


HighP5: Programming using Partitioned Parallel Processing Spaces

Muhammad Nur Yanhaona   [Brac University | nur.yanhaona@bracu.ac.bd]

Andrew Grimshaw  [University of Virginia | grimshaw@virginia.edu]

Shahriar Hasan Mickey  [Brac University | shahriar.mickey@bracu.ac.bd]

Received: 03 April 2024 • Accepted: 06 June 2024 • Published: 17 December 2024

Abstract HighP5 is a new high-level parallel programming language designed to help software developers to achieve three objectives simultaneously: programmer productivity, program portability, and superior program performance. HighP5 enables this by fostering a new programming paradigm that we call hardware-cognizant parallel programming. The paradigm uses a uniform hardware abstraction and a declarative programming syntax to allow programmers to write hardware feature-sensitive efficient programs without delving into the detail of those feature implementations. This paper is the first comprehensive description of HighP5's design rationale, language grammar, and core features. It also discusses the runtime behavior of HighP5 programs. In addition, the paper presents preliminary results on program performance from HighP5 compilers on three different architectural platforms: shared-memory multiprocessors, distributed memory multi-computers, and hybrid GPU/multi-computers.

Keywords: Parallel Programming, Programming Language, Type-Architecture, Declarative Programming, Portability, Productive Computing

1 Introduction

After several decades of parallel computing, the dominant parallel programming method is still low-level, platform-specific parallelization extensions (e.g., PThreads [Butenhof, 1997], MPI [Walker *et al.*, 1996], CUDA [Nickolls *et al.*, 2008]) to sequential languages such as C and FORTRAN. Frequently, the so-called ‘bookkeeping’ aspects of parallel computing, such as data decomposition, communication, synchronization, and load balancing, become a program’s dominant parts when written in these languages. A lack of language-level features to systematically deal with these aspects makes programming using these extensions challenging. In particular, eking out good performance becomes a laborious effort. Furthermore, paradigmatic differences between these low-level extensions do not allow code written for one hardware platform to port in another. Porting (and re-optimizing) parallel programs from platform to platform consumes additional time, money, and creative energy.

There are two conflicting elements at the heart of the above programming challenges. First, efficient parallel programming requires making the best use of the features of the target execution platforms. Second, these execution architectures are so distinct that low-level parallelization tools that enable their best use become necessarily different. Data decomposition, communication, synchronization, and load balancing implementations in a distributed memory environment that MPI targets are quite different from those in a multicore/multiprocessor CPU where threading dominates or in GPGPUs where we use CUDA.

High-level parallel programming tools or languages that hide the ‘bookkeeping’ aspects of programs, such as recent PGAS languages [El-Ghazawi and Smith, 2006] [Yelick *et al.*, 1998] [Charles *et al.*, 2005], OpenMP [Dagum and Menon, 1998], and OpenCL [Stone *et al.*, 2010], have the

problem of not generalizing across hardware architectures. Furthermore, there is a limit to how much a language compiler or library runtime can optimize the machine code without some directions from the programmer [Mallón *et al.*, 2009]. In his seminal work, ‘Type architectures, shared memory, and the corollary of modest potential,’ [Snyder, 1986] Snyder introduces the notion of type architecture as an interface that describes the hardware. As Snyder shows, unless the type architecture (and the programming model) exposes the relevant architectural features, a programmer’s algorithmic design decisions may unwittingly lead to an inefficient program. Furthermore, the compiler cannot bridge the gap if there is no efficient mapping of the programming model to the execution platform.

The recent decades-long trend toward deep memory hierarchies and hybrid architectures further intensifies the conflict between high performance, productivity, and portability. High-performance programming is challenging in present-day architectures, even using low-level parallelization primitives. The Von Neumann flat-memory architecture assumption of the underlying sequential languages forces MPI, PThreads, and CUDA programmers to apply hardware-based optimizations in sophisticated and ad-hoc manners [Merrill *et al.*, 2012] [Merrill and Garland, 2016]. The prominence of hybrid architectures in large-scale parallel computers then requires combining these ad-hoc optimizations for multiple parallelization primitives, which makes achieving performance at scale even more difficult [Pennycook *et al.*, 2011] and often leaves codes in a messy shape. Due to the pressing need for high performance, productivity and portability have become an afterthought and are rarely achievable in that way.

We believe that high performance is only achievable by writing programs sensitive to the execution platform’s characteristics. However, the programming model should ex-

pose the hardware features over a generic type-architecture interface to enable portability with high performance. Meanwhile, the language should make it easy to exploit the hardware features to ensure productivity and performance debugging. *HighP5* is the realization of this vision. It offers an alternative programming paradigm having the above characteristics. We call this programming paradigm hardware-cognizant. Hardware-cognizant programming is a middle-ground between hardware-dependent programming using low-level parallelization primitives and hardware-agnostic programming with high-level parallel programming languages. This new programming paradigm retains the benefits of both camps.

There are two components of HighP5's hardware-cognizant programming paradigm: a novel type architecture called *Partitioned Parallel Processing Spaces (PCubeS)* and a declarative language (also called HighP5). The PCubeS type architecture represents a machine as a hierarchical layering of *physical processing spaces (PPSs)*. Each PPS consists of *physical processing units (PPUs)* with memory capacity, computing capacity (FLOPS), intra-layer communication bandwidth and latency, and inter-layer communication bandwidth and latency. Each layer has overall memory, computing, and communication capabilities, which is the sum of its PPU capacities. PCubeS reflects present-day multi-layered (possibly heterogeneous) machines and is an excellent basis for revolutionizing parallel computing.

PCubeS engenders a programming model of multi-layered memory spaces reflecting the memory available in the PPUs of the execution platform. In HighP5, programmers associate data structures with one or more *logical processing spaces (LPSes)* and specify how to decompose or partition each data structure into segments (*LPU*s) where it is defined. Computations are also associated with LPSes and can operate on local data structures in respective LPUs. Moving data between spaces and re-partitioning data is a potentially expensive operation whose cost is evident to the programmer. The model combines coarse-grain task parallelism with data-parallel constructs expressing computations happening in multiple LPSes to simultaneously achieve high performance, productivity, and portability in four ways.

First, HighP5 has a declarative syntax where the programmer states what is to be computed but not how. This strategy gives the compiler the freedom to implement the semantics of the program using techniques that properly distribute work to processors and improve cache and memory bandwidth utilization. A declarative syntax also leads to a cleaner code that looks more like the underlying mathematical equations. The declarative syntax inspires productivity by allowing programmers to focus on what they want to accomplish rather than how to efficiently achieve it on many different architectures.

Second, HighP5 is a PCubeS language. By providing a programming model that closely maps to existing architectures and exposes salient architectural features and costs, HighP5 simplifies the task of writing performant programs. By highlighting the architectural costs in language structures, HighP5 permits the programmer to understand the performance ramifications of design choices, making it easier to write fast applications.

Third, HighP5 syntax separates program definition and semantics from mapping code and data structures to physical hardware components. This separation of concerns significantly enhances program portability. All that needs to change to port a program to a new architecture is to generate a new mapping file and recompile it. Similarly, the programmer may experiment with the performance ramifications of moving computations from one layer of the hardware to the other, e.g., from the cores in a multicore to the warps of a GPU, simply by changing the mapping file. No re-coding is required.

Finally, the HighP5 compiler ensures that all data accesses will be local. In other words, if programmers map an LPS to a particular cache level or SM on a GPU, the compiler will break up the computation into corresponding pieces and ensure that each piece's data element will fit into the specified hardware layer. The data staging happens on-demand at runtime, with proper indexes and loop boundary transformation of the declarative code. This mechanism improves performance by exploiting the performance benefits using the closest, fastest memory.

Preliminary HighP5 compilers exist for three different architecture types: shared memory multicore UMA/NUMA machines, shared memory multicore nodes connected by high-speed networks (i.e., distributed shared memory machines), and both of the above with embedded NVIDIA GPUs (i.e., hybrid architectures). Compiler support for diverse architectures is critical to supporting program portability. We claim that HighP5's approach addresses *Seven Ps*: writing portable, people-productive, predictable, and performant parallel programs. Specifically:

1. HighP5 programs port without modification from platform to platform.
2. HighP5 programs can achieve performance competitive with hand-written hardware-specific codes.
3. HighP5 programs are shorter and less complex than their C/(MPI, OpenMP, CUDA) counterparts.

We will present preliminary results that support these claims.

The organization of the rest of the paper is as follows. Section 2 does a brief survey of the related work in parallel programming. Section 3 delves into the detail of type architecture and describes PCubeS with examples. Section 4 explains parallel programming in HighP5 using program structure discussion and examples. Section 5 describes some core language features. Section 6 discusses the runtime behavior of a HighP5 program and illustrates how a generic runtime environment for HighP5 can facilitate efficient program execution in various PCubeS architectures. Section 7 presents preliminary performance results of some example HighP5 programs in three different parallel platforms and compares the performance with efficient hand-written code. Finally, Section 8 concludes the paper with a discussion of future language improvement and research directions.

2 Related Work

This section briefly surveys some dominant parallel programming models, primitives, and languages for various ar-

chitectural platforms of present and earlier days. It is not a comprehensive survey of the landscape of parallel programming tools and languages. The goal here is to illustrate how HighP5 differs from the existing modes of parallel computing. Therefore, the section ends with a discussion of HighP5's uniqueness and the advantage of hardware-cognizant programming that it promotes.

Parallel Programming on Multicore Machines

Intel's Cilk Plus [Blumofe *et al.*, 1995], followed by Open Cilk [Schardl *et al.*, 2018], is a C/C++ extension supporting fork-join task parallelism and vector operations on multicore CPUs. Initially designed for high-performance computing, it became a more general-purpose language. The programmer is responsible for identifying and exploiting parallelism using various features such as spawn, sync, parallel loop, and array extensions. The runtime engine decides how to distribute tasks to processors and vectorize array operations. Pthreads [Butenhof, 1997] is a popular library extension on C and Fortran for task-level parallelism. Pthreads has thread creation, joining, termination routines, and support for different forms of explicit synchronization of those threads. One can fine-tune the performance of a Pthreads program using programmatic thread affinity management, cache blocking, and combining NUMA memory allocation libraries in NUMA architectures. Pthreads is a low-level but flexible primitive. OpenMP [Dagum and Menon, 1998] [Chapman *et al.*, 2007] takes a very different route for parallelization from Cilk and Pthreads. OpenMP parallelizes a sequential C or Fortran program using OpenMP pragma constructs. There are parallel pragmas for loops, code blocks, and reductions with few attributes controlling their execution. Achieving good portable performance can be challenging with OpenMP unless the programmer explicitly tunes the code to the underlying physical architecture. In our experience, getting OpenMP code to run is easy; getting it to run fast is the trick.

Although programs using these tools and languages often achieve significant performance boost over their sequential bases through concurrency and vectorization, a lack of exposure to memory/cache hierarchies limits their efficiency. Furthermore, portability to other execution architectures, such as distributed-memory machines or accelerators, is not a concern for them.

Parallel Programming on Distributed Memory Machines

The Message Passing Interface (MPI) [Walker *et al.*, 1996] has been the standard for parallel programming in distributed memory for several decades. It implements Hoare's communicating sequential processes (CSP) [Hoare, 1978] model for distributed memory programming. All process-to-process interactions in MPI happen through pair-to-pair or collective communication. There is virtually no supercomputer and compute cluster that does not support MPI now. However, MPI programming is known to be complex and error-prone. Nevertheless, MPI remains popular because of its ability to perform well on purely distributed machines. The advent of

multicore CPUs and accelerators in supercomputers and compute clusters limit MPI's supremacy. MPI's treatment of the execution environment as a flat collection of processes with uniform characteristics no longer holds. There has been an effort to combine MPI with OpenMP, Pthreads, and other multicore parallelization tools [Rabenseifner *et al.*, 2009]. Unfortunately, this strategy makes programming even more laborious and more error-prone. Therefore, programmer productivity and difficulty of performance debugging is a major problem with MPI and its hybrids.

One can view the DARPA high productivity languages X10 [Charles *et al.*, 2005] and Chapel [Chamberlain *et al.*, 2007]; or PGAS languages such as Co-array Fortran [Numrich and Reid, 1998], UPC [El-Ghazawi and Smith, 2006], UPC++ [Bachan *et al.*, 2017], and Titanium [Yelick *et al.*, 1998] as efforts to find an alternative to MPI for flexible and expressive parallel programming paradigm without losing performance. These languages have a notion of local and remote memories and support operations over data structures residing on any of them. Both X10 and Chapel have numerous parallelization constructs over this basic foundation of a Partitioned Global Address Space (PGAS). Because of their flat memory partitioning, these languages suffer from the same performance problem in hierarchical and hybrid architectures as MPI. Given that they are not clear winners against MPI on Non-uniform Cluster Computers where PGAS [Shan *et al.*, 2012] [Blagojević *et al.*, 2010] is applicable, their future success in machines with deep memory hierarchies and non-uniform compute capacities is uncertain.

MPI and PGAS languages have the advantage that programs written in them are portable across multicore/multiprocessor CPUs and distributed memory machines, as one can treat a shared-memory environment as a distributed memory environment.

Languages for Co-processors

CUDA [Nickolls *et al.*, 2008] is the programming tool used for NVIDIA GPGPUs. Initially, it was deemed difficult, but GPGPUs popularity in high-performance computing gave it a rapid surge in acceptance. CUDA follows the footsteps of earlier SIMD/SPMD languages (e.g., pC++ [Mohr *et al.*, 1994], C** [Larus, 1993], DataParallel C [Adve *et al.*, 1994]) 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 instructions for data transfers between a CPU host and accompanying accelerators. Programmers need to understand the inner workings of the accelerator threads, programmable cache configuration, and efficient memory access patterns details to make their offloading functions behave correctly and efficiently.

The OpenCL [Stone *et al.*, 2010] standard has a strong CUDA heritage and originally targeted accelerator platforms exclusively. It has recently become more of a standard for parallel programming for multicore and GPUs. We believe that taking a special-purpose model and applying it to general-purpose computing has limitations. The model will be too complex if it allows all special-purpose attributes to retain efficiency in accelerators that are hardly useful in

general-purpose architectures. Otherwise, the model must discard its key special-purpose features and become inefficient.

There are some high-level alternatives to OpenCL for general-purpose GPU computing. For example, Futhark [Henriksen *et al.*, 2017] is a purely functional data-parallel language for GPU programming that generates efficient executables for AMD and NVIDIA GPU platforms. The language defines parallel operations on arrays such as map, reduce, and scan that the compilers combine and transform to generate OpenCL code.

A general problem with the accelerator programming models is that they do not model the cost of communication that dominates a program’s execution time in large-scale networked architectures.

Multi-Platform Languages

Julia is a dynamically typed programming language that has recently become popular for scientific computing on various platforms [Novosel and Slivnik, 2019] [Lin and McIntosh-Smith, 2021]. It is a multi-paradigm language in the sense that it combines elements from imperative, functional, and object-oriented programming. Julia supports multithreading for shared-memory computing and CSP-style distributed-memory parallel programming where processes interact using remote procedure calls. Recently Julia has added support for GPU programming on NVIDIA GPGPUs by providing extension libraries specifically designed to translate into efficient CUDA codes [Besard *et al.*, 2019]. Although it is possible to write Julia programs for different architectural platforms, the structure of the program and its elements differ depending on the target platform. Furthermore, the language itself does not facilitate reasoning about hardware features. Instead, the programmers rely on standard libraries for high performance and internalize performance-enhancing tricks specific to the language.

Julia’s approach to tackling architectural differences with different programming styles within a single language is comparable to the parallel programming features of the popular functional programming language Haskell. Haskell has parallel, distributed, and accelerator-offloading computation facilities as embedded sub-languages inside it [Chakravarty *et al.*, 2011] [Mainland and Morrisett, 2010] [Mainland and Morrisett, 2010]. Programmers use appropriate features depending on the target execution platform. The efficiency of the program largely depends on the efficiency of the feature implementations within those sub-languages. Therefore, programmers do not necessarily understand the execution time behavior of their program, and the same program may not be portable across platforms despite being written in the same language.

Julia and Haskell’s approach to supporting multiplatform parallel programming is comparable with hybrid MPI+OpenMP [Rabenseifner *et al.*, 2009], MPI+CUDA [Zhang *et al.*, 2023], or Hybrid OpenCL [Aoki *et al.*, 2011]. The idea is to allow programmers to use hardware-specific constructs in their programs without providing them with a single cohesive framework for parallel programming in arbitrary execution platforms. Consequently, programmer pro-

ductivity is compromised for the scheme’s portability across platforms, and achieving good performance is not necessarily easy.

Type Architecture Based Languages

Few languages focus on a type-architecture foundation as HighP5 does. Among them are ZPL [Chamberlain *et al.*, 2000] and Legion [Bauer *et al.*, 2012]. ZPL used Snyder’s original CTA type architecture [Snyder, 1986] and performed well on Cray machines. In our opinion, CTA is unsuitable for most hybrid and hierarchical parallel machines as it describes a parallel architecture as a connected network of uniform processors without further characterizing them. Legion is a functional language follow-on of Sequoia [Bauer *et al.*, 2011] that uses a type architecture called Parallel Memory Hierarchy (PMH) [Alpern *et al.*, 1993]. Surprisingly, the type-architecture aspect is not a highlight in Legion. Like HighP5, Legion supports hierarchical partitioning and programmer-controlled mapping of those partitions to different hardware layers. We suspect its fully-flexible nature of partitions and list-based language model is inappropriate for common high-performance computing problems that thrive on regularity and are replete in data parallelism. The PMH model has the additional problem of only supporting data movement up and down the memory hierarchy, which is not the case for many present-day architectures.

Performance Portability Programming Models

In recent years, researchers have proposed several programming models for addressing the portability problem of existing C and Fortran code across diverse hardware architectures [Marowka, 2022]. These models vary in their approach to the problem. Some are library-based, some work as a middle-layer, while some provide higher abstractions.

RAJA [Beckingsale *et al.*, 2019] is a C++ library based portability solution. Here programmers specify parallel loop kernels. Then against these loop kernels, they define an execution policy (seq, OpenMP, or CUDA). A notion of iteration spaces allow controlling index distributions of arrays and programmers write code snippets (called traversal templates) to be performed inside loop bodies based on specified execution policies and index distributions.

OpenACC [Herdman *et al.*, 2014] is a solution similar to OpenCL, but it does not require architectural knowledge of accelerators to the extent the latter does. OpenACC focuses on easy portability across accelerator-platforms using a pragma-based approach similar to the popular OpenMP for CPUs. In OpenACC, programmers place parallel and kernel directives/pragmas on C++ and Fortran codes that OpenACC translates into low-level accelerator code for NVIDIA CUDA, AMD GCN, and Intel Mic platforms.

StarPU [Augonnet *et al.*, 2009] offers a unified execution model for units of heterogeneous architecture involving manycore CPUs and GPUs. In StarPU, programmers define high-level tasks and task schedulers that submit those tasks to a pool of workers. Here the resources of the target heterogeneous machine serve as workers that fetch pending tasks and

execute them. A data management layer automates transfer of data throughout these workers.

KoKKOS [Edwards and Trott, 2013] is probably the most extensive among all the performance portability models and has the most significant community support. KoKKOS is a library and embedded language based solution. It has range of primitives for computation and data modeling such as execution spaces for mapping computation to hardware resources, execution patterns for parallel Instructions, execution policies for index distribution of arrays, memory layout for data assignment, and memory spaces for storage specification. Furthermore, advanced features such as multi-dimensional parallel index iteration, scratch memory, and vector parallelism are supported [Trott *et al.*, 2022]. However, hardware consideration is ad-hoc – not an integral part of the programming model – and all features do not have mapping in all execution platforms.

Domain-Specific Parallel Languages

Finally, domain-specific languages (DSL) and toolkits are a persistent trend in high-performance parallel programming [MacNeice *et al.*, 2000]. A DSL grows from a general language that 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 [MacNeice *et al.*, 2000] [Olukotun, 2014]. The performance of DSL programs is often competitive with general-purpose language implementations. Nevertheless, as their nature suggests, they cannot be the general solution for high-performance parallel computing.

Uniqueness and advantages of HighP5

The first noticeable difference between HighP5 and other existing parallel programming approaches is that HighP5 lets programmers systematically reason about the features of deeply hierarchical and complex machine architectures using a uniform type architecture interface. Therefore, predictable and high performance is achievable without sacrificing productivity. Then the critical aspect of portability using HighP5 is that the same program ports across hardware architectures without losing efficiency, which is quite different from enabling a language or tool to work across execution platforms. Finally, HighP5’s declarative syntax and clear separation of various parallel programming aspects enhance the readability of efficient codes and foster learning parallel programming in general. These are issues targeted by only a few other earlier paradigms, such as NESL [Blleloch, 1992].

3 Type Architecture and Programming Model

In this section, we first describe some type-architecture fundamentals as it is a relatively new concept to many readers, then discuss how the lack of it affects parallel programming, then explain the PCubeS type-architecture that HighP5

uses and provide PCubeS descriptions of some example execution platforms. The section concludes with a discussion of HighP5’s abstract machine model and how it aligns with PCubeS type architecture.

3.1 Type Architecture Fundamentals

The vast capacities of present-day massively parallel architectures may appear impressive, but historically their computational power lags behind the demand made by their contemporary applications. Most interesting scientific applications are quadratic or above in their runtime complexity. Therefore, we can achieve only a modest improvement in problem size and running time through a linear increase of processors that parallelism offers.

Snyder, in his 1986’s seminal article [Snyder, 1986], 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 keep up, even modestly, with the growing capacity demand of contemporary applications. He argues for a hardware-sensitive programming paradigm for writing parallel applications and introduces the notion of a Type Architecture.

According to Snyder, writing a program is 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. The performance of the program relies on the efficiency of both steps. Suppose 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 communication medium between him/her and the compiler. The abstraction can be prohibitively expensive depending on how the language defines it, making the programmer work against it to make a program efficient. Or, it can be so low-level that it allows programming for only a specific class of hardware.

There is a rift between high and low-level programming techniques regarding their underlying abstract machine models’ role. Most high-level languages present a simple abstract machine model of the execution environment to simplify coding and enhance the portability of written codes. However, that model provides nearly no guidance for efficiently exploiting the features of the execution environment. Thereby, getting good program performance in high-level languages is often a challenge. On the other hand, low-level programming techniques make the programmer deal with even minute details of the execution environment. Their abstract machine model is effectively the bare hardware. Hence efficiency is attainable 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. The interface should be 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 hardware facilities. The type-architecture description should bear

the following two characteristics to be effective.

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

Understanding the distinction between the type architecture and the abstract machine model of a programming language is essential. For example, the Von Neumann Architecture can be considered a type architecture for sequential machines. FORTRAN [Chapman *et al.*, 1992] and Lisp [McCarthy, 1978] 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 translate and perform in an execution platform. Successful programmers internalize the cost of model-to-architecture translation and choose their primitives and operations accordingly to write an efficient program.

Several type architecture proposals [Stone *et al.*, 2010] [Alexandrov *et al.*, 1995] [Culler *et al.*, 1993] [Alpern *et al.*, 1993] have appeared in literature since Snyder’s original proposal of CTA [Snyder, 1986] as a candidate type architecture for parallel machines. However, they either fail to generalize or lack an accompanying programming model that can address the challenges of contemporary parallel programming.

3.2 The Consequence of Lacking a Type Architecture

A mismatch between the assumptions of the abstract machine model of a parallel programming paradigm and the actual behavior of the features of the execution platform can lead to severe performance degradation. For example, programming in a non-uniform shared memory machine using a uniform shared memory programming paradigm may result in significant performance loss due to memory access inefficiency. Similarly, a distributed memory programming paradigm that treats communications among processes as pure information exchanges devoid of any performance characteristics can be grossly inefficient as it leaves the programmer in the dark about the proper choice of message size, nature, and frequency for a particular platform.

The ability to fully control the communication characteristics of a program is one of the most critical factors behind the success of the current de-facto standard of parallel computing, the Message Passing Interface (MPI). However, without a clear type-architecture foundation, writing good MPI codes often becomes equivalent to knowing a lot of ad-hoc performance tweaking. Therefore, it can be difficult for an average programmer to write high-performing MPI programs for typical building-block scientific problems. This problem was already evident when most supercomputers were plain distributed memory machines. The advent of accelerators and multicore CPUs has made this problem only more intense in recent decades.

The stakes are always high for extracting good performance from machines in any manner possible. Therefore

the standard practice in high-performance computing has become to “program against the machine [Snyder, 1986].” MPI has been coupled with various shared memory and offloading computing models, such as Pthreads, OpenMP, CUDA, and Cilk, violating MPI’s model of communicating sequential processes. To clarify this point through an example, if two different MPI processes running independently on two CPU cores vie for the same GPU resource during offloading computations using CUDA, the model of isolated processes falls apart. In such a situation, the programmer must rely on intuition rather than the machine model to determine what should run efficiently and what not.

A hybrid programming model is significantly more challenging for a typical programmer than a singular, holistic language alternative regarding reading, writing, and debugging a program. Unfortunately, that covers almost all aspects of programming. Furthermore, a hybrid model is particularly antagonistic to the portability of knowledge. To elaborate on the last point, programmers versed in C can easily rewrite their programs in FORTRAN once they learn the latter’s syntax. Their original programs were not portable, but their underlying logic is. Regrettably, programmers knowing how to write good MPI + Pthreads hybrid programs, on the other hand, may miserably fail if given an MPI + CUDA platform. This problem happens because, unlike the first sequential programming example, the peculiarities of underlying execution platforms are closely tied with the behavior of low-level primitives provided by Pthreads and CUDA in the second case. In other words, in the latter example, learning a different syntax and a new, entirely different abstract machine model is required. Thus the need for a unifying modeling framework, a facility description standard, aka a type architecture, is badly felt here.

Note that recent high-level PGAS [De Wael *et al.*, 2015] languages such as Chapel [Chamberlain *et al.*, 2007] or X10 [Charles *et al.*, 2005] and parallel extensions to sequential languages such as Co-array Fortran [Numrich and Reid, 1998] and UPC [El-Ghazawi and Smith, 2006] also need convincing type architecture foundations. Without a type architecture, features like ‘places,’ ‘locations,’ and so on that they propose for hardware-sensitive reasoning become abstract concepts for grouping program segments. They do not guide programmers toward efficient program design.

3.3 PCubeS Type Architecture

The formal description of HighP5’s 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.

PCubeS differs from Snyder’s notion of type architecture in two ways that are fundamental to its efficacy in describing modern hardware and its usefulness as the interface for reasoning about them.

1. **Programmability:** Snyder uses the term ‘structural feature’ and ‘facility’ interchangeably as he was more concerned about describing hardware features than their typical use in a program. We believe this approach of ‘describe first then derive programming models’ is incorrect as the type architecture’s purpose is to standardize the hardware-language interface and an interface design has to consider both sides’ demands. Therefore, PCubeS descriptions focus on the programmatic usage of hardware’s structural features rather than their actual working principles. For example, PCubeS cares not if a vector implementation is a pipeline or a SIMD lane. That the described hardware has vector computing capability is the primary concern. For the same reason, PCubeS describes both cache and RAM as memories. Furthermore, if the hardware is suitable for multiple modes of programming – as many contemporary supercomputers are – there may be multiple PCubeS descriptions presenting its features from different programming perspectives. For example, in a heterogeneous supercomputer having both CPUs and GPUs per node, there can be a PCubeS description exposing the CPU details and another exposing the GPU detail ignoring parallelism inside the CPU. The first will be ideal for programs having irregular task parallelisms while the second for programs having regular data parallelisms.
2. **Parameterization:** PCubeS uses parameterized type-architecture description to enable programmers accurately estimate their algorithm’s performance for a particular input set. To understand why it is essential, imagine Snyder’s CTA description of a parallel execution platform showing processors’ interconnection network is a fat tree [Leiserson, 1985]. From the description, programmers can deduce that, on 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 the individual message sizes should be to balance computation with communication, they need to know the network’s actual latency and bandwidth. PCubeS has a core model parameterized by actual values when describing an execution platform. Thus, programmers can use the model to design the algorithm and the parameter values to assess a program’s runtime performance.

3.3.1 Elements of PCubeS

Parallel Processing Space (PPS): The notion PPS describes any part of a parallel execution platform or the whole where computations can take place. For example, in a multicore CPU, both the CPU and its 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 (PPUs). A space can perform two fundamental operations: floating point arithmetic and data movement.

The Capacity of a PPS: Parallel processing and memory access capacities of a PPS are defined as the number of corresponding fundamental operations that the PPS can do 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 essential; instead, its programmatic manifestation is. This interpretation allows PCubeS to treat cores, SIMD lanes, and vector pipelines similarly. The speed of instruction execution is enough to expose their efficiency differences. A PPS may or may not have a memory. When it exists, the memory is characterized by its size and the number of transactions that can be done on it in parallel. PCubeS uses a transaction to represent a single load/store operation or communication. The volume of data it carries and latency further characterize a transaction. For example, if a read/write operation by a thread within a dual hyper-threaded CPU core involves 8 bytes of data, takes 15 clock cycles to complete, and operations from both threads can take place simultaneously then the core’s parallel memory access capacity is two transactions of width 8 bytes and latency 15 cycles.

Uniformity and Independence of Sub-spaces: Given that PCubeS view an execution platform as a hierarchy of spaces, there must be a guiding principle for breaking bigger hardware components into smaller components to form sub-spaces. Otherwise, any hardware can fit the description of a flat PCubeS hierarchy having a single space only. In that regard, PCubeS uses uniformity and independence as the defining factors. Uniformity requires that not only all sub-spaces of a particular PPS 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, PCubeS divides its capacities into sub-spaces; otherwise, it does 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 interpretation violates the uniformity requirement due to the difference in intra-CPU and inter-CPU information exchanges among CPU cores. There should be another space in-between that represents the individual CPUs to bring uniformity into the hierarchy. Similarly, the lock-step threads of an NVIDIA GPU thread block cannot form sub-spaces despite being uniform. The reason is that their operations are not independent.

Information Exchange: PCubeS characterizes interactions between sibling PPU as information exchanges instead of communication or other platform-specific terms. Consequently, PCubeS can treat shared memory and distributed memory systems uniformly. Furthermore, within the umbrella of distributed memory architecture, different execution platforms may have varying implementations of communication mechanisms. The use of an abstract term enables PCubeS to dissolve these differences. Information exchange is characterized solely by its latency. We believe latency is enough to capture the efficiency differences among different modes of interaction. For example, consider a shared memory environment of four CPU cores sharing an L-3 cache. The information exchange between a pair of CPU cores is equivalent to sending core writing data in the cache and re-

ceiving core reading it. The only difference between such an interaction and a read following a write by a single CPU core is that the sender must use a mutex operation to signal writing completion. So the latency of information exchange here 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. On the other hand, 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: PCubeS uses all three transactional attributes – latency, width, and bandwidth – in characterizing data movements between a parent space and its sub-spaces. Transaction latency and width expose the cost of a single data motion between the parent and one of its children. Meanwhile, transaction bandwidth indicates how many data motion operations can occur 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 simultaneously, 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, a width of 128 bytes, and a bandwidth of 15. When the PPS under concern represents a distributed memory segment of the hardware, PCubeS derives the transactional attributes from the communication channel’s capacity and the communication protocol’s settings.

3.4 PCubeS Description Examples

Now we will discuss the PCubeS description of a multicore cluster that we frequently use as a HighP5 target platform, a supercomputer, and a Tesla K20 General Purpose GPU to give the reader the feel for PCubeS descriptions.

Hermes Multicore Cluster

Hermes cluster has four 64-core nodes connected by a 10-GB Ethernet interconnect. Each node comprises four 16-core AMD Opteron 6276 server CPUs plugged in 4 sockets of a Dell 0W13NR motherboard. RAM per CPU is 64 GB. Figure 1 depicts a candidate PCubeS description of the system (the diagram expands only one PPU at each PPS, but the PPUs are uniform).

There are six levels/PPSes in the PCubeS hierarchy of Hermes. Since PCubeS treats caches as memories, there are four levels from the CPU to the processing cores that expose how each 64-core CPU is hierarchically broken down into smaller units. Further, notice that the breakdown assigns the non-zero processing capacity to the Core-Pair level. PCubeS does this because a pair of cores share a single floating point unit within a CPU. As one goes up from the Core to the Cluster level, both latency and transaction width generally increase. However, the information exchange latency between sibling PPUs of a PPS is uniform from the Core to the CPU layer, which is the latency of a compare-and-swap instruction. The latency is uniform because a synchronization via the RAM

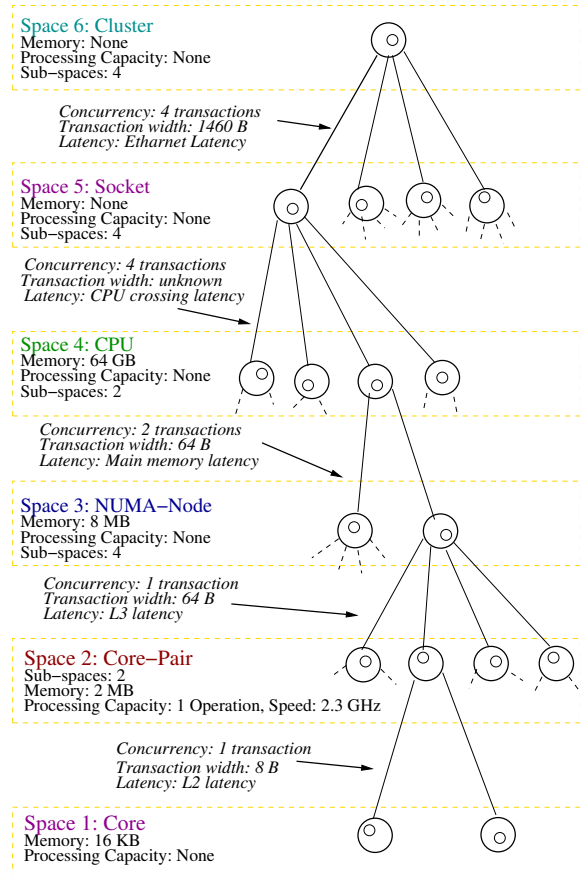


Figure 1. An illustration of PCubeS Description of the Hermes Cluster.

is essential for reliable cross-PPU interaction regardless of their proximity. Finally, there is no memory beyond the CPU level. Two essential characteristics of the PCubeS description are evident in Figure 1. First, PCubeS eliminates the distinction between shared and distributed memories by describing data storage and transfer attributes in terms of transactions. Second, PCubeS exposes every level of the hierarchical machine for programmers’ exploitation.

Tesla K20 General Purpose GPU

The NVIDIA K20, shown in Figure 2, has 2496 streaming cores running at 706 MHz distributed within 15 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 is accessible to programs. The streaming cores run in lock steps within each SM as a group of 32 threads known as warps. If we want to use the maximum amount of shared memory programmatically, each SM can roughly run up to 16 warps. Each shared memory load/store operation can process 16 32bit words, and a global memory load/store is twice that size.

The PCubeS description of the hardware depicts a 3-level space hierarchy. A warp represents a Space-1 unit. The parallel processing capacity of a Space-1 is 32 operations per cycle. However, the clock speed is only 44 MHz instead of 706 MHz. This bandwidth reduction is because warps execute as a pipeline instead of concurrently, and there are 16 of them. This clock speed setting in the PCubeS description contrasts NVIDIA’s advertised value as PCubeS focuses on actual performance characteristics of warps rather than mere numbers

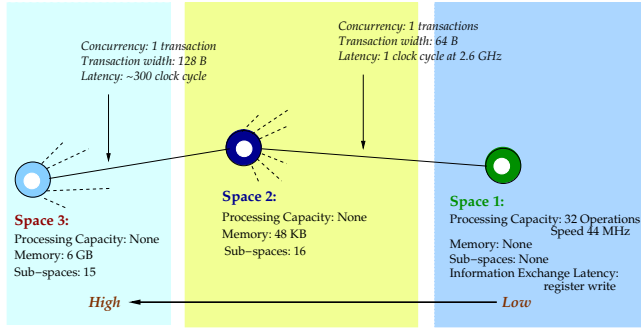


Figure 2. An illustration of PCubeS Description of an NVIDIA K20 GPU.

and structural details. A warp has no memory. Therefore, any computation must store its result in the closest space with memory. In this case, it is the parent Space-2. Space-2 represents an SM and holds 16 Space-1 PPUs. 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 one memory cycle. Sibling SMs cannot exchange information with one another. Therefore, information exchange latency is undefined. 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 15 Space-2 PPUs as sub-spaces.

The PCubeS description of the accelerator resembles the hardware abstraction popularized by NVIDIA’s CUDA programming model consisting of three levels. Nonetheless, there are noticeable 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 capability 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 divergence in a program must be sequential streams of fewer degrees of parallelism. So the programmer is aware of his wastage of processing capacity.

MIRA Supercomputer

The Mira Supercomputer [Schlagkamp *et al.*, 2016] in Argonne Leadership Computing Facility is a Blue Gene Q system. It has 48 compute racks hosting 49,152 IBM PowerPC92 A2 nodes. A 5D torus interconnection topology connects the nodes, and each node has 18 cores running at 1.6 GHz clock speed. Mira is a symmetric system of homogeneous nodes. Nonetheless, subtleties in the architecture reveal a rich hierarchy in the PCubeS description, as illustrated in Figure 3.

At the bottom, each CPU core can run up to 4 hyper-threads with 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 computation speed is only 400 MHz, instead of 1.6 GHz, as there are 4 Space-1 PPUs. Then a single core with its 16 KB L1 cache represents a Space-2 PPU. As the data path is 64 bits and threads’ data load/store happens in the L1 cache, transactions between Space-1 and Space-2 are 8 B

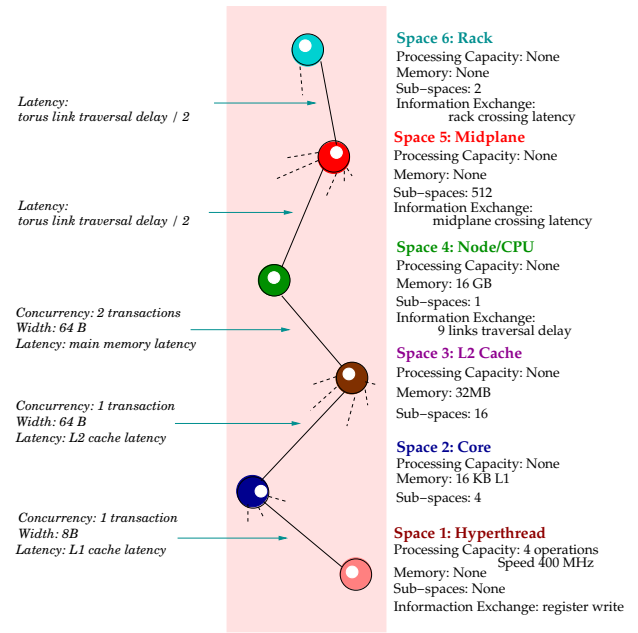


Figure 3. An illustration of PCubeS Description of the MIRA Supercomputer in Argonne National Lab. Only one PPU is shown in each level.

wide. Transaction concurrency is one as only one hyper-thread can issue 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 instead of 18 because only 16 out of the 18 cores can participate 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, there are levels in the hierarchy because of the non-uniform nature of node wiring.

A single compute rack of Mira holds 1024 nodes. A rack has 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 nine torus links. The average distance between a communicating pair lying in different mid-planes is thus ten links. Therefore, the PCubeS description has two additional levels on top of Space-4, representing a mid-plane and a rack as Space-5 and Space-6 PPUs. There are 48 racks in the system. Hence the PCubeS description has 48 Space-6 PPUs in total. 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 $4 \times 4 \times 4 \times 2$ torus topology. So an information exchange between a pair of Space-4 PPUs has, on average, nine torus links traversal delay. Mid-plane crossing adds one more link in the communication path of two nodes. Therefore, PCubeS equally divides the link traversal cost and assigns that as the communication cost between a Space-4 and its parent Space-5 PPU. This cost, augmented with any additional delay for mid-plane crossing (set as the information exchange latency in Space-5), reflects the average communication cost between nodes of opposite mid-planes. The same logic applies to nodes’ cross-rack communications in Space-6.

The PCubeS description of Mira exposes that the com-

plexity of the interconnection network in a supercomputer may lead to a deep hierarchy in its PCubeS modeling. However, careful attention to the network topology allows us to construct an accurate model of the cost of communications throughout the network. This capability is vital to avoid introducing hotspot contention [Hanawa *et al.*, 1996] in a program due to the uniform treatment of communication between any pair of nodes.

3.5 HighP5 Programming Model

HighP5 programming model assumes that a program executes in a hierarchy of logical processing spaces. As its name suggests, a Logical Processing Space (LPS) is an entity where a program can do computations over data structures. It is not directly associated with the computations or the data structures; instead, both get assigned to it. The best way to understand an LPS is how we treat variables and instructions in a traditional Von Neumann programming language. In a conventional Von Neumann machine, there is a flat memory. The hardware fetches instructions from memory and executes them. When an instruction executes, it may load and store variables in memory. The entire memory is visible to each instruction. The Von Neumann model is a single-space model.

HighP5 allows the programmer to define multiple spaces and hierarchical relationships between those spaces. Each LPS may have variables assigned to it. A variable assigned to an LPS is visible to computations executing in that space, much like variables in the single-space Von Neumann model are visible to programs. A program may assign a variable to multiple LPSes - and thus, it may be visible to code executing in different spaces.

An LPS may have one or more partitioning dimensions. When partitioned, each partitioning dimension has a cardinality. The partition breaks the LPS into Logical Processing Units (LPUs). The cross-product of the dimension cardinalities defines the number of LPUs in the LPS. Thus an LPS is not a characterless vacuum; rather, it is more like a geometric coordinate space. The programmer partitions data structures within each LPS and maps them to its LPUs. The same instruction stream executes in all LPUs of an LPS but on different parts of data structures.

Further, an LPS may divide another LPS. When this happens, the former is a subspace of the latter. Each LPU of the parent space has its complete sub-space. The sub-space also has some dimensionality (LPUs) and can be further subdivided. This hierarchy of LPS and LPU partitions may be arbitrarily deep. The dimensionality of LPS partitions is independent across levels of the LPS hierarchy but the same within a single level. Figure 4 illustrates the relations among LPSes using an example three-level LPS hierarchy.

A variable can coexist in multiple LPSes, and the program can partition the variable differently in different LPS levels. However, given variables are assigned to – not owned by – LPSes; any update of the shared variable in any LPS is also visible to all other relevant LPSes. From Figure 4, one might think that HighP5’s programming model is overly complicated. However, this model naturally corresponds with how programmers must distribute computation and data for

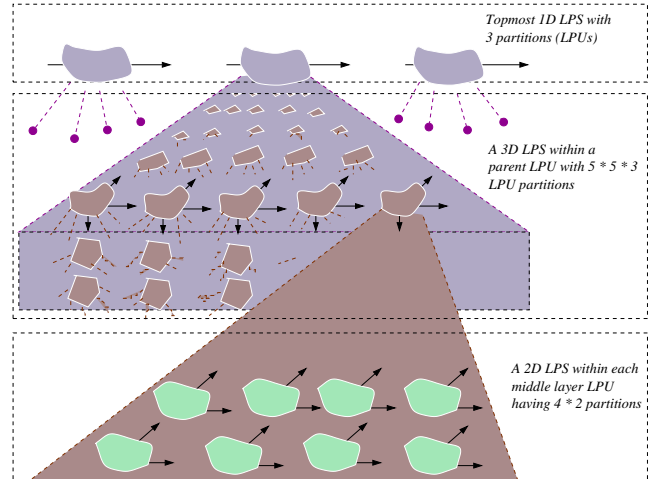


Figure 4. An example three-level LPS hierarchy with different dimensionality and LPU partition counts in each level.

the best performance in hierarchical machines. Furthermore, the declarative nature of the language makes it easy to write programs in terms of an LPS hierarchy and map the logical hierarchy of LPSes to the PPS hierarchy of the execution platform, as we will see in the following two sections.

4 Parallel Programming in HighP5

One can view HighP5 programming as parallel computations over multidimensional arrays where coarse-grained task parallelism encompasses fine-grained data parallelism. Philosophically, HighP5 adheres to the influential methodology proposed by Foster regarding developing parallel programs [Foster, 1995]. According to Foster, parallel programming combines four different activities: designing the algorithm, defining communication requirements among constituent parts, agglomerating parts into proper grained units and mapping those units to processors. A good decision regarding each of these activities requires considering the features of the execution platform. The essence of HighP5’s hardware-cognizant programming paradigm is to let programmers decide about all four aspects of parallel programming without being burdened with the specific implementation of their decision. Therefore, HighP5 has a declarative syntax that specifies ‘what’ not ‘how’ and ensures a clear separation of concerns in the program specification. HighP5 programming differs significantly from existing low-level and high-level parallel programming alternatives. In the following sub-sections, we first provide a general description of HighP5 programming and then discuss some example programs to clarify this distinction and expose some key features of the language. The current version of HighP5 language grammar is given in the Appendix.

4.1 HighP5 Program Structure

A HighP5 program is a collection of coarse-grained parallelizable tasks invoked and coordinated by a central controller. A task invocation in the program controller takes the following form:

```
execute(task: task-name;
        environment: ...; // environment-references
```

```

    initialize: ...;    // initialization-parameters
    partition: ...     // 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 non-blocking. Further, the invocation of the execute command does not necessarily launch the task immediately – instead, it schedules the task for future execution. The task can start only after all previous tasks, if any, manipulating its environmental data structures finish executing. Relating tasks through their environments provides 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. In other words, there is no need for accumulating intermediate results in a central place.

A HighP5 task is defined in terms of six distinct sections as follows:

```

Task 'Name of the Task':
Define:           // list of variable definitions
Environment:     // environmental variables, task input and output
Initialize <(optional initialization parameters)>:
    // variable initialization instructions
Stages:         // list of parallel procedures the task implements
Computation:    // the flow of computation stages, each in a LPS
Partition <(optional partition parameters)> :
    // specification of LPSes, their relationship,
    // and distribution of data structures in them

```

The Define, Environment, and Initialize sections do what you would imagine. The Define Section defines task-global variables. The Environment Section defines the variables received as part of the calling context and will be available to the caller when task execution is complete. The Initialize Section provides a mechanism to initialize task variables. Programmers define the list of parallel procedures – called compute-stages in HighP5 terminology – that are needed to implement the logic of a task in the Stages Section. The syntax for a compute-stage definition is as follows:

```

stage_name (...) //comma separated parameter names
{
    statements+
}

```

Similar to the Initialize Section, only the name of the stage parameters, not their types, are given. The difference is that a compute-stage must have at least one parameter. All stage parameters are passed-by-reference, and only task-global variables or constants can be the arguments for them during a stage invocation in the subsequent Computation Section. The compiler infers types of parameters and any local variable used inside a stage from the invocation context.

The Computation Section describes the logic of the task as a flow of compute-stages in LPSes. The basic idea is that the programmer specifies a control flow of LPS transitions and stage executions. Each stage execution occurs in the context of enclosing space in a data-parallel fashion. A stage may be executed in different spaces using different or the same parameters. HighP5 supports loops and conditional constructs with data-dependent dynamic conditions for controlling the LPS transitions. These LPS transitions can be nested. As shown in the example below:

```

Computation:
Space A {
    ...//statements including stage executions
Space B {
    ...//statements including stage executions
    Space C { statements }
}
Space C { statements }
}

```

Finally, the Partition Section specifies the configuration of the LPSes used in the computation flow, their relationships, and how to distribute data structure parts among LPUs of an LPS. Currently, HighP5 provides four standard partitioning functions as library support.

1. *block_size(s)*: divides an array dimension into partitions of size *s*.
2. *block_count(c)*: divides an array dimension into *c* parts.
3. *stride(l)*: distributes the indices of a dimension into processing units using strides of size *l*.
4. *block_stride(s,l)*: distributes the indices of a dimension into processing units using *s*-sized blocks of *l* strides.

The partition configuration for an LPS subdivides the space into a multi-dimensional coordinate system of LPUs discussed in the previous section. The number of data partitions, controlled by the partitioning arguments of the task execute command, determines the number of LPUs along different dimensions. An LPS may have named or un-named sub-partitions. An un-named sub-partition tells the compiler that all of the data in an LPU may not fit into memory. Therefore, the compiler should generate code to stage data in and out of the memory in chunks during compute-stage executions. A named sub-partition defines how a lower LPS divides each LPU of its parent LPS. The following example illustrates various features of hierarchical LPS partitioning.

```

Partition (k, l, m, n) {
Space A <1d> {
    matrix: block_count(k)
}
// Named sub-partitioning through LPS hierarchy
Space B <2d> divides Space A partitions {
    matrix: block_count(m, n)
    vector: replicated, block_count(n)

    // unnamed sub-partition for controlled loading
    // of data in an LPU
Sub-partition <1d> <unordered> {
    matrix<dim1>: block_size(l)
}
}
}

```

Control flow in the Computation Section is sequential but branches out horizontally in different LPUs independently in parallel. Different LPUs might be executing different compute-stages simultaneously on their respective data parts. When a transition between LPSes happens within the control flow, data structures used in both spaces are re-partitioned across the LPUs to ensure that all data access is local in the newly entered LPS. Re-partitioning may require moving data up or down through the LPS hierarchy or horizontally between LPUs in a particular LPS.

Let us now relate HighP5’s program specification to Foster’s parallel programming methodology. One would observe that the calculations within the compute-stages and the linear flow of stages within the Computation section define the parallel algorithm. The embedding of the control flow within a hierarchy of LPS dictates the communication needs among constituent parts. Then partitioning of that LPS hierarchy performs parts agglomeration. HighP5 excludes the last activity of mapping those parts to processors from the source code. Instead, HighP5 associates that activity with the compilation process. At that time, programmers explicitly dictate how the logical hierarchy of LPSes within different tasks of a program maps to the PPS hierarchy of the execution platform. We will discuss the mapping process in a subsequent section.

The compiler and run-time are responsible for keeping track of which data is where, moving data, and keeping track of all of the loop and index transformations required for proper compute-stage executions within their enclosing LPUs. However, given that the programmers have to map the LPS hierarchy to the PPS hierarchy of the execution platform, the cost of data movement and re-arrangement is evident to them.

4.2 Understanding HighP5 through Examples

We now look at the classic block matrix-matrix multiplication program as a comprehensive example of programming in HighP5. Listing 1 presents a single-task implementation of block matrix multiply to illustrate HighP5 concepts.

```

1 Program (args) {
2   //create an environment for matrix-matrix multiplication task
3   mmEnv = new TaskEnvironment(name: "Matrix Matrix Multiply")
4
5   // specify how input files are associated with environment
6   bind_input(mmEnv, "a", args.input_file_1)
7   bind_input(mmEnv, "b", args.input_file_2)
8
9   // execute the task
10  execute(task: "Matrix Matrix Multiply"; environment: mmEnv;
11         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 "Matrix Matrix Multiply":
18  Define:
19  a, b, c: 2d Array of Real double-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  //stage holding the logic of 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
35    // of Space A to have a block implementation as opposed
36    // to a traditional one
37    Repeat foreach sub-partition {

```

```

38    multiplyMatrices(c, a, b)
39  }
40 }
41 Partition (k, l, q):
42 // 2D partitioning of space giving a block of c in each
43 // partition along with a chunk of rows of a and a chunk
44 // of columns of b
45 Space A <2d> {
46   c: block_size(k, l)
47   a: block_size(k), replicated
48   b: replicated, block_size(1)
49   // block-by-block flow of data inside a PPU is governed
50   // by the sub-partition specification
51   Sub-partition <1d> <unordered> {
52     a<dim2>, b<dim1>: block_size(q)
53   }
54}

```

Listing 1: A Block Matrix-Matrix Multiplication HighP5 Program

The program consists of a coordinator program and a single task definition, *Matrix Matrix Multiply*. The coordinator program creates a new task environment (Line 3), binds the environment variables *a* and *b* to input files (Line 6 and 7), executes the single task (Line 10), and binds the result variable *c* to the output file (Line 14). The matrix-matrix multiplication task runs in a single space. The heart of the computation is the *multiplyMatrices* stage defined in the Stages Section. The code inside the compute-stage is a declarative specification for a set of vector dot products over the indices of the matrices. The *multiplyMatrices* compute-stage repeatedly executes in parallel within *Space A* in the Computation Section once for each sub-partition of *Space A* data structures defined in the partition section. Here each *Space A* LPU is responsible for computing a $k \times l$ block of *c*. For that, *k* rows of *a* is horizontally replicated in *l* LPUs and *l* columns of *b* is vertically replicated in *k* LPUs in the 2D *Space A* LPU partition. Finally, note that the for loop inside the compute-stage (Line 29 and 30) is a data-parallel loop. The compiler will translate the loop to ensure the maximum utilization of the vector computation capacity of the execution platform.

The reason for structuring a simple block matrix-matrix multiplication program as above becomes apparent when we consider the mapping of tasks to processors. Suppose we want to run the program in the Hermis cluster of Figure 1 utilizing the four CPUs of a single socket. Then a possible mapping configuration for the program can be as follows.

```

"Matrix Matrix Multiply" {
  Space Root: Socket
  Space A: Core-Pair
}

```

The mapping of the default Root LPS to the socket PPS limits the scope of the task to a single socket of Hermis. If the root LPS mapping is absent, the entire cluster engages in the matrix-matrix multiplication. The mapping of Space-A to the Core-Pair PPS instructs that each core pair will do a single LPU computation at a time. The runtime will pick an arbitrary core among the couple to do the computation during the task execution time and let it use the entire memory capacity of that PPU. As there are 16 core pairs within a single Hermis socket, the overall degree of parallelism for the multiplication task will be 16. The parallel for loop inside the compute-stage has to execute sequentially in this envi-

ronment as there is no vector processing capacity available in the cores.

Assume that we want to run the same matrix-matrix multiplication task on an NVIDIA K20 GPU (as shown in Figure 2) attached to a host computer. Then a possible alternative LPS to PPS mapping for the task can be as follows.

```

“Matrix Matrix Multiply” {
  Space Root: GPU
  Space A: Warp
}

```

In this mapping, each warp within the symmetric multiprocessor of the GPU will execute the compute-stage for individual Space-A LPUs. Given that a warp can perform 32 operations in parallel, the for loop of Line 29 in Listing 1 will run in a 32-way data-parallel fashion. However, as there is no memory associated with a warp, the data for LPUs (e.g., the blocks of matrices) will be stored in the nearest layer with a memory, the SM.

This simple program highlights the essence of HighP5’s hardware-cognizant declarative parallel programming. Programmers write a flexible program using a declarative syntax that they can map to the processing layers of their various target execution platforms. No code change is needed to port a single program to multiple platforms. The right decision for mapping LPSes to PPSes and the appropriate parameter values for data partitions within LPUs requires understanding the capacities of the hardware, which becomes easy due to PCubeS abstraction that describes different machine architectures uniformly.

To understand the role of the program coordinator in a HighP5 program, let us investigate a slightly more complicated problem of conjugate gradient calculation of a sparse system of linear equations. Conjugate gradient calculation is an iterative optimization problem involving vector addition, dot-product, and sparse matrix-vector multiplication in each iteration. Listing 2 illustrates a HighP5 implementation of this algorithm. All underlying tasks in the program are simple, so we only focus on the program coordinator’s behavior.

```

1 Program (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(
7     name: “CSR Matrix Vector Multiply”)
8   mvmEnv2 = new TaskEnvironment(
9     name: “CSR Matrix Vector Multiply”)
10
11  // make the argument sparse matrix stored in compressed
12  // row format from files to be read during first-time
13  // execution of the matrix-vector multiply task
14  bind_input(mvmEnv1, “columns”, args.arg_matrix_cols)
15  bind_input(mvmEnv1, “rows”, args.arg_matrix_rows)
16  bind_input(mvmEnv1, “values”, args.arg_matrix_values)
17
18  // bind the prediction (x_0) and the known vector (b)
19  // to the tasks’ environment that uses them initially
20  bind_input(vaEnv1, “u”, args.known_vector)
21  bind_input(mvmEnv1, “v”, args.prediction_vector)
22
23  // run the conjugate gradient logic
24  iteration = 0
25  maxIterations = args.maxIterations
26  do {
27    // calculate A * x_i
28
29    execute(task: “CSR Matrix Vector Multiply”;
30      environment: mvmEnv1; partition: args.r)
31
32    // determine the residual error r_i = b - A * x_i
33    vaEnv1.alpha = 1
34    vaEnv1.v = mvmEnv1.w
35    vaEnv1.beta = -1
36    execute(task: “Vector Addition”;
37      environment: vaEnv1; partition: args.b)
38
39    // determine the dot product of r_i to itself as
40    // the residual norm
41    dpEnv.u = dpEnv.v = vaEnv1.w
42    execute(task: “Vector Dot Product”;
43      environment: dpEnv; partition: args.b)
44    norm = dpEnv.product
45
46    // in the first iteration setup duplicate environment
47    // references for the sparse matrix components
48    if (iteration == 0) {
49      mvmEnv2.columns = mvmEnv1.columns
50      mvmEnv2.rows = mvmEnv1.rows
51      mvmEnv2.values = mvmEnv1.values
52    }
53
54    // determine A * r_i
55    mvmEnv2.v = vaEnv1.w
56    execute(task: “CSR Matrix Vector Multiply”;
57      environment: mvmEnv2; partition: args.r)
58
59    // determine dot product of r_i to A * r_i
60    dpEnv.v = mvmEnv2.w
61    execute(task: “Vector Dot Product”;
62      environment: dpEnv; partition: args.b)
63
64    // determine the next step size alpha_i as
65    // (r_i.r_i) / (r_i.(A * r_i))
66    alpha_i = norm / dpEnv.product
67
68    // calculate the next estimate x_i = x_i + alpha_i * r_i
69    vaEnv2.u = mvmEnv1.v
70    vaEnv2.alpha = 1
71    vaEnv2.v = vaEnv1.w
72    vaEnv2.beta = alpha_i
73    execute(task: “Vector Addition”;
74      environment: vaEnv2; partition: args.b)
75
76    // prepare x_i for the next iteration
77    mvmEnv1.v = vaEnv2.w
78    iteration = iteration + 1
79  } while iteration < maxIterations and norm > args.precision
80
81  // store the final solution vector in an output file
82  bind_output(vaEnv2, “w”, args.solution_vector)
83 }
84
85 Task “Vector Addition”:
86 Define:
87   u, v, w: 1d Array of Real double-precision
88   alpha, beta: Real double-precision
89 Environment:
90   u, v, alpha, beta: link
91   w: create
92 Initialize:
93   w.dimension = u.dimension
94 Stages:
95   addVectors(w, u, v, alpha, beta) {
96     do { w[i] = alpha * u[i] + beta * v[i] } for i in u
97   }
98 Computation:
99   Space A {
100     addVectors(w, u, v, alpha, beta)
101   }
102 Partition(b):

```

```

103 Space A <ld> {
104   u, v, w: block_size(b)
105 }
106
107 Task “Vector Dot Product”:
108 Define:
109   u, v: 1d Array of Real double-precision
110   product: Real double-precision Reduction
111 Environment:
112   u, v: link
113   product: create
114 Stages:
115   computeDotProduct(result, u, v) {
116     do { reduce(result, "sum", u[i] * v[i]) } for i in u
117   }
118 Computation:
119   Space B {
120     computeDotProduct(Space A: product, u, v)
121   }
122 Partition(b):
123   Space A <un-partitioned> {u, v}
124   Space B <ld> divides Space A partitions {
125     u, v: block_size(b)
126   }
127
128 Task “CSR Matrix Vector Multiply”:
129 Define:
130   columns, rows: 1d Array of Integer
131   values, v, w: 1d Array of Real double-precision
132 Environment:
133   values, columns, rows, v: link
134   w: create
135 Initialize:
136   w.dimension = rows.dimension
137 Stages:
138   multiply(w, v, rows, columns, values) {
139     start = rows.local.dimension1.range.min
140     if (start == 0) { start = -1 }
141     do {
142       if (i > 0) { beginIndex = rows[i - 1] + 1 }
143       else { beginIndex = 0 }
144       endIndex = rows[i]
145       do {
146         w[i] = w[i] + values[j] * v[columns[j]]
147       } for j in columns and j >= beginIndex and j <= endIndex
148     } for i in rows and i > start
149   }
150 Computation:
151   Space A {
152     multiply(w, v, rows, columns, values)
153   }
154 Partition(r):
155   Space A <ld> {
156     values, columns, v: replicated
157     rows: block_size(r) padding(1, 0)
158     w: block_size(r)
159   }

```

Listing 2: A HighP5 Program for Computing Sparse Matrix Conjugate Gradient

One can view a HighP5 program as a collection of tasks, each doing some computation in a logical environment consisting of data distributed over an LPS hierarchy. The program coordinator supplies any external input a task needs through the binding process (Line 11 to 21). The latter’s execution updates some environmental data and (or) creates new data in the environment. A task does not produce any output data per se. Instead, its output is the change in its environment. Tasks are interrelated to one another by their environmental data dependency. A dependent task gets the data it needs for calculation from an earlier task’s environment that created/updated it (Line 33, 47, 54, etc.). The programmer

extracts the program’s final output (or outputs) through the output binding process as in Line 82.

We believe this particular style of task coordination through environmental dependencies in a parallel program fosters efficiency. When programmers map LPS hierarchies of a program’s component tasks in nearby PPSes, a latter task can collect the pieces of distributed data directly from its predecessor’s environment. There is no need for data consolidation during task transitions. In the terminal case, where the partition specification and mapping configuration are the same between two tasks for a particular environmental data, there is no runtime cost of data re-distribution.

5 Major Language Features

HighP5 adheres to Niklaus Wirth’s classic advice of having a few generalizable features with similar structural abstractions that facilitate efficient program writing and compiler construction [Wirth, 1974]. The HighP5 language feature set will grow from its current inception phase to cover the common cases of high-performance parallel programming and to enhance the compactness of HighP5 programs. However, we are committed to maintaining a lean language core having simple but easy-to-understand structural features. This section briefly describes some basic features of the HighP5 language.

5.1 HighP5 Type System

HighP5 is a strongly-typed language that uses a mixture of implicit and explicit typing. Programmers must specify the types of environmental variables of component tasks of a program in the Define Section of those tasks. The compiler derives the types of all other variables (e.g., local variables in compute-stages and variables in the coordinator program) from environmental data types. The compilation process fails if the compiler cannot unambiguously determine the types of all variables from their context.

HighP5 supports common numerical and character data types and strings as basic and user-defined custom data types. A user-defined data type can have attributes of basic types, static arrays, and other custom types. Below is an example of a custom type declaration in HighP5.

```

Type Rectangle:
  Array[4] of Point
Type Point:
  x: Integer
  y: Integer

```

HighP5 custom types are a mechanism to group co-related data only. They are not like classes in an object-oriented language that supports behavior specification in terms of member functions. We adopt a simple strategy for custom types to ensure that efficient compilation is possible across different parallel architectures.

5.1.1 Function Types

All HighP5 functions are type polymorphic. The compiler generates proper implementation versions of a function based on the argument types found in its call contexts. These

functions are sequential and callable from both the coordinator program and compute-stages of tasks. Furthermore, programmers can embed sequential codes from other languages inside a function. The following example shows a function declaration that executes a certain number of Monte-Carlo sampling trials for area estimation within a rectangular cell of a larger area.

```
Function perform_sampling(cell, seed, trial_count) {
  cell_height = cell.top - cell.bottom + 1
  cell_width = cell.right - cell.left + 1
  internal_points = 0
  trial = 0
  do {
    // generate a point within the cell boundary and
    // calculate its position relative to the shape
    @Extern {
      @Language "C++"
      @Includes { math.h, cstdlib }
      $ {
        int x = rand_r(&seed) % cell_width + cell.left;
        int y = rand_r(&seed) % cell_height + cell.bottom;

        // tested polynomial is 10 sin x^2 + 50 cos y^3
        double r = 10 * sin(pow(x, 2)) + 50 * cos(pow(y, 3));
        if (r <= 0.0) {
          internal_points++;
        }
      } $
    }
    trial = trial + 1
  } while (trial < trial_count)
  return internal_points
}
```

Listing 3: A HighP5 function that uses an embedded C++ code to execute some Monte-Carlo area estimation sampling trials.

We understand that supporting interoperability with existing languages is essential to minimize the effort during a program migration. However, our conscious decision is to restrict language interoperability to sequential functions only for two reasons:

- First, allowing parallel code from other languages to mingle with a HighP5 code would promote the old way of thinking that we want to avoid.
- Second, the predictable program performance requirement from HighP5 demands that the HighP5 runtime accurately assesses the state of hardware resources when the program executes. If we allow arbitrary parallel codes to compete for resources with HighP5 tasks, then accurate resource status estimation is unattainable.

Finally, note that we do not encourage including codes written in other languages in a HighP5 program, as that would hurt the portability of a HighP5 program in architecture not supporting those languages.

5.2 Multi-versioned Data Management

Managing multiple versions of the same data structures is a common requirement in many scientific computing problems. For example, finite difference approximation for numerical fluid dynamics [Scannapieco and Harlow, 1995] typically requires new values of a cell being computed based on some previous values of that cell and its neighbors. Explicitly declaring these various versions of the same data hurts the readability of the code. HighP5 supports automatic multi-versioning to handle this programming pattern cleanly. The

compiler deduces whether or not a variable needs version management by checking the presence of expression of the following form in the code.

```
variable_name at (current - i)
```

Here the keyword *current* stands for the latest version and *i* is a positive numerical constant specifying how far in the past the compiler needs to track older versions of the underlying variable. The compiler searches for each such expression to determine what different versions of a variable should the runtime library maintain. Note that not all individual updates of a multi-versioned data structure may necessitate a version upgrade. It is up to the programmers to define the boundary of changes that lead to a new version using the *epoch* construct. The following code snippet shows the application of multi-versioning in a compute-stage definition of a 5-point iterative stencil task for heat propagation and corresponding version advancement control in the task's Computation Section.

```
...
Stages:
...
refineEstimates(plate) {
  localRows = plate.local.dimension1.range
  localCols = plate.local.dimension2.range
  do { plate[i][j] at (current)
    = 0.25 * (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:
...
Space B {
  ...
  Repeat for counter in [1...partition.p2] {
    ...
    // epoch needs to be advanced after each refinement step
    Epoch {
      refineEstimates(plate)
    }
    ...
  }
  ...
}
```

5.3 Built-in Support for Reduction

HighP5 takes reduction or data aggregation as a basic primitive (i.e., an expression type) of the language. This unique treatment of that parallel programming pattern is due to several reasons:

- First, various forms of reduction are frequent in parallel computing.
- Second, the performance of the reduction operations significantly contributes to the overall performance of a parallel program.
- Third, efficient implementation of a reduction requires careful consideration of the communication behavior of the execution platform, which may result in a performance portability issue for a HighP5 program when the target platforms are different in that regard.
- Finally, most large-scale parallel machines such as supercomputers employ a separate data aggregation network or hardware-level features to efficiently imple-

ment reductions and low-level reduction primitives [Stunkel *et al.*, 2020]. HighP5 compilers can directly use those primitives to ensure that the performance of a reduction operation in HighP5 is as good as it can be in the underlying execution platform.

To use reductions in a HighP5 task, programmers first define the target of a reduction operation using the *Reduction* keyword after the variable type declaration. Although efficient compilation does not require this demarcation of the reduction target, we decided to have it to ensure that programmers are aware of the distinction between a reduction variable and a standard variable regarding their memory allocation. For example, the count and dimensions of LPU partitions in the LPS a reduction takes place dictate the dimensions and cardinalities of the underlying reduction variable. On the other hand, its dimensions and cardinalities are either predefined or derived from other environmental variables for a standard environmental variable.

The reduction expression has the following general form:

```
for {
  reduce(result-variable,
        'accumulator-function-name', expression)
} index-range
```

This generic expression covers most standard forms of reductions common in parallel programming. The third argument of the reduction expression is an arbitrary expression to be evaluated on all elements of an array (or arrays) satisfying an index range expression. The second argument tells how the runtime should combine the values of those individual expressions. The first argument specifies the result variable that should receive the reduction output. Programmers should invoke the compute-stage having a reduction within the LPS that supplies the data for the third argument. The compiler determines the span of each reduction operation based on the first argument. If the result variable is in an ancestor LPS then there will be one separate reduction for each LPU in the ancestor LPS spanning across all LPUs of the descendent LPS that supplied the reduction input.

Let us now see an example to clarify the concept of reduction. Suppose we want to solve an area estimation problem using the Monte Carlo method [Quinn, 2003], and we structure the HighP5 task as illustrated in Figure 5. The uppermost level generates a single estimate for the overall area under the curve. Each LPU in the middle-layer LPS computes the part of a 2D grid cell that falls inside the curve. Finally, each LPU in the bottom LPS performs the Monte Carlo sampling for a horizontal stripe of the 2D grid cell belonging to its parent LPU. The goal is to do area estimation for each grid cell in the middle LPS repeatedly until further sampling trials do not improve an estimate.

Efficient implementation of this task should involve repeated reductions from the bottom LPS's LPUs to their middle LPS's LPU parents. Once all grid cell estimates are satisfactory, there should be a single reduction from the middle layer LPUs to the single top layer LPU. Listing 4 shows how programmers can implement this logic in the HighP5 task. Notice how the scope and target of each reduction are specified in Line 88 and 97 for the computation stages *monteCarloSampling* (Line 50) and *estimateTotalArea* (Line 73).

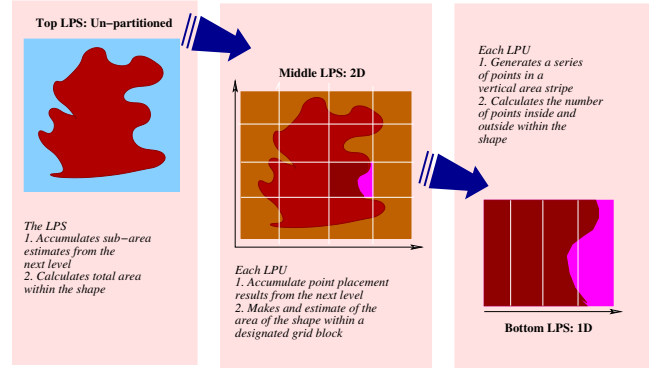


Figure 5. A logical breakdown of an area estimation task using the Monte Carlo method.

```
1 Task ‘Monte Carlo Area Estimation’:
2 Define:
3   grid: 2d Array of Rectangle
4   internal_points: 2d Array of Integer
5   placement_result: Integer Reduction
6   local_estimates, estimate_diffs:
7     2d Array of Real double-precision
8   round, cell_length, points_per_cell, max_rounds: Integer
9   precision_threshold: Real double-precision
10  area: Real double-precision Reduction
11 Environment:
12   area: create
13 Initialize(precision_threshold, max_rounds, cell_length,
14           grid_dim, points_per_cell):
15   grid.dimension1.range.min = 0
16   grid.dimension1.range.max = grid_dim - 1
17   grid.dimension2 = grid.dimension1
18   internal_points.dimension = grid.dimension
19   local_estimates.dimension1.range.min = 0
20   local_estimates.dimension1.range.max
21     = grid_dim / partition.b - 1
22   local_estimates.dimension2 = local_estimates.dimension1
23   estimate_diffs.dimension = local_estimates.dimension
24   // initialize the random number generator
25   init_rand()
26 Stages:
27   setupGridCells(grid, cell_length) {
28     do {
29       cell_height = cell_length
30       cell_width = cell_length
31       grid[i][j].left = cell_width * i
32       grid[i][j].right = cell_width * (i + 1) - 1
33       grid[i][j].bottom = cell_height * j
34       grid[i][j].top = cell_height * (j + 1) - 1
35     } for i, j in grid
36   }
37   initializeEstimateDiffs(diffs, threshold) {
38     do {
39       diffs[i][j] = threshold + 1
40     } for i, j in diffs
41   }
42   monteCarloSampling(sample_count, grid, points, result) {
43     do {
44       cell = grid[i][j]
45       seed = lpuId[0]
46       points[i][j] = perform_sampling(cell,
47                                     seed, points_per_cell)
48     } for i, j in grid
49     do {
50       reduce(result, ‘sum’, points[i][j])
51     } for i, j in points
52   }
53   estimateSubarea(internal_points,
54                 sample_count, round, diff, estimates) {
55     row = estimates.local.dimension1.range.min
56     col = estimates.local.dimension2.range.min
57     cell_size = cell_length * cell_length
58     curr_estimate =
```

```

59     cell_size * internal_points / sample_count
60     weight = 1.0 / round
61     old_estimate = estimates[row][col]
62     updated_estimate = curr_estimate * weight
63     + old_estimate * (1 - weight)
64     estimates[row][col] = updated_estimate
65     if (updated_estimate > old_estimate) {
66         diff[row][col] = updated_estimate - old_estimate
67     } else {
68         diff[row][col] = old_estimate - updated_estimate
69     }
70 }
71 estimateTotalArea(result, sub_area_estimates) {
72     do {
73         reduce(result, "sum", sub_area_estimates[i][j])
74     } for i, j in sub_area_estimates
75 }
76 Computation:
77     Space A {
78         Space B {
79             Space C {
80                 setupGridCells(grid, cell_length)
81             }
82             initializeEstimateDiffs(estimate_diffs,
83                 precision_threshold)
84             Repeat for round in [1..max_rounds] {
85                 If estimate_diffs[lpuId[0]][lpuId[1]]
86                     > precision_threshold {
87                     Space C {
88                         monteCarloSampling(points_per_cell,
89                             grid, internal_points,
90                             Space B: placement_result)
91                     }
92                     estimateSubarea(placement_result,
93                         points_per_cell, round,
94                         estimate_diffs, local_estimates)
95                 }
96             }
97             estimateTotalArea(Space A: area, local_estimates)
98         }
99     }
100 Partition(b):
101     Space A <unpartitioned> { local_estimates }
102     Space B <2d> divides Space A partitions {
103         grid, internal_points: block_size(b, b)
104         local_estimates, estimate_diffs: block_size(1, 1)
105     }
106     Space C <2d> divides Space B partitions {
107         grid, internal_points: block_size(1, 1)
108     }

```

Listing 4: A HighP5 task with three layer LPS hierarchy for solving an area estimation problem using Monte Carlo method.

5.4 Automatic Data Synchronization

Data synchronization happens in the LPS boundaries of a task. When the control flow of a task moves from one LPS to another LPS, then the HighP5 runtime synchronizes data. Furthermore, data synchronization happens only when it is needed. A HighP5 compiler performs static analysis of the control flow to determine whether the enclosed computation within the subsequent LPS depends on any data modified by any computation in earlier LPSes. Then it inserts appropriate data synchronization code before the LPS transition. The exact nature of the synchronization implementation depends on the mapping of LPSes to the PPSes of the underlying execution platform. Note that HighP5 supports overlapping data partitioning among the LPUes of a single LPS.

In that case, programmers must put explicit LPS transition boundaries between successive computations within a single LPS, as shown in the following code snippet (Listing 5) taken from an iterative stencil program for estimating heat propagation. Overlapping data partitions is the only special case for data synchronization that programmers need to be conscious about. In this code, the repeat statements are outside the LPS boundaries in Line 4 and 9 to ensure that the runtime synchronizes overlapping boundary regions of plate data in neighboring LPUes of Space A and B in regular intervals.

```

1 Computation:
2 // the whole computation should iterate for max_iterations
3 // number of times
4 Repeat for counter_1 in [1..max_iterations] {
5 // after partition.p1/partition.p2 upper level iterations
6 // the flow should exit for upper level padding
7 // synchronization
8     Space A {
9         Repeat for counter_2 in [1..partition.p1]
10             step partition.p2 {
11                 // after partition.p2 iterations the flow exits
12                 // Space B for lower level padding synchronization
13                 Space B {
14                     Repeat for counter_3 in [1..partition.p2] {
15                         // epoch needs to be advanced after each
16                         // refinement step
17                         Epoch {
18                             refineEstimates(plate)
19                         }
20                     }
21                 }
22             }
23     }
24 }
25 Partition (k, l, m, n, p1, p2):
26     Space A <2d> {
27         plate: block_count(k, l) padding(p1)
28     }
29     Space B <2d> divides Space A partitions {
30         plate: block_count(m, n) padding(p2)
31     }

```

Listing 5: Computation and Partition Sections of a 5-point stencil program for estimating heat propagation on a 2D plate.

We consider the uniform treatment of all data synchronization requirements a significant advantage of HighP5's style of parallel programming. Data synchronization is a frequent source of errors and performance bottlenecks in parallel programs. HighP5 ensures that the cognitive burden for dealing with data synchronization is minimal and its implementation is efficient also by delegating that responsibility to the compiler. Note that although programmers do not deal with the nitty-gritty of data synchronization, they can debug its cost based on the PCubeS description of the execution platform and their mapping of LPS hierarchies to PPSes. The data transfer latency and bandwidth-related attributes reflect the cost of moving data along the PPS hierarchy and within PPUes of identical PPS.

6 Run-time Environment

A central element of HighP5's hardware cognizant programming paradigm is that the performance of a HighP5 program must be predictable. That requires that programmers can

assess the runtime behavior of their programs, and HighP5 compilers generate executable codes that meet programmers' expectations. In this section, we first explain the runtime behavior of a HighP5 program and then discuss the common runtime model all HighP5 compilers implement to achieve both predictable and high performance in their respective target execution platforms.

6.1 HighP5 Program Behavior

Each HighP5 task specifies a flow of computation along a hierarchy of LPSes happening concurrently on the LPUs of those LPSes. An LPU execution takes place on data local to that LPU. Therefore, appropriate data synchronization, movement, and reorganization are precursors to a successful computation. Programmers' mapping of the LPS hierarchy to the PPS hierarchy of the execution platform dictates the actual data synchronization, movements, and reorganizations at runtime corresponding to the declarative task specification and the degree of parallelism during its execution. The degree of parallelism for the overall program varies from task to task as programmers do LPS-PPS mapping for individual tasks separately. Programmers' LPS-PPS mapping for a task is a binding regarding hardware resource allocation on the compiler such that not only the computation and memory capacity of those mapped PPSes but also the communication capacity of the network (sub-network) connecting their PPU are committed to the task during its execution. Consequently, two tasks in a HighP5 program do not execute concurrently even with no data dependency between them if their LPS to PPS mapping does not allow concurrent execution without resource conflicts. This characteristic makes it easy to debug the program's overall performance based on the analysis of individual tasks.

6.1.1 Degree of Parallelism in a Task

When mapping the LPS hierarchy of a task to the PPS hierarchy of the execution platform, programmers can skip PPSes and map multiple LPSes to a single PPS. The degree of parallelism within the task varies depending on the number of PPUs available in a PPS. The LPUs of the LPS get multiplexed into the PPUs of the PPS. Therefore for a specific LPS to PPS mapping, the degree of parallelism is proportional to the number of PPUs in the PPS. Consider the following three mappings of LPSes of the 5-point iterative stencil task (Listing 5) in the Hermes cluster (Figure 1) as shown in Table 1:

In Case A, the degree of parallelism for Space A computations is 16, as there are 16 CPUs if we start counting from the top cluster level. Within the confinement of each Space A LPU, the degree of parallelism for Space B computation is eight, as there are eight core-pairs within each CPU. Therefore, the overall degree of parallelism for Space B LPU computations is $16 \times 8 = 128$. Notice that in Hermes's PCubeS model (Figure 1), there is no computation capacity at the CPU level. HighP5 compilers deal with LPS mappings on PPSes with no computation or memory capacity using the following *PPS Substitution Rule*:

1. If an LPS is assigned to a PPS having no compute capacity, then a single PPU in the nearest lower/upper-level

PPS that is capable of computation executes all LPUs of that LPS.

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

Therefore, a single core-pair will execute the LPUs of Space A on behalf of its owner CPU for the first LPS-PPS mapping scenario, and all core-pairs will execute the LPUs of each Space B confined within a single Space A LPU. From so far discussion, it is clear that in Case B mapping, the degree of parallelism for both Space-A and Space-B computations is $4 \text{ sockets} \times 4 \text{ CPUs} \times 2 \text{ NUMA Nodes} = 32$. The LPS-PPS mapping in Case C is similar to that of Case A, except that the entire task's resource allocation is restricted to a single socket of the cluster. Therefore, the degree of parallelism for Space A and B computations is four and 32 respectively. HighP5 compilers search scope for concurrent task executions only in the last case.

6.1.2 Intra-Task Data Movement and Synchronization

A PPU executes instruction streams on local data or data parts. Therefore, the compiler ensures that data belonging to an LPU being multiplexed to the PPU is available before a computation needing that data begins. We call this execution model local-compute. This model is different from the owner-computes model [Lee and Kedem, 2002] as the same data may exist in different LPSes that programmers may have mapped to different PPSes. If two PPUs have overlapped or shared data, the compiler ensures that their common data part is synchronized between compute-stage executions. The implementation of data synchronization varies depending on the nature of physical memory. If the two PPUs share a memory, then only a single version of the data exists that the runtime passes between the PPUs. Otherwise, there are multiple versions of the same data that the runtime updates as needed.

In both cases, the compiler ensures that the data synchronization initiates at the sender's end as soon as the update is done, and the receiver's end joins in the data synchronization process as late as possible. The sender's side is asynchronous, while the receiver's side is not. The compiler tries to fit in as many independent compute-stages as possible between the send and receive to maximize the overlapping of computation with communication. The overall approach of local-compute ensures that the memory access latency for any computation is predictable and the inter-PPU communication-related parameters accurately reflect the worst-case data movement cost of the underlying execution platform.

6.1.3 Inter-task Data Dependency Resolution

When there is a data dependency between two tasks, the dependent task waits for the completion of the data updater task. Even in the case of a conditional data dependency, HighP5 compilers take the conservative approach and generate code to schedule the probably independent task later. Data dependency resolution is typically costlier for inter-dependent

Table 1. Three different LPS to PPS mapping configuration of 5-point Stencil task to the Hermes cluster.

<pre> “5-point Stencil” { Space A: CPU Space B: Core-Pair } </pre>	<pre> “5-point Stencil” { Space A: NUMA-Node Space B: NUMA-Node } </pre>	<pre> “5-point Stencil” { Spare Root: Socket Space A: CPU Space B: Core-Pair } </pre>
A	B	C

tasks than for inter-dependent LPS computations within a single task.

In the case of intra-task data movement and synchronization, once the inputs and partitioning parameters are available during task initialization, the runtime can readily decide the sizes of various data communication buffers and the nature of synchronization and data movement primitives. Therefore, there is no additional overhead during the actual data communication/synchronization. However, doing the same for inter-task data dependency resolution is difficult as that would require monitoring the progress of an ongoing task and setting up communication/synchronization resources for the subsequent dependent task premeditatively. This strategy is both complex and potentially wasteful due to conditional data dependencies. Furthermore, the potential benefit is slim as the communication/synchronization resources have a lower chance of repeated use.

Currently, tasks scheduled for executions from the coordinator program remain in a task pool. Whenever a running task completes its execution, the runtime checks what other tasks from the pool can execute next and then select one or more depending on hardware resource availability. If there is any data dependency to resolve before running a new task, then the necessary data movement/synchronization happens before the task starts. The only optimization in the process for data dependency resolution is that if the PPU’s responsible for handling computation using the dependent data parts are the same for the earlier and the later task, then no data movement happens.

6.2 HighP5 Runtime Model

As long as the language model and mapping configuration requirements are satisfied, different implementations are possible for the HighP5 runtime engine (RTE) for tasks and PPU handling. It is better to choose an implementation model that can utilize the strength of the underlying low-level programming primitives a HighP5 compiler uses to generate the executable in a specific PCubeS platform. Nonetheless, the three compilers we have developed so far implement a common RTE.

The runtime process model for HighP5 is straightforward. A single process executes on each computing node of the target execution environment. The process runs the HighP5 program that includes references to the HighP5 runtime libraries. The main program initializes global variables and then starts the program controller.

Applications compiled with the multicore compiler have just a single process running on the host. At run time, parallelism is achieved by using Pthreads threads that play the role of PPUs. LPUs are assigned to PPUs, and PPUs run a

loop in which they execute the LPUs that have been given to them and synchronize as necessary.

Applications compiled for distributed memory machines use one MPI process per node with one corresponding 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 are achieved using shared memory and Pthreads barriers. Processes perform inter-node communication and synchronization using MPI.

Applications that use accelerators, particularly 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 and execute them as required. If a set of LPUs is to be executed on the GPU, a single popup controller thread starts that manages interactions with the GPU. The GPU controller thread is responsible for copying LPU data structures onto GPU card memory, initiating the kernel calls on the GPU, and staging data back from the cards.

An LPU management code running within each GPU SM manages LPU executions on the GPU. The code stages data into the SM from the card memory, executes LPUs, then 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 complete their execution, 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 LPUs to the GPU. This process repeats until there are no more LPUs.

7 Experiments

HighP5 claims that programmers can effortlessly write portable, high-performant parallel programs with predictable runtime behavior using a declarative, hardware-cognizant paradigm. Our approach to proving that claim was to build multiple HighP5 compilers for different present-day architectures and then enable building block parallel programs on those platforms. We currently have three preliminary HighP5 compilers for shared-memory multicore machines, distributed shared-memory supercomputers and commodity clusters, and hybrid supercomputers having multicore CPUs and NVIDIA GPUs in their nodes. We choose the building block problems from the characteristic application classes identified by the landmark paper ‘The Landscape of Parallel Computing Research: A view from Berkeley.’[Asanovic

et al., 2006] The paper identifies seven core and six emerging characteristic classes, or dwarfs, that capture different computation and communication patterns common in important applications. HighP5 paradigm supports all core dwarfs. However, the compiler support is lacking for some features in the hybrid platform. In general, all three compilers generate code competitive with efficient hand-written code. However, they are still not optimized in-depth to compete with complex low-level library implementations of representative problems from the dwarf application classes.

We asked some competent programmers to write efficient low-level C++ code parallelized with Pthreads and MPI for five problems from four dwarf application classes to use as the reference implementations in multicore and distributed shared-memory platforms. Then we asked an experienced industry expert to verify those implementations and remove any obvious inefficiencies. These building block problems are block matrix-matrix multiplication, block LU factorization, conjugate gradient on a sparse matrix, 5-point iterative stencil on a regular grid, and Monte Carlo area estimation. Table 2 describes the characteristics of these dwarf applications. This paper has already discussed HighP5 programs for these problems, except for the LU factorization code (which is shared in the Appendix). We emphasize that we used the same HighP5 programs discussed in this paper with different mapping configurations to compile and execute them on various platforms.

For the hybrid platform, we used highly optimized single GPU CUDA programs. The hybrid HighP5 compiler is currently in an elementary stage. It still does not support multi-task programs and point-to-point lateral communications in a single LPS. Consequently, we could not test conjugate gradient and iterative stencil in this platform and only collected implementations and tested the performance of other three applications. Table 3 presents the line of code (LoC) comparison between implementing the tested applications using low-level tools and using HighP5. Note that the low-level implementations are quite lengthy due to various optimizations, such as, cache blocking, thread pinning, and non-blocking communications. Although the core algorithmic logic is portable across platforms, compared to HighP5 versions, the time and effort required to implement the individual low-level programs is significantly high.

Finally, note that HighP5 compilers are source-to-source compilers. They generate C++ code parallelized with Pthreads, MPI, and CUDA depending on the mapping configuration and target hardware platform. Integrated GCC, MPICC, and NVCC compilers then compile a HighP5 compiler's output to produce the final executable. To ensure parity, we compiled the hand-written reference programs with the same compiler optimization flags enabled as we have for HighP5 compilers. All HighP5 compilers use -O3 compiler optimization for the underlying GCC, MPICC, and NVCC compilers.

7.1 Experiments on a Multicore Environment

We used an Intel(R) Xeon(R) Gold 6438Y+ machine with 128 cores and 251 GB RAM for our multicore platform experiments. It is a NUMA machine with cores divided into

two groups of 64 having 60 MB L3 cache each. 64 cores of each NUMA node are then divided into core pairs. Each pair of cores share a 2 MB L2 cache and 48 KB L1 cache. The Instruction cache size for a core pair is 32 kB.

As mentioned earlier, we compared HighP5 programs against handwritten optimized Pthreads programs. Different degrees of parallelism for both HighP5 and reference implementations have been achieved by progressively doubling the number of threads. Consequently, we have results for 1, 2, 4, 8, 16, 32, 64, and 128-way parallel versions. We ran each version of the executables 50 times and measured the end-to-end execution time. The maximum standard deviations among different degrees of parallelism for the five applications are given in Table 4. The maximum standard deviations on LU factorization code is quite high in both Pthreads and HighP5 programs, which occurred for single-threaded execution. We do not know the exact reason for these variations. The standard deviations for higher degree parallel versions of this program are significantly lower than that of the single-thread case.

Figure 6 illustrates the average execution time difference of reference Pthreads and HighP5 executables for all applications. Here the horizontal axis presents thread counts in logarithmic scale ($2^0 = 1 \dots 2^7 = 128$).

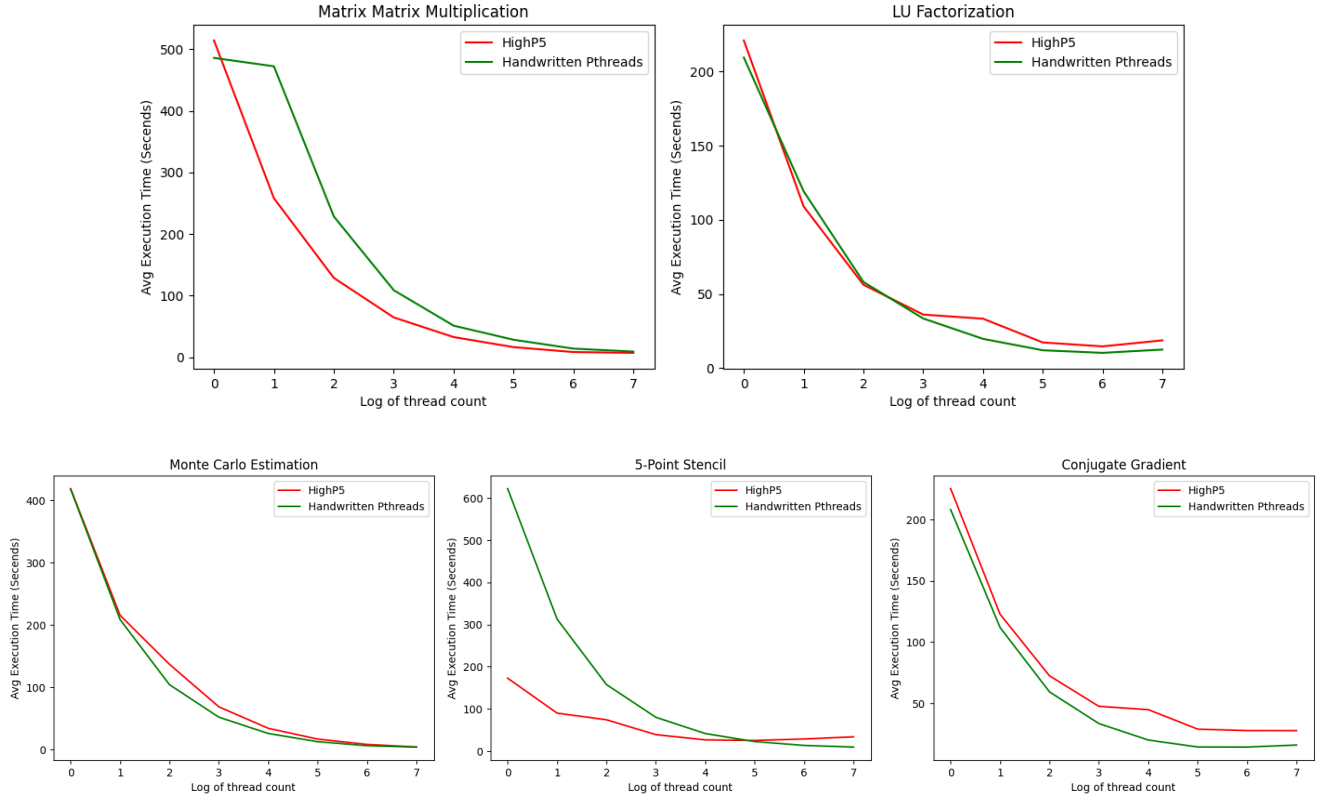
We observe that HighP5 and reference implementations follow almost the same trajectory for the Monte Carlo area estimation problem. There is a deviation in matrix-matrix multiplication problem for $2^1 = 2$ threads. The performance is otherwise similar with HighP5 always slightly outperforming the reference implementation. On the other hand, reference implementations slightly outperform HighP5 for high degrees of parallelism in conjugate gradient and LU factorization. The reason is that HighP5 compiler generates synchronization code based on data dependency analysis. Although a lot of optimizations are in place inside the compiler to avoid redundant synchronizations, there are still a few additional synchronizations in the HighP5 versions that do not exist in the reference code. Upon investigation, we found that the reference implementations do some duplicate local computations on scalar variables to reduce synchronization overhead. This is an optimization that HighP5 compiler currently does not support. However, we believe the overall performance in these two problems is quite satisfactory.

The most interesting result is observed for the 5-point stencil problem. The HighP5 implementation starts with a much better performance for $2^0 = 1$ thread and slowly improves in performance up to $2^4 = 16$ threads. Then its performance becomes flat. On the other hand, the reference implementation keeps getting better until it crosses HighP5 at $2^5 = 32$ threads then its improvement slows down. This divergence in behavior is due to the different nature of the two implementations. HighP5 compiler generates code for overlapping data movement in the stencil program¹ using data distribution analysis. For a single thread, there is no data to share: consequently, no synchronization. However, the reference implementation does not treat it as a corner case and two Pthreads barrier synchronizations happen at periodic in-

¹ Parallel implementations of 5-Stencil must have overlapping partitions among neighboring threads/processes.

Table 2. Descriptions of the Dwarf Applications Used for Performance Experiments

Application Name	ID	Sequential Time Complexity	Communication or Synchronization	Memory Access
Matrix-Matrix Multiplication	mmult(M,N)	$\mathcal{O}(n^3)$ for M,N of $\mathcal{O}(n^2)$	None	Regular
Block LU Decomposition	bluf(A)	$\mathcal{O}(n^3)$ for A of $\mathcal{O}(n^2)$	Iterative Collective Comm/Sync	Regular
Monte Carlo Estimation	monte(curv, count)	$\mathcal{O}(count^k)$ for curve of $\mathcal{O}(k)$	Infrequent Collective	None
Iterative Stencil	stencil(plate, iter)	$\mathcal{O}(n^2 \times iter)$ for a plate of $\mathcal{O}(n^2)$	Iterative point-to-point	Regular
Conjugate Gradient	conjgrad(M, V, iter)	$\mathcal{O}(nm \times iter)$ for M $\mathcal{O}(n)$ and V $\mathcal{O}(m)$	Iterative Collective	Irregular

**Figure 6.** Performance comparisons of HighP5 and Pthreads implementations. Problem sizes: (a) 7168×7168 square matrices for matrix-matrix multiplication (b) 8192×8192 square argument matrix for LU Factorization (c) 1024×1024 square grid and 2000 samples per grid cell for Monte Carlo Estimation (d) 8192×8192 square plate and 2000 iterations for 5-point stencil (e) 30720×30720 symmetric matrix with 90% sparsity for conjugate gradient.**Table 3.** Comparison Between Line of Code in Writing Efficient Implementation of the Dwarf Applications using HighP5 and Low-Level Alternatives

App ID	Low Level LoC				HighP5 LoC
	Pthreads	MPI	CUDA	Total	
mmult	157	290	315	762	51
bluf	431	588	580	1599	166
monte	162	113	154	429	124
stencil	300	336	N/A	636	59
conjgrad	308	360	N/A	686	152

tervals despite no data copying taking place. On the other hand, the same two barrier synchronizations are used to protect data exchanges among neighboring threads for higher degree of parallelism that seems to scale well as the thread count increases. HighP5 implementation has twice the number of threads pair-wise semaphore-based synchronizations that seems to scale poorly as their number grows. This is an issue we want to address in the future for better code generation in multicore platforms.

Table 4. The Maximum Standard Deviations in Execution Time for Different Degrees of Parallelism of HighP5 and Handwritten Pthreads Programs

App ID	Pthreads		HighP5	
	Parallelism	STD	Parallelism	STD
mmult	4	0.0673s	4	0.86s
bluf	1	23.212s	1	23.21s
monte	1	0.067s	8	0.207s
stencil	2	0.166s	1	2.085s
conjgrad	64	0.737s	32	0.34s

7.2 Experiments on a Distributed Shared-memory Environment

We faced a machine access issue when doing the experiments on distributed shared-memory environment. We first tested strong and weak scalability of HighP5 executables compared to efficient sequential baseline in a compute cluster. The strong scalability test shows how performance of a HighP5 executable improves compared to a sequential baseline as we increase the number of processors. The weak scalability test, on the other hand, shows whether the efficiency gain of par-

allelism remains consistent as we increase the problem size proportionally with the processor count. We were able to do scalability testing up to 1000 CPU cores. However, later we loose access to the cluster and could not do performance comparison with parallel MPI code and HighP5 executable on the same platform. Rather, we had to do the comparative analysis on just 8 machines connected in a 1 GBPs Ethernet LAN network in a larger research lab. We provide the results of scalability testing in the Appendix so that readers can examine the performance of HighP5 executables at high degrees of parallelism.

Our configured small cluster setup has 8 machines with 13th Gen Intel(R) Core(TM) i5-13600KF processor nodes where per node RAM is 15 GB. This is a hybrid architecture with 6 dual-way hyper-threaded high performance cores and 8 single-threaded efficient cores. This gives us two different PCubeS models for the cluster, one using the performance cores and another using efficient cores per node. We found that the efficient cores are 3 to 4 times slower than the performance cores in single node application performance. Consequently, we only used the performance cores model for the comparative performance analysis.

We installed OpenMPI 4.1.2 in the cluster. However, we faced the main difficulty with the slow LAN network with a large collision domain. Although the connection is 1GB Ethernet, we got less than 10 MBps speed for cross node communication. Consequently, MPI communication primitives' performance was quite poor and increasing parallelism did not improve program performance in applications as much as it would do otherwise. Hence, we encourage readers to rely on the scalability test of Appendix for assessing the efficiency of HighP5 in distributed shared-memory platforms and use the comparative performance analysis with the hand-written MPI programs given below as a tool for evaluating only the relative merits of HighP5 against low-level programming.

As in the multicore platform case, we ran each version of the reference MPI and HighP5 implementations 50 times. We increased the degree of parallelism by increasing the number of nodes by 1 from 1 to 8 nodes. For matrix-matrix multiplication, Monte Carlo area estimation, and 5-point stencils; each node ran a 6-way multi-threaded HighP5 MPI process. The reference implementations are, on the other hand, pure MPI code. Consequently, we had 6 MPI processes per node for the reference runs. Figure 7 shows the average end-to-end execution times of HighP5 and handwritten MPI implementations for these three applications. In all three applications, HighP5 and the handwritten implementations' performance follow the same trajectory.

We faced a strange scalability problem with OpenMPI collective communication primitives when running the other two applications. Both LU factorization and conjugate gradient use multiple collective communications per iteration. Consequently, good performance of the MPI collective communication primitives is critical for overall good performance of these applications. However, even with just two/three processes per node, the collective communications were getting slow to the extent that increasing node counts was consistently decreasing performance for both HighP5 and Handwritten MPI implementations. Hence, we had to ran these two applications with one MPI process (and thread) per node

configuration. Figure 8 shows the performance comparison between HighP5 and Handwritten MPI for this configuration.

We observe that HighP5 implementation maintains a consistent lead with the handwritten MPI version for all degrees of parallelism for LU factorization. There is no clear explanation of this behavior other than that the larger number of collective communications in the handwritten version that HighP5 compiler handled using direct sends/receives slowed the former more (Currently, HighP5 compiler can only translates reduction operations to MPI broadcasts and other forms of group communications are translated into multiple non-blocking MPI send/receives.). There is a large difference in running time at single-node configuration also. This happens because HighP5 implementation avoid calls to MPI communication primitives in this terminal case but the handwritten implementation does not.

The difference in behavior at the single-node terminal case is even more prominent in the conjugate gradient application performance. Readers should also notice a large spike in HighP5 implementation's execution time as we go from a single node to 2 nodes. This reflects the extreme slow performance of the group communication primitives. Despite doubling the number of nodes halving per-node computation cost, the overhead of communication actually made the execution time twice as slow. Afterwards, HighP5 and the handwritten MPI implementations follow similar trajectory for higher degrees of parallelism.

Overall, we would say that the experiments show that the HighP5 compiler does not have any inherent limitation that would make it difficult to generate executables that are competitive with efficient handwritten low-level code in the distributed shared-memory environment. However, there are rooms to improve inside the compiler regarding better communication code generation.

The maximum standard deviations of the Handwritten MPI and HighP5 implementations are given in Table 5 for the sake of completeness.

Table 5. The Maximum Standard Deviations in Execution Time for Different Degrees of Parallelism of HighP5 and Handwritten MPI Programs

App ID	HandWritten MPI		HighP5	
	Parallelism	STD	Parallelsim	STD
mmult	8	0.185s	1	0.1089s
bluf	2	7.042s	1	18.724s
monte	8	0.193s	2	0.948s
stencil	2	4.6649s	1	1.2858s
conjgrad	2	1.2335s	4	3.0118s

7.3 Experiments on a Hybrid Environment

We used the *gpu* queue in a supercomputer (we will call it Super 1 for anonymity) of another University as the backend 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

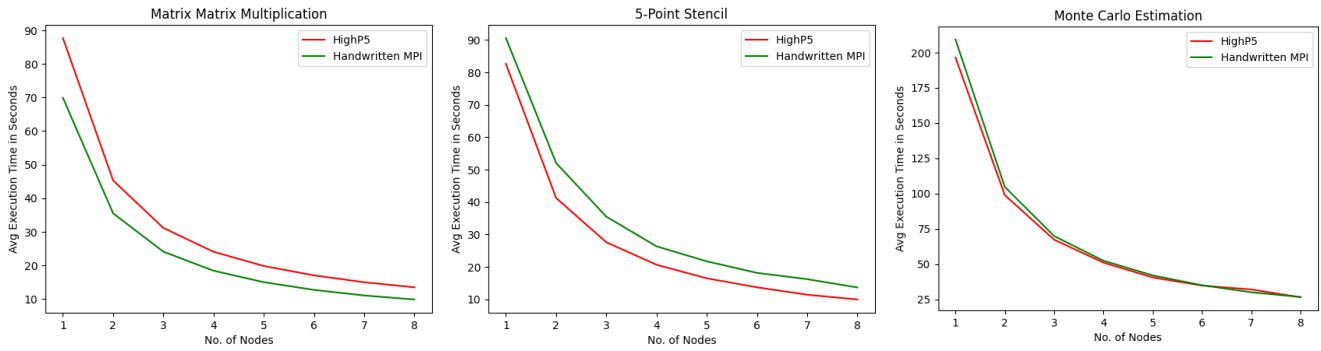


Figure 7. The Average Execution Time Comparison Between HighP5 and Handwritten MPI implementations of a) Matrix-Matrix Multiplication, b) 5-point Stencil, and c) Monte Carlo Area Estimation

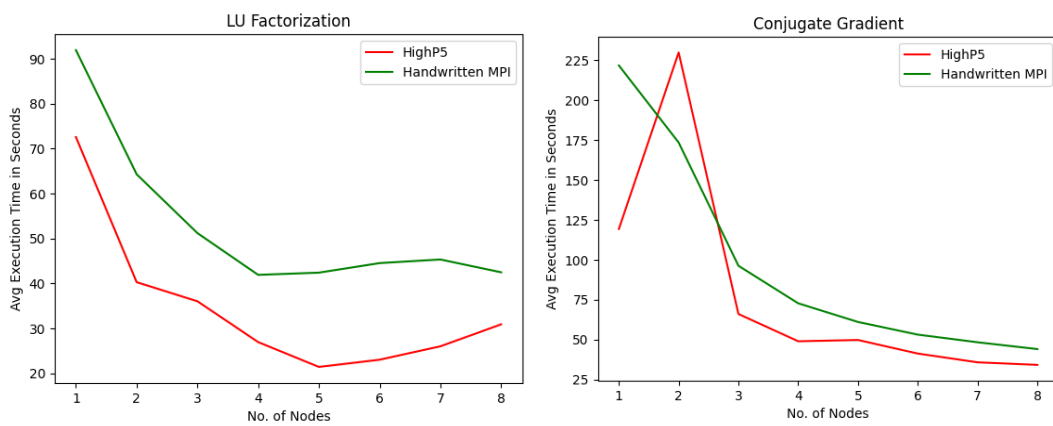


Figure 8. The Average Execution Time Comparison between HighP5 and Handwritten MPI implementation of a) LU Factorization and b) Conjugate Gradient Applications.

has a 6 GB card memory. A K20 GPU has 13 symmetric multiprocessors (SMs) instead of 15 of the K20 GPU described in Figure 2. The remaining configurations are the same in both GPUs. Due to availability restrictions, we have used a maximum of 8 nodes to run the programs. Super 1 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 cray++, and the MPI implementation was cray-MPICH/7.3.2. The CUDA compiler was NVCC 7.0, V7.0.27. In addition, CUDA codes have been compiled with compute architecture setting 3.5 (`arch=sm_35` parameter).

As the hybrid HighP5 compiler is only in its initial stage and it lacks some features for cross-task data sharing and data synchronization within a task for overlapping LPU partitions, we cannot run the conjugate gradient and 5-point stencil programs in the hybrid platform yet. Furthermore, when doing the experiments, our LPS-PPS mappings of the HighP5 tasks exploit parallelisms in the GPUs only. The part of the code that runs on the host CPUs is single-threaded for each program. We chose this mapping strategy as we already knew the performance in multicore and distributed-memory architectures from the earlier experiments and were interested in investigating HighP5 program performance on GPUs. However, a multi-GPU computation in Super 1 requires the host CPUs to handle communication across machines and staging

in and out of data to/from the GPUs. Hence, we recorded the host execution time, other overheads at the host for communication and resource setup, and the data transfer time for the host-GPU interaction besides the GPU code (kernel) execution time.

Figure 9 depicts the strong and weak scaling results for the HighP5 block matrix-matrix multiplication program. The strong scalability experiment is done for 10, 240×240 square input matrices. 1-GPU version is almost six times slower than the CUDA reference implementation, 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 HighP5 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, the performance gain will be substantial.

The host-level overhead for the HighP5 implementation is also higher than that of the reference implementation. The overhead is more prominent because managing small data parts for LPUs involves more computation overhead than initialization of a single large block of memory as done in the reference implementation. However, this overhead does not alarm us as it exhibits a downward trend with increasing parallelism. The weak scalability results are consistent with earlier results from the parallel cluster. Thus we believe the current hybrid compiler is adequate in that regard.

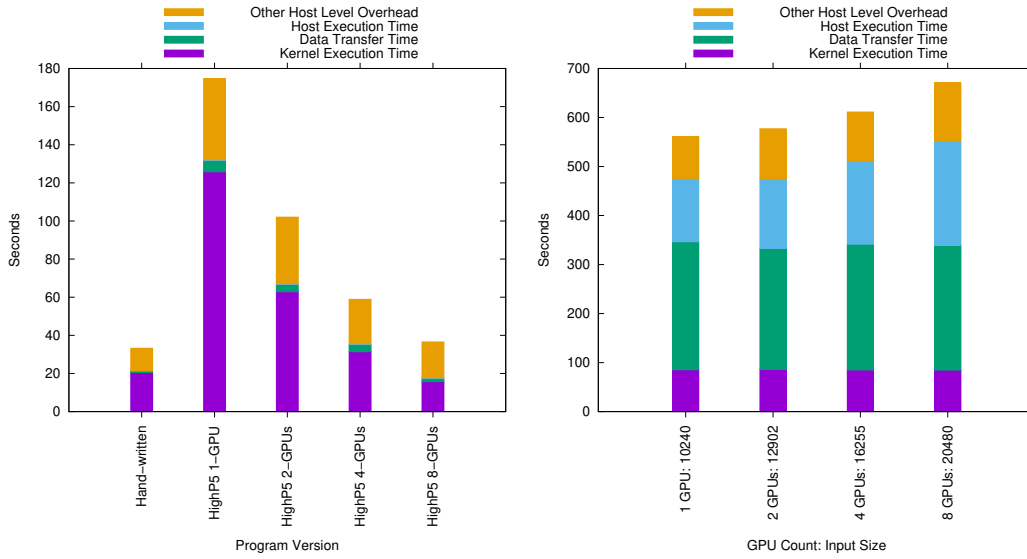


Figure 9. Strong and Weak scaling results for block matrix-matrix multiplication on Super 1

Figure 10 shows the strong and weak scaling results for HighP5 block LU factorization. Again, the input argument matrix size for strong scalability experiment is $10,240 \times 10,240$.

The most noticeable feature in both graphs is the large data transfer and host execution costs. Other than that, the results are consistent with the findings from the previous matrix-matrix multiplication experiments. The host execution time part is significant in the HighP5 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 executed in the host has become the more significant contributor to runtime cost 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 emphasize this issue most in the future development of the hybrid compiler.

Despite having a downward trend, the data transfer time is also high due to the memory-less nature of the host and GPU interactions. The HighP5 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 are updated in-between successive encounters. With a more sophisticated data transfer mechanism, we should be able to eliminate most of this data transfer cost.

Finally, Figure 11 illustrates the strong and weak scaling results for HighP5 Monte Carlo area estimation. We investigated the same curve from earlier multicore and distributed shared memory experiments. The code again divided the bounding area into a square grid of $10,240 \times 10,240$ cells for the strong scalability test. The samples per cell are, 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 11. Unlike the previous two programs, the 1-GPU version of the HighP5 program performs nearly as well as the hand-written 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 HighP5 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 calculations for LPU generation. This result suggests that the kernel execution time slowdown we have observed for the earlier two programs is probably related to sub-optimal memory accesses instead of overhead computations.

Overall, we take the current experiments in the hybrid platform as proof that we can port the same HighP5 programs in this architecture and a strong indicator that we have a feasible target for performance improvement to be competitive with efficient hand-written code for this platform.

8 Conclusion

Hybrid and deeply hierarchical machine architectures have become increasingly common in parallel and everyday computing. While writing efficient programs for these increasingly complex machines is becoming harder and harder as good program performance requires careful utilization of these machines' computing, memory, and communication capacities. In this paper, we described a new paradigm for parallel computing that we named hardware cognizant programming. In this paradigm, a uniform type architecture description exposes the salient features of the hardware platform. Programmers reason over that common abstraction and write platform-feature sensitive, portable code using a declarative programming language. That makes productivity and portability attainable across diverse parallel platforms without sacrificing performance.

HighP5 programming language is the embodiment of our vision. The paper shows how HighP5 realizes that vision by discussing its underlying type architecture, programming model, language features, and runtime environment. Experiments on three different architectural platforms using build-

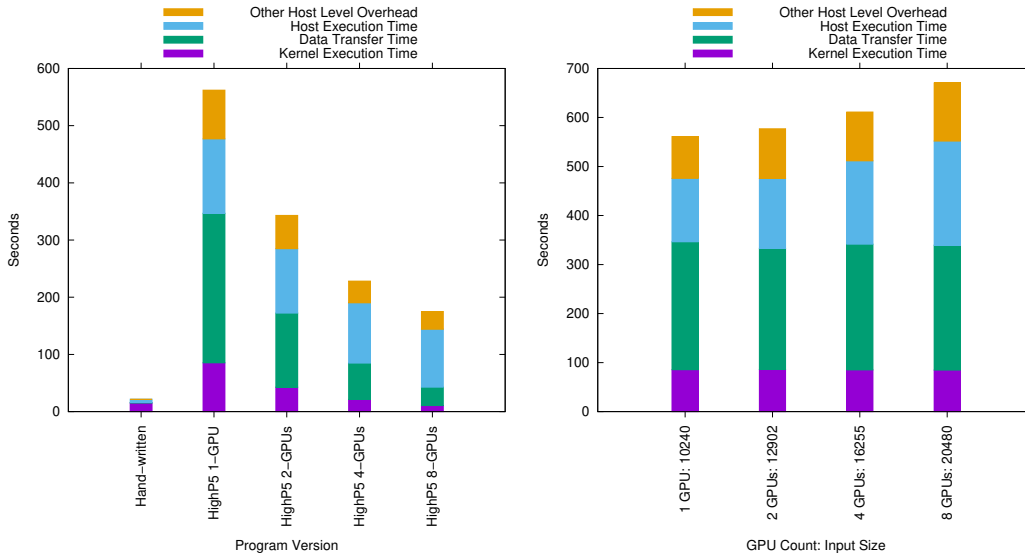


Figure 10. Strong and Weak scaling results for block LU factorization on Super 1

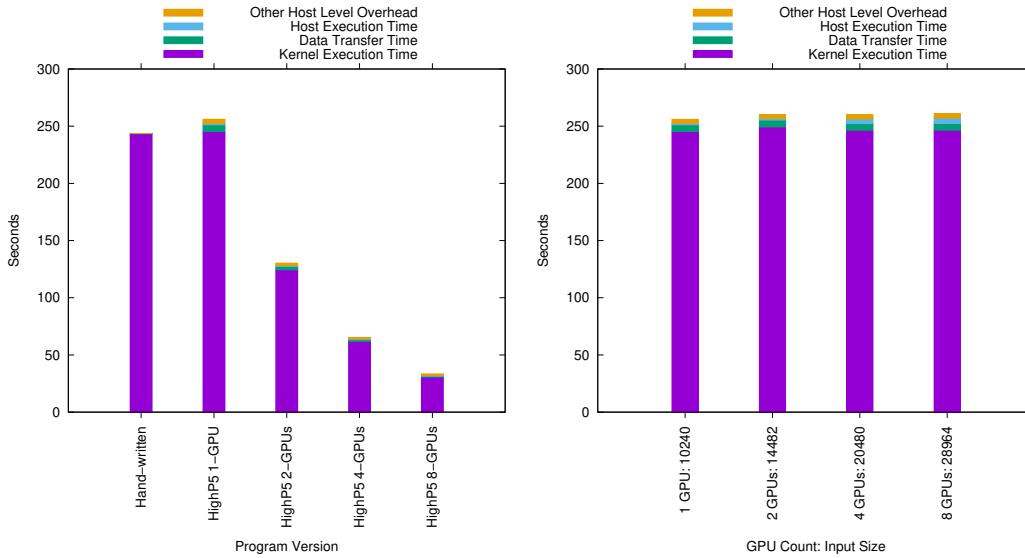


Figure 11. Strong and Weak scaling results for Monte Carlo area estimation on Super 1

ing block parallel applications shows the utility and potential of the HighP5 style of parallel programming. In particular, the simplicity of HighP5 programming and program portability across execution platforms is evident from the discussion and experiments. The performance results are promising, but there is significant scope for achieving general competitiveness against efficient low-level library codes.

HighP5 is currently in its inception phase. We must solve several crucial future research problems to make HighP5 an acceptable alternative to the dominant low-level platform-specific parallelization techniques and the high-level hardware-agnostic programming languages. We recognize the following targets for our immediate future research on HighP5:

1. Reduce the runtime overhead of data dependency resolution during transitions between tasks of a multi-task HighP5 program in distributed memory and hybrid environments.

2. Enable efficient forms of data synchronization for code running inside the GPU and optimize data transfers between host CPU and attached GPUs in hybrid computing platforms.
3. Improve HighP5 loop translations in CPUs that support vectored instructions.
4. Enhance IO handling features in the language.
5. Incorporate some fundamental and efficient features for better expressing irregular parallelisms in the language.

Feature streamlining to reduce the amount of typing, multi-file source code compilations, and library import support are things we want to consider after we solve the problems mentioned above.

Declarations

Acknowledgements

The authors would like to thank Rich Knepper and Craig Stewart of Indiana University, K.S.M. Tozammel Hossain of University of North Texas, and Golam Rabiul Alam of Brac University for providing access to their machines, compute cluster, and supercomputer for various experiments.

Funding

This work is partially funded by the XSEDE: Extreme Science and Engineering Discovery Environment project of National Science Foundation, USA. The fund was granted to Andrew Grimshaw for his investigation on GFFS: a Global Federated File System that connects compute and storage resources across university campuses and supercomputing centers for fostering research collaboration.

Authors' Contributions

The first author, Muhammad Nur Yanhaona, is the main contributor of this work. The paper is a result of his PhD. thesis under the supervision of the second author, Andrew Grimshaw. The third author, Shahriar Hasan Mickey, helped in conducting performance experiments and manuscript preparation.

Competing interests

On behalf of all authors, the corresponding author states that there is no conflict of interest.

Availability of data and materials

Source code version of the three compilers and the HighP5 reference programs used in experiments described earlier have been provided with this article.

References

- Adve, V., Carle, A., Granston, E., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Mellor-Crummey, J., and Warren, S. (1994). Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58. DOI: 10.1109/M-PDT.1994.329801.
- Alexandrov, A., Ionescu, M. F., Schauer, K. E., and Scheiman, C. (1995). Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation. DOI: 10.1145/215399.215427.
- Alpern, B., Carter, L., and Ferrante, J. (1993). Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings*, pages 116–123. DOI: 10.1109/PMMP.1993.315548.
- Aoki, R., Oikawa, S., Nakamura, T., and Miki, S. (2011). Hybrid opencl: Enhancing opencl for distributed processing. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 149–154. DOI: 10.1109/ISPA.2011.28.
- Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., Yelick, K. A., Demmel, M. J., Plishker, W., Shalf, J., Williams, S., and Yelick, K. (2006). The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY. Available at: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
- Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In Sips, H., Epema, D., and Lin, H.-X., editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-03869-3_80.
- Bachan, J., Bonachea, D., Hargrove, P. H., Hofmeyr, S., Jacquelin, M., Kamil, A., van Straalen, B., and Baden, S. B. (2017). The upc++ pgas library for exascale computing. In *Proceedings of the Second Annual PGAS Applications Workshop, PAW17*, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3144779.3169108.
- Bauer, M., Clark, J., Schkufza, E., and Aiken, A. (2011). Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. *SIGPLAN Not.*, 46(8):13–24. DOI: 10.1145/2038037.1941558.
- Bauer, M., Treichler, S., Slaughter, E., and 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*, pages 66:1–66:11, Los Alamitos, CA, USA. IEEE Computer Society Press. DOI: 10.1109/SC.2012.71.
- Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., Ryujin, B. S., and Scogland, T. R. (2019). Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. DOI: 10.1109/P3HPC49587.2019.00012.
- Besard, T., Foket, C., and De Sutter, B. (2019). Effective extensible programming: Unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841. DOI: 10.1109/TPDS.2018.2872064.
- Blagojević, F., Hargrove, P., Iancu, C., and Yelick, K. (2010). Hybrid pgas runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS '10*, pages 3:1–3:10, New York, NY, USA. ACM. DOI: 10.1145/2020373.2020376.
- Blelloch, G. E. (1992). *NESL: A Nested Data-Parallel Language*. Carnegie Mellon University, USA. Book.
- Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95*, page 207–216, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/209936.209958.

- Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., USA. Book.
- Chakravarty, M. M., Keller, G., Lee, S., McDonnell, T. L., and Grover, V. (2011). Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1926354.1926358.
- Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312. DOI: 10.1177/1094342007078442.
- Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. (2000). Zpl: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211. DOI: 10.1109/32.842947.
- Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press. Book.
- Chapman, B., Mehrotra, P., and Zima, H. (1992). Programming in vienna fortran. *SCIENTIFIC PROGRAMMING*, 1(1):31–50. DOI: 10.1155/1992/258136.
- Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538. DOI: 10.1145/1103845.1094852.
- Culler, D., Karp, R., Patterson, D., Sahay, A., Schauer, K. E., Santos, E., Subramonian, R., and 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, pages 1–12, New York, NY, USA. ACM. DOI: 10.1145/155332.155333.
- Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55. DOI: 10.1109/99.660313.
- De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., and De Meuter, W. (2015). Partitioned global address space languages. 47(4). DOI: 10.1145/2716320.
- Edwards, H. C. and Trott, C. R. (2013). Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24. DOI: 10.1109/XSW.2013.7.
- El-Ghazawi, T. and Smith, L. (2006). Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM. DOI: 10.1145/1188455.1188483.
- Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Available at: https://edoras.sdsu.edu/~mthomas/docs/foster/Foster_Designing_and_Building_Parallel_Programs.pdf.
- Hanawa, T., Fujiwara, T., and Amano, H. (1996). Hot spot contention and message combining in the simple serial synchronized multistage interconnection network. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, SPDP '96, page 298, USA. IEEE Computer Society. DOI: 10.1109/SPDP.1996.570347.
- Henriksen, T., Serup, N. G. W., Elsmann, M., Henglein, F., and Oancea, C. E. (2017). Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571. DOI: 10.1145/3140587.3062354.
- Herdman, J. A., Gaudin, W. P., Perks, O., Beckingsale, D. A., Mallinson, A. C., and Jarvis, S. A. (2014). Achieving portability and performance through openacc. In *2014 First Workshop on Accelerator Programming using Directives*, pages 19–26. DOI: 10.1109/WACCPD.2014.10.
- Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677. DOI: 10.1145/359576.359585.
- Larus, J. (1993). C**: A large-grain, object-oriented, data-parallel programming language. In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Languages and Compilers for Parallel Computing*, pages 326–341, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/3-540-57502-2_56.
- Lee, P. and Kedem, Z. M. (2002). Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. Program. Lang. Syst.*, 24(1):1–50. DOI: 10.1145/509705.509706.
- Leiserson, C. E. (1985). Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901. DOI: 10.1109/TC.1985.6312192.
- Lin, W.-C. and McIntosh-Smith, S. (2021). Comparing julia to performance portable parallel programming models for hpc. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 94–105. DOI: 10.1109/PMBS54543.2021.00016.
- MacNeice, P., Olson, K. M., Mobarry, C., de Fainchtein, R., and Packer, C. (2000). Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354. DOI: 10.1016/S0010-4655(99)00501-9.
- Mainland, G. and Morrisett, G. (2010). Nikola: Embedding compiled gpu functions in haskell. *Haskell '10*, page 67–78, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1863523.1863533.
- Mallón, D. A., Taboada, G. L., Teijeiro, C., Touriño, J., Fraguera, B. B., Gómez, A., Doallo, R., and Mouriño, J. C. (2009). Performance evaluation of mpi, upc and openmp on multicore architectures. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 174–184, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-03770-2_24.
- Marowka, A. (2022). On the performance portability of openacc, openmp, kokkos and raja. In *International Conference on High Performance Computing in Asia-*

- Pacific Region*, HPCAsia '22, page 103–114, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3492805.3492806.
- McCarthy, J. (1978). *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA. DOI: 10.1145/800025.1198360.
- Merrill, D. and Garland, M. (2016). Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '16, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2851141.2851190.
- Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '12, page 117–128, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2145816.2145832.
- Mohr, M., Malony, A., Mohr, B., Beckman, P., Gannon, D., Yang, S., and 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*, pages 75–85. Society Press. DOI: 10.1109/IPPS.1994.288316.
- Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2):40–53. DOI: 10.1145/1365490.1365500.
- Novosel, R. and Slivnik, B. (2019). Beyond classical parallel programming frameworks: Chapel vs julia. In Rodrigues, R., Janousek, J., Ferreira, L., Coheur, L., Batista, F., and Oliveira, H. G., editors, *8th Symposium on Languages, Applications and Technologies, SLATE 2019, June 27-28, 2019, Coimbra, Portugal*, volume 74 of *OASICs*. DOI: 10.4230/OASICs.SLATE.2019.12.
- Numrich, R. W. and Reid, J. (1998). Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31. DOI: 10.1145/289918.289920.
- Olukotun, K. (2014). Beyond parallel programming with domain specific languages. 49(8):179–180. DOI: 10.1145/2692916.2557966.
- Pennycook, S. J., Hammond, S. D., Jarvis, S. A., and Mudalige, G. R. (2011). Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark. *SIGMETRICS Perform. Eval. Rev.*, 38(4):23–29. DOI: 10.1145/1964218.1964223.
- Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Education Group. DOI: 10.5555/1211440.
- Rabenseifner, R., Hager, G., and Jost, G. (2009). Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. DOI: 10.1109/PDP.2009.43.
- Scannapieco, E. and Harlow, F. H. (1995). Introduction to finite-difference methods for numerical fluid dynamics. DOI: 10.2172/212567.
- Schardl, T. B., Lee, I.-T. A., and Leiserson, C. E. (2018). Brief announcement: Open cilk. SPAA '18, page 351–353, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3210377.3210658.
- Schlagkamp, S., Ferreira da Silva, R., Allcock, W., Deelman, E., and Schwiegelshohn, U. (2016). Consecutive job submission behavior at mira supercomputer. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 93–96, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2907294.2907314.
- Shan, H., Wright, N. J., Shalf, J., Yelick, K., Wagner, M., and Wichmann, N. (2012). A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for pgas languages and comparison with mpi. *SIGMETRICS Perform. Eval. Rev.*, 40(2):92–98. DOI: 10.1145/2381056.2381077.
- Snyder, L. (1986). Type architectures, shared memory, and the corollary of modest potential. In Traub, J. F., Grosz, B. J., Lampson, B. W., and Nilsson, N. J., editors, *Annual Review of Computer Science Vol. 1, 1986*, pages 289–317. Annual Reviews Inc., Palo Alto, CA, USA. Available at: <http://dl.acm.org/citation.cfm?id=17814.17826>.
- Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73. DOI: <http://dx.doi.org/10.1109/MCSE.2010.69>.
- Stunkel, C. B., Graham, R. L., Shainer, G., Kagan, M., Sharkawi, S. S., Rosenburg, B., and Chochia, G. A. (2020). The high-speed networks of the summit and sierra supercomputers. *IBM Journal of Research and Development*, 64(3/4):3:1–3:10. DOI: 10.1147/JRD.2020.2967330.
- Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817. DOI: 10.1109/T-PDS.2021.3097283.
- Walker, D. W., Walker, D. W., Dongarra, J. J., and Dongarra, J. J. (1996). Mpi: A standard message passing interface. *Supercomputer*, 12:56–68. Available at: https://www.researchgate.net/publication/2809665_Mpi_A_Standard_Message_Passing_Interface.
- Wirth, N. (1974). On the design of programming languages. In *IFIP Congress*, volume 74, pages 386–393. Available at: https://people.csail.mit.edu/feser/pld-s23/Wirth_Design.pdf.
- Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A high-performance java dialect. In *In ACM*, pages 10–11. Available at: <https://theory.stanford.edu/~aiken/publications/papers/java98.pdf>.
- Zhang, X., Guo, X., Weng, Y., Zhang, X., Lu, Y., and Zhao, Z. (2023). Hybrid mpi and cuda paralleled finite volume unstructured cfd simulations on a multi-gpu sys-

tem. *Future Generation Computer Systems*, 139:1–16.
DOI: <https://doi.org/10.1016/j.future.2022.09.005>.

Appendix

HighP5 Language Grammar

$\langle \text{program} \rangle ::= \langle \text{components} \rangle$	$\langle \text{repeat_cycle} \rangle ::= \text{Repeat } \langle \text{repeat_control} \rangle \{ \langle \text{compute_flow} \rangle \}$
$\langle \text{components} \rangle ::= \langle \text{component} \rangle \langle \text{components} \rangle \langle \text{component} \rangle$	$\langle \text{repeat_control} \rangle ::= \text{For } \langle \text{Variable_Name} \rangle \text{ In } \langle \text{expr} \rangle \langle \text{step_expr} \rangle$ $\text{Foreach Sub_Partition} \text{While } \langle \text{expr} \rangle$
$\langle \text{component} \rangle ::= \langle \text{tuple} \rangle \langle \text{task} \rangle \langle \text{function} \rangle \langle \text{coordinator} \rangle$	$\langle \text{condition_block} \rangle ::= \text{Where } \langle \text{expr} \rangle \{ \langle \text{compute_flow} \rangle \}$ $\text{Where } \langle \text{field} \rangle \text{ In } \langle \text{expr} \rangle \{ \langle \text{compute_flow} \rangle \}$
$\langle \text{tuple} \rangle ::= \text{Type } \langle \text{Type_Name} \rangle \{ \langle \text{element_defs} \rangle \}$	$\langle \text{epoch_block} \rangle ::= \text{Epoch } \{ \langle \text{compute_flow} \rangle \}$
$\langle \text{element_defs} \rangle ::= \langle \text{element_def} \rangle \langle \text{element_defs} \rangle \langle \text{element_def} \rangle$	$\langle \text{stage_invoke} \rangle ::= \langle \text{Variable_Name} \rangle \{ \langle \text{args} \rangle \}$
$\langle \text{element_def} \rangle ::= \langle \text{names} \rangle \{ \langle \text{static_type} \rangle \}$	$\langle \text{partition} \rangle ::= \text{Partition } \langle \text{arguments} \rangle \{ \langle \text{partition_specs} \rangle \}$
$\langle \text{names} \rangle ::= \langle \text{Variable_Name} \rangle \langle \text{names} \rangle \{ \langle \text{Variable_Name} \rangle \}$	$\langle \text{partition_specs} \rangle ::= \langle \text{partition_spec} \rangle \langle \text{partition_specs} \rangle \langle \text{partition_spec} \rangle$
$\langle \text{static_type} \rangle ::= \langle \text{scalar_type} \rangle \langle \text{static_array} \rangle$	$\langle \text{partition_spec} \rangle ::= \text{Space } \langle \text{Space_ID} \rangle \{ \langle \text{Dimensionality} \rangle \langle \text{dynamic} \rangle \}$ $\text{Space } \langle \text{Space_ID} \rangle \{ \langle \text{Unpartitioned} \rangle \{ \langle \text{names} \rangle \} \}$
$\langle \text{scalar_type} \rangle ::= \langle \text{numeric_type} \rangle \text{String} \text{Range} \langle \text{Type_Name} \rangle$	$\langle \text{dynamic} \rangle ::= \{ \langle \text{Dynamic} \rangle \}$
$\langle \text{numeric_type} \rangle ::= \text{Integer} \text{T_Real Single} \text{Real Double} \text{Character} \text{Boolean}$	$\langle \text{divides} \rangle ::= \{ \text{Divides Space } \langle \text{Space_ID} \rangle \text{ Partitions} \}$ $\{ \text{Divides Space } \langle \text{Space_ID} \rangle \langle \text{Sub_Partitions} \rangle \}$
$\langle \text{static_array} \rangle ::= \text{Array } \langle \text{static_dims} \rangle \text{ Of } \langle \text{scalar_type} \rangle$	$\langle \text{main_dist} \rangle ::= \langle \text{data_spec} \rangle \langle \text{main_dist} \rangle \langle \text{data_spec} \rangle$
$\langle \text{static_dims} \rangle ::= \langle \text{static_dims} \rangle \{ \langle \text{Integer} \rangle \} \{ \langle \text{Integer} \rangle \}$	$\langle \text{data_spec} \rangle ::= \langle \text{var_list} \rangle \{ \langle \text{instr_list} \rangle \langle \text{relativity} \rangle \}$
$\langle \text{task} \rangle ::= \text{Task String } \{ \langle \text{define} \rangle \langle \text{environment} \rangle \langle \text{initialize} \rangle \langle \text{stages} \rangle \}$ $\langle \text{computation} \rangle \langle \text{partition} \rangle$	$\langle \text{var_list} \rangle ::= \langle \text{var} \rangle \langle \text{var_list} \rangle \{ \langle \text{var} \rangle \}$
$\langle \text{define} \rangle ::= \text{Define } \{ \langle \text{definitions} \rangle \}$	$\langle \text{var} \rangle ::= \langle \text{Variable_Name} \rangle \langle \text{Variable_Name} \rangle \{ \langle \text{dimensions} \rangle \}$
$\langle \text{definitions} \rangle ::= \langle \text{definition} \rangle \langle \text{definitions} \rangle \langle \text{definition} \rangle$	$\langle \text{dimensions} \rangle ::= \{ \text{dim} \langle \text{dim} \rangle [0-9]^+ \text{dimensions } \langle \text{dim} \rangle [0-9]^+ \}$
$\langle \text{definition} \rangle ::= \langle \text{names} \rangle \{ \langle \text{dynamic_type} \rangle \} \langle \text{names} \rangle \{ \langle \text{static_type} \rangle \}$ $\langle \text{names} \rangle \{ \langle \text{numeric_type} \rangle \} \text{Reduction}$	$\langle \text{instr_list} \rangle ::= \langle \text{instr} \rangle \langle \text{instr_list} \rangle \{ \langle \text{instr} \rangle \}$
$\langle \text{dynamic_type} \rangle ::= \langle \text{list} \rangle \langle \text{dynamic_array} \rangle$	$\text{nstr} ::= \text{Replicated } \langle \text{Variable_Name} \rangle \{ \langle \text{partition_args} \rangle \}$ $\langle \text{Variable_Name} \rangle \{ \langle \text{partition_args} \rangle \} \text{Padding } \langle \text{partition_args} \rangle \}$
$\langle \text{list} \rangle ::= \text{List Of } \langle \text{static_type} \rangle$	$\langle \text{partition_args} \rangle ::= \langle \text{partition_arg} \rangle \langle \text{partition_args} \rangle \{ \langle \text{partition_arg} \rangle \}$
$\langle \text{dynamic_array} \rangle ::= \langle \text{Dimensionality} \rangle \text{ Array Of } \langle \text{static_type} \rangle \langle \text{format} \rangle$	$\langle \text{partition_arg} \rangle ::= \langle \text{Variable_Name} \rangle \langle \text{Integer} \rangle$
$\langle \text{format} \rangle ::= \{ \text{Format } \langle \text{Type_Name} \rangle \}$	$\langle \text{relativity} \rangle ::= \{ \langle \text{Relative_To Space } \langle \text{Space_ID} \rangle \}$
$\langle \text{environment} \rangle ::= \text{Environment } \{ \langle \text{linkages} \rangle \}$	$\langle \text{sub_dist} \rangle ::= \{ \text{Sub_Partition } \langle \text{Dimensionality} \rangle \langle \text{nature} \rangle \langle \text{data_sub_dist} \rangle \}$
$\langle \text{linkages} \rangle ::= \langle \text{linkage} \rangle \langle \text{linkages} \rangle \langle \text{linkage} \rangle$	$\langle \text{nature} \rangle ::= \text{Ordered} \text{Unordered}$
$\langle \text{linkage} \rangle ::= \langle \text{names} \rangle \{ \langle \text{mode} \rangle \}$	$\langle \text{data_sub_dist} \rangle ::= \langle \text{data_sub_spec} \rangle \langle \text{data_sub_dist} \rangle \langle \text{data_sub_spec} \rangle$
$\langle \text{mode} \rangle ::= \text{Link} \text{Create} \text{Link_or_Create}$	$\langle \text{data_sub_spec} \rangle ::= \langle \text{var_list} \rangle \{ \langle \text{ordered_instr_list} \rangle \}$
$\langle \text{initialize} \rangle ::= \{ \text{Initialize } \langle \text{arguments} \rangle \{ \langle \text{code} \rangle \} \}$	$\langle \text{ordered_instr_list} \rangle ::= \langle \text{ordered_instr} \rangle \langle \text{ordered_instr_list} \rangle \{ \langle \text{ordered_instr} \rangle \}$
$\langle \text{arguments} \rangle ::= \{ \langle \text{names} \rangle \}$	$\langle \text{ordered_instr} \rangle ::= \langle \text{instr} \rangle \langle \text{order} \rangle$
$\langle \text{stages} \rangle ::= \text{Stages } \{ \langle \text{stage_list} \rangle \}$	$\langle \text{order} \rangle ::= \{ \text{Ascends} \text{Descends} \}$
$\langle \text{stage_list} \rangle ::= \langle \text{compute_stage} \rangle \langle \text{stage_list} \rangle \langle \text{compute_stage} \rangle$	$\langle \text{coordinator} \rangle ::= \text{Program } \langle \text{Variable_Name} \rangle \{ \langle \text{meta_code} \rangle \}$
$\langle \text{compute_stage} \rangle ::= \text{Variable_Name } \{ \langle \text{names} \rangle \} \{ \langle \text{code} \rangle \}$	$\langle \text{meta_code} \rangle ::= \langle \text{stmt_block} \rangle$
$\langle \text{computation} \rangle ::= \text{Computation } \{ \langle \text{compute_flow} \rangle \}$	$\langle \text{create_obj} \rangle ::= \text{New } \langle \text{dynamic_type} \rangle \text{New } \langle \text{static_type} \rangle \{ \langle \text{obj_args} \rangle \}$
$\langle \text{compute_flow} \rangle ::= \langle \text{flow_part} \rangle \langle \text{compute_flow} \rangle \langle \text{flow_part} \rangle$	$\langle \text{obj_args} \rangle ::= \langle \text{named_args} \rangle$
$\langle \text{flow_part} \rangle ::= \langle \text{lps_transition} \rangle \langle \text{repeat_cycle} \rangle \langle \text{condition_block} \rangle \langle \text{epoch_block} \rangle \langle \text{stage_invoke} \rangle$	$\langle \text{named_args} \rangle ::= \langle \text{named_arg} \rangle \langle \text{named_args} \rangle \langle \text{named_arg} \rangle$
$\langle \text{lps_transition} \rangle ::= \text{Space } \langle \text{Space_ID} \rangle \{ \langle \text{compute_flow} \rangle \}$	$\langle \text{named_arg} \rangle ::= \langle \text{Variable_Name} \rangle \{ \langle \text{expr} \rangle \}$

$\langle \text{task_invocation} \rangle ::= \text{Execute } \langle ' \langle \text{multi_args} \rangle ' \rangle$
 $\langle \text{multi_args} \rangle ::= \langle \text{multi_arg} \rangle | \langle \text{multi_args} \rangle \langle ' ; ' \rangle \langle \text{multi_arg} \rangle$
 $\langle \text{multi_arg} \rangle ::= \langle \text{Variable_Name} \rangle \langle ' : ' \rangle \langle \text{invoke_args} \rangle$
 $\langle \text{invoke_args} \rangle ::= \langle \text{expr} \rangle | \langle \text{invoke_args} \rangle \langle ' , ' \rangle \langle \text{expr} \rangle$

 $\langle \text{function} \rangle ::= \text{Function } \langle \text{Variable_Name} \rangle \langle ' (\langle \text{function_args} \rangle) ' \rangle \langle \{ \langle \text{code} \rangle \} ' \rangle$
 $\langle \text{function_args} \rangle ::= | \langle \text{function_arg} \rangle | \langle \text{function_args} \rangle \langle ' , ' \rangle \langle \text{function_arg} \rangle$
 $\langle \text{function_arg} \rangle ::= \langle \& \rangle \langle \text{Variable_Name} \rangle | \langle \text{Variable_Name} \rangle$

 $\langle \text{extern_block} \rangle ::= \text{Extern } \langle \{ \rangle \text{Language } \langle \text{String} \rangle \langle \text{header_includes} \rangle \langle \text{extern_links} \rangle \langle \text{native} \rangle \langle \} \rangle$
 $\langle \text{header_includes} \rangle ::= | \text{Header_Includes } \langle \{ \rangle \langle \text{includes} \rangle \langle \} \rangle \langle \text{new_lines} \rangle$
 $\langle \text{includes} \rangle ::= \langle \text{String} \rangle | \langle \text{includes} \rangle \langle ' , ' \rangle \langle \text{String} \rangle$
 $\langle \text{extern_links} \rangle ::= | \text{Library_Links } \langle \{ \rangle \langle \text{library_links} \rangle \langle \} \rangle$
 $\langle \text{library_links} \rangle ::= \langle \text{String} \rangle | \langle \text{library_links} \rangle \langle ' , ' \rangle \langle \text{String} \rangle \langle \text{new_lines} \rangle$

 $\langle \text{code} \rangle ::= \langle \text{stmt_block} \rangle$
 $\langle \text{stmt_block} \rangle ::= \langle \text{stmt} \rangle | \langle \text{stmt} \rangle \langle \text{new_lines} \rangle | \langle \text{stmt} \rangle \langle \text{new_lines} \rangle \langle \text{stmt_block} \rangle$
 $\langle \text{new_lines} \rangle ::= \text{New_Line} | \text{New_Line } \langle \text{new_lines} \rangle$
 $\langle \text{stmt} \rangle ::= \langle \text{parallel_loop} \rangle | \langle \text{sequential_loop} \rangle | \langle \text{if_else_block} \rangle |$
 $\langle \text{extern_block} \rangle | \langle \text{return_stmt} \rangle | \langle \text{reduction} \rangle | \langle \text{expr} \rangle$
 $\langle \text{return_stmt} \rangle ::= \text{Return } \langle \text{expr} \rangle$
 $\langle \text{reduction} \rangle ::= \text{Reduce } \langle ' (\langle \text{Variable_Name} \rangle \langle ' , ' \rangle \langle \text{String} \rangle \langle ' , ' \rangle \langle \text{expr} \rangle) ' \rangle$
 $\langle \text{sequential_loop} \rangle ::= \text{Do In Sequence } \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle \text{ For } \langle \text{id} \rangle \text{ In } \langle \text{sloop_attr} \rangle$
 $\langle \text{sloop_attr} \rangle ::= \langle \text{field} \rangle \langle \text{step_expr} \rangle | \langle \text{field} \rangle \text{ O_AND } \langle \text{expr} \rangle$
 $\langle \text{parallel_loop} \rangle ::= \text{Do } \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle \text{ For } \langle \text{index_ranges} \rangle$
 $| \text{Do } \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle \text{ While } \langle \text{expr} \rangle$
 $\langle \text{index_ranges} \rangle ::= \langle \text{index_range} \rangle | \langle \text{index_ranges} \rangle \langle ' ; ' \rangle \langle \text{index_range} \rangle$
 $\langle \text{index_range} \rangle ::= \langle \text{names} \rangle \text{ In } \langle \text{Variable_Name} \rangle \langle \text{restrictions} \rangle | \langle \text{names} \rangle \text{ In } \langle \text{Variable_Name} \rangle \langle ' , ' \rangle \langle \text{Dimension_No} \rangle \langle \text{restrictions} \rangle$
 $\langle \text{restrictions} \rangle ::= | \text{O_AND } \langle \text{expr} \rangle$
 $\langle \text{if_else_block} \rangle ::= \text{If } \langle ' (\langle \text{expr} \rangle) ' \rangle \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle \langle \text{else_block} \rangle$
 $\langle \text{else_block} \rangle ::= | \langle \text{else} \rangle | \langle \text{else_if} \rangle \langle \text{else_block} \rangle$
 $\langle \text{else} \rangle ::= \text{Else } \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle$
 $\langle \text{else_if} \rangle ::= \text{Else If } \langle ' (\langle \text{expr} \rangle) ' \rangle \langle \{ \rangle \langle \text{stmt_block} \rangle \langle \} \rangle$
 $\langle \text{step_expr} \rangle ::= | \text{Step } \langle \text{expr} \rangle$
 $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \langle \text{binary_operator} \rangle \langle \text{expr} \rangle | \langle ' ! ' \rangle \langle \text{expr} \rangle | \langle \{ \rangle \langle \text{expr} \rangle \langle \dots \rangle \langle \text{expr} \rangle \langle \} \rangle$
 $| \langle \text{constant} \rangle | \langle \text{field} \rangle | \langle \text{function_call} \rangle | \langle \text{task_invocation} \rangle | \langle \text{create_obj} \rangle$
 $| \langle \text{expr} \rangle \text{ At } \langle ' (\langle \text{epoch_lag} \rangle) ' \rangle | \langle ' (\langle \text{expr} \rangle) ' \rangle$
 $\langle \text{b_op} \rangle ::= \langle ' + ' \rangle | \langle ' - ' \rangle | \langle ' * ' \rangle | \langle ' / ' \rangle | \langle ' \% ' \rangle$
 $| \text{LSHIFT} | \text{RSHIFT} | \text{POWER}$
 $| \text{BITWISE_AND} | \text{BITWISE_OR} | \text{BITWISE_XOR}$
 $| \text{LT} | \text{GT} | \text{OR} | \text{AND} | \langle ' == ' \rangle | \langle ' != ' \rangle | \text{LTE} | \text{GTE} | \langle ' = ' \rangle$

$\langle \text{constant} \rangle ::= \langle \text{sign} \rangle \langle \text{number} \rangle | \text{Boolean} | \text{Character} | \text{String}$

$\langle \text{sign} \rangle ::= | \langle ' - ' \rangle$

$\langle \text{number} \rangle ::= \text{Integer} | \text{Real_Single} | \text{Real_Double}$

$\langle \text{field} \rangle ::= \langle \text{id} \rangle \text{Field} | \langle \text{field} \rangle \langle ' . ' \rangle \langle \text{id} \rangle | \langle \text{field} \rangle \langle \{ \rangle \langle \text{array_index} \rangle \langle \} \rangle$

$\langle \text{array_index} \rangle ::= \langle \text{expr} \rangle | \langle \text{expr} \rangle \langle ' . . . ' \rangle \langle \text{expr} \rangle | \langle ' . . . ' \rangle$

$\langle \text{function_call} \rangle ::= \text{Variable_Name } \langle ' (\langle \text{args} \rangle) ' \rangle$

$\langle \text{arg} \rangle ::= \text{Space Space_ID } \langle ' : ' \rangle \text{Variable_Name} | \langle \text{expr} \rangle$

$\langle \text{args} \rangle ::= | \langle \text{arg} \rangle | \langle \text{args} \rangle \langle ' , ' \rangle \langle \text{arg} \rangle$

$\langle \text{epoch_lag} \rangle ::= \text{Current} | \text{Current } \langle ' - ' \rangle \text{Integer}$

$\langle \text{id} \rangle ::= \text{Variable_Name} | \text{Dimension_No} | \text{Range} | \text{Local} | \text{Index}$

Scalability Experiments on Distributed Shared-memory Environment

We used the *parallel* partition in a compute cluster of our organization as the back-end for the distributed-shared memory compiler. The *parallel* partition has two 2.50 GHz Intel Xeon E5-2670 CPUs in each computing node. An Infiniband interconnect connects these nodes where each CPU has ten cores and three cache levels. Ten CPU cores share a 25 MB L3 cache segment. Then each has a 256 KB L2 and a 32 KB L1 cache. In the

Although the *parallel* partition has many more nodes, our allocation restriction limited the nodes count to 50 in the experiments. Thus the maximum number of cores we used in the experiments is 1000. The native MPI compiler in this platform is mpic++, which 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 HighP5 executables nor reference implementations. Therefore, we ignored that capacity.

We did both strong scaling experiments by gradually increasing the number of cores (aka. parallelism) for a fixed problem size and weak scaling experiments by increasing the problem size proportionately with the increasing number of CPU cores. Furthermore, we distinguished two speedups for the HighP5 programs against the sequential baseline code in the distributed shared memory execution platform. The computation speedup only considers the running time of the tasks, and the execution speedup includes any other overhead for resource preparation and communication buffers setup added with the task computation time in the parallel versions.

Figure 12 shows the strong and weak scaling results for the block matrix-matrix multiplication problem. 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 we consider the additional overhead. The execution speedup against the sequential implementation for the 1000-Cores version is only 238 times. 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 due to a maximum memory per node limitation set on the cluster 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. We should be able to reduce the cost drastically in the future by parallelizing the steps of the resource preparation process.

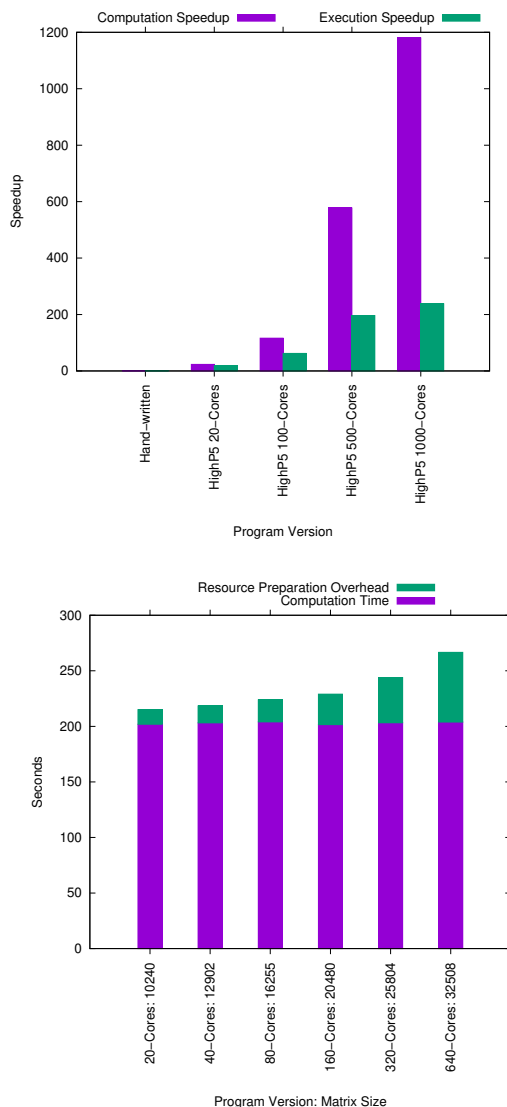


Figure 12. Strong (matrix size 10, 239 × 10, 239) and weak scaling results for block matrix-matrix multiplication on parallel cluster

The speedups for the strong scaling experiment with the block LU factorization problem (Figure 13) are relatively modest compared to the block matrix-matrix multiplication problem. We expected this as each iteration of LU factorization involves multiple collective MPI communications

on top of intra-node CPU core synchronizations. The differences between computation and execution speedups are minor as the cost of data structures and communication resources setup is negligible compared to the cost of actual task computation. We further distinguished between time spent on actual computation and communication for the weak scaling experiment to understand program behavior better. There are fluctuations in the computation time, but the overall trend is flat. On the other hand, communication time tends to increase gradually as the cost of collective communication increases with increasing problem size and processor count.

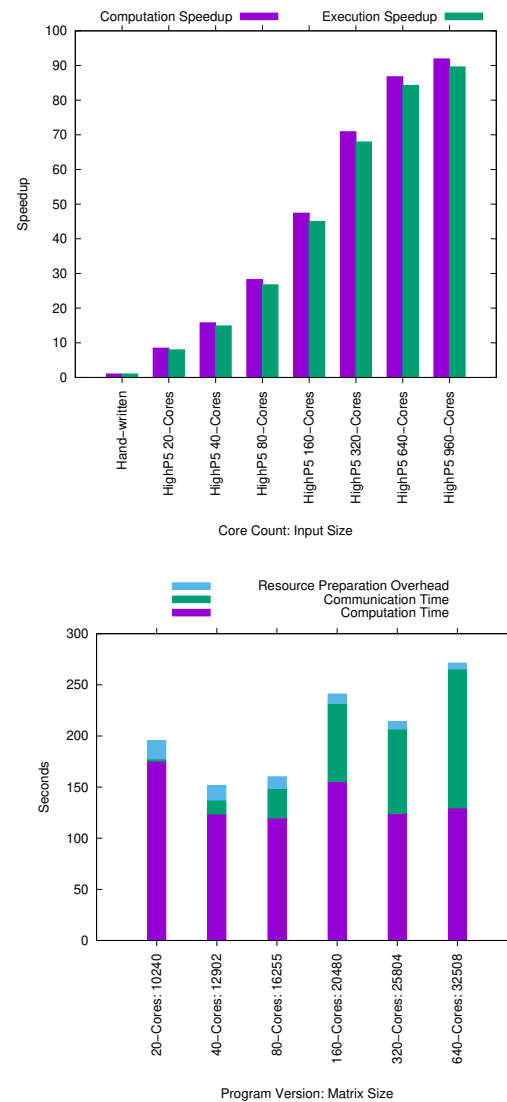


Figure 13. Strong (argument matrix size 20, 480 × 20, 480) and Weak scaling results for block LU Factorization on parallel cluster

For strong scaling experiments with Monte Carlo area estimation and 5-point iterative stencil problems, we used the same problem configurations we used for respective experiments in the multicore platform. Figures 14 and 15 show that the results are promising for both problems. Furthermore, both problems exhibit almost perfect weak scaling behavior by keeping the speedup flat for a proportionate increase of problem size and the degree of parallelism. This behavior is because the former is embarrassingly parallel, and the latter uses localized communications only whose cost gets evenly

distributed with increasing parallelism.

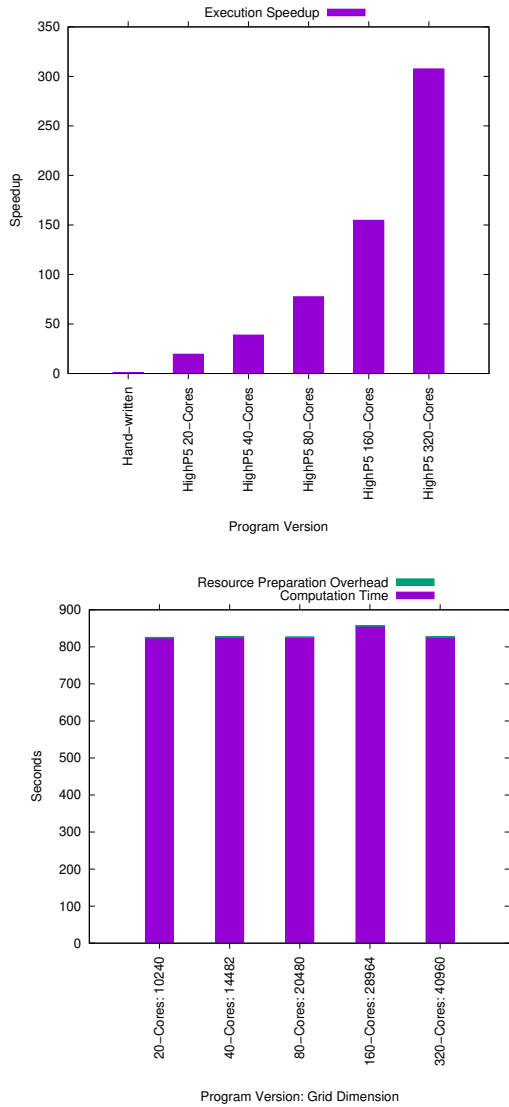


Figure 14. Strong and Weak scaling results for Monte Carlo area estimation (1000 samples per cell) on parallel cluster

We did only the strong scaling experiment for the conjugate gradient problem. Given our lack of improvement for increasing threading parallelism per process in the multicore environment, we decided to scale up the resources in terms of node count instead of 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 as shown in Figure 16 are the most significant revelation in the performance testing in this platform.

The computation does improve with increasing parallelism, but the increasing cost of the concomitant overhead calculation tramples any performance gain in the overall execution. Our investigation revealed that the repeated creation of LPU data structures and communication resources for resolving inter-task data dependency is much higher than the running time of the actual tasks that collaboratively solve the conjugate gradient problem. This observation reveals that minimization of inter-task transitions overhead should be our core focus on compiler improvement in the distributed shared memory platform.

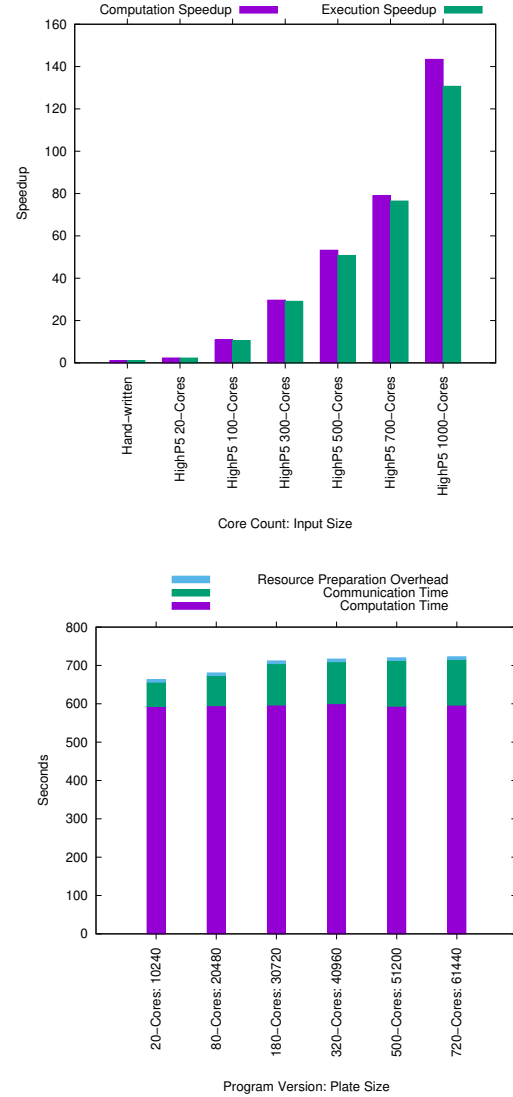


Figure 15. Strong and Weak scaling results for 5-point stencil (10,000 iterations) on parallel cluster

HighP5 LU Factorization Code

```

1 Program (args) {
2   luEnv = new TaskEnvironment(name: "Block LU Factorization")
3   bind_input(luEnv, "a", args.input_matrix_file)
4   execute(task: "Block LU Factorization";
5     environment: luEnv; partition: args.block_size)
6   bind_output(luEnv, "u", args.upper_matrix_file)
7   bind_output(luEnv, "l", args.lower_matrix_file)
8   bind_output(luEnv, "p", args.pivot_matrix_file)
9 }
10
11 Task "Block LU Factorization":
12   Define:
13     a, u, l: 2d Array of Real double-precision
14     p: 1d Array of Integer
15     l_row, l_column, p_column: 1d Array of Real double-precision
16     u_block, l_block: 2d Array of Real double-precision
17     pivot: Integer Reduction
18     k, r, block_size: Integer
19     row_range: Range
20   Environment:
21     a: link
22     u, l, p: create
23   Initialize:
24     u.dimension1 = l.dimension1 = a.dimension2
25     u.dimension2 = l.dimension2 = a.dimension1

```

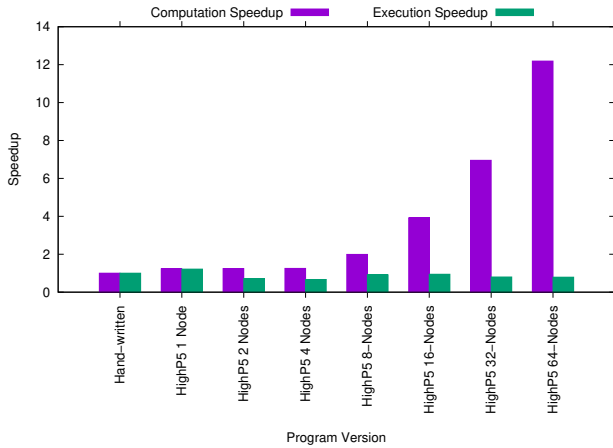


Figure 16. Strong scaling results for conjugate gradient on parallel cluster

```

26 p.dimension = a.dimension1
27 block_size = partition.b
28 l_row.dimension = l.dimension2
29 l_column.dimension = p_column.dimension = l.dimension1
30 u_block.dimension1 = u.dimension1
31 u_block.dimension2.range.min
32   = l_block.dimension1.range.min = 0
33 u_block.dimension2.range.max
34   = l_block.dimension1.range.max = block_size - 1
35 l_block.dimension2 = l.dimension2
36 Stages:
37 prepareLU(a, u, l) {
38   do { u[j][i] = a[i][j] } for i, j in a
39   do { l[i][i] = 1 } for i in l
40 }
41 calculateRowRange(a, row_range, block_size) {
42   last_row = r + block_size - 1
43   if (last_row > a.dimension1.range.max) {
44     last_row = a.dimension1.range.max
45   }
46   row_range.min = r
47   row_range.max = last_row
48 }
49 selectPivot(pivot, u, k) {
50   do { reduce(pivot, "max_entry", u[k][j])
51     } for j in u and j >= k
52 }
53 storePivot(p, k, pivot) {
54   p[k] = pivot
55 }
56 interchangeRows(pivot, k, u, l) {
57   do { pivot_entry = u[i][k]
58     u[i][k] = u[i][pivot]
59     u[i][pivot] = pivot_entry
60   } for i in u and i >= k
61   do { pivot_entry = l[i][k]
62     l[i][k] = l[i][pivot]
63     l[i][pivot] = pivot_entry
64   } for i in l and i < k
65 }
66 updateL(l, k, l_row) {
67   do { l[k][j] = u[k][j] / u[k][k]
68     u[k][j] = 0
69     l_row[j] = l[k][j]
70   } for j in l and j > k
71 }
72 updateURowsBlock(u, l_row, k, row_range) {
73   do { u[i][j] = u[i][j] - l_row[j] * u[i][k]
74     } for i, j in u and i > k
75     and i <= row_range.max and j > k
76 }
77 collectLColParts(l_column, l, k, row_range) {
78   do { l_column[i] = l[i][k]
79     } for i in l and i >= row_range.min and i < k
80 }
81 generatePivotColumn(p_column, l_column, row_range, k) {

```

```

82 do { p_column[i] = l_column[i]
83 } for i in l_column and i >= row_range.min and i < k
84 }
85 updateUColsBlock(u, p_column, k, row_range) {
86 do {u[i][k] = u[i][k] - u[i][j] * p_column[j]
87 } for i, j in u and i > row_range.max
88   and j >= row_range.min and j < k
89 }
90 copyUpdatedUBlock(u_block, row_range, u) {
91 do {column = j - row_range.min
92   u_block[i][column] = u[i][j]
93 } for i, j in u and i > row_range.max
94   and j >= row_range.min and j <= row_range.max
95 }
96 copyUpdatedLBlock(l_block, row_range, l) {
97 do {row = i - row_range.min
98   l_block[row][j] = l[i][j]
99 } for i, j in l and i >= row_range.min
100   and i <= row_range.max and j > row_range.max
101 }
102 saxpy(u, u_block, l_block, row_range) {
103 do {
104   total = 0.000000
105   do {total = total + u_block[i][m] * l_block[m][j]
106     } for m in u_block
107   u[i][j] = u[i][j] - total
108 } for i, j in u and i > row_range.max and j > row_range.max
109 }
110 Computation:
111 Space A {
112   Space B {
113     prepareLU(a, u, l)
114   }
115   Repeat for r in a.dimension1.range step block_size {
116     calculateRowRange(a, row_range, block_size)
117     Repeat for k in row_range {
118       Space B {
119         Where k in u.local.dimension1.range {
120           selectPivot(Space A: pivot, u, k)
121         }
122       }
123       storePivot(p, k, pivot)
124       Space B {
125         If k != pivot { interchangeRows(pivot, k, u, l) }
126         Where k in l.local.dimension1.range {
127           updateL(l, k, l_row)
128         }
129         updateURowsBlock(u, l_row, k, row_range)
130         collectLColParts(l_column, l, k, row_range)
131       }
132       generatePivotColumn(p_column, l_column, row_range, k)
133       Space B {
134         updateUColsBlock(u, p_column, k, row_range)
135       }
136     }
137   }
138   copyUpdatedUBlock(u_block, row_range, u)
139   Where r in l.local.dimension1.range {
140     copyUpdatedLBlock(l_block, row_range, l) }
141   Space C {
142     Repeat foreach sub-partition {
143       saxpy(u, u_block, l_block, row_range)
144     }
145   }
146 }
147 }
148 Partition(b):
149 Space A <un-partitioned> {
150   a, p, l_column, l_row, p_column, l_block, u_block
151 }
152 Space B <ld> divides Space A partitions {
153   a<dim2>, u<dim1>, u_block<dim1>,
154   l<dim1>, l_column: block_stride(b)
155   l_row, p_column, l_block: replicated
156 }

```

```
157 Space C <2d> divides Space B partitions {
158   u: block_size(b, b)
159   u_block: block_size(b), replicated
160   l_block: replicated, block_size(b)
161   Sub-partition <1d> <unordered> {
162     u_block<dim2>, l_block<dim1>: block_size(b)
163   }
164 }
```

Listing 6: HighP5 Code for Block LU Factorization.