made from. We found this elaborate mechanism actually contributes little to the overall run-time of the program.

6.3.4 Synchronization

Since data structures are allocated as whole, all data dependency resolution problems become synchronization problems in the multicore back-end. There is a set of synchronization primitives for each dependency arc in the flow graph. Each primitive is intended for a particular group of Composite PPU Controller threads having LPUs with overlapped/replicated data. There is a deterministic association between a thread's data inter-dependency with others and the CPU core the thread occupies. This association is the foundation of synchronization grouping.

Let us examine a simple example to understand this association. Assume that a task has three hierarchically related LPSes: Space-A, B, and C. Further assume that a single array is partitioned in Space-A, then those partitions are replicated among the LPUs of Space-B, then Space-C LPUs further divide those replicated Space-B partitions. Finally, assume that the programmer wants to run the task using a single CPU of the Hermes cluster of Figure 5.1 with a mapping configuration that maps Space-A, B, and C to NUMA-Node, Core-Pair, and Core PPS levels respectively.

Given the above configuration, if some threads executing compute stages of Space-C update the concerned array then all threads within a single NUMA-Node confinement are affected and should be notified. This is required due to the replication of the array parts in Space-B LPUs. Threads of different NUMA-Nodes do not affect one-another as the array is divided into disjoint partitions in Space-A. Therefore for this particular

example, two synchronization primitives will be needed for the two NUMA-Nodes of the CPU, each primitive will have 8 participants for the 8 Composite PPU Controller threads running on the cores of a NUMA-Node, with the threads aware of what particular primitive is intended for them.

The discovery that synchronization grouping can be done in this manner for arbitrary task partition hierarchies once the LPS-PPS mapping is known is one of the most important findings during multicore back-end compiler development. This characteristic is a result of the close correspondence of the abstract programming model of *IT* and *PCubeS* machine description.

The back-end compiler annotates the dependency arcs of the intermediate code based on a mapping dependent static analysis. Afterwards it generates library routines for creating and accessing the synchronization primitives by parsing those annotations. These synchronization primitives and their access functions form the Dependency Resolvers Registry for the multicore back-end.

The Task Executor initializes the primitives in the Dependency Resolvers Registry during a task launch. Subsequently, the Flow Executor of a Composite PPU Controller thread retrieves its synchronization primitive for a particular situation in the Computation Flow Template and issues the proper signal irrespective of other participant threads. The synchronization primitive ensures that the Flow Executors of different threads are blocked and released at proper times. In other words, threads cooperate in a synchronization without being conscious of their cooperation. The compiler generated code snippet of Listing 6.5 illustrates this mechanism in the context of the *LU Factorization with Row Pivoting* task of Figure 6.4.

The two synchronizations from Line 20 to 32 in Listing 6.5 are for the two data dependency arcs from the *Interchange Columns* and *Update Lower* stages (invoked at Line 5 and 11) to the *Update Upper* stage. Note that a dependency arc is for an LPU-LPU interaction in the flow graph of Figure 6.4 but the compiler implements

```
1       while((lpu = threadState->getNextLpu(Space_B, Space_Root, spaceBLpuId)) != NULL) {
2               spaceBLpu = (SpaceB_LPU*) lpu;
3               if (threadState->isValidPpu(Space_B)) {
4                       // invoking user computation
5                       int stage5Executed = interchange_columns(spaceBLpu,
6                                       arrayMetadata, taskGlobals, threadLocals, partition);
7                       uStage5No3 += stage5Executed;
8               }
9               if (threadState->isValidPpu(Space_B)) {
10                      // invoking user computation
11                      int stage6Executed = update_lower(spaceBLpu,
12                                      arrayMetadata, taskGlobals, threadLocals, partition);
13                      l_rowStage6No1 += stage6Executed;
14              }
15              spaceBLpuId = spaceBLpu->id;
16              spaceBIteration++;
17      }
18      } // scope exit for iterating LPUs of Space B
19
20      // resolving synchronization dependencies
21      if (uStage5No3 > 0 && threadState->isValidPpu(Space_B)) {
22              threadSync->uStage5No3DSync->signal(repeatIteration);
23              uStage5No3 = 0;
24      } else if (threadState->isValidPpu(Space_C)) {
25              threadSync->uStage5No3DSync->wait(repeatIteration);
26      }
27      if (l_rowStage6No1 > 0 && threadState->isValidPpu(Space_B)) {
28              threadSync->l_rowStage6No1DSync->signal(repeatIteration);
29              l_rowStage6No1 = 0;
30      } else if (threadState->isValidPpu(Space_C)) {
31              threadSync->l_rowStage6No1DSync->wait(repeatIteration);
32      }
```

Listing 6.5: A Code Snippet from LUF with Row Pivoting Task's Flow Template

a PPU-PPU synchronization that is derived from the arc characteristics and LPU distribution information. Implementing an LPU-LPU synchronization mechanism would be far more complicated due to the overhead related to tracking individual LPU executions.

6.3.5 Array Index Transformation

One of the most critical aspects of the back-end compilation is to associate array indices to their storage locations (and vice versa) within the functions for compute stages in the presence of array index reordering in the *Partition Section*. This is a concern regardless of the particular memory model being chosen by the back-end compiler as the hierarchical LPU metadata descriptions for arrays have to be generated by recursively applying the partition functions. Given multicore back-end compiler is the first compiler we have developed, we had to solve this issue here. The two subsequent compilers apply the same solution strategy developed for the first compiler with minor adjustments for the memory models they implement.

The index transformation problem becomes apparent if one reflects on the *IT* code samples being presented so far in this writing. For portability, array indexing within a compute stage is necessarily oblivious of the actual physical locations the indices refer to. The programmer only ensures that indices are valid for the underlying LPU given the specification provided in the *Partition Section*. Take the *Block Matrix-Matrix Multiplication* task of Listing 4.1 as an example. It is not unrealistic to have the input matrices partitioned using stride in Space-A then within the sub-partition using block-stride when that task is being used in conjunction with other tasks to solve a larger problem. This means indices in both dimensions of the matrices will be shuffled and distributed among LPUs in that manner. Nevertheless, the *Compute Section* remains the same. It will remain the same even if we add more LPSes and have multiple shuffles per dimension. It is the compiler's responsibility to transform

all index comparisons and array accesses to ensure that the correct storage address is retrieved for an index value and vice versa.

The transformation is done by associating parameterized expressions with each partition function definition. There are three such transform expressions: *one for converting an original array index to its transformed index within an LPU part, another for doing the reverse translation, and a final expression for determining if an arbitrary integer lies within the transformed index range of an array part*. Different expressions are needed in different statement contexts the concerned array and its indices are being used within a compute stage. During code generation, the compiler replaces parameters with appropriate arguments in a transform expression, combines transform expressions for different dimensions, and puts the resulting expression as a replacement for direct index accesses based on what is appropriate for a particular scenario. If a particular array dimension has been reordered multiple times along the LPS hierarchy then the expression for that dimension is generated by repeatedly transforming the transform expression. This is done using an expression stack where each popped expression replaces one or more parameters in the subsequent expression.

The mathematics of transforming and reverse transforming an array element index is founded on the logic of function composition[fnC]. If $f: X \to Y$ and $g: Y \to Z$ are two functions then their composition $g \circ f: X \to Z$ is a function for all $x \in X$ with the property $g \circ f = g(f(x))$. Each partition function provides its own associated functions for the aforementioned three types of index transformations that may be needed in an *IT* executable. For a hierarchical partition, appropriate associated functions can be applied one after another in a compositional manner.

The problem with straightforward function applications for an index transformation is that function invocations are costly. In particular, array index transformations often happen inside deeply nested loops whose

execution time can be affected drastically if burdened with unwanted function calls. The novelty of our index transformation technique is that it involves no function calls. This feat is achieved using the aforementioned parameterized expressions for the transformation functions.

Note that an array index transformation does not result in a compile-time static expression. This is because the array/storage index being transformed as well as the arguments for the underlying partition functions are all run-time variables. Rather the final expression being generated by the compiler as a replacement for a straightforward index access in the source code is itself a parameterized expression. The parameters of that final expression are, however, available as local variables in the context the expression is evaluated. Consequently, at run-time the expression is always evaluated to proper integer value.

Depending on the complexity of the partition functions and the depth of the partition hierarchy, an index transformation may result in a lot of integer operations. Integer operations are, however, cheap compared to floating point arithmetics that dominate high performance parallel computing. Furthermore, the native compiler's optimizer should detect and eliminate any redundancy in the transformation expressions at the third compilation step. Nonetheless, measures have been taken in the back-end compiler to hoist index transformation expressions to appropriate loop level to avoid redundant computations.

6.4 The Segmented Memory Back-end Compiler

The second back-end compiler, called the Segmented Memory *IT* Compiler, generates executables for compute clusters or supercomputers composed of multicore CPUs. Since we already had an *IT* compiler for multicore CPUs, the plan was to build the new compiler on top of that. Then aspects of the *IT* run-time that involves

cross CPU interactions will be handled by an added part and aspects of the *IT* run-time local to individual CPUs will be delegated to the existing multicore compiler.

This strategy provides a communicating processes model for a segmented memory executable where each participant process is a multi-threaded shared memory program that communicates with other processes by sending/receiving explicit messages. There is a justification for this particular design choice. The conventional wisdom is shared memory programming techniques are ideal for shared memory environments as distributed memory programming techniques are for distributed memory systems. Consequently, hardware architectures that connect shared memory component pieces through a distributed network should be programmed using a hybrid as opposed to a uniform programming technique.

The chosen strategy has one problem: the multicore compiler's memory model that allocates and manages data structures as a whole is inapplicable in the new environment where CPUs are expected to hold only partial data relevant to local LPU computations. Hence we changed the memory model of the multicore compiler into a new model where individual data parts are allocated separately. This change invalidates some parts of previous LPU generation scheme. So the implementation of the LPU Generator and its interaction mechanism with the Data Parts Registry are updated accordingly. Apart from those, the multicore compiler remains mostly unchanged and the larger segmented memory compiler gets the feature breakdown of Figure 6.5.

The segmented memory compiler generates C++ MPI code to implement the added part of the run-time and the multicore part is still C++ parallelized with Pthreads. MPI has been chosen for the added part as it is the standard for distributed memory programming for decades. The native code in this scenario is a collection of MPI + Pthreads hybrid C++ source files that are compiled and linked into the executable with whatever MPI C++ compiler is installed in the target machine. Given two standard parallelization techniques have been

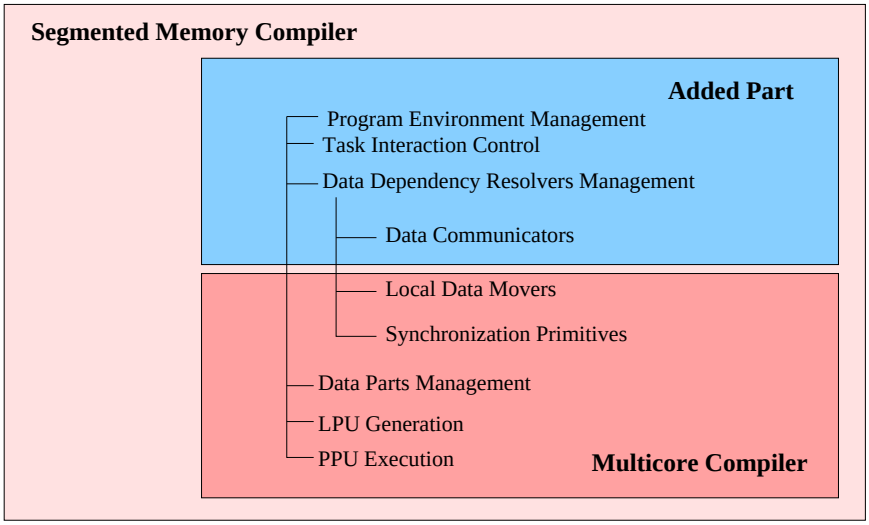


Figure 6.5: Breakdown of the Segmented Memory Back-end Compiler

used, *IT* segmented memory executables can run in virtually all contemporary supercomputers and compute clusters.

6.4.1 RTE Implementation

Referring back to the run-time model of Chapter 5, the Program Coordinator Module in this case is an MPI process running on each cluster/supercomputer node. Accordingly, Environment Managers of different processes use MPI communications to do any data reshuffling needed before a task launch.

The Task Executor in the segmented memory back-end is very similar to that of the multicore back-end with one addition: the new Task Executor needs to set up Data Dependency Resolvers for dependencies that cannot be resolved using synchronization primitives.

Dependencies that involve inter-process communications have resolvers implemented over MPI. The MPI communication within a resolver is tailored based on the nature of the dependency. That is, different MPI

collective and point-to-point communication primitives are used for different kinds of dependencies. Dependencies that involve no communication still use the previous synchronization primitives.

There is one more possibility: some dependencies may involve no inter-process communications but still need to exchange elements among data parts local to individual MPI processes. This situation may arise as data structures are not allocated as a whole anymore; therefore, overlapped data parts need to be synchronized by explicitly copying elements from one to another. This form of dependencies are handled with a third kind of Data Dependency Resolvers. The Flow Executor is oblivious of the difference among different resolver types. It uses them from the Data Dependency Resolvers Registry without knowing the underlying mechanism of dependency resolution.

A Composite PPU Controller thread's behavior does not change at all from the previous case. It still executes the Computation Flow Template, checks its PPU Role Assignments to determine what to execute and what not to, retrieves new LPUs from the LPU Generator, and issues signals on designated resolvers. The only addition is that the Task Executor also assigns higher level PPU IDs to the Composite PPU Controller based on the position of the underlying CPU core in the overall network. Hence, compute stages of LPSes mapped above the CPU/node level PPS are only executed within selective MPI processes.

To summarize, the segmented memory back-end compiler is architected in a way so that the MPI features can be peeled off from it, leaving a new multicore compiler implementation with a different memory model. This particular design has the further advantage that the communication infrastructure can be switched from MPI to something else, if deemed beneficial, without affecting the rest of the compiler.

Subsequent sections describe the major feature changes and additions in the run-time model introduced by the segmented memory compiler.

6.4.2 Memory Model

The new memory model allocates memory only for data parts of LPUs that execute locally. As LPUs and data parts within them are hierarchically defined, it is important to make allocation decisions prudently to avoid data redundancy and performance overhead during LPU execution. The current solution makes allocation decisions based on a data access analysis done on the intermediate flow graph generated by the front end compiler. A data structure within an LPS is tagged as needing parts allocation by the analysis if the following conditions hold.

1. The data structure is accessed in some compute stage that executes in the current LPS.
2. The data structure has not already been tagged for allocation in some ancestor LPS of the current LPS.
3. If (2) is false, the data structure has been accessed multiple times in stages of the current LPS and has been reordered by some partition function since the last time it has been tagged for allocation.

These conditions eliminate data parts allocations for LPSes whose LPUs are just bounding box for descendant LPUs. Furthermore, single access scenarios that are unlikely to be benefited from new parts allocation are also skipped. For example, according to the above conditions, the upper triangular matrix, *u*, of the *LU Factorization* task of Listing 6.3 will be allocated for Space-B LPUs but not for Space-C LPUs despite both LPSes have stages that access *u*. Once all allocation tags are placed properly, the compiler generates library functions for data parts allocation.

Immediately after its launch, the Task Executor invokes these library functions to generate the data parts needed for the LPUs to be executed within the current MPI process and populates the Data Parts Registry. Note that there are many-to-one relationships from LPU IDs to data part IDs because of the possibility of replicated data parts along some LPS dimensions. Thus given an LPU ID, the IDs of the various data parts the

LPU needs can be determined. Furthermore, as the hierarchical partition configurations for individual data structures are known, the dimension ranges of array data parts can be calculated from their part IDs. Thus the parts can be allocated properly. This process, unfortunately, requires generating IDs of all LPUs of the current MPI process before any computation starts. To avoid holding LPU IDs – which can be numerous – the library function processes LPU IDs one by one and generates any new data part identified.

This scheme of managing a collection of unique parts for individual data structures has two important consequences. First, there is no duplication of data. If multiple Composite PPU Controller threads within a single MPI process access a common data part, any update to that data part can be synchronized among those threads using the previous thread synchronization technique – no data movement is involved. Second, the memory consumption per MPI process for data parts is exactly necessary and sufficient for all LPU computations the MPI process is responsible for. Hence, the scheme is space optimal and has the best scaling characteristics.

As *PCubeS + IT* programming paradigm encourages a programmer to partition arrays to fit in hardware caches, the data parts can be quite small and their number for large input sizes can be considerably big. So an efficient mechanism is needed to identify an LPU data part in the Data Parts Registry. This is done by storing the parts of an array into an ordered tree hierarchy called the *Part Container Tree*. Each level in the tree corresponds to a particular LPS and array dimension combination. The nodes of that level are ordered based on the entry in a part's ID for that combination. The leaves of the tree hold the data parts. Thus a data part can be retrieved in logarithmic time given its ID. Figure 6.6 illustrates the Part Container Tree search procedure for the *Block Matrix-Matrix Multiplication* task of Listing 4.1.

Note that data part search in the registry is logarithmic in the worst case. A further optimization is made based on the observation that most of the time data parts needed for successive LPUs of a PPU lie next to each

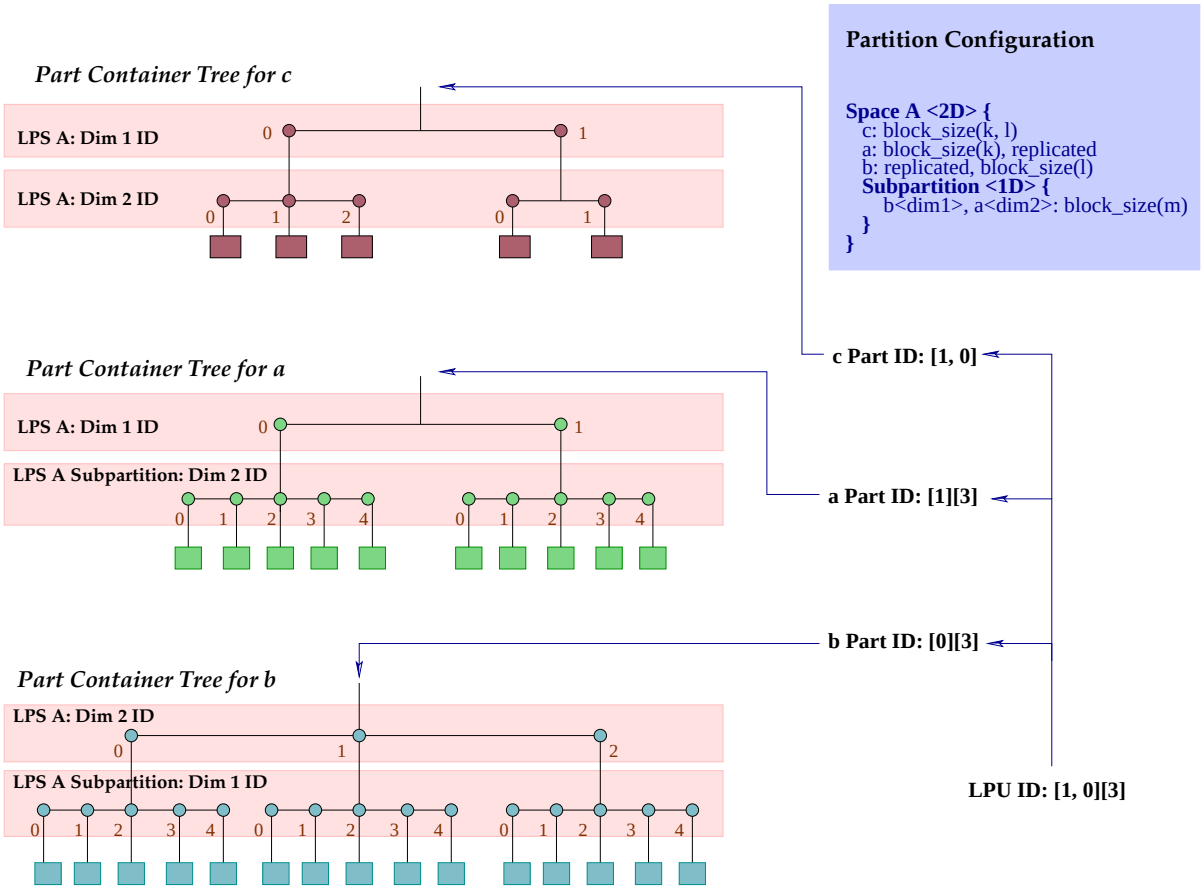


Figure 6.6: The Process of Locating Data Parts in Part Container Trees using the LPU ID

other in the part container tree or are the same. This is a consequence of the particular policy the run-time applies for multiplexing LPUs to PPUs. The LPU Generator takes advantage of this by maintaining stateful iterators for the Part Container Trees of the Data Parts Registry. During an LPU construction process, the iterators are investigated first to check if a particular data part is at the current or next location of an iterator. The Part Container Tree is searched only if that checking fails [a]. This strategy makes LPU generation process a constant time operation as it was in the previous compiler.

6.4.3 Communication

The most difficult aspect of the segmented memory back-end compiler was to solve data communication among MPI processes for dependencies involving partitioned arrays [b]. The problem asks for a generic solution that is accurate regardless of the partition configuration of the communicated data structure, and the solution's efficiency was critical for the paradigm's success in the target architecture. Algorithm whose complexity is even logarithmic in the problem size or the number of LPUs will negatively impact scalability. The solution we implemented is quite sophisticated and involves many steps. Regardless, below we provide an overview in case the reader might be interested to know how it works.

Algorithmic Foundation

The central idea behind the solution is to construct precise mathematical descriptions in terms of multidimensional interval functions (as arrays can be multidimensional) of the data content of each MPI process for both

[a] A search also moves the iterator to the searched position.

[b] Proper communication mechanism for scalar data structures and non-partitioned collections can be easily determined from the information carried by the dependency arcs.

ends of a data dependency arc. An MPI process has to communicate an update to another process if the former's sender-end data description has a non-empty intersection with the letter's receiver-end data description and the intersection function precisely describes the data that needs to be communicated. Given a pair of MPI processes i and j, the data i has to send to j for a dependency can be described mathematically as follows.

$$desc(send_{i,j}) = \left(\bigcup_{part_{updater\_LPS} \in i} desc(part_{updater\_LPS}) \right) \cap \left(\bigcup_{part_{receiver\_LPS} \in j} desc(part_{receiver\_LPS}) \right) \tag{6.1}$$

There are several questions associated with this solution strategy. Among them the two mathematical questions are: how to construct a multidimensional interval description of an MPI process's data content for an LPS, and how to compute the intersection between two arbitrary interval descriptions.

Note that the partition functions are nothing but mathematical instructions for dividing an array dimension given its index range. So an interval description is straightforward to construct for a single data part of an isolated LPS. Problems arise when there are more than one LPS in the hierarchy leading to the LPS relevant to the data dependency. This is because each ancestor LPS along the hierarchy can partition every dimension of the concerned array. In that regard, we invented an algorithm that can create a precise mathematical description of a hierarchically partitioned array by recursively applying mathematical transformations over an original periodic interval description in a bottom up fashion.

Any partition instruction p can be mathematically described as an interval function $f_p < \beta, \varrho, \delta, \eta >: I \rightarrow B$ mapping the integer range I to boolean values B indicating if a particular point in the integer domain is included or not included. The included points form the index range of p. In this formation β is the starting point, ϱ is

the period, δ is the number of contiguous points included within each periodic interval, and η is the number of times the periodic interval repeats. Then letting p to divide the index range of a higher level partition instruction q having interval function f_q is mathematically equivalent to transforming the domain of f_p with the range of f_q. This transformation gives a set of interval functions $\sum_{i=0}^{N} f_i < \beta_i, \varrho_i, \delta_i, \eta_i >$ representing the hierarchical partition instruction composed of p and q. If q itself divides the index range of an even higher level partition instruction r then the domain of individual functions f_i can be transformed to get the final interval description. This is the logical foundation of our recursive interval description generation algorithm.

This algorithm provides a way to generate an interval description for the total data content for an LPS. The union of all data part descriptions (used in Equation 6.1) is a valid description for the MPI process's data content. The problem, however, is that the data parts can be numerous. So the aggregate description itself can become unwieldy and computing all the individual interval descriptions can be prohibitively expensive too.

To circumvent both problems, we enhance the recursive interval description generation algorithm to work over a compact string representation of the entire Part Container Tree for an LPS instead of over individual data parts the tree holds. We call the algorithm for constructing the string representation from a Part Container Tree the *Folding Algorithm* [a]. Although the details of the Folding Algorithm are skipped here, it is important to understand the rationale behind it which is again founded on the properties of periodic interval functions.

Assume that the index range of a 1D array a has been divided applying the partition function p that produces the set of periodic interval functions $\sum_{i=0}^{N} f_i < \beta_i, \varrho, \delta_i, \eta >$ for consecutive data parts a_i. Then if an MPI process contains all the parts from a_j to a_{j+n}, the data content of the process can be described by the single inter-

[a] The algorithm is called a folding algorithm as it progresses as a recursive process of combining part IDs horizontally within a level then vertically across levels.

val function $< \beta_j, \varrho, \delta_j + \cdots + \delta_{j+n}, \eta >$ [a]. For a D dimensional array b being divided by multiple partition functions $p_1 \cdots p_D$ scenario, a single interval function description can be generated only if the underlying data parts have a common interval function description in all but a single dimension. The Folding Algorithm applies this principle to combine data part IDs and form the most compact representation of the Part Container Tree as a hierarchical part ID range description. Then the recursive interval description generation algorithm converts the part ID range into a functional form.

The compactness of the output of the Folding Algorithm impacts the efficiency of all subsequent algorithmic and implementation-specific aspects of inter-process communication. Importantly, the compactness or the lack of it is a property of the LPU distribution scheme. Thus *IT*'s systematic LPU multiplexing to PPUs and the efficiency of communication go hand-in-hand.

Along each dimension, the multidimensional interval description is a collection of periodic interval functions that tell us what indices are included and what are not. We calculate the intersection of two such interval descriptions along any dimension by calculating intersections of those periodic interval functions. For this we invented an algorithm that calculates the intersection of two periodic interval functions with arbitrary but finite number of interval occurrences in time logarithmic of the periods. The algorithm compares LCM of the period lengths number of interval occurrences in the worst case; hence the logarithmic time complexity. If the intersection along any dimension is empty then the two multidimensional interval descriptions do not intersect. To the best of our knowledge, both the algorithm for interval description generation and the algorithm for calculating their intersections are novel discovery of this research [b].

[a] Some adjustment is needed in this logic when p does not evenly divide the index range of a.

[b] We have not published those two algorithms though. The source codes for the algorithms can be found in our github project for the compilers in src/utils/interval.cpp and src/partition-lib/partition.cpp files within the segmented memory

Optimizations for a Practical Implementation

Once the mathematical groundwork is done, we need to address the practical concerns. How does a process determine what other processes have for the two ends of the data dependency so that it can determine whom to send local updates to and where to receive remote updates from? One possibility is to let all processes exchange their interval descriptions among one another. That strategy will be costly for a large number of PPUs and can be wasteful too as most processes may have to communicate with only a few other processes.

We tackle this problem by constructing a complete *Distribution Tree* of data part IDs of all MPI processes in each process then generating interval descriptions for relevant other processes from that Distribution Tree. Distribution Tree construction involves no communication. The Task Executor just pretends that it has the PPUs of other MPI processes, generates part IDs from those PPUs' LPU IDs, and then stores the IDs in the Distribution Tree. One Distribution Tree is made for every array that has some dependency arcs requiring communication in the intermediate code. This is a one time cost during program execution. The time and space complexities of Distribution Tree construction can be a scalability problem in a large network, in particular, when the data parts are numerous. This is an issue we will address in the future.

The key to keep the cost of inter-process data exchange requirement calculation manageable is to let each MPI process search only the locations in the Distribution Tree relevant to it. This is done by investigating the Part Container Tree and identifying where in the Distribution Tree the current MPI process's data parts fit into. We call this identification process *Confinement Construction*.

A confinement is basically a sub-tree of the Distribution Tree; and depending on the nature of the data

compiler directory.

partitions, there might be multiple confinements relevant to the current MPI process. The process can have data parts overlapping with only those other MPI processes that have some data parts in the confinements the former participates into. We skip the mathematical derivation here, but the rationale for confinement-based data overlapping calculation follows from the same rationale of synchronization grouping in the multicore back-end compiler discussed earlier. Once all confinements are identified, compact string representations of the confinement sub-trees are made for each participating process using the previous Folding Algorithm, an interval description is generated for the string representation, and current process computes its data intersection with the other process. At the end, each MPI process knows exactly what to send and receive and to/from where without any communication.

The Mechanism of Communication

Once the data content of communications are determined then the mechanism for transferring the data needs to be implemented. Only at this step does MPI become a part of the solution. MPI has a wide range of primitives for different types of collective and point-to-point communications. Therefore, the front-end compiler's classification of dependency arcs based on different kinds of data transfer needs comes in handy at this step. In fact, there is a one-to-one correspondence between *IT* dependency arc types and specific MPI communication primitives. The back-end compiler takes full advantage of this association by using the most appropriate primitive for a particular dependency arc type.

Since elements of the data to be communicated may come from many different data parts and moreover from non-adjacent memory addresses within those parts, a separate communication buffer is needed to collect those elements together before an MPI communication takes place. In that regard, proper communication

buffers for holding data items coming from different data parts are created. In addition, an efficient mechanism for elements exchange between data parts in the Part Container Trees and the communication buffers is implemented that is linear to the length of the buffers.

Groups of interacting processes get their own MPI Communicator resource if it is deemed that a collective communication primitive is ideal for the particular dependency and group. Some MPI collective communications such as *MPI_Gatherv* and *MPI_AllReduce* require buffer aggregation. A communication buffer aggregation mechanism is implemented for those. All these features are wrapped inside a resolver that is placed in the Data Dependency Resolvers Registry.

During task execution time, a Flow Executor only retrieves the resolver from the registry and issues a proper signal on the resolver. The resolver ensures that it has received all send/receive signals it is supposed to receive form Flow Executors local to the MPI process before proceeding any further. Once all signals have been received the resolver copies data to be sent from data parts to communication buffers, performs MPI communications, copies any data received from the communication buffers to the data parts, and then releases the blocked Flow Executors.

It is important to recognize that the sophisticated resolver setup process takes place only once per task execution, before the Composite PPU Controller threads start running. The overhead for subsequent use of the resolver by the Flow Executors is not significantly higher than direct MPI communications programmers typically implement in their MPI programs.

6.4.4 Program Environment Management

The Environmental Data Structures Repository holds a data structure in the format the completed task used it during its execution, i.e., as an ordered list of data parts accessible through a Part Container Tree. Along with the parts list and the Part Container Tree, the hierarchical partition configuration that led to those data parts are stored as a metadata. The data parts are kept in this manner to leverage the existing technique of data communicating resolvers for Environment Managers' interactions during any cross-process data shuffling before a task launch.

There is one exception in the communication scheme for Environment Managers though. Since the Environment Managers do not maintain Part Distribution Trees, they have to exchange their data descriptions with one another instead of computing them locally. This change is made as the set of MPI processes that participate in a task execution may be different for different tasks; consequently, some MPI processes responsible for the upcoming task may have no information about the current locations of up-to-date data parts. Furthermore, once a task completes, a participating MPI process's knowledge about the data contents of the rests does not necessarily hold true anymore. The take-home message is, in the current implementation, a data shuffling during task transitions is costlier than data dependency resolution during task execution.

Another important responsibility of the Environment Manager is to keep track of the freshness of data parts lists. This issue occurs because if an upcoming task requires a different parts list arrangement for an existing array in the environment, we create a new parts list for the new task and retain the old parts list as it is. So depending on the nature of task partitions, multiple parts list versions of a single array may exist in the environment – some of those versions may be fresh and some others stale. This strategy is adopted as *IT* allows

an entire task to operate on only a small part of a larger array created by other task. Hence the execution of the former can only modify a fraction of the data parts stored for the letter. If a transition is subsequently made to the earlier task then those stale data parts are updated only [a].

Referring back to the *IT* RTE of Chapter 5, a newly instantiated Task Executor populates the Data Parts Registry with both environmental and non-environmental data structures. For environmental data structures, the Data Parts Registry acts only as an access provider. The Composite PPU Controllers of the task subsequently access all data from the Data Parts Registry and are oblivious of the data type differences. We implement a mechanism to support a seamless integration of environmental data owned by the Environment Manager and non-environmental data owned by the Data Parts Registry to realize this semantics.

The Program Coordinator Module of the MPI process uses the Environment Manager only to register environmental instructions with the Task Executor before launching it. Later when the Task Executor initializes the Data Parts Registry, these instructions become activated one after another and a transfer of control happens from the Data Parts Registry to the Environment Manager for each instruction activation. The Environment Manager then uses a proper handler to execute that instruction and returns the control. This technique allows the Environment Manager and Data Parts Registry to operate on the same data parts references in the memory for environmental data structures and eliminates any data transfer between those two RTE components.

Finally, since partitioned arrays are maintained as independent parts list versions, the reference count based garbage collection also treats those versions separately. For scalars and other non-partitioned data structures there is still just one copy.

[a] With minor changes and a few language enhancements, this facility can be used to solve many iterative refinement problems where some regions of an object are refined more often and in finer grain than the others due to differences in their complexity or composition.

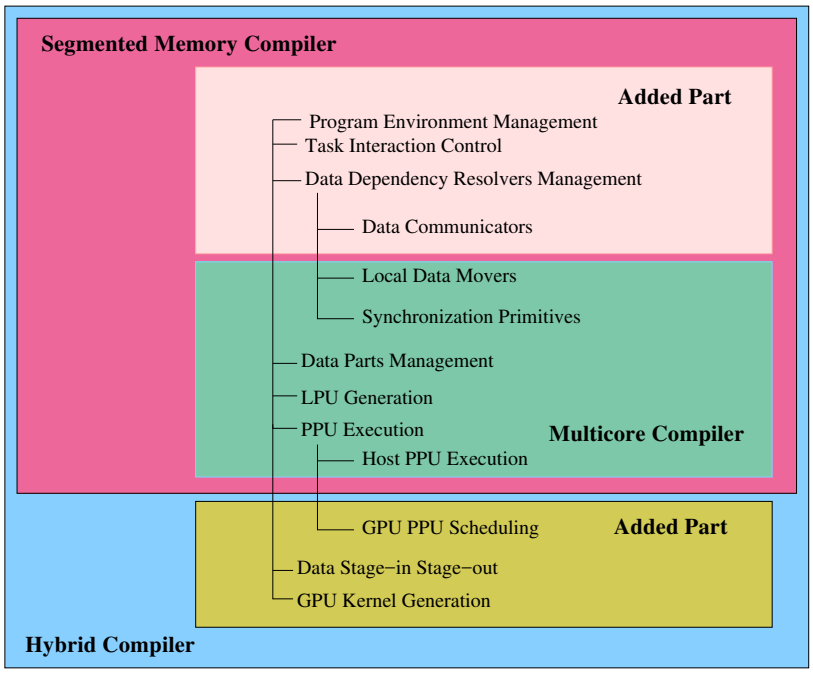


Figure 6.7: Breakdown of the Hybrid Back-end Compiler

6.5 The Hybrid Back-end Compiler

The third *IT* back-end compiler, called the Hybrid *IT* Compiler, generates executables for compute clusters and supercomputers that have both multicore CPUs and NVIDIA GPUs as compute nodes. As the segmented memory compiler is constructed by augmenting new features to the multicore compiler, the hybrid compiler is constructed by augmenting features to the segmented memory compiler. There is a difference in the nature of the addition though. The news features of the segmented memory compiler work as a canopy over the features of the multicore compiler, but the new features of the hybrid compiler work more like extensions to the multicore compiler features. This is so because the PPUs of the GPU in a hybrid machine reside below the PPUs of the multicore host. Figure 6.7 depicts the feature breakdown of the hybrid compiler.

The compiler generates CUDA code for the parts of the *IT* source code to be executed inside the GPU. So the overall native program is a hybrid of MPI, Pthreads, and CUDA C++ source files. The CUDA files are compiled and linked together using the NVCC compiler, the MPI + Pthreads files are compiled with the MPI C++ compiler, then the MPI compiler links all object files to generate the executable. We believe this is the best strategy for generating executables for a hybrid architecture as the strategy effectively combines the programming tools ideal for the individual architectural components. Accordingly, we have the utmost optimization opportunities for future performance improvements. Furthermore, the use of standard tools at each level makes the *IT* executable portable across all hybrid supercomputers or compute clusters composed of multicore CPUs and NVIDIA GPUs.

As illustrated in the Titan Supercomputer case study of Chapter 3, a hybrid machine is modeled by *PCubeS* in three different ways. The first model, called the *host-only* model, ignores the GPU and portrays machine's multicore capacities only; the second model, called the *GPU model*, portrays the GPU as to be controlled by a host CPU as a whole; and the last model, called the *fragmented GPU model*, gives each CPU core access to a portion of the GPU capacity. The current compiler supports the first two *PCubeS* models only [a]. When mapping the tasks of his/her program, the programmer can use different models for different tasks. Note that even in the GPU model, some parts of a task may execute in the CPU hosts if the corresponding LPSes are mapped to PPSes above the first GPU PPS in the model. A host CPU merely behaves as a single-core hardware when executing those parts.

[a] The current compiler supports a single GPU per node. Extending it to support multiple GPUs per node should not be difficult.

A Short Note on the RTE Implementation

Referring back to the run-time model of Chapter 5, in the context of the hybrid compiler, all the machineries for program coordination, environment management, and data dependency resolution remain the same as they were in the segmented memory back-end compiler. If a task is mapped to the host-only model then the machineries for task execution remain the same also. If a task is mapped to the GPU model, however, then a new type of task executor is launched that can do both CPU and GPU computations. We call this task executor *The Hybrid Task Executor*. It still uses the mechanism implemented for a regular task executor for host-side computations but it has the additional capacity to integrate host LPUs with GPU LPUs and it has a very different mechanism for executing LPUs inside the GPU. To understand the source of the difference some background in CUDA programming is needed. The next section provides a brief overview of the CUDA programming model before we delve into the details of the Hybrid Task Executor. We also encourage the reader to review the *PCubeS* description of a single GPU from Chapter 3 case study on the Titan Supercomputer.

6.5.1 The CUDA Programming Model

In a typical CUDA program, the host CPU copies data into the GPU card memory then schedules one or more computations inside the GPU, called CUDA kernels, to be done over the copied data. After the kernels finish executing, the host copies the updated data from the GPU card memory back to the host memory. The paradigm also allows host memory pinning which enables the GPU threads to directly access the host memory during a kernel computation, but that is an inefficient way of using the GPU and generally not recommended.

A CUDA kernel executes in SPMD (Single Program Multiple Data) fashion in the threads of GPU's Sym-

metric Multiprocessors (SM). Each SM executes a programmable number of warps within it. The number of SMs to be used by a launched kernel and the number of warps to run within them are launching parameters. A warp is a group of 32 lockstep threads. The threads of the warp cannot diverge and must all be computing the same instruction in the same function at a time in a SIMD fashion. Different warps in an SM may compute different functions.

The GPU card memory is accessible from all SMs and computations can be directly done on that, but it is at the range of 100 times slower than the small 64 KB shared memory or L1 cache (it can be used in either way) per SM. The standard approach to CUDA programming is to programmatic-ally load data from the card memory to the shared memory or to access data from the former in a way to ensure that the data is cached in the SM for the most part of the warps execution.

Just caching data in the shared memory is not enough for good data access performance. An SM's memory is divided into memory banks and multiple banks can be accessed simultaneously but not multiple addresses in the same bank. Consequently, depending on the data access patterns of the threads of a warp only 1 data item as opposed to 32 items may be read at a time. Thus reducing memory bank conflict is essential for good performance.

A further limitation of the shared memory is that it does not allow dynamic memory allocation – only constant length arrays and scalar variables are supported. The only option that comes close to a dynamic memory is a single extern per-SM memory block whose size is determined at the kernel launching time. This single block of shared memory can be used as a programmer controlled cache where spaces for different dynamic variables can be allocated as needed.

Intra-SM warps synchronization is possible using a *__syncthreads()* primitive. Cross-SM synchronization

is, however, not supported. That is, if the same card address is accessed and/or modified from warps of two different SMs, the behavior of the code is undefined.

Apart from not supporting thread divergence within warps, recursive function calls from threads are either not supported (older GPU cards) or inefficient (versions starting from Tesla K-20). Newer GPUs support direct cross-GPU data exchanges, but that is in their white paper, and to the best of our knowledge that feature is not popular in most GPU applications.

Note that at the end of a kernel execution, updates made by all threads in all warps from all SMs get written to the card memory and the shared memory (or cache depending on the use case) gets cleared for the next kernel launch. In other words, shared memory variables do not persist across kernel launches.

6.5.2 The Hybrid Task Executor

Given the prohibitive cost of accessing the host CPU memory from GPU threads, the previous scheme of the Task Executor creating a fixed number of Composite PPU Controllers that execute LPUs multiplexed to PPUs of different PPSes is no longer an option. Any computation to be done in an LPS mapped to a PPS above the GPU level is thereby kept separate and designated to a *Host PPU Controller* thread. When a transition is made from an LPS mapped to host CPU or above to an LPS mapped to one of the three PPSes [a] of the GPU, the following things happen.

1. The Task Executor accumulates data parts for the LPUs multiplexed to the PPUs of upcoming GPU PPS and copies those data parts into the GPU card memory.
2. Then it launches one or more CUDA kernels that emulate a Composite PPU Controller's LPU execution logic inside the GPU SMs/warps for LPSes mapped inside the GPU.

[a] These PPSes are GPU, SM, and Warp in that order in the *PCubeS* description.

3. After all the launched kernels ran to their completion, the Task Executor copies any updated data parts back to the host CPU memory and removes all data from GPU card memory.

Then the Task Executor switches back to the Host PPU Controller thread. A location in the Computation Flow Template that causes such switchings between host and GPU execution modes is called a *GPU Offloading Context*. Since there is only one Host PPU Controller thread in this scheme, the Task Executor directly runs the logic of that Host PPU Controller instead of delegating responsibilities to a separate thread.

There are two interrelated problems associated with copying LPU data parts into GPU card memory: a host to GPU data part transfer is costly so should be done in bulk, but the capacity of the GPU card memory is usually significantly smaller than that of the host memory so LPU data parts may not fit in the GPU all at once. To balance between these two opposing problems, the Task Executor maintains a host-side data buffer per data structure needed by the GPU kernels. It accumulates contents from LPU data parts into those buffers until adding more LPU data oversubscribes the GPU memory or there are no LPUs left. Then it copies individual data buffers as a whole into the card memory.

Each data buffer has some associated metadata buffers that are also copied into the card memory. GPU SMs/warps access those metadata buffers to determine the location of a particular data part's content in the data buffer. At the end, the data copy-in/kernel execution/data copy-out sequence described previously may actually repeat for a number of times at a GPU Offloading Context to account for all LPUs before the Task Executor can switch back to the Host PPU Controller mode [a].

If the LPUs have overlapping data parts or share some data then data dependencies may arise among GPUs of different MPI processes within the current part of the flow template that is intended for GPU execution.

[a] A possible optimization is to pipeline the data transfers and overlap GPU execution with data movements.

Any inter-process communication, however, needs to be mediated by the hosts using the Data Dependency Resolvers Registry mechanism described before. So the intermediate flow graph generated by the front-end compiler cannot be directly translated into the Computation Flow Template in the hybrid back-end. Instead, the flow graph needs to be modified first so that any sub-graph intended for GPU execution is broken down into separate GPU Offloading Contexts with intervening transitions back to the host if the original sub-graph execution introduces cross-process dependencies.

Flow graph breakdown for identifying all GPU Offloading Contexts and generating CUDA kernels for them were the two most critical problems in the hybrid back-end compiler. Subsequent sections discuss their current solutions.

6.5.3 Identifying GPU Offloading Contexts

To understand how the intermediate flow graph is modified for GPU execution and how GPU kernel boundaries are set within it, assume that an arbitrary task has the computation flow of Listing 6.6 and has been mapped to the GPU model of a hybrid back-end as shown in Figure 6.8. Further assume that a particular array, *m*, has been accessed in all the stages of the computation flow and all other arguments to the stages are unique for each stage.

Given the mapping of Figure 6.8, it is clear that sub-flow from Line 4 to 15 of Listing 6.6 should execute inside the GPU and the rest of the computation should be done in the host. Whether there should be just one GPU Offloading Context for the sub-flow of Line 4 to 15 or more depends first on how *m* has been partitioned in Space-B. If the partitions of *m* are disjoint in Space-B then a modification done on *m* within any stage between Line 4 to 15 only affects the individual GPUs doing the computation. So there will be just one GPU Offloading

```
1 Computation:
2         Space A {
3                 stage1(m, ...)
4                 Space B {
5                         stage2(m, ...)
6                         Space C {
7                                 stage3(m, ...)
8                                 Space D {
9                                         stage4(m, ...)
10                                        stage5(m, ...)
11                                }
12                                stage6(m, ...)
13                        }
14                        stage7(m, ...)
15                }
16                Space E {
17                        stage8(m, ...)
18                }
19        }
```

Listing 6.6: Computation Flow of an Imaginary Task

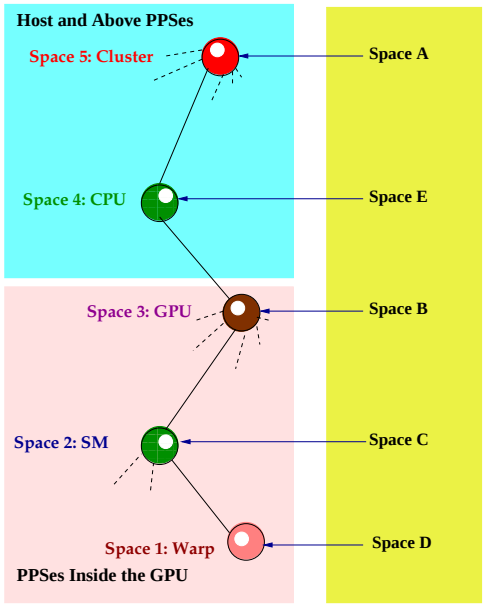


Figure 6.8: Mapping LPSes of a Task to a Hybrid Machine

Context. If, however, *m* has overlapping partitions in Space-B then the number of GPU Offloading Contexts depends next on how *m* has been accessed within the sub-flow. Consider two cases for example, first *stage1* modifies *m* and all other stages use *m* as read-only, and second *stage1* and some stages within Line 6 to 13 modify *m*. In the first case there will be two and in the second case will be 3 GPU Offloading Contexts. So after modification, flow graph for the two cases should appear as if it has been generated for the following two computation flows of Listing 6.7 and 6.8 instead.

```
1 Computation:
2       Space A {
3               stage1(m, ...)
4               Space B { // GPU offloading context 1
5                       stage2(m, ...)
6               }
7               Space B { // GPU offloading context 2
8                       Space C {
9                               stage3(m, ...)
10                              Space D {
11                                      stage4(m, ...)
12                                      stage5(m, ...)
13                              }
14                              stage6(m, ...)
15                      }
16                      stage7(m, ...)
17              }
18              Space E {
19                      stage8(m, ...)
20              }
21      }
```

Listing 6.7: Case 1 Transformation

```
1 Computation:
2       Space A {
3               stage1(m, ...)
4               Space B { // GPU offloading context 1
5                       stage2(m, ...)
6               }
7               Space B { // GPU offloading context 2
8                       Space C {
9                               stage3(m, ...)
10                              Space D {
11                                      stage4(m, ...)
12                                      stage5(m, ...)
13                              }
14                              stage6(m, ...)
15                      }
16              }
17              Space B { // GPU offloading context 3
18                      stage7(m, ...)
19              }
20              Space E {
21                      stage8(m, ...)
22              }
23      }
```

Listing 6.8: Case 2 Transformation

Two Semantics Preserving Transformations of the Code in Listing 6.6

Note that both transformations are semantic preserving. In other words, Listing 6.6, 6.7, and 6.8 computations produce the same result. It is important to understand how the intent of the program remains unaltered even after the flow transformation.

The *IT* programming model offers a semantics where compute stages are executed on an LPU irrespec-

tive of the states of other LPUs. The only requirement is that during a stage execution the LPU has access to the most up-to-date data. It is the responsibility of the compiler to implant appropriate instructions in-between stage transitions to ensure strong data consistency among LPUs accessing shared or overlapped data. Therefore, as long as the LPU nesting hierarchy is respected and the programmer specified relative ordering of inter-dependent compute stages is preserved, any rearrangement of the original computation flow upholds the semantics of the source task.

Our flow transformation algorithm only affects the sequencing of LPUs without affecting the execution of any of them. Accordingly, the transformation does not compromise the original semantics. To explain the transformation in terms of the LPU traversal process described in Section 6.3.3, the transformation algorithm only converts the original DFS-like exploration of LPUs, to a hybrid of *BFS*[34] and DFS exploration.

Finally, note that it does not matter how many times *m* has been modified in the sub-flow executing inside Space-C in Listing 6.6. There will be still just one GPU offloading context in the Case 2 transformation spanning from Line 7 to 16 of Listing 6.8. This is because the entire Space-C resides within a partition of Space-B. Hence any change made to *m* within the former does not need to be propagated to other MPI processes. When the control comes out of the Space-C confinement, however, Space-B data parts across MPI processes need to be synchronized with one another. Consequently, *stage7* (Line 18 of Listing 6.8) has been put into a separate GPU Offloading Context.

6.5.4 CUDA Kernel Generation

Determining the Kernel Boundaries

Once a GPU offloading context is identified, the next concern is to determine the number of kernels the context needs. It should be obvious in Case 2 of Listing 6.8 that the first and the third offloading contexts need just one kernel each as the contexts have only one compute stage to execute. So let us focus on the second GPU Offloading Context that encircles the Space-C computation. According to Figure 6.8 mapping configuration, Space-C has been mapped to SMs and Space-D to warps. Updates made from warps of a single SM can be synchronized with one another inside a single CUDA kernel but not updates made across SMs. Updates made by one kernel are, however, visible to any subsequent kernel. This fact can be exploited to implement inter-SM synchronization. The run-time has to exit the current kernel and launch a new kernel to make each SM's update visible to all. So assuming *m* has overlapping partitions for Space-C LPUs and is modified in *stage3* and read in the others, the kernel boundaries for Case 2 of Listing 6.8 will be as shown in Listing 18.

Although discussed as such, in reality the flow-graph modification process does not require investigating the partition configurations of data structures and how they are being accessed within the compute stages. The dependency arcs carry enough information that viewed in the light of the mapping configuration is sufficient to determine all GPU Offloading Contexts and CUDA kernel boundaries. Therefore, instead of changing the flow graph generated by the front-end compiler, those boundaries can be directly applied during native code generation.

```
1 Computation:
2         Space A {
3                 stage1(m, ...)
4                 Space B { // GPU offloading context 1
5                         Kernel {
6                                 stage2(m, ...)
7                         }
8                 }
9                 Space B { // GPU offloading context 2
10                        Kernel {
11                                Space C {
12                                        stage3(m, ...)
13                                }
14                        }
15                        Kernel {
16                                Space C {
17                                        Space D {
18                                                stage4(m, ...)
19                                                stage5(m, ...)
20                                        }
21                                        stage6(m, ...)
22                                }
23                        }
24                }
25                Space B { // GPU offloading context 3
26                        Kernel {
27                                stage7(m, ...)
28                        }
29                }
30                Space E {
31                        stage8(m, ...)
32                }
33        }
```

Listing 6.9: Kernel boundaries within GPU Offloading Contexts for the Computation Flow of Listing 6.6

Code Generation for a CUDA Kernel

Lack of efficient support for recursive functions, significant cost of a non-coalesced GPU card memory access, consequence of shared memory bank-conflicts, and last but not least the need to exploit the lock-step threads of SMs/warps for good performance all make the previous mechanism of Composite PPU Controller execution unsuitable for a CUDA kernel implementation. A radically different LPU execution mechanism was needed and accordingly implemented. Let us take the kernel from Line 15 to 23 of Listing 6.9 as the example to discuss the deviation from the previous LPU execution model. Had the entire code been running on a segmented

memory or multicore back-end hardware then that portion of the flow graph would be translated as the code snippet of Listing 6.10.

```
1 while (spaceCLpu = getNextLpu(Space_C, Space_B, ...) != NULL) {
2         ... // Space C LPU dependent argument processing
3         while (spaceDLpu = getNextLpu(Space_D, Space_C, ...) != NULL) {
4                 ... // Space D LPU dependent argument processing
5                 if (threadState->isValidPpu(Space_D)) {
6                         stage4_function(spaceDLpu, ...);
7                         ... // any synchronization needed before stage transition
8                         stage5_function(spaceDLpu, ...);
9                 }
10        }
11        ... // any synchronization needed before returning to Space C
12        if (threadState->isValidPpu(Space_C)) {
13                stage6_function(spaceCLpu, ...);
14        }
15 }
```

Listing 6.10: Corresponding Code for the Discussed Kernel if the Task was Mapped to a Host-only Model

The Flow Executor of a Composite PPU Controller thread would execute the code snippet. During the *getNextLpu* calls of line 1 and 3, control would be passed to the LPU Generator that would construct the LPU by invoking many functions and doing a recursive LPU hierarchy traversal if needed. The LPU Generator is configured properly for individual threads beforehand. So a Flow Executor would only receive the LPUs designated for it. The sequential stage functions (Line 6, 8, and 13) would do the LPU computation. During those function executions, locality of LPU data in memory would be tackled by the hardware's caching machinery.

For the CUDA kernel, on the other hand, the generated code has the structure of Listing 6.11 (the entire kernel is basically described in comments because of the lengthiness of the component logic in the compiler generated code).

```
1 __global__ void cuda_kernel(LpuDataBuffers spaceBLpuBuffers, KernelMetadata metadata) {
2
3         extern __shared__ char memoryPanel[];
4         __shared__ int panelIndex;
5
6         ... // calculate SM, warp, and thread IDs of current thread
```

```
7
8       __shared__ spaceCLpu;
9       if (warp_ID == 0 && thread_ID == 0) {
10              ... // assign dynamic shared memory space for space C LPU data parts from memoryPanel
11      }
12      __syncthreads();
13
14      for (int spaceBLpuId = metadata.id_range.min; spaceBLpuId <= metadata.id_range.max; spaceBLpuId++) {
15
16              if (warp_ID == 0 && thread_ID == 0) {
17                      ... // retrieve Space B LPU data part references from spaceBLpuBuffer
18                      ... // calculate Space C LPU count spaceCLpuCount
19              }
20              __syncthreads();
21
22              for (int spaceCLpuId = sm_ID; spaceCLpuId < spaceCLpuCount; spaceCLpuId += sm_Count) {
23
24                      if (warp_ID == 0 && thread_ID == 0) {
25                              ... // calculate metadata for Space C LPU data parts
26                      }
27                      __syncthreads();
28
29                      ... // copy data from Space B LPU parts in the card memory to Space C LPU parts in shared memory
30
31                      if (warp_ID == 0 && thread_ID == 0) {
32                              ... // calculate Space D LPU count spaceDLpuCount
33                      }
34                      __syncthreads();
35
36                      __shared__ spaceDLpu[warp_Count];
37                      for (int spaceDLpuId = warp_ID; spaceDLpuId < spaceDLpuCount; spaceDLpuId += warp_Count) {
38
39                              if (thread_ID == 0) {
40                                      ... // calculate metadata for Space D LPU data parts
41                                      ... // assign data references for Space D LPU data parts within Space C LPU parts
42                              }
43
44                              stage4_device_function(spaceDLpu[warp_ID]);
45                              stage5_device_function(spaceDLpu[warp_ID]);
46                      }
47                      __syncthreads();
48
49                      if (warp_ID == 0) {
50                              stage6_device_function(spaceCLpu);
51                      }
52                      __syncthreads();
53
54                      ... // copy data from Space C LPU parts in shared memory to Space B LPU parts in the card memory
55              }
56      }
57 }
```

Listing 6.11: Structure of a CUDA Kernel

The Flow Executor and LPU Generator are fused together in the CUDA kernel. This is done by breaking down the recursive LPU traversal logic described earlier into nested for loops and associated metadata calculations with flow execution logic inserted in-between at appropriate places. Note that due to the presence of non-adjacent LPS transitions, LPU generation for the original computation sub-flow to be executed by an arbitrary kernel may not be directly convertible into a nested loop expansion as done in Listing 6.11. Therefore, a static analysis phase further transforms any non-adjacent LPS transition within the sub-flow into a series of transitions between adjacent LPSes before the kernel generation initiates.

Since one LPU is generated for an entire SM or warp at a time and most parts of the LPU generation process is sequential, only the first thread in a warp or an entire SM takes care of that part. Thus sections for LPU count calculation, LPU metadata construction, etc. of Listing 6.11 (such as Line 16-19, Line 24-26, and Line 39-42) are done within condition blocks that filter out all but one thread. As in the first multicore compiler, LPU generation within CUDA kernel is basically a metadata calculation process. Except for retrieving the metadata of the topmost LPS's LPU that is copied into GPU card memory by the host, no part of the LPU generation process accesses the card memory. This strategy works because smaller Space-C and Space-D LPUs operate on regions of the larger Space-B LPU.

The stages, on the other hand, are collectively executed by all the warp and SM threads. Therefore, there is no condition block surrounding Line 44 and 45 in Listing 6.11. Execution of *stage6* is handled by the threads of the first warp (Line 49 to 51). This is so as Space-C is mapped to the SM (Figure 6.8) which has no compute capacity [a] and LPUs of such a PPU is supposed to be executed by the first descendant or ancestor PPU that can

[a] In an alternative *PCubeS* description of the GPU, the compute capacity can be given to the SM directly without an additional warp PPS. Then the *PCubeS* description will have two PPS levels: GPU and SM. We are considering supporting both descriptions for a GPU as different descriptions are ideal for different algorithms.

compute. In Listing 6.10, the conditional block for PPU role checking from Line 12 to 14 does the same thing for the host-only mapping.

The CUDA device functions for compute stages are parallel as opposed to their counter parts in the host-only mapping. Iteration indexes of the parallel for loops of an *IT* source stage are distributed among the threads of the SM or warp. The efficiency of the index distribution is crucial for reducing shared memory bank conflicts among the threads during the loop execution. Currently we do a rudimentary analysis of the source *IT* stage to make the distribution decision. The analysis needs further improvements.

The programmatic control of the shared memory through an extern memory panel from Line 3 to 11 and then in Line 29 is another important aspect of the CUDA kernel generation. We reserve enough space in the shared memory for the topmost LPU data parts needed within each SM then assign different data parts different sections of the memory panel. Thus despite Space-B LPUs being shipped into the card from the host, memory is reserved only for holding a single Space-C LPU as an SM only needs that much memory during kernel execution. In this regard, we have a mechanism to determine the highest memory requirement for a descendant LPU given an ancestor LPU.

Finally, note that any data transfer between the GPU card memory and the SM shared memory is done collaboratively by all threads of the SM (Line 29 and 54) and in a manner that minimizes non-coalesced read/write of the card memory.

6.6 Future Work

Developing the compilers was like a race against time. We had to implement all three of them to prove the portability of *PCubeS* + *IT* paradigm in the architectures popular in present day high performance parallel computing, but there was not enough time and resources to optimize any compiler in particular. Some features are yet not implemented in one or two. Furthermore, the front-end compiler needs to be updated to support the new *IT* syntax. Among all these issues, some are critical for performance. As good performance is one of the major objectives of the paradigm, here we briefly discuss some performance critical issues that we want to address in the future.

IO Optimization

Currently any file IO operation is very inefficient in both segmented memory and hybrid compiler. We implemented a mechanism for initializing data parts from files using random file reads but the strategy does not scale. Moreover, file writing is now serialized among MPI processes. Several solutions are at table to address the IO issue such as memory mapped files, a new parallel data file format for *IT*, using MPI IO, and mapping a file as another LPS in an *IT* task.

Parallelizing Resource Setup Process

In all three compilers, creation of the data parts and preparation of the dependency resolvers are sequential operations. For large input sizes these operations can be a large part of a task's overall running time as they have many internal steps that are linear to the data parts count. Many of these steps can be made parallel. Given the

back-end architectures we are targeting are parallel; parallelizing these operations is a natural choice.

Improving GPU Execution

In the current implementation of the hybrid compiler, each GPU Offloading Context causes one or more host to GPU data copy-in, kernel execution, GPU to host data copy-out cycles. If the GPU Offloading Context appears within a repeat loop in the flow graph then the cycles repeat within each loop iteration. In other words, the GPU Offloading Context handling is now a memory-less process. Often times the data to be processed within a GPU does not change across repeat loop iterations. Some other time an entire data structure can be copied into the GPU and kept there for the lifetime of the task. As a data exchange between host CPU memory and the GPU card memory is a costly operation, it is important that we take advantage of all such opportunities.

Further, CUDA supports pipelining of offloading kernels through its streaming feature but currently we are launching the kernels in a lock-step manner. Similarly, some supercomputer nodes have more than one GPU but the current compiler only supports one GPU per node. We want to address these issues in the future also.

Programmatic Cache Control

Currently we are entirely relying on multicore CPUs cache machinery for LPU data locality. Our so far experiments indicate that sometimes data that we expect to be in the cache is probably not in there. This is first a violation of the paradigm and second it makes assessing what changes need to be made in the compilers for better code generation very difficult. So we want to introduce some level of programmatic cache control by the *IT* run-time in the future compilers. For example, we can use some modern hardware's support for pinning memory blocks in the caches for this purpose.

7

Experiments

This chapter describes the experiments we have done to assess the practicality of the *PCubeS + IT* programming paradigm as a portable and efficient alternative to contemporary high-performance parallel computing techniques. After a brief discussion on the objective and nature of the experiments; the chapter presents the test application suite, experimental platforms, and measurement techniques. Finally, it expounds the test results on individual platforms.

7.1 Objective and Nature of Experiments

If we paraphrase the thesis statement, the goal of this research is to provide a new direction for parallel computing through a novel hardware cognizant programming paradigm that has the following characteristics.

- The paradigm is portable across architectural platforms common in contemporary parallel computing.
- Enables writing programs whose efficiency is on-par with equivalent programs written using standard platform specific programming techniques.
- Simplifies debugging the runtime performance of a program.
- Makes it easy to write efficient parallel programs.

The experiments we have conducted [a] are aimed towards that goal. A comprehensive study of the effectiveness of a new programming paradigm requires significant time and human resource investment, and is outside the scope of this research. Consequently, we have relied on a representative set of applications to evaluate the paradigm. Experiments done on the representative set do not go all the way proving the stated goal has been reached, rather they show that the goal is achievable and the paradigm has the potential to achieve it.

Our intention was to choose the representative set from existing, well-known, and widely-used programming patterns so that a comparative analysis to platform specific low-level implementations can be made easily. Furthermore, we wanted the experiment results to be illustrative of the paradigm's effectiveness to average parallel computing enthusiasts. In that regard, we have chosen five building block problems from four characteristic application classes identified in the landmark paper 'The Landscape of Parallel Computing Research: A View from Berkeley[16].' The paper identifies 13 characteristic classes, or dwarfs, that capture different patterns of com-

[a] Most experiments are conducted by my adviser, Professor Andrew Grimshaw, and fellow graduate student, Swaroopa Dola

putation and communication that are common in important applications. We choose 4 out of those 13 to keep the effort of development and experiment manageable.

Our chosen dwarfs are: Dense Linear Algebra, Sparse Linear Algebra, Regular Grid, and Monte Carlo. The problems we have chosen are Matrix-Matrix Multiplication and LU Factorization from Dense Linear Algebra, Conjugate Gradient from Sparse Linear Algebra, 5-point Iterative Stencil from Regular Grid, and an Area Estimation problem from Monte Carlo. Each of these five problems has a different memory, computation, and communication characteristics that demands a careful consideration of target hardware's features in construing an efficient solution. Hence their choice helps to assess the effectiveness of *PCubeS* as an interface to the hardware as well as the efficiency of *IT* as the medium of programming.

We have used well-known and efficient, but not complicated algorithms for the chosen problems for both *IT* and reference implementations. The reference programs are hand-written sequential C++ programs for the multicore and segmented memory back-ends. For the hybrid back-end, we have compared *IT* programs against hand-written CUDA programs. To summarize, our objective is to show what a knowledgeable parallel programmer can achieve using *IT* at a reasonable effort – not how the *IT* programs fare against sophisticated library implementations done by experts. We believe our choice of algorithms is practical and adequate at this phase of the compiler and language development.

The experiments are done on three machines with three different architecture types in accordance to the three back-end compilers being discussed before. We have used the same *IT* source codes in all three back-ends to make the portability aspects of the paradigm obvious [a]. There is, however, one restriction: the hybrid

[a] Some IT programs for the hybrid back-end have deeper partition hierarchies in tasks than the corresponding tasks in programs for the remaining two back-ends. Any IT task with a deep partition hierarchy transforms into an equivalent task with a narrow hierarchy when consecutive LPSes are collapsed into one by mapping of them to the same PPS.

compiler does not yet support all features needed for the 5-point Iterative Stencil and Conjugate Gradient. Hence, only the remaining three programs are tested on the hybrid back-end.

We now discuss the individual problems and algorithms to understand their characteristics and what features of the hardware impacts their respective solutions. The *IT* source codes of all programs are given in Appendix A.

7.2 Application Suite

7.2.1 Matrix-Matrix Multiplication

Matrix-Matrix Multiplication is, arguably, the most important problem in linear algebra and a building block element to numerous larger applications. In this problem, a result matrix is computed by multiplying two argument matrices. Each $(i, j)_{th}$ entry in the result is the dot product of i_{th} row and j_{th} column of the first and second argument matrices respectively. The problem has $\mathcal{O}(mn^2p)$ asymptotic time complexity for argument matrices of dimensions $m \times n$ and $n \times p$. Matrix-Matrix Multiplication is a memory bound problem and its straightforward implementation suffers from severe cache misses for larger matrices. We implemented an alternative algorithm that divides the matrices into blocks, multiplies the blocks, and then accumulates the partial results to generate the final output (check the supplementary material 'Using Blocking to Increase Temporal Locality' of the book[24] for the algorithm). This incremental result construction process makes the program more compute bound. We refer to the algorithm as the *Block Matrix-Matrix Multiplication*. As explained in Chapter 3, *PCubeS* stresses on exposing the memory and cache hierarchy of a described hardware in detail. An *IT* programmer exploits the hierarchy by mapping LPSes of the *IT* program to proper PPSes and by se-

lecting partition parameters for the LPUs appropriately to fit them into the PPUs. The *Block Matrix-Matrix Multiplication* tests how effective this strategy of cache exploitation is in practice.

7.2.2 LU Factorization

LU Decomposition or Factorization is another common problem is linear algebra. In this problem, the argument matrix of a system of linear equations is decomposed into an upper and a lower triangular matrix to facilitate Gaussian Elimination. The technique is originally invented by Alan Turing[87], and many algorithms have been developed for it due to the importance of the problem. All these algorithms progress in a diagonal fashion by adding one or more entries in the upper and lower triangular matrices in each step based on computation done on unaltered parts of the original argument matrix. For numerical stability, most algorithms permute the order of either rows or columns of the argument matrix during the decomposition process. The permutation technique is called pivoting, and the algorithms are, accordingly, classified as LU Factorization with Partial Pivoting.

All parallel implementations of LU Factorization involve heavy communications of shared data update. This is the reason we have chosen LU Factorization as our second test problem. *IT* implementations of LU Factorization shows that the automatic data dependency resolution demanded by *IT* can be realized efficiently in the back-end compilers. Furthermore, this is a problem where index reordering data partitioning is essential for computation load balancing. Therefore, the implementations expose the cost of *IT*'s automatic index transformation also.

We have several *IT* programs implementing different LU Factorization algorithms. For the performance experiment, we only used a blocked algorithm that has an embedded SAXPY ($c = \alpha c + \beta a \times b$) operation

for better temporal locality (the algorithm is referred as the Blocked *kji*-SAXPY Algorithm in [90]). We call the algorithm *Block LU Factorization*.

7.2.3 Conjugate Gradient on Random Sparse Matrix

Conjugate Gradient is the most prominent mechanism for solving a sparse system of linear equations [80]. The normal strategy of Gaussian Elimination using matrix factorization is unsuitable for most sparse systems as sparse matrices in practical problems are usually several factors larger than their dense counterparts. Consequently, the factoring may be impossible because of limited memory or very time consuming when it is doable. Alternative algorithms for sparse matrices exploits the fact that most entries in a sparse matrix are zeros and avoid storing and computing over those entries.

Conjugate Gradient is an iterative method applicable for symmetric, positive-definite matrices. It applies a logic of *steepest descent of the gradient* of a quadratic form of the sparse matrix to quickly converge into the solution. We have implemented the algorithm published in [80] for random sparse matrices stored in *compressed row format (CSR)* [Dongarra]. Although the sparse matrices of practical problems are often regular structures (e.g., a diagonal band matrix), we have decided to implement the algorithm to work on the most generic and random CSR format. This is because an algorithm on a regular, partition-able sparse matrix does not reveal anything new about the *PCubeS + IT* paradigm after Matrix-Matrix Multiplication and LU Factorization. In particular, we wanted to investigate the impact of irregular memory accesses on the performance of an IT program as we increase the degree of parallelism.

7.2.4 5-point Iterative Stencil

5-point iterative stencil is a numerical differentiation technique on two-dimensional grids. In this problem, the next value of a point in the grid is computed from its current value and values of its four neighbors[8]. 2D and 3D stencil problems are common in Molecular Dynamics, Image Processing, Cellular Automata and many other fields. A critical aspect of all parallel stencil algorithms is that they involve boundary sharing among neighboring data partitions and consequent exchange of updated points in the overlapped regions. This is a difficult feature to implement in a generic way – nonetheless, *IT* supports that. We wanted to verify the effectiveness of the feature implementation.

The algorithm we have implemented uses *Jacobi iteration*[8] with *Dirichlet boundary condition*[Dir] to measure heat propagation on a simulated heated plate problem. Like most stencil programs, our implementation is memory bound. Note that straightforward stencil with Jacobi iteration has the problem of slow convergence and many applications similar to our simulated problem use more advance techniques such as Adaptive Mesh Refinement and Multigrid[41]. Supporting basic stencil is the first step toward enabling those techniques in *IT*.

7.2.5 Monte Carlo Area Estimation

Monte-Carlo method is a broad class of algorithms that use repeated random sampling to obtain numeric results for problems that are difficult or impossible to solve using other means[Mon]. Monte Carlo algorithms are embarrassingly parallel and generally compute bound. We implemented an area under the curve estimation problem that is a common use case of Monte Carlo simulation. We had to implement parallel reductions using MPI collectives and Pthread synchronizations to support this problem, but it has mainly been included in the

application suite as the sole compute bound problem.

7.3 Experimental Platforms

7.3.1 Multicore Back-end: Hermes Machines

We have used *Hermes* machines of Computer Science, University of Virgina (UVA) as the back-end platforms for the multicore compiler. There are four *Hermes* machines in the Computer Science, UVA cluster having the same configuration. We have run the tests on whatever machine we found available. Each 64-core *Hermes* machine consists of four 16-core AMD Opteron 6276 server processors on four socket of a Dell 0W13NR motherboard. RAM per CPU is 64 GB, giving a total of 256 GB main memory. There are three cache levels in an AMD Opteron 6276. A 6 MB L3 cache segment is shared among a group of 8 cores. Then a pair of cores share a 2 MB L2 cache. Finally, each core has a 16 KB L1 cache. Note that there are only 8 floating point units in a CPU despite the number of cores being 16. Each pair of cores share a floating point unit. The *PCubeS* description of a *Hermes* machine is illustrated previously in Figure 5.1.

The native C++ compiler we have used to generate the executables is gcc version 4.9.4 (Ubuntu4.9.4-2 ubuntu1 14.04.1). All codes have been compiled with O3 optimization flag enabled. Finally, note that Hermes machines have NUMA memory allocation enabled but the multicore compiler allocates data structures assuming a uniform memory. Hence, there is an inherent inefficiency in memory accesses done by *IT* executables that the sequential reference implementations do not suffer.

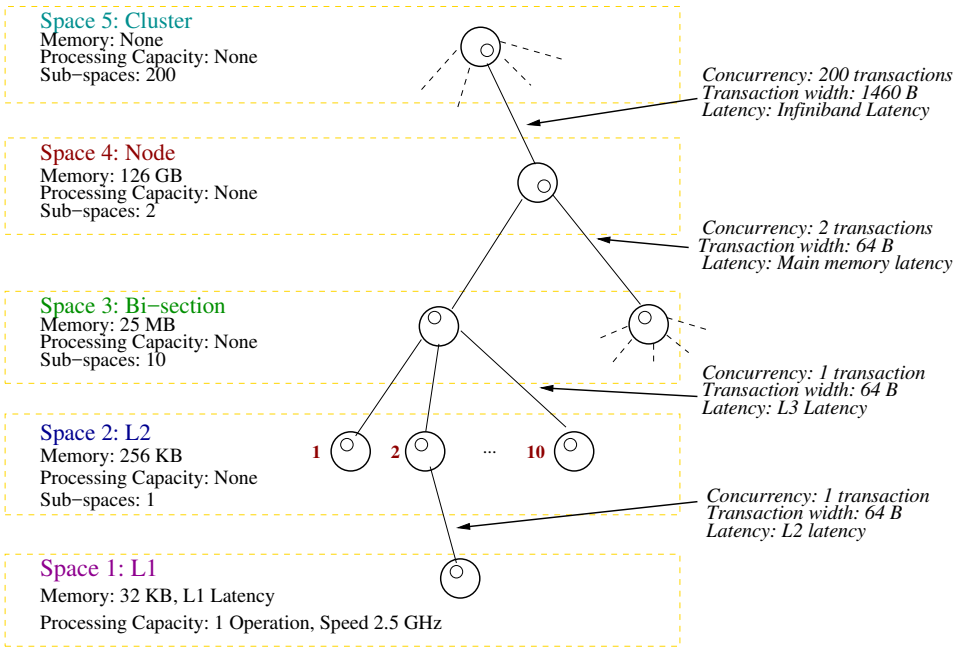


Figure 7.1: PCubeS Description of the Parallel Partition of Rivanna Cluster

7.3.2 Segmented-memory Back-end: Rivanna Compute Cluster

We have used the *parallel* partition in the *Rivanna* compute cluster of ARCS, UVA as the back-end for segmented memory compiler. The *parallel* partition has two 2.50 GHz Intel Xeon E5-2670 CPUs in each compute node. These nodes are connected by an Infiniband interconnect. Each CPU has 10 cores and three cache levels. A 25 MB L3 cache segment is shared among 10 cores. Then each core individually has a 256 KB L2 and a 32 KB L1 cache. Memory per node is 126 GB. The *PCubeS* description of the *parallel* partition is shown in Figure 7.1.

Although the *parallel* partition has many more nodes, due to our allocation restriction, we have limited the nodes count to 50 in the experiments. Thus the maximum number of cores being used in the experiments is

1000. The native MPI compiler in this platform is mpic++ that uses Intel's icpc version 14.0.2 C++ compiler underneath and the Open MPI Intel implementation (openmpi/intel/1.8.4) for message passing. Finally, note that Intel Xeon E5-2670 supports vector instructions but automatic vectorization through the underlying icpc compiler was working for neither *IT* executables nor reference implementations. Therefore, we ignore that capacity.

7.3.3 Hybrid Back-end: Big Red II GPU Cluster

We have used the *gpu* queue in the *Big Red II* supercomputer of Indiana University as the back-end for the hybrid compiler. Each node in the queue has a 16-core AMD Opteron 6276 server processor (the same CPU used in the Hermes machines) and an NVIDIA K20 GK110 GPU. Host CPU memory per node is 31 GB and the GPU has a 6 GB card memory. A K20 GPU has 13 symmetric multiprocessors (SMs) as apposed 14 of the K20X GPU described in Section 3.4.1. The remaining configurations are the same in both GPUs. All three *IT* programs we have run on the queue are single-task programs mapped to the *GPU model* of the machine [a]. The *PCubeS* description of the queue for the *GPU model* is illustrated in Figure 7.2.

We have used a maximum of 8 nodes to run the programs due to availability restrictions. Big Red II is a Cray machine that provides an integrated Cray environment for compiling and running programs. At the time of the experiments, the environment had PrgEnv-cray/5.2.82 set as the default. The underlying C++ compiler was crayc++ and the MPI implementation was cray-mpich/7.3.2. The CUDA compiler was nvcc 7.0, V7.0.27. As in the previous two platforms, back-end compilers had O3 optimization flag turned on. In addition, CUDA codes have been compiled with compute architecture setting 3.5 (*arch=sm_35* parameter).

[a] Remember a hybrid machine has 3 different *PCubeS* descriptions or models based on its intended use (Section 6.5).

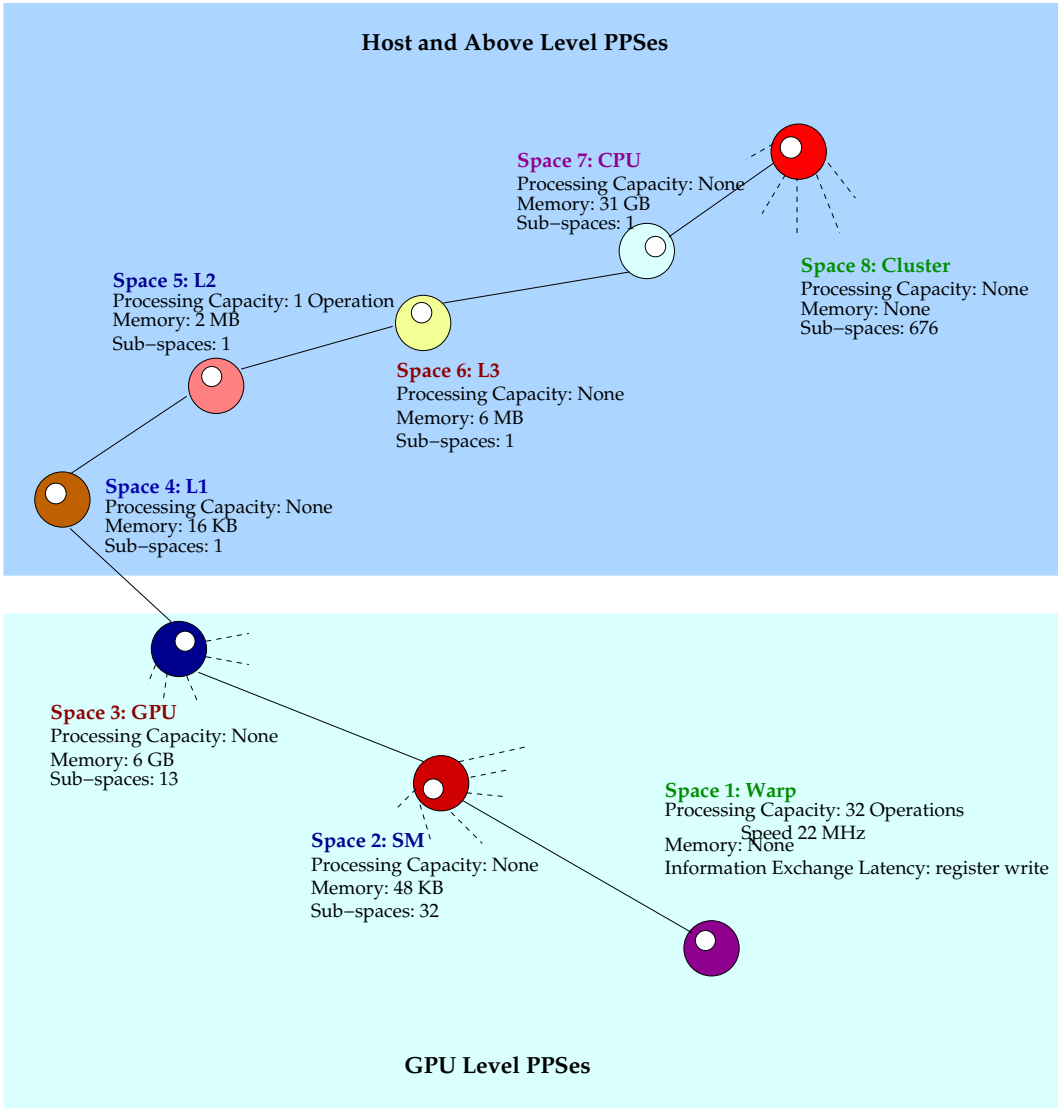


Figure 7.2: The PCubeS Description of GPU Queue of Big Red II

7.4 Measurement Criteria and Techniques

All measurements and analyses have been done on programs' running time. Lately it has become common to measure the floating point operations per second (FLOPs) in computational science problems as opposed to the running time. In our opinion, the former metric may be appropriate for assessing the efficiency gain through a particular program optimization but analyses based on the latter is the right choice when assessing a language. Therefore, we stick to the traditional and more intuitive running time measurement.

We have run each program 5 times and have taken the average to account for random performance fluctuations in the target machines at the time of the experiments. We were interested in three analyses of programs' running time.

1. *Speedup*: is the ratio between the sequential execution time and the parallel execution time.
2. *Scaled Speedup*: is the testing of retainment of speedup with proportional increase in execution resources and input size.
3. *Parallel Overhead*: is the portion of the running time spent on communication, redundant computations, and other overhead.

The speedup is generally measured for a fixed input size and different degrees of parallelism. This metric is of interest when there is a large problem at hand whose performance may be improved by parallelism. The asymptotic speedup of a parallel program is limited by the percentage of time it executes serially – the famous finding known as the Amdahl's Law[77]. If the performance of a program steadily improves with an arbitrary increase of the degree of parallelism then the program is called *strongly scalable*. The programs in our applications suite have different asymptotic speedups and only two of them, *Block Matrix-Matrix Multiplication* and *Monte Carlo Area Estimation* are theoretically strongly scalable. We were, however, interested only in actual speedup

of IT programs compared to reference sequential (or CUDA) implementations. For these experiments, different degrees of parallelism have been achieved from the same *IT* program by mapping the LPSes differently during compilation or increasing/decreasing the nodes count at runtime.

The scaled speedup has been proposed by Gustafson [49] as an alternative to Amdahl's asymptotic speedup when the objective for more parallelism is to solve larger problems – not the same small problem more efficiently. If the performance of a program remains steady with proportional increase of the input size and parallel execution resources then the program is called *weakly scalable* and is a good candidate for running on large-scale parallel architectures. Except for the *Conjugate Gradient*, all programs in our application suite are theoretically weakly scalable. We wanted to test if that property holds in practice for IT implementations. Since scaled speedup experiments require an increase in execution resources both in terms of memory and computation power, we performed them on the segmented memory and hybrid back-ends only. We skipped the multicore back-end as there memory and cache resources remain constant for different degrees of parallelism.

Finally, we have included the parallel overhead as the third metric to identify sources of inefficiencies that may hinder potential performance improvement in executables generated from the *IT* programs. This metric is important specifically for the segmented memory and hybrid back-ends as the generated executables for most programs in the application suite involve considerable data movements and communications. The multicore back-end has synchronization overhead only.

Input data for all experiments have been generated randomly as we were not interested in the results. This strategy does not affect the runtime behavior of any program in the application suite – except for the *Conjugate Gradient*, which requires a positive definite matrix with a spectral radius less than 1. To circumvent the problem, we have modified the iterative algorithm to run for a maximum number of iterations as opposed to until the

solution has been reached.

Further, all programs in the application suite use double-precision floating points for relevant array elements. Therefore, all running time measurements presented subsequently are for double-precision arithmetic.

Finally, note that we either did not measure the time consumed on file IO or disabled file read/write on programs when feasible. Thus the aforementioned running time is the time taken by the programs except for IO. We admit that file IO support is sub-optimal in all three compilers at present, and tackling this issue is an important future work (Section 6.6), but the issue is orthogonal to the objective of the current experiments.

The upcoming sections discuss the results of the experiments on the chosen platforms. We use graphical representations of the results in these sections to focus on trends as opposed to exact values of our metrics of interest. The actual timing data for all experiments are available in Appendix B.

7.5 Experiment Results on Hermes

7.5.1 Block Matrix-Matrix Multiplication

Figure 7.3 depicts the speedups of *IT Block Matrix-Matrix Multiplication* program compared to the sequential reference implementation. We used $10,239 \times 10,239$ square matrices [a] as inputs and 64×64 blocks to fit data into the caches during LPU computation.

Note that there is no 2 or 16-Cores version of the *IT* program as the *Hermes* machines' *PCubeS* description does not allow such breakups of the CPU cores.

[a] In the multicore back-end, input size that is an exact multiple of 1024 is causing *IT* programs to run slower than usual. We discovered this problem very late during the experiments. We are currently investigating the reason for this strange behavior.

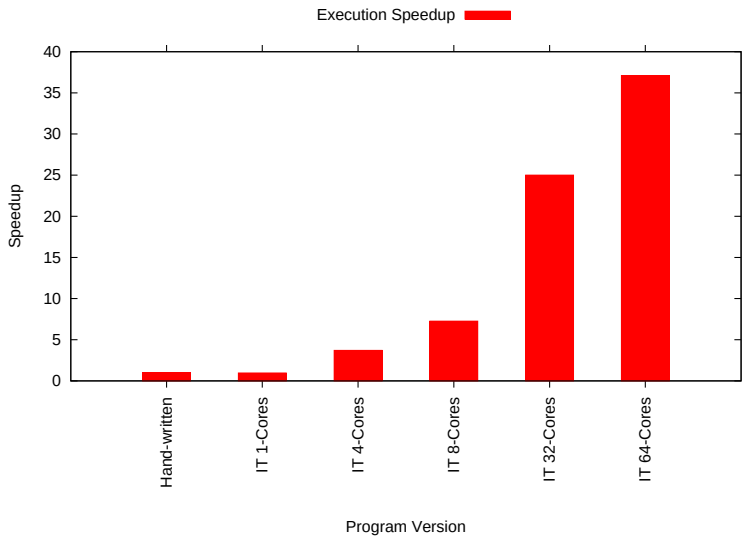


Figure 7.3: Strong Scaling Results for Block Matrix-Matrix Multiplication on Hermes for $10,239 \times 10,239$ Square Matrices

The *IT* versions achieve consistent and almost linear speedups with increase of parallelism up to 32 cores. The 37 times speedup for 64 cores is particularly interesting. The *Hermes* machines have only 32 floating point units for their 64 cores – still we got an speedup over 32. This happens due to better cache utilization with proper block size configuration.

Further, notice that the 1-Core *IT* version is almost as efficient (a slowdown of 4%) as the reference sequential program. This indicates the efficiency of our LPU generation process. For the given input and block size configuration, $160 \times 160 \times 160 = 4,096,000$ LPUs were generated by the *IT* program, but the impact of that on the running time appears to be marginal.

7.5.2 Block LU Factorization

Figure 7.4 depicts the speedups of *IT Block LU Factorization* program compared to the sequential reference implementation. We used an $10,239 \times 10,239$ square matrix as the input and a strided-block partition (with

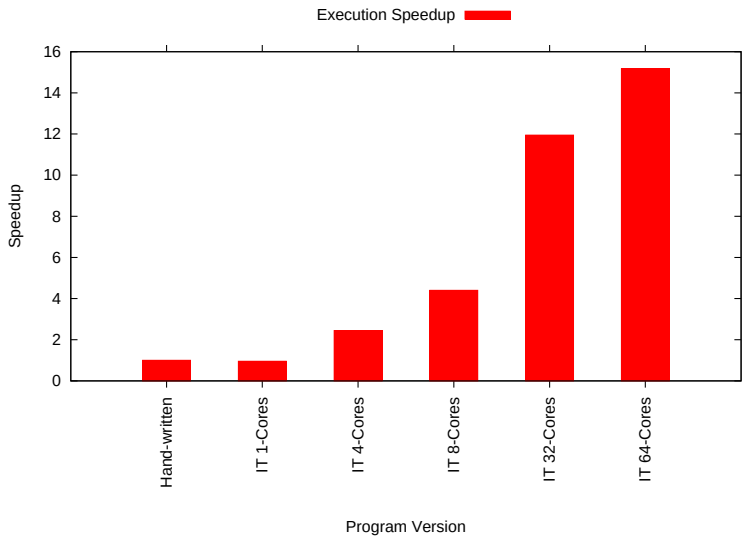


Figure 7.4: Strong Scaling Results for Block LU Factorization on Hermes for a 10, 239 × 10, 239 Square Argument Matrix

64 rows per stride) for load balancing [a].

The *IT* versions achieve consistent speedup with the increase of parallelism. The degree of speedup for *Block LU Factorization* is lesser than that of *Block Matrix-Matrix Multiplication*, as the former has two sequential stages per iteration and involves synchronization of shared data when the latter has no such bottlenecks.

The 1-Core *IT* version again approximates (a slowdown of 5%) the performance of the sequential reference implementation despite the former applying several index transformations when accessing array elements. This result illustrates the efficiency of our index transformation process for index reordering partition functions.

7.5.3 Conjugate Gradient on Random Sparse Matrix

Figure 7.5 shows the performance of *IT Conjugate Gradient* program compared to the sequential reference implementation for a 99.95% sparse 320, 000 × 320, 000 symmetric argument matrix. The experiment is done

[a] In a strided-block partition, a core gets stripes of rows/columns from the matrix as opposed to a sequential chunk.

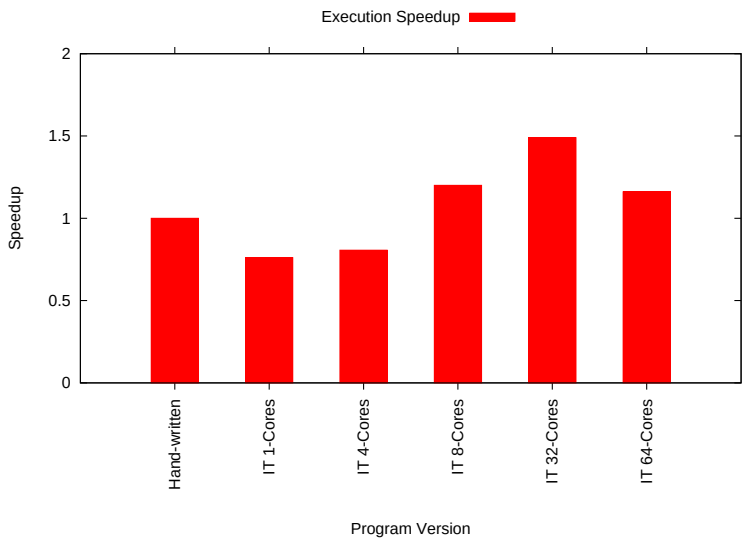


Figure 7.5: Strong Scaling Results for Conjugate Gradient on Hermes for a 99.95% Sparse $320,000 \times 320,000$ Symmetric Matrix

for 1000 iterations. For this problem, there is hardly any speedup with increasing parallelism. Given that most parts of the underlying algorithm are parallel and there is little LPU generation overhead in the component tasks, the results are seemingly disappointing.

So we investigated the runtime behavior of the *IT* program using the Linux *top* command. We have found that the CPU utilization hardly ever reaches 100% (the maximum is 3,200% due to 32 floating points unit) no matter what the degree of parallelism is. This suggests that the irregular data accesses from the sparse matrix as demanded by the underlying algorithm result in poor cache utilization in the program. The cores are spending most of the time waiting for data to arrive in caches as opposed to doing computation. In addition, space conflicts in the shared L2 and L3 caches are likely to intensify with more cores. The fall-off of speedup from the 32 to 64-Cores version further substantiate this conclusion.

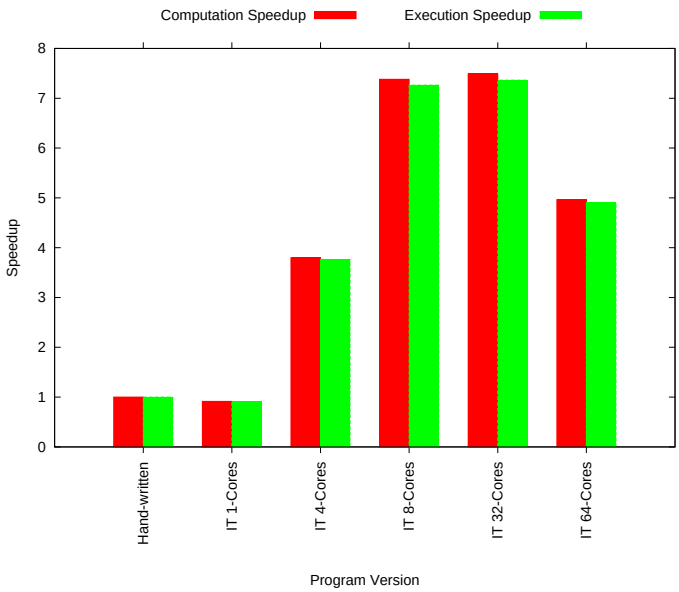


Figure 7.6: Strong Scaling Results for 5-point Iterative Stencil on Hermes for a $10,239 \times 10,239$ Plate and 10,000 Iterations

7.5.4 5-point Iterative Stencil

Figure 7.6 illustrates the speedup of *IT 5-point Iterative Stencil* program compared to the sequential implementation for a simulated heated plate problem of size $10,239 \times 10,239$. Each program version ran for 10,000 iterations. The graph displays two speedups per version as this problem involves data copying among neighboring partitions for overlapping boundary region synchronization. The *Computation Speedup* ignores the data copying cost which is included in the *Execution Speedup*. There were 4 overlapping boundary rows among data partitions requiring a synchronization in every 4^{th} iteration.

It is apparent from the results that data synchronization overhead is insignificant compared to the execution time in all parallel versions. We observe a near linear speedup up to 8-Cores where it peaked. This trend is expected as the cores are competing for spaces in the shared L2 and L3 caches in *Iterative Stencil* instead of co-

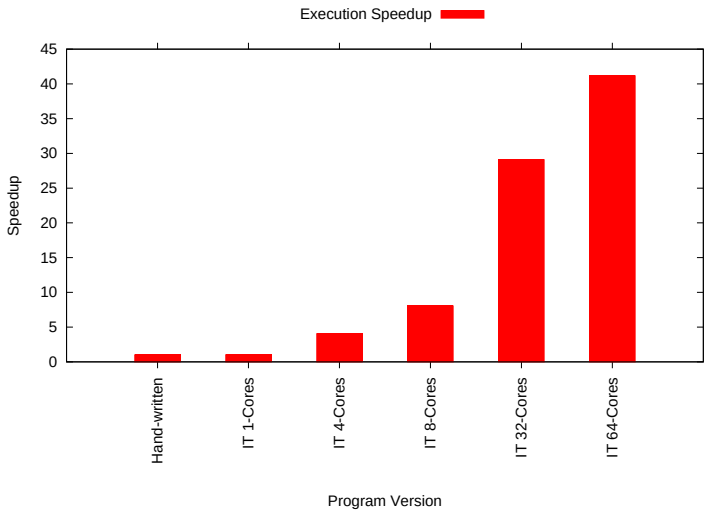


Figure 7.7: Strong Scaling Results for Monte Carlo Area Estimation on Hermes for a $10,239 \times 10,239$ Square Grid and 250 Samples Per Grid Cell

operating as they do in *Block Matrix-Matrix Multiplication* and *LU Factorization*. Up to the 8-Cores version, the L2 and L3 caches are unique for individual cores, justifying the near linear speedup.

7.5.5 Monte Carlo Area Estimation

Figure 7.7 presents the speedup of *IT Monte Carlo Area Estimation* program compared to the sequential implementation. The bounding area for the problem was divided into a grid of $10,239 \times 10,239$ cells and 250 random samples were generated per cell to estimate the area under the curve represented by the equation, $10\sin(x^2) + 50\cos(y^3)$.

As shown in Figure 7.7, *IT* versions have a near linear speedup compared to the sequential reference implementation up to the 32-Cores version. There is a further improvement in the 64-Cores version. This is similar to the behavior we have observed for *Block Matrix-Matrix Multiplication*. For *Monte Carlo Area Estimation*, the improvement is more likely due to faster integer operations rather than better cache utilization. Although

a pair of cores in *Hermes* share a single floating point unit, each has its own integer unit. In the future, we would like to investigate if exposing the integer capacity in the *PCubeS* description of a hardware along with the floating point capacity has any significant benefit.

7.6 Experiment Results on Rivanna

7.6.1 Block Matrix-Matrix Multiplication

Figure 7.8 depicts the strong and week scaling results of *IT Block Matrix-Matrix Multiplication* program on *Rivanna* compute cluster. The strong scaling results have been presented as the speedup of IT program versions compared to the sequential reference implementation. Here the *Computation Speedup* only considers the computation time and the *Execution Speedup* considers the computation time and any additional overhead. The input matrices for strong scaling were $10,239 \times 10,239$ squares and the block size was fixed to 64×64. The weak scaling results for the *IT* versions are shown as a breakdown of the actual running time for different input size and core count combinations.

We observe a super-linear speedup of computation with increasing parallelism. The 1000-Cores version performs 1180 times better than the sequential implementation. The speedup tapers off when additional overhead is considered. The execution speedup for the 1000-Cores version is only 238 times compared to the sequential implementation. Our investigation suggests that this happens because of the drastic reduction of the computation time that makes the overhead cost a larger percentage of the overall running time. The actual <computation time, overhead cost> combinations for the 500-Cores and 1000-Cores versions are <3.34861 sec, 6.49955 sec> and <1.638 sec, 6.46373 sec> respectively. We could not run the experiment for a much larger input size

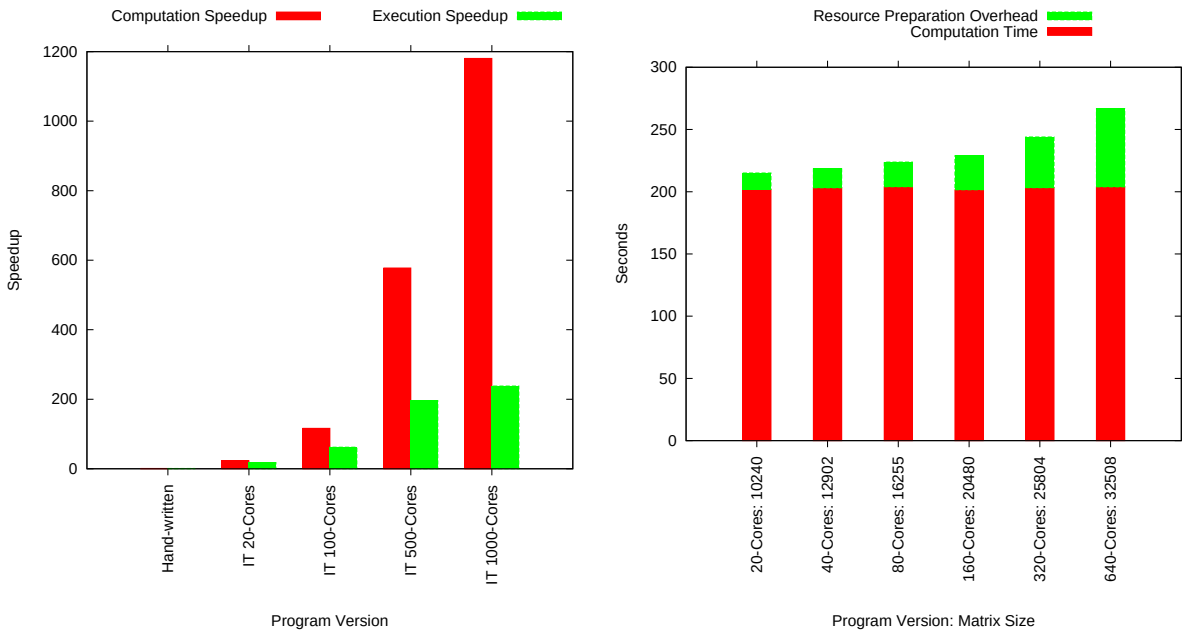


Figure 7.8: Strong (Matrix Size 10,239) and Weak Scaling Results for Block Matrix-Matrix Multiplication on Rivanna

due to a maximum memory per node limitation set on Rivanna by its job scheduling system.

We used a more gradual increase of core counts to capture the trend for the weak scaling experiment. As shown in the graph, the computation time remains almost flat. These are the best possible results. The overhead computation per node increases with larger input sizes. For this problem, the entire overhead is due to the initial serial preparation of array data parts and auxiliary management data structures. Much of this cost can be eliminated in the future by parallelizing the steps of the resource preparation process.

7.6.2 Block LU Factorization

Figure 7.9 presents the strong and weak scaling results for *IT Block LU Factorization* on *Rivanna* compute cluster. This time the input argument matrices were $20,480 \times 20,480$ squares for the strong scaling experi-

ment.

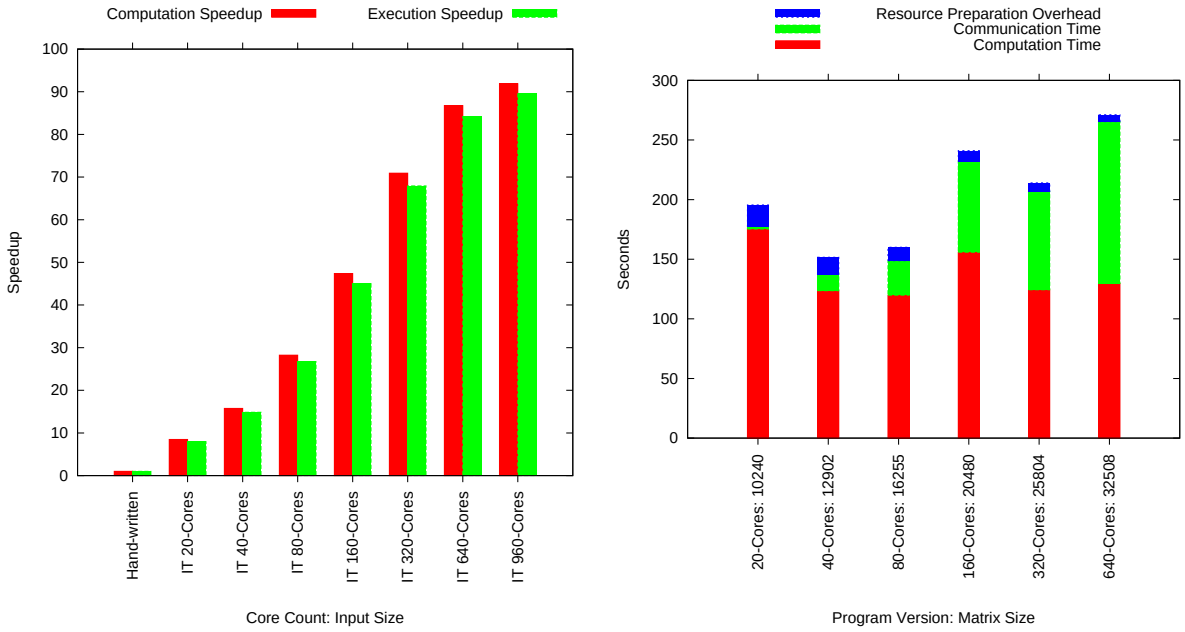


Figure 7.9: Strong (Matrix Size 20,480) and Weak Scaling Results for Block LU Factorization on Rivanna Cluster

Compared to the previous *Block Matrix-Matrix Multiplication* program, we observe only a modest improvement of performance with increasing parallelism. This is expected as the IT implementation now involves multiple collective MPI communications per iteration on top of the intra-node CPU core synchronization for serial compute stages. The computation speedup (including both computation and communication as both are parts of the algorithmic logic) is not drastic and the difference between the two speedup metrics is minor. This is because the additional overhead cost for data structures and communication resources setup is insignificant compared to the cost of actual computation.

For the weak scalability testing, we further distinguish between time spent on actual computation and time spent on communication. As shown in Figure 7.9, there are fluctuations in the computation times but the

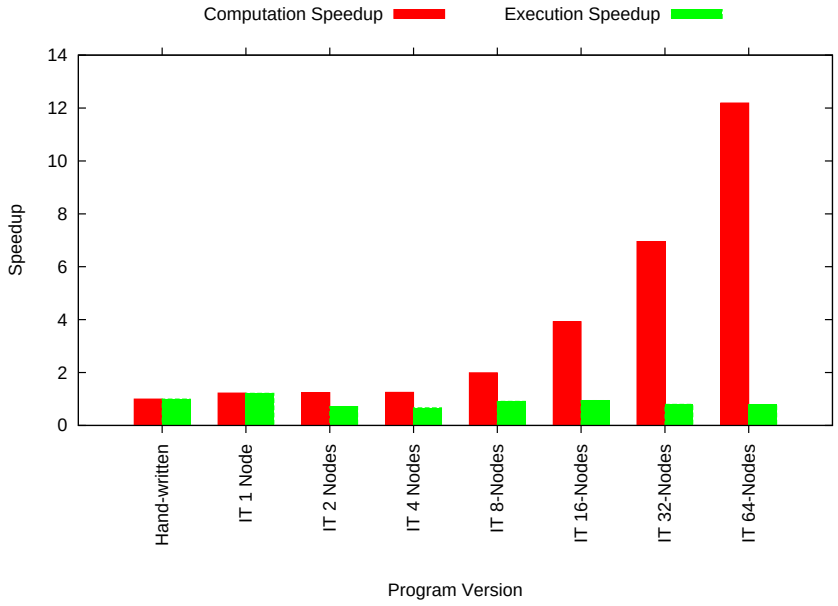


Figure 7.10: Strong Scaling Results for Conjugate Gradient on Rivanna for a 99.95% Sparse 320, 000 × 320, 000 Matrix

overall trend is flat. The communication time goes, predictably, up with larger problem sizes due to increasing cost of communicating larger data parts among more nodes. The initial sequential resource preparation process seems to have a slight decreasing trend. We need to investigate the reason for this performance improvement.

7.6.3 Conjugate Gradient on Random Sparse Matrix

Figure 7.10 shows the performance of *IT Conjugate Gradient* program compared to the sequential reference implementation for the same 99.95% sparse 320, 000 × 320, 000 argument matrix used during experimenting on *Hermes*. Again the experiment is done for 1000 iterations. Given our multicore experiment did not give significant speedup with increasing parallelism, we decided to scale up the resources in terms of node count as opposed to core count and let each node do its part of the computation sequentially. We expected a steady increase of speedup with more parallelism with this strategy.

The results are the biggest revelation in the segmented-memory performance testing. The computation does improve with increasing parallelism but the increasing cost of the concomitant overhead calculation completely tramples any performance gain in the overall execution. We found that most of the overhead cost is associated with *Environment Managers'* interaction during task transitions. As discussed in Chapter 6, we knew that interactions among *Environment Managers* are inefficient compared to *PPU Controllers'* interaction within a single task, but we did not expect the difference to be so significant. Arguably, had we implemented *Conjugate Gradient* as a single task as opposed to a program with three different tasks and six task invocations per iteration, the overhead cost would be a tiny fraction of that of the current overhead. Such a change is, however, unjustified due to productivity reasons. The results rather suggest that optimizing the environment management part of the *IT* RTE implementation should be a central concern during future development.

7.6.4 5-point Iterative Stencil

Figure 7.11 presents the strong and weak scaling results for *IT 5-point Iterative Stencil* on *Rivanna* compute cluster. The simulated heated plates were $10,239 \times 10,239$ squares for the strong scaling experiment. Further, for the strong scaling experiment, boundary overlapping among inter-node and intra-node data parts were 16 rows and 4 rows respectively. The LPSes were made 1D by the runtime partition configuration despite the source IT task has 2D partitioning in both LPSes. For the weak scalability experiment, on the other hand, we kept the original 2D partitioning for the upper LPS to stabilize the communication overhead and used 4 overlapping boundary rows and columns in the inter-node data parts and just 1 overlapping row for the intra-node data parts.

The results are good for both experiments as illustrated in Figure 7.11. The results also show the importance

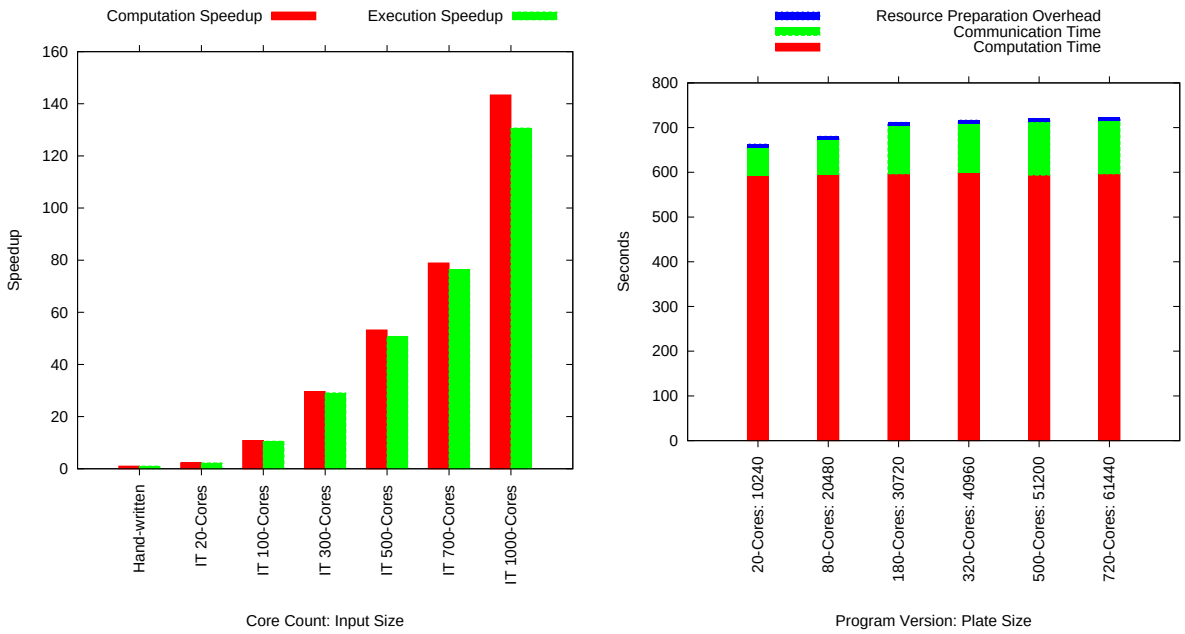


Figure 7.11: Strong and Weak Scaling Results for 5-point Iterative Stencil (10,000 Iterations) on Rivanna

of choosing appropriate partitioning parameters. We did some weak scalability testing using 1D partitioning also and experienced performance deterioration in larger problems due to a linear increase of the communication overhead. A 2D partitioning keeps the per-node communication overhead more or less constant as can be seen in Figure 7.11.

These results also illustrate the flexibility the *PCubeS* + *IT* paradigm offers in shaping a program's behavior. A generic 2D partitioned program of the *5-point Iterative Stencil* is much harder to implement for an average programmer than a program using 1D partitioning. In *IT*, however, the difference is just a one line change.

7.6.5 *Monte Carlo Area Estimation*

Figure 7.12 depicts the strong and weak scaling results for *IT Monte Carlo Area Estimation*. The strong scalability experiment is done for the same curve and grid configuration as in the case of corresponding experiment in *Hermes*. The weak scalability experiment only increases the grid dimension. Both experiments generate 1000 samples per grid cell.

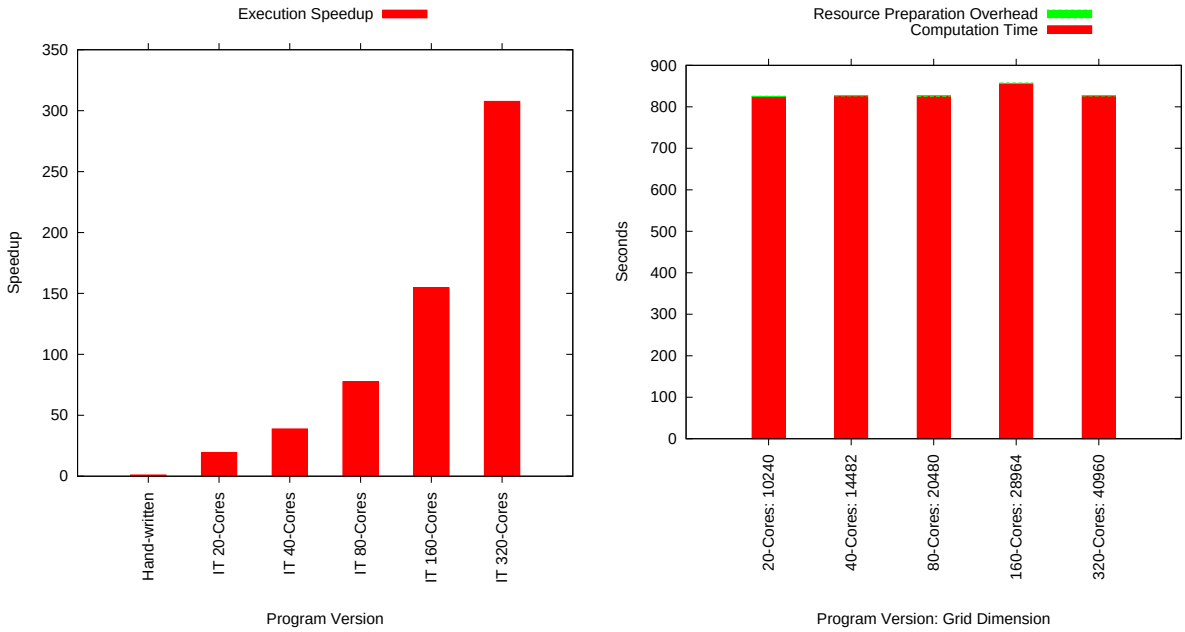


Figure 7.12: Strong and Weak Scaling Results for Monte Carlo Area Estimation (1000 Samples Per Cell) on Rivanna

The results are unsurprisingly good for both experiments. We observe a near linear increase of speedup with increasing parallelism on a fixed input size, and flat running times for proportional increase in parallel resources and the input size for this embarrassingly parallel program.

7.7 Experiment Results on Big Red II

Before we discuss the performance results on *Big Red II*, we like to point out that the hybrid *IT* compiler is currently in its initial stage. We have some programs running but we are still struggling to come up with a performance model that will inform us what code generation strategy for the GPU kernels is ideal and what data transfer policy is better for the CPU host and GPU interactions.

In addition, the reference implementations are highly optimized single GPU CUDA programs that assume the program data always resides in the GPU card memory once copied into it. The *IT* programs moves data in and out of the GPU card memory to support much larger input sizes than that can be fit into a single GPU all at once. The impact of this behavior difference is significant and we do not have an apple-to-apple comparison between the hand-written reference and *IT* implementations.

7.7.1 Block Matrix-Matrix Multiplication

Figure 7.13 depicts the strong and weak scaling results for *IT Block Matrix-Matrix Multiplication*. The Results of both experiments are shown as a breakdown of overall running time to expose the underlying components of inefficiencies, if exists. The strong scalability experiment is done for $10,240 \times 10,240$ square input matrices.

The *Other Host Level Overhead* component in the graphs represent the cost of data preparation and auxiliary resource setup. The *Host Execution Time* represents the time spent doing computation/communication in the host that cannot be pushed down to the GPU due to some algorithmic restriction or, in the case of the *IT* implementation in particular, due to a lack of support in the current hybrid compiler. The remaining two timing components are self explanatory.

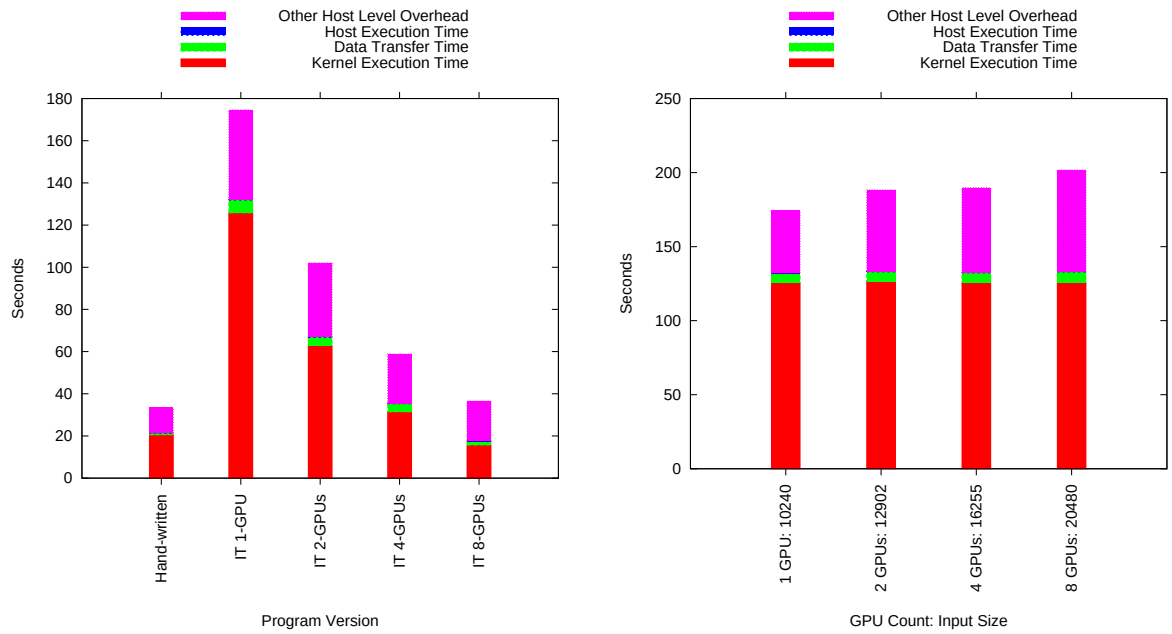


Figure 7.13: Strong and Weak Scaling Results for Block Matrix-Matrix Multiplication on Big Red II

As shown in Figure 7.13, the *IT* 1-GPU version is almost 6 times slower than the CUDA reference implementation and that is mostly because of the generated kernel's relative inefficiency compared to the hand-written version. We are currently investigating the reason for this slowdown. The *IT* implementation's performance almost linearly improves with increasing GPU count. This result indicates that if we can solve the inefficiency problem for the 1-GPU version then the performance gain will be substantial.

The host level overhead for the *IT* implementation is also larger than that of the reference implementation. This is expected as managing small data parts for LPUs separately involves more computation overhead than does initialization of a single large block of memory as done in the reference implementation. Further, this overhead does not alarm us as it also exhibits a downward trend with increasing parallelism.

The weak scalability results are good and consistent with earlier results from *Rivanna*. Thus the current

hybrid compiler is adequate in that regard.

7.7.2 Block LU Factorization

Figure 7.14 shows the strong and weak scaling results for *IT Block LU Factorization*. The strong scalability experiment is again done for 10,240 × 10,240 square input argument matrices.

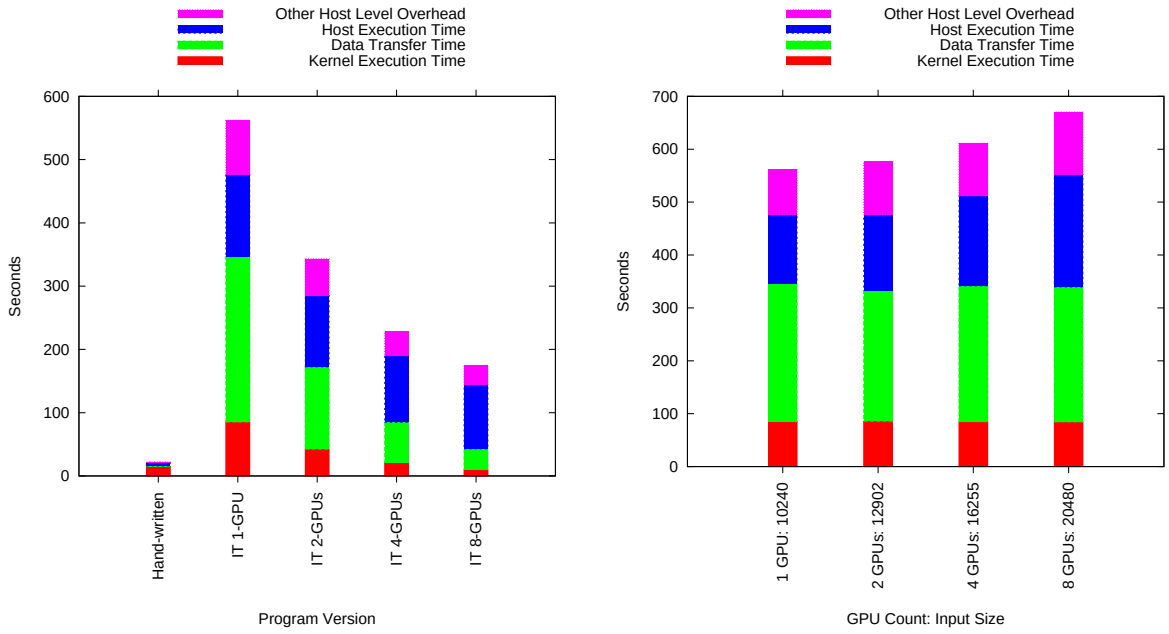


Figure 7.14: Strong and Weak Scaling Results for Block LU Factorization on Big Red II

The most noticeable feature in both graphs are the large data transfer and host execution costs. Other than that, the results are consistent with the findings from the previous *Block Matrix-Matrix Multiplication* experiments. The host execution time part is so significant in the *IT* versions primarily because the hybrid compiler still cannot generate CUDA kernels for stages with reduction operations and epoch version updates. Consequently, the near 5% part of the computation that executes in the host has become the larger contributor of

the runtime due to a massive speed difference in the kernel and host code executions. This part of the running time is not scaling with increasing parallelism either. We need to give this issue the most emphasis in future development of the hybrid compiler.

Despite having a downward trend, the data transfer time is also unacceptably large. This happens due to the memoryless nature of the host and GPU interaction. The *IT* implementation exchanges two whole matrices between the GPU and the host in each encounter of the same *GPU Offloading Context* when only a tiny fraction of the matrices is updated in-between successive encounters. With a more sophisticated data transfer mechanism we should be able to eliminate most of this data transfer cost.

7.7.3 Monte Carlo Area Estimation

Finally, Figure 7.15 illustrates the strong and weak scaling results for *IT Monte Carlo Area Estimation* on *Big Red II*. The curve being investigated is the same curve from earlier multicore and segmented memory experiments. The bounding area is again divided into a square grid of $10,240 \times 10,240$ cells for the strong scalability test. The samples per cell is, however, set to 10,000 to produce enough work for the warps of the GPU SMs.

The results of both experiments are excellent as can be seen in Figure 7.15. Unlike *Block Matrix-Matrix Multiplication* and *Block LU Factorization*, the 1-GPU version of the *IT* program performs nearly as well as the handwritten CUDA program. Then there is a linear reduction of running time – equivalently, a linear increase of speedup – with more GPUs. The difference between the *Monte Carlo Area Estimation IT* kernel and that of the previous two programs is that the former has an insignificant memory access overhead compared to the latter. The three kernels involve comparable overhead calculation for LPU generation. This result suggests that the kernel execution time slowdown we have observed for the earlier two programs is probably related to

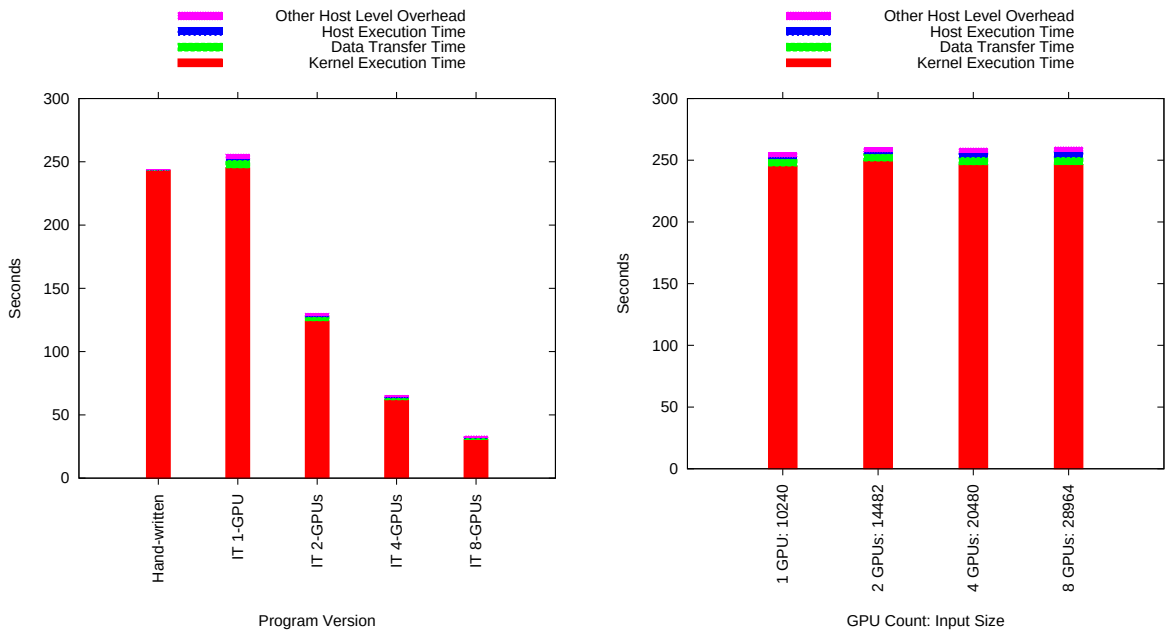


Figure 7.15: Strong and Weak Scaling Results for Monte Carlo Area Estimation (10,000 Samples Per Cell) on Big Red II

sub-optimal memory accesses as opposed to overhead computations.

8

Conclusion

My desire to undertake this research was ignited from a personal experience with parallel computing in a graduate course I took for that subject several years ago. The students were asked to implement a simple *5-point Iterative Stencil* program that uses 1D partitioning in three different architectural platforms in three consecutive homework assignments. I found the program difficult to implement efficiently – every single time – due to the differences in the target architectures and the programming tools we had to use. This was a troubling

experience given the underlying algorithmic and data partitioning strategies that lead to an efficient parallel solution for the *5-point Iterative Stencil* remain the same irrespective of the target hardware. Even more worrying was the observation that I spent most of my effort getting the parallelization tools right than thinking about a better solution.

I agree with Andrews' statement[15] that "concurrent programs are to sequential programs what chess is to checkers." Nevertheless, I found placing the chess pieces on the board and understanding their rule of engagement to be harder than actually playing the game! The predominant low-level parallelization tools such as MPI, Pthreads, and CUDA have made parallel computing harder than what it inherently is. The present high-level language based alternatives to those tools, on the other hand, have not appreciated that inherent difficulty enough and are trying to make a chess out of checkers. Thus the overall outcome is either discouragement or disappointment for people who could be benefited from learning parallel programming, and there are a lot of them now.

With the end of rapid single-processor performance improvement, there is a proliferation of parallel hardware everywhere. At the same time, costly compute intensive applications and large-scale data analytics have become the norm in many areas of sciences, arts, and industry. The potential benefits of parallel computing is enormous at present. We just need proper paradigms to make parallel computing enjoyable, productive, and above all rewarding enough to avail the current opportunity.

The broad objective of this research was to provide such a parallel programming paradigm and in doing so proving that the aforementioned vision is practical. The central idea behind the *PCubeS + IT* programming paradigm is to provide an unifying abstraction, called a type architecture, that exposes the salient features and their associated costs of any parallel hardware; then describe a parallel program in terms of that generic

abstraction using a medium that seamlessly integrates reasoning about hardware features with expressing the algorithmic logic of the program. I hoped that this paradigm will enable high performance and portable parallel programming without losing programmer productivity. That hope led the my research claim:

> *A high level parallel programming paradigm based on type architecture will enable portability, simplify learning and performance debugging, and approximate the efficiency of contemporary low level programming techniques.*

This research has produced the *PCubeS* type architecture that describes a parallel hardware as a finite hierarchy of parallel processing spaces each having fixed, possibly zero, compute and memory capacities and containing a finite set of uniform, independent sub-spaces. This research has also given the *IT* language for high performance parallel computing in which computations are defined to take place in a corresponding hierarchy of logical processing spaces, each of which may impose a different partitioning of data structures. In addition, three *IT* compilers for three architectural back-ends – multicore CPUs, distributed memory supercomputers and compute clusters, and hybrid supercomputers having both multicore CPUs and NVIDIA GPUs as nodes – have been developed. All these took closely 4 years of research and development. Now it is the right time to ask how far the work goes in proving my stated claim.

As illustrated in Chapter 3, *PCubeS* can describe most present day parallel architectures effectively regardless of their heterogeneity. Thus an unifying machine abstraction for present day parallel hardware is an evident reality. That we have three *IT* compilers and are able to run the same IT program in three different architectural back-ends is also a proof of the *PCubeS* + *IT* paradigm's portability. The qualitative attributes of the language such as readability, simplicity, and programmer productivity have not been verified yet using some human

study. Regardless, I will argue that the strong separation of concerns and the use of a declarative syntax that give IT programs a pseudo-code like appearance, and the brevity of an *IT* program compared to equivalent hand-written parallel programs written using existing techniques are indicative of *IT*'s success in those regards.

Regarding efficiency, I believe it is fair to say that the *PCubeS + IT* paradigm has promising prospects. The results presented in Chapter 7 for five well-known building block applications are generally good. Yes, we discovered performance issues in the *IT* programs, and we are not yet ready to compete with existing parallelization techniques that are in decades of development, but we have not discovered any problem during experiments that appears to be an obstacle for future performance improvement. Rather the opportunity of performance improvement through better cache control, sophisticated environment management, context-aware GPU kernel scheduling, and many more makes me excited about a bright future.

I would like to work on those optimizations but I would prefer to pass the baton and let other burgeoning, student researchers to investigate future opportunities. For example, implementing an efficient parallel file IO mechanism for the *PCubeS + IT* paradigm should be a rewarding project on its own. One can also pick a particular back-end architecture and investigate all possible optimizations for the corresponding *IT* compiler. Optimizing communication and environment management overhead in the *IT* RTE for Exascale Computing should also be interesting. In my view, this research answers as many questions as it introduces. I want fresh new ideas to enrich the paradigm through the involvement of new student researchers as opposed to having the project started and ended with me.

I did not want the *PCubeS + IT* paradigm to be just an academic endeavor, answering questions asked only for the sake of curiosity. I wanted it to be a real solution to real-world problems – not within the 4 years of my research but definitely in the future. So in conclusion it is appropriate to ask what need to be done to

make that happen. Real world applications are not like the building block programs we did the experiments on. Rather, they have many components and demand a wide range of features. Therefore, the *IT* language will require significant feature enhancement. Regardless, the future researchers must be cautious about bringing in new features without properly investigating what can be implemented efficiently across the board and what not. The central theme of a lean language core, a declarative syntax, and a clear separation of concerns must be respected to retain the flavor of the paradigm. This is an alternative line of research I encourage my successors to examine.

For the near future, I believe support for adaptive mesh refinements and multi-grid methods and an efficient list implementation coupled with an improved file IO and some synthetic features will take *PCubeS + IT* a long way towards solving many critical real-world problems for which efficient solutions are yet unknown or too difficult to construct using existing techniques. We need research fundings and collaboration with domain scientists to make progress in that direction.

Finally, I want to thank the reader for coming this far. That you have read it is my reward, and I treasure it dearly. If you have learned anything from my writing or have gotten even a single chuckle out of it, that is my humble gift to you in return. Thank you.

A

IT Source Codes

A.1 Block Matrix-Matrix Multiplication

```
1 Program (args) {
2
3       // create an environment object for the matrix–matrix multiplication task
4       mmEnv = new TaskEnvironment(name: ``Block Matrix–Matrix Multiply")
5
6       // specify how external input files are associated with the environmental objects
7       bind_input(mmEnv, ``a", args.input_file_1)
8       bind_input(mmEnv, ``b", args.input_file_2)
9
10      // execute the task
```

```
    execute(task: ``Block Matrix–Matrix Multiply"; environment: mmEnv; partition: args.k, args.l, args.q)

    // specify where the output should be written to
    bind_output(mmEnv, ``c", args.output_file)
}

Task ``Block Matrix–Matrix Multiply":
    Define:
        a, b, c: 2d Array of Real double–precision
    Environment:
        a, b: link
        c: create
    Initialize:
        c.dimension1 = a.dimension1
        c.dimension2 = b.dimension2
    Stages:
        // a single computation stage embodying the logic of the matrix–matrix multiplication
        multiplyMatrices(x, y, z) {
            do { x[i][j] = x[i][j] + y[i][k] * z[k][j]
            } for i, j in x; k in y
        }
    Computation:
        Space A {
            // the stage has to be repeated for each sub–partition of Space A to have a block implementation
            // as opposed to a traditional one
            Repeat foreach sub–partition {
                multiplyMatrices(c, a, b)
            }
        }
    Partition (k, l, q):
        // 2D partitioning of space giving a block of c in each partition along with a chunk of rows of a
        // and a chunk of columns of b
        Space A <2d> {
            c: block_size(k, l)
            a: block_size(k), replicated
            b: replicated, block_size(l)
            // block–by–block flow of data inside a PPU is governed by the sub–partition specification
            Sub–partition <1d> <unordered> {
                a<dim2>, b<dim1>: block_size(q)
            }
        }
```

A.2 Block LU Factorization

```
Program (args) {
    luEnv = new TaskEnvironment(name: ``Block LU Factorization")
    bind_input(luEnv, ``a", args.input_matrix_file)
    execute(task: ``Block LU Factorization"; environment: luEnv; partition: args.block_size)
    bind_output(luEnv, ``u", args.upper_matrix_file)
    bind_output(luEnv, ``l", args.lower_matrix_file)
    bind_output(luEnv, ``p", args.pivot_matrix_file)
}
```

```
Task ``Block LU Factorization'':
    Define:
        a, u, l: 2d Array of Real double-precision
        p: 1d Array of Integer
        l_row, l_column, p_column: 1d Array of Real double-precision
        u_block, l_block: 2d Array of Real double-precision
        pivot, k, r, block_size: Integer
        row_range: Range
    Environment:
        a: link
        u, l, p: create
    Initialize:
        u.dimension1 = l.dimension1 = a.dimension2
        u.dimension2 = l.dimension2 = a.dimension1
        p.dimension = a.dimension1
        block_size = partition.b
        l_row.dimension = l.dimension2
        l_column.dimension = p_column.dimension = l.dimension1
        u_block.dimension1 = u.dimension1
        u_block.dimension2.range.min = l_block.dimension1.range.min = 0
        u_block.dimension2.range.max = l_block.dimension1.range.max = block_size - 1
        l_block.dimension2 = l.dimension2
    Stages:
        prepareLU(a, u, l) {
            do { u[j][i] = a[i][j] } for i, j in a
            do { l[i][i] = 1 } for i in l
        }
        calculateRowRange(a, row_range, block_size) {
            last_row = r + block_size - 1
            if (last_row > a.dimension1.range.max) {
                last_row = a.dimension1.range.max
            }
            row_range.min = r
            row_range.max = last_row
        }
        selectPivot(pivot, u, k) {
            do { pivot = reduce (``maxEntry'', u[k][j]) } for j in u and j >= k
        }
        storePivot(p, k, pivot) {
            p[k] = pivot
        }
        interchangeColumns(pivot, k, u, l) {
            do {    u[i][k] at (current) = u[i][pivot] at (current - 1)
                    u[i][pivot] at (current) = u[i][k] at (current - 1)
            } for i in u and i >= k
            do {    l[i][k] at (current) = l[i][pivot] at (current - 1)
                    l[i][pivot] at (current) = l[i][k] at (current - 1)
            } for i in l and i < k
        }
        updateL(l, k, l_row) {
            do {    l[k][j] = u[k][j] / u[k][k]
                    u[k][j] = 0
                    l_row[j] = l[k][j]
            } for j in l and j > k
        }
        updateURowsBlock(u, l_row, k, row_range) {
            do {    u[i][j] = u[i][j] - l_row[j] * u[i][k]
```

```
67                  } for i, j in u and i > k and i <= row_range.max and j > k
68          }
69          updateUColsBlock(u, p_column, k, row_range) {
70                  do {    u[i][k] = u[i][k] - u[i][j] * p_column[j]
71                  } for i, j in u and i > row_range.max and j >= row_range.min and j < k
72          }
73
74          // three addditional stages are needed to prepare the blocks of updated upper and lower triangular
75          // matrices for the embedded SAXPY operation
76          collectLColParts(l_column, l, k, row_range) {
77                  do {    l_column[i] = l[i][k]
78                  } for i in l and i >= row_range.min and i < k
79          }
80          generatePivotColumn(p_column, l_column, row_range, k) {
81                  do { p_column[i] = l_column[i]
82                  } for i in l_column and i >= row_range.min and i < k
83          }
84          copyUpdatedLBlock(l_block, row_range, l) {
85                  do {    row = i - row_range.min
86                          l_block[row][j] = l[i][j]
87                  } for i, j in l and i >= row_range.min and i <= row_range.max and j > row_range.max
88          }
89
90          saxpy(u, u_block, l_block, row_range) {
91                  do {    do { total = reduce ("sum", u_block[i][m] * l_block[m][j])
92                          } for m in u_block
93                          u[i][j] = u[i][j] - total
94                  } for i, j in u and i > row_range.max and j > row_range.max
95          }
96  Computation:
97          Space A {
98                  Space B {
99                          prepareLU(a, u, l)
100                 }
101                 Repeat for r in a.dimension1.range step block_size {
102                         calculateRowRange(a, row_range, block_size)
103                         Repeat for k in row_range {
104                                 Space B {
105                                         Where k in u.local.dimension1.range { selectPivot(pivot, u, k) }
106                                 }
107                                 storePivot(p, k, pivot)
108                                 Space B {
109                                         Where k != pivot {
110                                                 Epoch { interchangeColumns(pivot, k, u, l) }
111                                         }
112                                         Where k in l.local.dimension1.range  { updateL(l, k, l_row) }
113
114                                         // only a block of rows are being updated from the upper triangular
115                                         // matrix per step instead of the entire matrix
116                                         updateURowsBlock(u, l_row, k, row_range)
117
118                                         collectLColParts(l_column, l, k, row_range)
119                                 }
120                                 generatePivotColumn(p_column, l_column, row_range, k)
121                                 Space B {
122                                         // similarly only a block of columns are updated per step from the
123                                         // upper triangular matrix
124                                         updateUColsBlock(u, p_column, k, row_range)
```

```
                                        }
                                }
                                // unaltered portion of the upper triangular matrix is updated using the SAXPY operation
                                // after the desired number of steps of partial LU Factorization
                                Space B {
                                        Where r in l.local.dimension1.range { copyUpdatedLBlock(l_block, row_range, l) }
                                        Space C {
                                                Repeat foreach sub-partition {
                                                        saxpy(u, u_block, l_block, row_range)
                                                }
                                        }
                                }
                        }
                }
        Partition(b):
                Space A <un-partitioned> {
                        a, p, l_column, l_row, p_column, l_block, u_block
                }
                Space B <1d> divides Space A partitions {
                        a<dim2>, u<dim1>, u_block<dim1>, l<dim1>, l_column: block_stride(b)
                        l_row, p_column, l_block: replicated
                }
                // cache friendly partitioning for the LPS where SAXPY operation takes place
                Space C <2d> divides Space B partitions {
                        u: block_size(b, b)
                        u_block: block_size(b), replicated
                        l_block: replicated, block_size(b)
                        Sub-partition <1d> <unordered> {
                                u_block<dim2>, l_block<dim1>: block_size(b)
                        }
                }
```

A.3 Conjugate Gradient on Random Sparse Matrix

```
1Program (args) {
2       // creating environment objects for component tasks
3       vaEnv1 = new TaskEnvironment(name: ``Vector Addition")
4       vaEnv2 = new TaskEnvironment(name: ``Vector Addition")
5       dpEnv = new TaskEnvironment(name: ``Vector Dot Product")
6       mvmEnv1 = new TaskEnvironment(name: ``CSR Matrix Vector Multiply")
7       mvmEnv2 = new TaskEnvironment(name: ``CSR Matrix Vector Multiply")
8
9       // make the argument sparse matrix stored in compressed row format from files to be read during first-time
10      // execution of the matrix-vector multiply task
11      bind_input(mvmEnv1, ``columns", args.arg_matrix_cols)
12      bind_input(mvmEnv1, ``rows", args.arg_matrix_rows)
13      bind_input(mvmEnv1, ``values", args.arg_matrix_values)
14
15      // bind the prediction (x_0) and the known vector (b)  to the tasks' environment that use them initially
16      bind_input(vaEnv1, ``u", args.known_vector)
17      bind_input(mvmEnv1, ``v", args.prediction_vector)
18
19      // run the conjugate gradient logic
20      // note that here iteration should continue until the estimate for solution vector converges to its actual
```

```
21          // value. That should happen if the residual error is zero. But we are doing an max-iterations based
22          // termination as we do not know if the symmetric, sparse matrix is positive-definite with a spectral radius
23          // less than 1
24          iteration = 0
25          maxIterations = args.maxIterations
26          do {
27                  // calculate A * x_i
28                  execute(``CSR Matrix Vector Multiply"; mvmEnv1; Partition: args.r)
29
30                  // determine the current residual error as r_i = b - A * x_i
31                  vaEnv1.alpha = 1
32                  vaEnv1.v = mvmEnv1.w
33                  vaEnv1.beta = -1
34                  execute(``Vector Addition"; vaEnv1; Partition: args.b)
35
36                  // determine the dot product of r_i to itself as the residual norm
37                  dpEnv.u = dpEnv.v = vaEnv1.w
38                  execute(``Vector Dot Product"; dpEnv; Partition: args.b)
39                  norm = dpEnv.product
40
41                  // in the first iteration setup duplicate environmental references for the sparse matrix components
42                  if (iteration == 0) {
43                          mvmEnv2.columns = mvmEnv1.columns
44                          mvmEnv2.rows = mvmEnv1.rows
45                          mvmEnv2.values = mvmEnv1.values
46                  }
47
48                  // determine A * r_i
49                  mvmEnv2.v = vaEnv1.w
50                  execute(``CSR Matrix Vector Multiply"; mvmEnv2; Partition: args.r)
51
52                  // determine dot product of r_i to A * r_i
53                  dpEnv.v = mvmEnv2.w
54                  execute(``Vector Dot Product"; dpEnv; Partition: args.b)
55
56                  // determine the next step size alpha_i as (r_i.r_i) / (r_i.(A * r_i))
57                  alpha_i = norm / dpEnv.product
58
59                  // calculate the next estimate x_i = x_i + alpha_i * r_i
60                  vaEnv2.u = mvmEnv1.v
61                  vaEnv2.alpha = 1
62                  vaEnv2.v = vaEnv1.w
63                  vaEnv2.beta = alpha_i
64                  execute(``Vector Addition"; vaEnv2; Partition: args.b)
65
66                  // prepare x_i for the next iteration
67                  mvmEnv1.v = vaEnv2.w
68                  iteration = iteration + 1
69
70          } while iteration < maxIterations and norm > args.precision
71
72          // store the final solution vector in an output file
73          bind_output(vaEnv2, ``w", args.solution_vector)
74  }
75
76  Task ``Vector Addition":
77          Define:
```

```
78              u, v, w: 1D Array of Real double-precision
79              alpha, beta: Real double-precision
80      Environment:
81              u, v, alpha, beta: link
82              w: create
83      Initialize:
84              w.dimension = u.dimension
85      Stages:
86              addVectors(w, u, v, alpha, beta) {
87                      do { w[i] = alpha * u[i] + beta * v[i] } for i in u
88              }
89      Computation:
90              Space A {
91                      addVectors(w, u, v, alpha, beta)
92              }
93      Partition(b):
94              Space A <id> {
95                      u, v, w: block_size(b)
96              }
97
98  Task ``Vector Dot Product":
99      Define:
100             u, v: 1D Array of Real double-precision
101             product: Real double-precision
102     Environment:
103             u, v: link
104             product: create
105     Stages:
106             computeDotProduct(result, u, v) {
107                     do { result = reduce(``sum", u[i] * v[i]) } for i in u
108             }
109     Computation:
110             Space B {
111                     computeDotProduct(Space A: product, u, v)
112             }
113     Partition(b):
114             Space A <un-partitioned> {u, v}
115             Space B <id> divides Space A partitions {
116                     u, v: block_size(b)
117             }
118
119 Task ``CSR Matrix Vector Multiply":
120     Define:
121             columns, rows: 1D Array of Integer
122             values, v, w: 1D Array of Real double-precision
123     Environment:
124             values, columns, rows, v: link
125             w: create
126     Initialize:
127             w.dimension = rows.dimension
128     Stages:
129             multiply(w, v, rows, columns, values) {
130                     start = rows.local.dimension1.range.min
131                     if (start == 0) { start = -1 }
132                     do {    if (i > 0) {
133                                     beginIndex = rows[i - 1] + 1
```

```
                        } else {
                                beginIndex = 0
                        }
                        endIndex = rows[i]
                        do {
                                w[i] = w[i] + values[j] * v[columns[j]]
                        } for j in columns and j >= beginIndex and j <= endIndex
                } for i in rows and i > start
        }
    Computation:
        Space A {
                multiply(w, v, rows, columns, values)
        }
    Partition(r):
        Space A <id> {
                values, columns, v: replicated
                rows: block_size(r) padding(1, 0)
                w: block_size(r)
        }
```

A.4 5-point Iterative Stencil

```
Program (args) {
    stencilEnv = new TaskEnvironment(name: ``Five Points Stencil")
    bind_input(stencilEnv, ``plate", args.input_file)
    execute(task: ``Five Points Stencil"; environment: stencilEnv;
                    initialize: args.iterations;                                    \
                    partition: args.k, args.l, args.m, args.n, args.p1, args.p2)    \
    bind_output(stencilEnv, ``plate", args.output_file)
}

Task ``Five Points Stencil":
    Define:
        plate: 2d Array of Real double-precision
        max_iterations: Integer
        counter_1, counter_2, counter_3: Integer
    Environment:
        plate: link
    Initialize (max_iterations):
    Stages:
        refineEstimates(plate) {
                localRows = plate.dimension1.range
                localCols = plate.dimension2.range
                do { plate[i][j] at (current)                                       \
                        = 1/4 * (plate[i-1][j]                                      \
                        + plate[i+1][j]                                             \
                        + plate[i][j-1]                                             \
                        + plate[i][j+1]) at (current - 1)
                } for i, j in plate                                                 \
                  and (i > localRows.min and i < localRows.max)                     \
                  and (j > localCols.min and j < localCols.max)
        }
    Computation:
```

```
32              // the whole computation should iterate for max_iterations number of times
33              Repeat for counter_1 in [1...max_iterations] {
34                      // after partition.p1 / partition.p2 upper level iterations the flow should exit for upper
35                      // level padding synchronization
36                      Space A {
37                              Repeat for counter_2 in [1...partition.p1] step partition.p2 {
38                                      // after partition.p2 iterations the flow should exit Space B for lower
39                                      // level padding synchronization
40                                      Space B {
41                                              Repeat for counter_3 in [1...partition.p2] {
42                                                      // epoch needs to be advanced after each refinement step
43                                                      Epoch {
44                                                              refineEstimates(plate)
45                                                      }
46                                              }
47                                      }
48                              }
49                      }
50              }
51      Partition (k, l, m, n, p1, p2):
52              Space A <2d> {
53                      plate: block_count(k, l) padding(p1)
54              }
55              Space B <2d> divides Space A partitions {
56                      plate: block_count(m, n) padding(p2)
57              }
```

A.5 Monte Carlo Area Estimation

```
1 Program (args) {
2       mcEnv = new TaskEnvironment(name: ``Monte Carlo Area Estimation")
3       execute(``Monte Carlo Area Estimation"; mcEnv;                                  \
4               Initialize: args.cell_length, args.grid_dim, args.points_per_cell;      \
5               Partition: args.b)
6 }
7
8 Class Rectangle:
9       top, bottom, left, right: Integer
10
11 Task ``Monte Carlo Area Estimation":
12      Define:
13              grid: 2D Array of Rectangle
14              sub_area_estimates: 2D Array of Real double-precision
15              cell_length, points_per_cell: Integer
16              cell_size: Real double-precision
17              area: Real double-precision Reduction
18      Environment:
19              area: create
20      Initialize(cell_length, grid_dim, points_per_cell):
21              grid.dimension1.range.min = 0
22              grid.dimension1.range.max = grid_dim - 1
23              grid.dimension2 = grid.dimension1
24              sub_area_estimates.dimension = grid.dimension
```

```
25          cell_size = cell_length * cell_length
26  Stages:
27          setupGridCells(grid, cell_length) {
28                  do {
29                          grid[i][j].left = cell_length * i
30                          grid[i][j].right = cell_length * (i + 1) - 1
31                          grid[i][j].bottom = cell_length * j
32                          grid[i][j].top = cell_length * ( j + 1) - 1
33                  } for i, j in grid
34          }
35          initiateRandGenerator() {
36                  @Extern {
37                          @Language "C++"
38                          @Includes { time.h, cstdlib }
39                          ${ srand(time(NULL)); }$
40                  }
41          }
42          estimateSubarea(cell_length, grid, sub_area_estimates, points_per_cell) {
43
44                  // iterate parallely over the grid cells
45                  do {
46                          cell = grid[i][j]
47                          internal_points = 0
48                          seed = lpuId[0]
49
50                          // undertakes points_per_cell number of point placement trails
51                          trial = 0
52                          do {
53                                  // generate a point within the cell boundary and calculate its position
54                                  // relative to the shape
55                                  @Extern {
56                                          @Language "C++"
57                                          @Includes { math.h, cstdlib }
58                                          ${
59                                                  int x = rand_r((unsigned int *) &seed) % cell_length + cell.left;
60                                                  int y = rand_r((unsigned int *) &seed) % cell_length + cell.bottom;
61
62                                                  // tested polynomial is 10 sin x^2 + 50 cos y^3
63                                                  double result = 10 * sin(pow(x, 2)) + 50 * cos(pow(y, 3));
64                                                  if (result <= 0.0) {
65                                                          internal_points++;
66                                                  }
67                                          }$
68                                  }
69                                  trial = trial + 1
70                          } while (trial < points_per_cell)
71
72                          // estimate the part of the polynomial within the grid cell
73                          sub_area_estimates[i][j] = cell_size * internal_points / points_per_cell
74
75                  } for i, j in grid
76          }
77          estimateTotalArea(result, sub_area_estimates) {
78                  do {
79                          result = reduce(``sum", sub_area_estimates[i][j])
80                  } for i, j in sub_area_estimates
81          }
82  Computation:
```

```
83              Space A {
84                      Space B {
85                              // calculate the cell boundaries based on cell dimension length and position in the grid
86                              setupGridCells(grid, cell_length)
87
88                              // initialize the random number generator for sampling
89                              initiateRandGenerator()
90
91                              // estimate the sub-area under the polynomial within each grid cell by a fixed number of
92                              // random samples
93                              estimateSubarea(cell_length, grid, sub_area_estimates, points_per_cell)
94
95                              // reduce the sub-area estimates into a single final result in Space A
96                              estimateTotalArea(Space A:area, sub_area_estimates)
97                      }
98              }
99      Partition(b):
100             Space A <un-partitioned> { grid }
101             Space B <id> divides Space A partitions {
102                     grid<dim1>, sub_area_estimates<dim1>: block_count(b)
103             }
```

B

Experimental Data

B.1 Hybrid Back-end: Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
Hand-written	10240	20.57 sec	0.83 sec	0 sec	11.83 sec
IT 1-GPU	10,240	125.8 sec	6.03 sec	0.3 sec	42.45 sec
IT 2-GPUs	10,240	62.89 sec	3.86 sec	0.32 sec	34.94 sec
IT 4-GPUs	10,240	31.45 sec	3.81 sec	0.3 sec	23.28 sec
IT 8-GPUs	10,240	15.72 sec	1.67 sec	0.33 sec	18.8 sec

Table B.1: Strong Scaling Timing Results of Block Matrix-Matrix Multiplication on Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
IT 1-GPU	10,240	125.79 sec	5.95 sec	0.31 sec	42.19 sec
IT 2-GPUs	12,902	126.62 sec	6.30 sec	0.31 sec	54.98 sec
IT 4-GPUs	16,255	125.97 sec	6.26 sec	0.32 sec	57.23 sec
IT 8-GPUs	20,480	125.77 sec	7.11 sec	0.31 sec	68.455 sec

Table B.2: Weak Scaling Timing Results of Block Matrix-Matrix Multiplication on Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
Hand-written	10,240	15.1 sec	0.7 sec	5.84 sec	0 sec
IT 1-GPU	10,240	85.3861 sec	261.315 sec	130.1919 sec	85.193 sec
IT 2-GPUs	10,240	42.5174 sec	129.7985 sec	112.5431 sec	58.269 sec
IT 4-GPUs	10,240	21.2178 sec	63.5662 sec	105.4520 sec	37.987 sec
IT 8-GPUs	10,240	10.5833 sec	32.1989 sec	101.2508 sec	30.907 sec

Table B.3: Strong Scaling Timing Results of Block LU Factorization on Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
IT 1-GPU	10,240	85.4046 sec	261.065 sec	129.2294 sec	85.412 sec
IT 2-GPUs	12,902	85.8069 sec	247.2143 sec	142.5728 sec	101.360 sec
IT 4-GPUs	16,255	85.0394 sec	256.612 sec	169.9956 sec	99.504 sec
IT 8-GPUs	20,480	84.7429 sec	254.733 sec	212.4211 sec	118.437 sec

Table B.4: Weak Scaling Timing Results of Block LU Factorization on Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
Hand-written	10,240	243.4 sec	0.21 sec	0 sec	0 sec
IT 1-GPU	10,240	245.248 sec	6.020438 sec	1.255562 sec	3.504 sec
IT 2-GPUs	10,240	124.431 sec	2.985667 sec	.768333 sec	2.031 sec
IT 4-GPUs	10,240	61.8975 sec	1.501463 sec	.554037 sec	1.3919 sec
IT 8-GPUs	10,240	30.7099 sec	.739322 sec	.690778 sec	1.2982 sec

Table B.5: Strong Scaling Timing Results of Monte Carlo Area Estimation on Big Red II

Program Version	Input Size	Kernel Execution	Data Transfer	Host Execution	Other Host Overhead
IT 1-GPU	10,240	245.299 sec	5.963885 sec	1.254115 sec	3.482 sec
IT 2-GPUs	14,482	249.348 sec	5.943742 sec	1.510258 sec	3.439 sec
IT 4-GPUs	20,480	246.413 sec	5.888004 sec	3.967996 sec	3.670 sec
IT 8-GPUs	28,964	246.494 sec	5.915936 sec	4.672064 sec	3.790 sec

Table B.6: Weak Scaling Timing Results of Monte Carlo Area Estimation on Big Red II

B.2 Segmented-memory Back-end: Rivanna Compute Cluster

Program Version	Input Size	Resource Preparation Overhead	Computation Time	Other Overhead
Hand-written	10,239	0 sec	1933.47 sec	0 sec
IT 20-Cores	10,239	24.6157 sec	83.5525 sec	0 sec
IT 100-Cores	10,239	14.4593 sec	16.6397 sec	0 sec
IT 500-Cores	10,239	6.49955 sec	3.34861 sec	0 sec
IT 1000-Cores	10,239	6.46373 sec	1.638 sec	0 sec

Table B.7: Strong Scaling Timing Results of Block Matrix-Matrix Multiplication on Rivanna

Program Version	Input Size	Computation Time	Resource Preparation Overhead
IT 20-Cores	10,240	201.907 sec	13.1619 sec
IT 40-Cores	12,902	203.153 sec	15.2231 sec
IT 80-Cores	16,255	204.013 sec	19.8148 sec
IT 160-Cores	20,480	201.53 sec	27.4318 sec
IT 320-Cores	25,804	203.236 sec	40.7699 sec
IT 640-Cores	32,508	203.914 sec	62.6531 sec

Table B.8: Weak Scaling Timing Results of Block Matrix-Matrix Multiplication on Rivanna

Program Version	Input Size	Resource Preparation Overhead	Computation Time	Other Overhead
Hand-written	20,480	0 sec	9671.27 sec	0 sec
IT 20-Cores	20,480	71.51628 sec	1141.984 sec	1.20172 sec
IT 40-Cores	20,480	35.83 sec	613.518 sec	1.588 sec
IT 80-Cores	20,480	18.048 sec	342.416 sec	1.392 sec
IT 160-Cores	20,480	9.005 sec	204.068 sec	1.7079 sec
IT 320-Cores	20,480	4.60 sec	136.38 sec	1.3638 sec
IT 640-Cores	20,480	2.392 sec	111.46 sec	1.006 sec
IT 960-Cores	20,480	1.942 sec	105.247 sec	0.745 sec

Table B.9: Strong Scaling Timing Results of Block LU Factorization on Rivanna

Program Version	Input Size	Computation Time	Communication Overhead	Resource Preparation Overhead
IT 20-Cores	10,240	175.349 sec	2.27343 sec	17.9142 sec
IT 40-Cores	12,902	123.638 sec	13.7526 sec	14.2393 sec
IT 80-Cores	16,255	119.847 sec	28.8984 sec	11.3341 sec
IT 160-Cores	20,480	155.833 sec	76.169 sec	9.04663 sec
IT 320-Cores	25,804	124.369 sec	82.3926 sec	7.2741 sec
IT 640-Cores	32,508	129.688 sec	135.719 sec	5.73743 sec

Table B.10: Weak Scaling Timing Results of Block LU Factorization on Rivanna

Program Version	Input Size	Iterations	Computation Time	Environment Management Overhead
Hand-written	320,000	1000	573.67 sec	0 sec
IT 1-Node	320,000	1000	467.687 sec	4.91 sec
IT 2-Nodes	320,000	1000	461.464 sec	337.57 sec
IT 4-Nodes	320,000	1000	459.224 sec	399.29 sec
IT 8-Nodes	320,000	1000	287.975 sec	342.11 sec
IT 16-Nodes	320,000	1000	146.265 sec	461.79 sec
IT 32-Nodes	320,000	1000	82.4862 sec	637.02 sec
IT 64-Nodes	320,000	1000	47.0628 sec	680.77 sec

Table B.11: Strong Scaling Results for Conjugate Gradient on Rivanna for a 99.95% Sparse Matrix

Program Version	Input Size	Iterations	Resource Preparation Overhead	Computation Time	Other Overhead
Hand-written	10,239	10,000	0 sec	2059 sec	0 sec
IT 20-Cores	10,239	10,000	13.12 sec	898.342 sec	9.354 sec
IT 100-Cores	10,239	10,000	2.77 sec	190.114 sec	2.638 sec
IT 300-Cores	10,239	10,000	0.48 sec	69.511 sec	0.751 sec
IT 500-Cores	10,239	10,000	0.615 sec	38.6915 sec	1.26 sec
IT 700-Cores	10,239	10,000	0.242 sec	26.093 sec	0.60 sec
IT 1000-Cores	10,239	10,000	0.296 sec	14.361 sec	1.1 sec

Table B.12: Strong Scaling Timing Results of 5-point Iterative Stencil on Rivanna

Program Version	Input Size	Iterations	Resource Preparation Overhead	Computation Time	Other Overhead
IT 20-Cores	10,240	10,000	6.54344 sec	593.084 sec	63.3062 sec
IT 80-Cores	20,480	10,000	6.86589 sec	595.204 sec	78.4019 sec
IT 180-Cores	30,720	10,000	6.89374 sec	596.716 sec	108.073 sec
IT 320-Cores	40,960	10,000	6.58999 sec	600.35 sec	109.621 sec
IT 500-Cores	51,200	10,000	6.58432 sec	593.632 sec	119.584 sec
IT 720-Cores	61,440	10,000	6.76767 sec	596.599 sec	119.182 sec

Table B.13: Weak Scaling Timing Results of 5-point Iterative Stencil on Rivanna

Program Version	Grid Dimension	Samples Per Cell	Resource Preparation Overhead	Computation Time
Hand-written	10,240	1000	0 sec	16,065.1 sec
IT 20-Cores	10,240	1000	1.08475 sec	824.56 sec
IT 40-Cores	10,240	1000	0.576924 sec	413.214 sec
IT 80-Cores	10,240	1000	0.362021 sec	206.942 sec
IT 160-Cores	10,240	1000	0.282253 sec	103.575 sec
IT 320-Cores	10,240	1000	0.295342 sec	51.9234 sec

Table B.14: Strong Scaling Timing Results of Monte Carlo Area Estimation on Rivanna

Program Version	Grid Dimension	Samples Per Cell	Computation Time	Resource Preparation Overhead
IT 20-Cores	10,240	1000	824.56 sec	1.08475 sec
IT 40-Cores	14,482	1000	826.441 sec	1.10838 sec
IT 80-Cores	20,480	1000	825.705 sec	1.16423 sec
IT 160-Cores	28,964	1000	856.2 sec	1.17591 sec
IT 320-Cores	40,960	1000	826.174 sec	1.30623 sec

Table B.15: Weak Scaling Timing Results of Monte Carlo Area Estimation on Rivanna

B.3 Multicore Back-end: Hermes Machines

Program Version	Input Size	Computation Time	Resource Preparation Overhead
Hand-written	10,239	2844.81 sec	0 sec
IT 1-Cores	10,239	2962.38 sec	0 sec
IT 4-Cores	10,239	767.838 sec	0 sec
IT 8-Cores	10,239	392.293 sec	0 sec
IT 32-Cores	10,239	113.778 sec	0 sec
IT 64-Cores	10,239	76.6735 sec	0 sec

Table B.16: Strong Scaling Timing Results of Block Matrix-Matrix Multiplication on Hermes

Program Version	Input Size	Computation Time	Resource Preparation Overhead
Hand-written	10,239	904.993 sec	0 sec
IT 1-Cores	10,239	948.612 sec	0 sec
IT 4-Cores	10,239	369.022 sec	0 sec
IT 8-Cores	10,239	205.249 sec	0 sec
IT 32-Cores	10,239	75.6677 sec	0 sec
IT 64-Cores	10,239	59.5962 sec	0 sec

Table B.17: Strong Scaling Timing Results of Block LU Factorization on Hermes

Program Version	Input Size	Iterations	Computation Time	Resource Preparation Overhead
Hand-written	320,000	1000	338.479 sec	0 sec
IT 1-Cores	320,000	1000	444.587 sec	0 sec
IT 4-Cores	320,000	1000	420.205 sec	0 sec
IT 8-Cores	320,000	1000	282.078 sec	0 sec
IT 32-Cores	320,000	1000	227.544 sec	0 sec
IT 64-Cores	320,000	1000	291.473 sec	0 sec

Table B.18: Strong Scaling Timing Results of Conjugate Gradient on Hermes for a 99.95% Sparse Matrix

Program Version	Input Size	Iterations	Computation Time	Resource Preparation Overhead
Hand-written	10,239	10,000	4211.06 sec	0 sec
IT 1-Cores	10,239	10,000	4609.13 sec	9.11768 sec
IT 4-Cores	10,239	10,000	1108.33 sec	9.99017 sec
IT 8-Cores	10,239	10,000	568.358 sec	9.11788 sec
IT 32-Cores	10,239	10,000	552.255 sec	9.59435 sec
IT 64-Cores	10,239	10,000	1270.21 sec	9.56709 sec

Table B.19: Strong Scaling Timing Results of 5-point Iterative Stencil on Hermes

Program Version	Input Size	Samples Per Cell	Computation Time	Resource Preparation Overhead
Hand-written	10,239	250	11,604.5 sec	0 sec
IT 1-Cores	10,239	250	11,351.167 sec	0 sec
IT 4-Cores	10,239	250	2,859.51 sec	0 sec
IT 8-Cores	10,239	250	1,441.92 sec	0 sec
IT 32-Cores	10,239	250	398.106 sec	0 sec
IT 64-Cores	10,239	250	280.963 sec	0 sec

Table B.20: Strong Scaling Timing Results of Monte Carlo Area Estimation on Hermes

References

[1] Blue gene - wikipedia. http://en.wikipedia.org/wiki/Blue_Gene. Accessed: 2014-08-19.

[Dir] Dirichlet boundary condition. https://en.wikipedia.org/wiki/Dirichlet_boundary_condition. Accessed: 2016-10-15.

[fnC] Function composition. https://en.wikipedia.org/wiki/Function_composition. Accessed: 2016-12-13.

[mir] Mira user guide. http://www.alcf.anl.gov/user-guides/mira-cetus-vesta. Accessed: 2014-08-19.

[Mon] Monte carlo method. https://en.wikipedia.org/wiki/Monte_Carlo_method. Accessed: 2016-10-15.

[ana] Nvidia's geforce gtx titan, part 1: Titan for gaming, titan for compute. http://www.anandtech.com/show/6760/nvidias-geforce-gtx-titan-part-1. Accessed: 2016-09-19.

[sta] Stampede user guide. https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide. Accessed: 2014-08-19.

[8] Stencil code. https://en.wikipedia.org/wiki/Stencil_code. Accessed: 2016-10-15.

[top] Tianhe-2 (milkyway-2). https://www.top500.org/featured/systems/tianhe-2/. Accessed: 2016-08-05.

[tit] Titan cray xk7. https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/. Accessed: 2014-08-19.

[11] Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[12] Alexandrov, A., Ionescu, M. F., Schauser, K. E., & Scheiman, C. (1995). Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation.

[13] Alpern, B., Carter, L., & Ferrante, J. (1993). Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings* (pp. 116–123).

[14] Alpern, B., Carter, L., & Ferrante, J. (1995). Space-limited procedures: a methodology for portable high-performance. In *Programming Models for Massively Parallel Computers, 1995* (pp. 10–17).

[15] Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.

[16] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., & Yelick, K. A. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical report, TECHNICAL REPORT, UC BERKELEY.

[17] Backus, J. (1978). Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8), 613–641.

[18] Bauer, M., Clark, J., Schkufza, E., & Aiken, A. (2011). Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. *SIGPLAN Not.*, 46(8), 13–24.

[19] Bauer, M., Treichler, S., Slaughter, E., & Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12 (pp. 66:1–66:11). Los Alamitos, CA, USA: IEEE Computer Society Press.

[20] Bell, C. G., Broadley, W., Wulf, W., Newel, A., Pierson, C., Reddy, R., & Rege, S. (1971). *Cmmp: The CMU Multi-mini-processor Computer*. Technical report.

[21] Blelloch, G. E. (1996). Programming parallel algorithms. *Commun. ACM*, 39(3), 85–97.

[22] Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., & Weatherford, S. (1994). Polaris: The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING* (pp. 10–1).: Springer-Verlag, Berlin/Heidelberg.

[23] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., & Dongarra, J. (2011). Dague: A generic distributed dag engine for high performance computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (pp. 1151–1158).

[24] Bryant, R. E. & O'Hallaron, D. R. (2010). *Computer Systems: A Programmer's Perspective*. USA: Addison-Wesley Publishing Company, 2nd edition.

[25] Butenhof, D. R. (1997). *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[26] Cappello, F. & Etiemble, D. (2000). Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00 Washington, DC, USA: IEEE Computer Society.

[27] Carriero, N. & Gelernter, D. (1989). Linda in context. *Commun. ACM*, 32(4), 444–458.

[28] Chakravarty, M. M., Keller, G., Lechtchinsky, R., & Pfannenstiel, W. (2001). Nepal – nested data-parallelism in haskell. In *IN EURO-PAR □01* (pp. 524–534).: Springer-Verlag.

[29] Chamberlain, B. (2015). *Programming Models for Parallel Computing*. The MIT Press.

[30] Chamberlain, B., Callahan, D., & Zima, H. (2007). Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3), 291–312.

[31] Chapman, B., Mehrotra, P., & Zima, H. (1992). Programming in vienna fortran. *SCIENTIFIC PROGRAMMING*, 1(1), 31–50.

[32] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., & Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10), 519–538.

[33] Cooper, K., Hall, M. W., Hood, R., Kennedy, K., McKinley, K., Mellor-Crummey, J., Torczon, L., & Warren, S. (1993). The parascope parallel programming environment. *Proceedings of the IEEE*, 81(2), 244–263.

[34] Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

[35] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., & von Eicken, T. (1993). Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93 (pp. 1–12). New York, NY, USA: ACM.

[36] Dagum, L. & Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1), 46–55.

[37] De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., & De Meuter, W. (2015). Partitioned global address space languages. *ACM Comput. Surv.*, 47(4), 62:1–62:27.

[38] Deelman, E., Singh, G., hui Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., & Katz, D. S. (2005). Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *SCIENTIFIC PROGRAMMING JOURNAL*, 13, 219–237.

[39] Dennis, J. B. (1980). Data Flow Supercomputers. *Computer*, 13(11), 48–56.

[Dongarra] Dongarra, J. Sparse matrix storage formats.

[41] Douglas, C. C. (1996). Multigrid methods in science and engineering. *IEEE Comput. Sci. Eng.*, 3(4), 55–68.

[42] El-Ghazawi, T. & Smith, L. (2006). Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06 New York, NY, USA: ACM.

[43] Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D., Leem, L., Park, J., Ren, M., Aiken, A., Dally, W., & Hanrahan, P. (2006). Sequoia: Programming the memory hierarchy. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (pp. 4–4).

[44] Feldman, J. A. (1979). High level programming for distributed computing. *Commun. ACM*, 22(6), 353–368.

[45] Feo, J. T., Cann, D. C., & Oldehoeft, R. R. (1990). A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10(4), 349–366.

[46] Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[Gehani & Roome] Gehani, N. H. & Roome, W. D. Concurrent c. *SOFTWARE–PRACTICE AND EXPERIENCE*, 16, 821–844.

[48] Gurd, J. (1985). The manchester dataflow machine. *Computer Physics Communications*, 37(1–3), 49 – 62.

[49] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5), 532–533.

[50] Hall, M., Anderson, J., Amarasinghe, S., Murphy, B., wei Liao, S., Bugnion, E., & Lam, M. (1996). Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12), 84–89.

[51] Halstead, Jr., R. H. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 501–538.

[52] Harris, T., Marlow, S., & Jones, S. P. (2005). Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05 (pp. 49–61). New York, NY, USA: ACM.

[53] Hinsen, K. (2009). The promises of functional programming. *Computing in Science Engineering*, 11(4), 86–90.

[54] Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., & Tseng, C.-W. (1992a). An overview of the fortran d programming system. In U. Banerjee, D. Gelernter, A. Nicolau, & D. Padua (Eds.), *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science* (pp. 18–34). Springer Berlin Heidelberg.

[55] Hiranandani, S., Kennedy, K., & Tseng, C.-W. (1992b). Compiling fortran d for mimd distributed-memory machines. *Commun. ACM*, 35(8), 66–80.

[56] Hoare, C. A. R. (1973). *Hints on Programming Language Design.* Technical report, Stanford, CA, USA.

[57] Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8), 666–677.

[58] Horel, T. & Lauterbach, G. (1999). Ultrasparc-iii: Designing third-generation 64-bit performance. *IEEE Micro*, 19(3), 73–85.

[59] Hull, M. E. C. (1987). Occam—a programming language for multiprocessor systems. *Comput. Lang.*, 12(1), 27–37.

[60] Karczmarczuk, J. (1999). Scientific computation and functional programming. *Computing in Science and Engineering*, 1(3), 64–72.

[61] Keltcher, C. N., McGrath, K. J., Ahmed, A., & Conway, P. (2003). The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2), 66–76.

[62] Kennedy, K., Koelbel, C., & Zima, H. (2007). The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III (pp. 7–1–7–22). New York, NY, USA: ACM.

[63] Kung, H. (2003). Systolic array.

[64] Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), 39–55.

[65] LUSK, E. & YELICK, K. (2007). Languages for high-productivity computing: The darpa hpcs language project. *Parallel Processing Letters*, 17(01), 89–102.

[66] MacNeice, P., Olson, K. M., Mobarry, C., de Fainchtein, R., & Packer, C. (2000). Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3), 330 – 354.

[67] McCarthy, J. (1978). History of lisp. *SIGPLAN Not.*, 13(8), 217–223.

[68] McGraw, J. R. (1982). The val language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1), 44–82.

[69] Meissner, L. P. (1982). The fortran programming language: Recent developments and a view of the future. *SIGPLAN Fortran Forum*, 1(1), 3–8.

[70] Mohr, M., Malony, A., Mohr, B., Beckman, P., Gannon, D., Yang, S., & Bodin, F. (1994). Performance analysis of pc++: A portable data-parallel programming system for scalable parallel computers. In *Proc. 8th Int. Parallel Processing Symb. (IPPS), Canc'un, Mexico, IEEE Computer* (pp. 75–85).: Society Press.

[71] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2), 40–53.

[72] Notkin, D., Snyder, L., Socha, D., Bailey, M. L., Forstall, B., Gates, K., Greenlaw, R., Griswold, W. G., Holman, T. J., Korry, R., Lasswell, G., Mitchell, R., & Nelson, P. A. (1988). Experiences with poker. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*, PPEALS '88 (pp. 10–20). New York, NY, USA: ACM.

[73] Numrich, R. W. & Reid, J. (1998). Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2), 1–31.

[74] Olukotun, K. (2014). Beyond parallel programming with domain specific languages. *SIGPLAN Not.*, 49(8), 179–180.

[75] Peña, A. J., Carvalho, R. G. C., Dinan, J., Balaji, P., Thakur, R., & Gropp, W. (2013). Analysis of topology-dependent mpi performance on gemini networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13 (pp. 61–66). New York, NY, USA: ACM.

[76] Pfister, G. F. & Norton, V. A. (1994). : chapter "Hot Spot" Contention and Combining in Multistage Interconnection Networks, (pp. 276–281). Los Alamitos, CA, USA: IEEE Computer Society Press.

[77] Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

[78] Rieger, C., Trigg, H., & Bane, B. (1981). Zmob: A new computing engine for ai. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81 (pp. 955–960). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[79] Schwartz, J. T. (1980). Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4), 484–521.

[80] Shewchuk, J. R. (1994). painless-conjugate-gradient.pdf (obiekt application/pdf).

[81] Siegel, H. (1979). A model of simd machines and a comparison of various interconnection networks. *Computers, IEEE Transactions on*, C-28(12), 907–917.

[82] Snyder, L. (1986). : chapter Type Architectures, Shared Memory, and the Corollary of Modest Potential, (pp. 289–317). Palo Alto, CA, USA: Annual Reviews Inc.

[83] Steele, GuyL., J., Allen, E., Chase, D., Flood, C., Luchangco, V., Maessen, J.-W., & Ryu, S. (2011). Fortress (sun hpcs language). In D. Padua (Ed.), *Encyclopedia of Parallel Computing* (pp. 718–735). Springer US.

[Stitt] Stitt, T. An introduction to the partitioned global address space (pgas) programming model.

[85] Stone, J. E., Gohara, D., & Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3), 66–73.

[86] Trinder, P., w. Loidl, H., & Pointon, R. (2002). Parallel and distributed haskells.

[87] Turing, A. M. (1948). Rounding-off errors in matrix processes. 1 (part 3), 287–308.

[88] Turner, D. A. (1981). The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81 (pp. 85–92). New York, NY, USA: ACM.

[89] Veen, A. H. (1986). Dataflow machine architecture. *ACM Comput. Surv.*, 18(4), 365–396.

[90] von Laszewski, G., Parashar, M., Mohamed, A. G., & Fox, G. C. (1992). On the parallelization of blocked lu factorization algorithms on distributed memory architectures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92 (pp. 170–179). Los Alamitos, CA, USA: IEEE Computer Society Press.

[91] Walker, D. W., Walker, D. W., Dongarra, J. J., & Dongarra, J. J. (1996). Mpi: A standard message passing interface. *Supercomputer*, 12, 56–68.

[92] Weiss, S. & Smith, J. E. (1994). *IBM Power and PowerPC*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

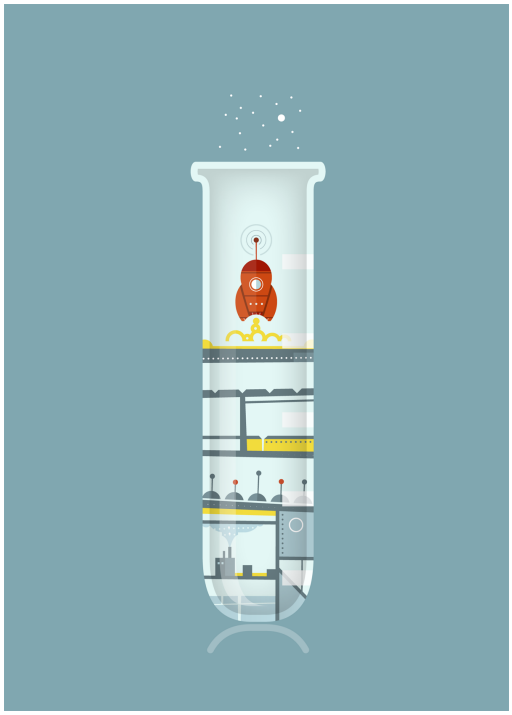
[93] Wilhelm, R., Alt, M., Martin, F., & Raber, M. (1997). Parallel implementation of functional languages. In M. Dam (Ed.), *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science* (pp. 279–295). Springer Berlin Heidelberg.

[94] Yanhaona, M. N. & Grimshaw, A. S. (2014). *The Partitioned Parallel Processing Spaces (PCubeS) Type Architecture*. Technical report, Technical Report, UVA Computer Science.

[95] Yanhaona, M. N. & Grimshaw, A. S. (2015). A roadmap for a type architecture based parallel programming language. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, COSMIC '15 (pp. 7:1–7:10). New York, NY, USA: ACM.

[96] Yanhaona, M. N. & Grimshaw, A. S. (2016). *IT: Machine Independent Programming on Hierarchically Nested Machines*. Technical report, Technical Report, UVA Computer Science.

[97] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., & Aiken, A. (1998). Titanium: A high-performance java dialect. In *In ACM* (pp. 10–11).



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, *Science Experiment 02*, was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.