IT: Machine Independent Programming on Hierarchically Nested Machines

Muhammad N. Yanhaona & Andrew S. Grimshaw

Contents

Introduction 3
Related Work 5
Parallel Programming on Multicore 5
Parallel Programming for Distributed Memory 5
Co-processor Languages 6
Type Architecture Based Languages 6
Domain Specific Parallel Languages 6
PCubeS + IT Programming Paradigm 7
The PCubeS Type Architecture 9
IT Parallel Programming Language 10
Data Structures 11
The Coordinator Program & Task Execution 12
File I/O 13
IT Tasks 14
Logical Processing Spaces and Units 14
Execution 15
Task Definition: 17
Define Section 17
Environment Section 17
Initialize Section 18
Stages Section 18
A Note on the Declarative Syntax 19
Computation Section 20
Partition Section 21
Mapping Logical Spaces to Physical Spaces 23
Degrees of Parallelism in an IT Program 24

IT Compilers 25
Compiler for Multicore CPUs 25
Compiler for Distributed + Shared Memory Hybrid Machines 25
Conclusion 26
Appendix A: IT Grammar –Version 1 27
Program Definition 27
Class Definition 27
Task Definition 27
Define Section 27
Environment Section 28
Initialize Section 28
Stages Section 28
Computation Section 28
Partition Section 28
Expressions & Statements 29
Coordinator Program Definition 29
Function Definition 29
Appendix B: IT Data File Formats 30
Appendix C: Program Examples 30
LU Factorization 30
Iterative Stencil 33
Radix Sort 35
Bibliography 37

Introduction

Andrews in his "Concurrent Programming: Principles and Practice" expresses that "concurrent programs are to sequential programs what chess is to checkers" [1]. We believe people experienced in both kinds of programming will largely agree to his statements. But what makes parallel programming so difficult? Although there are differences of opinions, none blames the difficulty of devising a parallel algorithm, as opposed to a sequential algorithm, as the core problem.

It is true that construing a parallel algorithm sometimes requires considerably more ingenuity compared to its sequential counterpart, but one can expect that a programmer who is knowledgeable about his problem can come up with a parallel algorithm (or get help from his community) to solve it. Rather, most problems occur in the process of translating the parallel algorithm into a program, as deftly described by Snyder in his seminal work "Type Architectures, Shared Memory, and the Corollary of Modest Potential" [25].

Correctly implementing interactions between parallel pieces of the program can be difficult. So much so that sometimes the original algorithm gets lost in the plethora of interaction primitives. The biggest problem is, however, extracting good performance from a parallel machine. Features like memory organization, cache hierarchies, interconnection network structure, speed and bandwidth of memory access and data transfers that are orthogonal to the algorithm by and large determine the performance of its program implementation. The influence of hardware in the performance of a parallel program is so profound and universally agreed that that the present trend in parallel computing is to devise algorithms specific to hardware architectures, e.g., MAGMA for GPUs [28] or MKL for Intel processors.

The original Von Neumann architecture that portrays a computer with a flat memory and a fetch/execute computation cycle works well for most forms of sequential programming. That architecture is, however, inadequate for parallel programming for the same reason it is so successful for sequential computing: it relieves the programmer from reasoning about hardware features. Further, it is inappropriate to treat a parallel machine as a uniform collection of Von Neumann machines given that contemporary execution platforms have shared memories with deep cache hierarchies, distributed shared memories, distributed machines with one or more co-processors, and so on. Exploiting all these features appropriately is essential for good performance of a parallel program.

An accelerated demand for better performance on compute intensive problems compounded by a consistent trend towards increasing hardware complexity leads to the situation where many researchers concluded that "developing programming models that productively enable development of highly efficient implementations of parallel applications is the biggest challenge facing the deployment of future many-core systems [2]." This is a monumental shift from the era when Gelernter & Carriero's statement that "the real world programmer has better things to do than writing/rewriting programs in a new parallel language [17]" could be debated upon.

Although hardware's importance in shaping the performance of a parallel program has been unanimously admitted, the approaches taken to address this issue widely differ in various programming models. The currently dominant parallel programming strategy is to extend some conventional sequential programming language (in particular C and Fortran) with parallelization primitives as done in MPI [29] for distributed memory machines, CUDA [21] for GPCPUs, and Pthreads [7] for multi-core CPUs. These tools and primitives may be perfect in their designated environments, but with the advent of hybrid and hierarchical architectures they fall short of utilizing many important hardware features. For example, neither MPI nor Pthreads addresses cache hierarchies. Combining these primitives in a program for hybrid platforms – though feasible -- is often difficult for an average programmer. Further, there is a long-standing complaint about the programming difficulty when using these primitives.

The strategy taken by some high-level programming language initiatives such as Co-array Fortran [22], UPC [15], X10 [12], Chapel [10], or Fortress [26] is to fix a model of the target execution platform (in these cases the partitioned global address space (PGAS) model [14]) and provide primitives to use features of the model that are implemented with appropriate low level primitives available in the platform. Consequently the programmer is relieved from the specific details of a particular hardware platform. This strategy reduces the programming complexity, but the limitation of the machine model has become a barrier for these initiatives to achieve good performance in hardware architectures with complex memory hierarchies and hybrid execution units. Recent efforts to extend PGAS for hybrid and hierarchical architectures [5] [16] show some promise, but do not develop a coherent architectural model and associated programming paradigm.

We are developing the IT programming language for high-performance, mostly data parallel programming (IT supports other forms of parallelism but the focus is on data parallelism) as an alternative to both low-level parallel tools and contemporary high-level parallel languages discussed above [31]. At its core, IT has a new type architecture, called PCubeS [30] that describes a parallel execution platform as a hierarchy of processing spaces each capable of doing computation and holding data. A PCubeS description not only captures the processing hierarchy of spaces but also important programming attributes such as SIMD capacity, memory access bandwidth, and data transfer latency between spaces. Thus PCubeS provides a highly detailed and performance oriented representation of parallel architectures that is applicable to most machines used in parallel computing at the present.

In an IT program, computations take place in corresponding hierarchy of logical processing spaces each of which may impose a different partitioning for a data structure. To generate an executable the programmer needs to explicitly map these logical spaces to physical spaces of PCubeS. Efficient partitioning and mapping is critical for the good performance of a program. The implementation of computation and communication that is determined by the compiler depends on the features of the physical spaces computations have been mapped to.

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

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

IT's contribution is in its 1) explicit program mapping to multilayered and hybrid physical architectures that we believe will scale to deeper and deeper architectures, and 2) a clear separation of concerns between describing the computation and describing how the data will be decomposed and scattered across architectural components, while 3) leaving the required complex communication and synchronization up to the compiler and run-time system.

Before we dive into the detail of the programming paradigm, and the syntax and features set of the language, we discuss some contemporary related work on parallel programming techniques to provide some context.

Related Work

Parallel Programming on Multicore

Intel's Cilk Plus [6] is a C/C++ extension supporting fork-join task parallelism and vector operations on multicore CPUs. It was originally designed for high-performance computing then later became a more general purpose language. The programmer is responsible for identifying and exploiting parallelism in the code using features like spawn, sync, parallel loop, array extensions; and the runtime engine decides how to distribute parallel tasks into processors and vectorize array operations.

A popular library extension on C and Fortran for task level parallelism is POSIX threads or Pthreads [7]. Pthreads has thread creation, join, termination routines and the support for different forms of explicit synchronization of those threads. Performance of a Pthreads program can be fine-tuned using programmatic thread affinity management, combining NUMA memory allocation libraries in NUMA architectures and so on. Pthreads is a very low level but flexible primitive.

OpenMP [13], an extremely popular parallelization technique in multicore, takes a very different route for parallelization from Cilk and Pthreads. In OpenMP, a sequential C or Fortran program is parallelized using OpenMP pragma constructs. There are parallel pragmas for loops, code blocks, and reductions with few attributes controlling their execution; but for the most part the underlying parallel execution is controlled by the OpenMP runtime that implements a thread-based parallelism. The types of parallelism that can be expressed using OpenMP are limited, but it remains immensely popular because of its flexibility and simplicity. Achieving good portable performance can be quite a challenge with OpenMP though unless the programmer explicitly tunes the code to the underlying physical architecture. In our experience, particularly with students, getting the parallel code to run is easy; getting it to run quickly is the trick.

Although programs using these tools and languages often achieve significant performance boost up over their sequential bases through concurrency and vectorization, there is a lack of exposure of memory/cache hierarchies that we believe may limit their efficiency, in particular, given the trends toward more cache levels and larger cache capacities in modern multicore CPUs.

Parallel languages and tools for multicore often focus less on extracting the best performance from a machine and more on enabling interesting concurrent paradigms; Scala [23] and Erlang [9] are two good examples of the latter. Scala combined java with functional parallelism. In Scala functions are first-class language objects. Scala supports actor based concurrency, parallel collections, and asynchronous programming using features such as future and promise. The philosophy is to support a variety of sequential and parallel primitives to enable a wide variety of applications.

Like Scala, Erlang is a functional programming language. Its focus is fault-tolerance and real-time execution. An Erlang program is written as a collection of processes (that are not operating system threads or processes but something in between those two) that communicate using message passing.

Parallel Programming for Distributed Memory

The Message Passing Interface (MPI) [29] has been the standard for parallel programming in distributed memory for almost two decades. MPI is an implementation of the communicating sequential processes [18] over C and Fortran. 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. Programming using MPI is, however, consistently been accused for being difficult and error-prone. Nevertheless, MPI remains popular because of its ability to provide good performance on pure distributed machines. The advent of multicore CPUs and accelerators in supercomputers and compute cluster limits MPI's supremacy. MPI's treatment of the execution environment as a flat collection of processes with uniform characteristics no longer holds. To overcome the shortcoming MPI has been combined with OpenMP, Pthreads, and other multicore parallelization tools [8]. Unfortunately, this hybrid strategy makes programming even harder and more error-prone.

The DARPA high productivity languages X10 [12] and Chapel [10]; or PGAS languages such as Co-array Fortran [22], UPC [15], and Titanium [32] can be viewed as efforts to find an alternative to MPI for flexible and expressive parallel programming paradigm without losing performance. All 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.

These languages suffer from the same performance problem in hierarchical and hybrid architectures as MPI does because their flat partitioning of the memory. Given that they are not clear winners against MPI on Non-uniform Cluster Computers where PGAS [12] is applicable, their future success in machines with deep memory hierarchies and non-uniform compute capacities is uncertain.

Co-processor Languages

CUDA [21] is the programming tool used for NVIDIA GPGPUS. Initially it was deemed to be difficult, but GPGPUs popularity in high performance computing gave it a rapid surge in acceptance. CUDA follows the footsteps of earlier SIMD/SPMD languages (C*, pC++, C**, DataParallel C) with additional features to manipulate memory unique to NVIDIA architectures. A CUDA program is a C or Fortran program with functions to be offloaded to the accelerators and additional instructions for data transfers between a CPU host and accompanying accelerators. The programmer needs to understand the inner working of the accelerator threads and details of memory access to make his offloading functions to behave correctly and efficiently.

The OpenCL [27] standard has strong CUDA heritage and was originally targeted for accelerators. Recently, it has become more of a standard for parallel programming for multicore CPUs and GPUs alike. In our opinion, taking a special purpose model and applying it to general purpose computing has its limitations. It will be either too complex to allow all special purpose attributes to retain efficiency in accelerators that are hardly useful in general purpose architectures; or the model has to shed off its key special purpose attributes and become inefficient. Therefore, we are doubtful about OpenCL's future success beyond its use in accelerators and embedded applications. Further, the OpenCL model is inappropriate for distributed machines.

Type Architecture Based Languages

There are few recent languages that focus on a type architecture foundation as IT does. Among them ZPL [11] and Legion [4] are mentionable. ZPL uses Snyder's original CTA type architecture [25] and found to perform 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 [4] is a functional language follow-on of Sequoia [3] that uses a type architecture called Parallel Memory Hierarchy. Surprisingly, the type architecture aspect is not highlighted in Legion. Similar to IT, Legion supports hierarchical partitioning and programmer controlled mapping of those partitions to different layers of a hardware. 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 replete in data parallelism. The central problem is multi-dimensional arrays are much better suited for representing data structures in these problems than lists. Representing arrays as lists is possible but that compromises both program's performance and its readability.

Domain Specific Parallel Languages

Finally, developing domain specific languages (DSL) and toolkits is an emerging trend in high performance parallel programming [19]. A DSL grows from a general purpose language providing 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 [24], and the performance of DSL programs are often competitive to general purpose language implementations. Nevertheless, as their nature suggests, they cannot be the solution for high performance parallel computing in general.

PCubeS + IT Programming Paradigm

The PCubeS + IT programming paradigm asks a programmer to break a program in a top-down manner into cooperating tasks, then individual tasks into one or more sequences of data parallel computation stages (computations are similar to procedures in conventional programming languages). The execution order of tasks is determined by their data dependencies and availability of physical resources in the target platform. The execution platform and the data structures located in it constitute a program's environment. The execution of a task changes the environment by updating existing data structures and adding new structures that will be needed at later time for subsequent tasks. By-products of tasks that are not finally written to external files are automatically garbage collected. An example of IT task breakdown can be a conjugate gradient program where matrix-vector multiplication, vector dot-products and so on are the individual tasks. The granularity of the individual tasks is subjective, but the guideline is to restrict tasks to independent program units that use a particular arrangement of data.

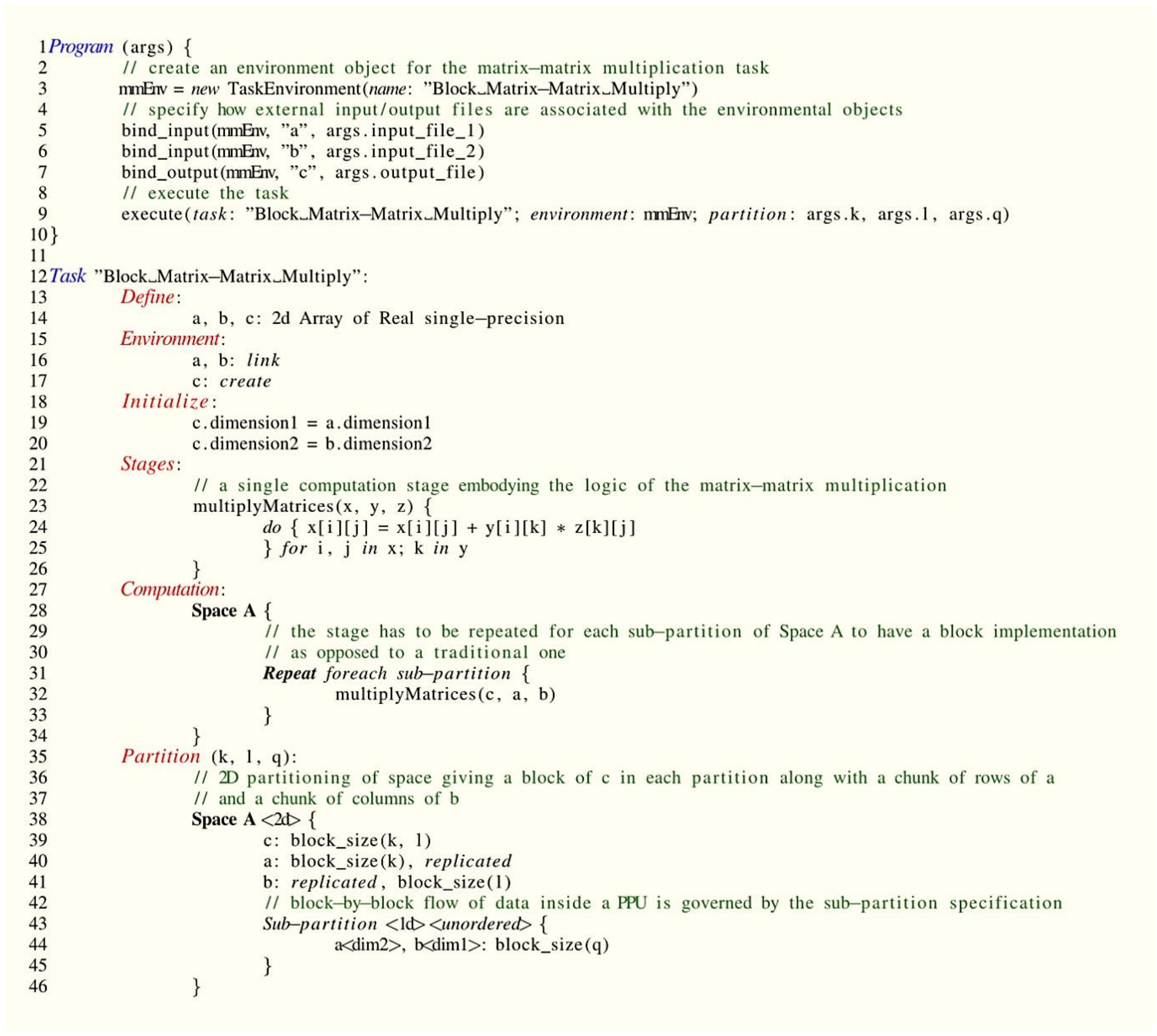
Beneath the task level parallelism resides the data parallelism of computation stages within a task. A task specifies the logic of the parallel algorithm as a sequence of computation stages executing in one or more ***Logical Processing Spaces (LPS)*** within a ***Computation Section***. The relationship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

Finally, the calculation inside a compute stage is written using a declarative syntax supporting parallel loops, reductions, and conventional arithmetic and logical operators. When writing a compute stage, the programmer assumes the locality of any data structure used within and is encouraged to focus on identifying instruction level parallelisms through the declarative parallel constructs. Depending on the support available in the target platform these parallel constructs may get translated into vectorized or SIMD instructions or just execute sequentially. Again, the nature of the translation, consequently its efficiency, depends on the mapping of LPSes to PPSes which is under programmer's control. The declarative pseudo-code like syntax (we discuss it later) is essential for IT's portability across diverse platforms.

Listing 1 presents an example IT single-task program as an illustration of the aforementioned concepts.

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

More formerly, the computation flow (Line 27 to 34) specifies that within the confinement of Space-A partition the compute stage *multiplyMatrices* should execute in each of its sub-partitions one after another. The partition section (Line 35 to 46) dictates that each independent Space-A partition will have a *k-by-l* blocks of *c*, *k* rows of *a* and *l* columns of *b*. Then within a partition, *q* columns of *a* and *q* rows of *b* will be operated at once for that *k* rows and *l* columns respectively.



```
Program (args) {
        // create an environment object for the matrix–matrix multiplication task
        mmEnv = new TaskEnvironment(name: "Block_Matrix–Matrix_Multiply")
        // specify how external input/output files are associated with the environmental objects
        bind_input(mmEnv, "a", args.input_file_1)
        bind_input(mmEnv, "b", args.input_file_2)
        bind_output(mmEnv, "c", args.output_file)
        // execute the task
        execute(task: "Block_Matrix–Matrix_Multiply"; environment: mmEnv; partition: args.k, args.l, args.q)
}

Task "Block_Matrix–Matrix_Multiply":
        Define:
                a, b, c: 2d Array of Real single–precision
        Environment:
                a, b: link
                c: create
        Initialize:
                c.dimension1 = a.dimension1
                c.dimension2 = b.dimension2
        Stages:
                // a single computation stage embodying the logic of the matrix–matrix multiplication
                multiplyMatrices(x, y, z) {
                        do { x[i][j] = x[i][j] + y[i][k] * z[k][j]
                        } for i, j in x; k in y
                }
        Computation:
                Space A {
                        // the stage has to be repeated for each sub–partition of Space A to have a block implementation
                        // as opposed to a traditional one
                        Repeat foreach sub–partition {
                                multiplyMatrices(c, a, b)
                        }
                }
        Partition (k, l, q):
                // 2D partitioning of space giving a block of c in each partition along with a chunk of rows of a
                // and a chunk of columns of b
                Space A <2d> {
                        c: block_size(k, l)
                        a: block_size(k), replicated
                        b: replicated, block_size(l)
                        // block–by–block flow of data inside a PPU is governed by the sub–partition specification
                        Sub–partition <1d> <unordered> {
                                a<dim2>, b<dim1>: block_size(q)
                        }
                }
```

Listing 1: An IT Block Matrix-Matrix Multiplication Program

For example, if the sole space of Listing 1 is mapped to the symmetric multiprocessor (SMs) of a GPGPU then the degree of parallelism will be equal to the number of SMs in the hardware and the Space-A LPUs will be multiplexed on them to be executed one after another. The threads of an SM will execute the compute stage in SIMD fashion over data loaded on SM's shared memory. The partition arguments *k*, *l*, and *q* should be chosen so that data for a sub-partition can be held within the limited shared memory.

On the other hand, if Space-A is mapped to the cores of a multicore CPU then the degree of parallelism will be equal to the number of cores and each core will execute the instructions of *multiplyMatrices* stage sequentially. Then a different setting for the partition arguments, based on cache capacities, may be appropriate.

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

The problem is it is too much to expect from an average programmer to be able to make sense of various features of contemporary parallel architectures that are widely different. Exploiting those features is feasible only after a proper understanding of the hardware.

This is where the PCubeS type architecture comes into play. It provides a uniform standard to conveniently describe salient programming features of most parallel architectures, as we discuss in the next section.

The PCubeS Type Architecture

The formal description of the Partitioned Parallel Processing Spaces (PCubeS) type architecture is as follows.

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

PCubeS measures the compute capacity of a PPS as the number of concurrent floating point operations it can do. Transfer of data between PPSes as well as memory accesses within a single PPS are expressed in terms of the number of concurrent transactions. A transaction is characterized by the volume of data it carries (its width) and its latency. The memory capacity of a PPS is the size of its physical memory along with the memory transaction characteristics. Finally, note that PPUs are uniform within a single PPS – not necessarily across multiple PPSes.

The philosophy behind PCubeS, detailed discussion of its individual attributes, and how it can describe a wide variety of parallel architectures are available here [30]. Here we will discuss the PCubeS description of a single multicore cluster that we frequently use as an IT target platform.

The cluster is called *Hermes*. It has 4 64-core nodes connected by a 10-GB Ethernet interconnect. Each node is composed of 4 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 (Only one PPU has been expanded at each PPS in the above figure, but the PPUs are uniform).

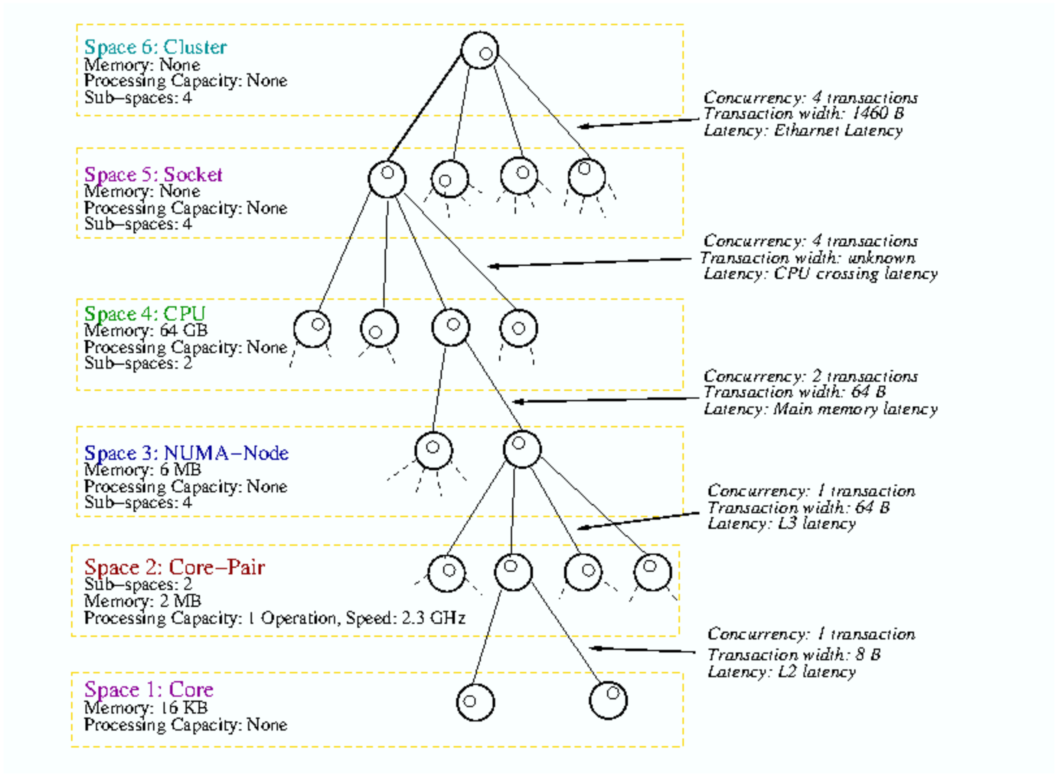

Space 6: Cluster
Memory: None
Processing Capacity: None
Sub–spaces: 4
Concurrency: 4 transactions
Transaction width: 1460 B
Latency: Ethernet Latency
Space 5: Socket
Memory: None
Processing Capacity: None
Sub–spaces: 4
Concurrency: 4 transactions
Transaction width: unknown
Latency: CPU crossing latency
Space 4: CPU
Memory: 64 GB
Processing Capacity: None
Sub–spaces: 2
Concurrency: 2 transactions
Transaction width: 64 B
Latency: Main memory latency
Space 3: NUMA–Node
Memory: 6 MB
Processing Capacity: None
Sub–spaces: 4
Concurrency: 1 transaction
Transaction width: 64 B
Latency: L3 latency
Space 2: Core–Pair
Sub–spaces: 2
Memory: 2 MB
Processing Capacity: 1 Operation, Speed: 2.3 GHz
Concurrency: 1 transaction
Transaction width: 8 B
Latency: L2 latency
Space 1: Core
Memory: 16 KB
Processing Capacity: None

Figure 1: A PCubeS Description of the Hermes Cluster

There are 6 levels/PPSes in the PCubeS hierarchy of Hermes. Since PCubeS treats caches as memories, there are four levels from CPU to Core. Further notice that the breakdown assigns the non-zero processing capacity to the Core-Pair level. This is done because within a CPU a pair of cores shares a single floating point unit. As one goes up from Core to Cluster level, both the latency and width of transaction generally increases. Finally there is no memory beyond the CPU level.

Two important 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, every level of the deeply hierarchical machine is exposed for programmer's exploitation.

To give an example of how to reason about the execution platform when mapping LPSes of an IT program to PPSes of the hardware, consider the matrix-matrix multiplication example of Listing 1 to be executed in Hermes. The programmer can quickly determine that mapping the sole LPS (Space-A) of the program at the Core-Pair level will maximize concurrency. Going to one level below to Cores will not increase parallel processing power – as there is no processing capacity at the lowest level – but may bring advantage in load/store because of the fast 16 k local memory. That to be true, the size of the partitions should be chosen in a way that the data can fit within that memory.

The expectation in the IT + PCubeS paradigm is that once a programmer learns to read PCubeS descriptions of machines, he should be able to assess the possible impact of the nature of computation, memory access, and data transfer of his program as deterministic functions of hardware attributes. This knowledge will guide him to learn to write programs in a flexible way that can be tuned for specific target architecture.

This is a hardware conscious paradigm for parallel programming, but not hardware specific. The actual implementations of compute stages, synchronization, and communications are determined by the compiler once the LPS-to-PPS mapping is known, but the programmer is oblivious of that: he understands the costs of hardware features not their implementation.

Next we investigate the features of the IT language to understand the semantics of this new type-architecture based programming paradigm.

IT Parallel Programming Language

Following things can be said about the IT language overall to put it in the context of related work discussed before.

- IT is a language intended for high performance parallel programming
- IT is mostly a data parallel language with support for course-grained task parallelism.
- Parallelism in IT scales proportional to the number of parallel physical processing resources in the target execution platform.
- The language imposes no specific model for parallel execution (e.g., bulk synchronous) and synchronization and communication are implicit. The compiler for a target back-end chooses the execution and parallelism model deems appropriate.
- IT is portable across all PCubeS type hardware architectures.
- IT employs a declarative pseudo-code like syntax that highlights separation of programming concerns.

Let us new examine individual elements of the language. The current version of IT grammar is given at the end of this article.

Data Structures

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

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

A variable declaration takes the following form "Variable-name: Type" as shown in the following examples.

```
average: Real double-precision
matrix: 2d Array of Integer
coordinate: List of Point
```

In the above, matrix is a dynamic array as its dimensionality but not lengths of individual dimensions are known from the declaration. The dimension lengths are specified using an *${array-name}.dimension* attribute and must be set before data can be read or written to and from the array. Below is an example of a user defined type.

```
Class Room {
        size: Integer[3]
        material: String
        owner: Person
        rent: Real single-precision
}
```

Here the size attribute is a fixed length array of 3 integers. The attributes of a user defined object type are accessed using the *${object-name}***.***${attribute-name}* syntax.

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

```
a, b, c: 2d Array of Real single-precision
i, j: Integer

c[i][j] = a[i][j] * b[i][j]
```

An entire section of an array can be accessed and assigned to other arrays using a sub-range expression as shown below.

```
a, b: 2d Array of Real single-precision
c: 1d Array of Real single-precision
i, j, k: Integer

b = a[i…j]
c = b[…][k]
```

In the above, *b* gets all the elements of *a* from rows *i* to *j* and *c* gets the *k*th column of *b*.

List elements are, however, only accessed and manipulated sequentially from beginning to end using built-in list functions. The *get_cursor* function returns a cursor at the beginning of the list; the *value* function returns the element at the cursor location; the *advance* function moves the cursor one step forward and returns a Boolean value indicating if stepping was successful. The following code snippet illustrates the use of different list functions.

```
polynomial: List of Coefficient
point: Point
result: Real double-precision

cursor = get_cursor(polynomial)
result = 0.0
do {
        result = result + evaluate(point, value(cursor))
} while (advance(cursor))
```

Unlike arrays whose dimensions get fixed once being set, lists can grow and shrink on-demand. A new element can be added at the current cursor location using the *insert_at* function and the existing element at the current location can be removed using the *remove_at* function. A list however cannot grow beyond a predefined maximum capacity which is defined using the *set_capacity* function. These three functions have the following signatures.

```
insert_at(list, cursor, element)
remove_at(list, cursor)
set_capacity(list, capacity)
```

The result of concurrent updates on a list is undefined.

Finally, IT is a strongly and statically typed language that uses type inference to deduce types of local variables found within the coordinator program, function bodies, and tasks' compute-stages that are not explicitly declared.

The Coordinator Program & Task Execution

As discussed before, an IT program is a collection of 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-reference;
        initialize: comma separated initialization-parameters;
        partition: comma separated integer partition parameters)
```

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

There is no name for the program controller and its syntax is.

```
Program (argument-name) {
        <statement-sequence>+
}
```

The sole argument of the program controller holds all command line arguments passed at the beginning. The command line arguments are passed as key, value pairs in the form 'key=value.' A task's environment contains the set of data structures that establishes a task's relationship with other tasks and external input/outputs. The controller for block LU factorization program of Listing 2 illustrates the role a task's environment plays.

The LU factorization program consists of three tasks. First it executes the *Initiate LU* task (Line 7) to initiate upper and lower triangular matrices from elements of the argument matrix that is read from an external file (Line 4). Then the outputs of the first task are assigned to the environment of the second, *LU Factorization*, task (Line 14 and 15) that executes repeatedly inside a loop. After the completion of one round of the *LU Factorization* task, a *SAXPY* task (Line 37) updates portion of the upper triangular matrix using another portion of the same matrix and a portion of the lower triangular matrix. Since there are environmental dependencies between the *LU Factorization* and *SAXPY* tasks, the latter cannot start before the former finish. For the same reason, the next iteration of the loop cannot re-launch the former until the latter finishes for the previous iteration.

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

Listing 2: The Coordinator Program for Block LU Factorization

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

File I/O

Just like computation, the principal mechanism for file I/O in IT is parallel. The coordinator program only specifies the external input/output files – we call this action *binding* – and the actual read/write happens in parallel inside the individual tasks. For example, the *bind_input* command at Line 4 only attaches appropriate instruction to the *Initiate LU* task that causes the argument matrix to be read when that task starts execution. Similarly, the three *bind_output* commands of Line 10 to 12 results in the outputs of the *LU factorization* task to be written to files in parallel at the end of the program. The *bind_input* and *bind_output* commands are sensitive to the type of the data structure and read/write data accordingly. The file format must be appropriate for *bind_input*'s correct behavior though. The format for different kinds of data files are given at the end of this document.

If a program requires file read/write to be done in the coordinator then the alternative commands for that are *load* and *store*. The latter two commands result in sequential file read/write as opposed to parallel as in the former two. The syntax for all four commands is as follows.

```
bind_input(environment-reference, object-name, input-file-name)
bind_output(environment-reference, object-name, output-file-name)
load(object-reference, input-file-name)
store(object-reference, output-file-name)
```

Note that none of the file I/O commands can be used inside the compute-stages of a task. That is, their usage is restricted as statements in the program controller. This restriction is important as the PPS that may be assigned to execute a compute-stage having I/O instructions may not have the hardware features to carry out those instructions.

IT Tasks

A task in IT consists of six sections: a *Define* section where variables (scalars, structures, lists, and arrays) are defined, an *Environment* section where some locally defined variables are bound to the task environment, an *Initialize* section that defines how unbound variables or their properties should be initialized, a Stages section where parallel procedures (called compute-stages in IT) are defined, a *Computation* section that expresses the underlying algorithm a task implements as a flow of compute-stages in logical spaces (LPSes), and finally a *Partition* section that defines how parts of variables are distributed and replicated to LPUs of LPSes.

The notion of LPS and LPUs are central to IT programming and its most innovative feature. So we discuss these concepts in detail before we proceed to discussing the individual sections of a task.

Logical Processing Spaces and Units

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

In a traditional Von Neumann machine, there is a memory and instructions. Instructions are fetched from the memory and executed. When an instruction executes it may load and store variables in the memory. The entire memory is visible to each instruction - whether a variable is in the scope of the current programming context or not, or whether it is a local variable or a global.

Figure 2 illustrates this concept. The address space of the Von Neumann computer is shown as a slate. Within the memory three variables *a*, *b*, and *c* have been defined. Instructions operating in the Von Neumann machine can load and store variables in the address space. In essence all instructions can access all memory, this makes the Von Neumann model a single-space model.

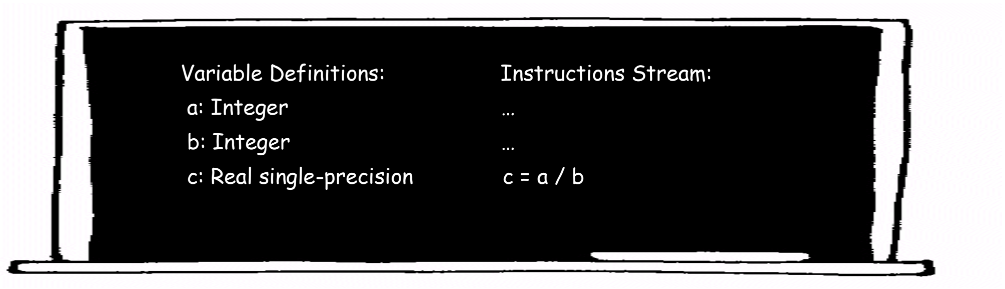


Figure 2: A Von Neumann Space

IT supports multiple spaces in a program where both variables and computations can be assigned (notice the distinction between definition and assignment). Figure 3 depicts a scenario with two spaces. Here the variables *average*, *median*, and *earning_list* are defined externally; the first two reside in Space-A and the second two in

Space-B. Therefore the *earning_list* is shared between the two spaces. The update done on the *earning_list* in Space-A is visible in Space-B, but the variable *average* is not accessible from the latter by any instruction.

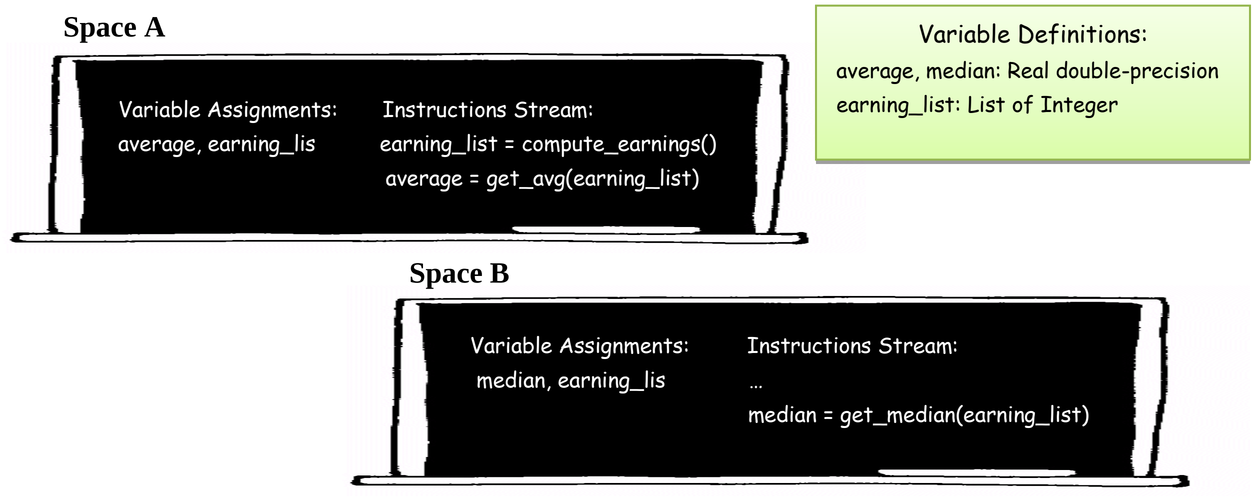

Space A
Variable Assignments:
average, earning_lis
Instructions Stream:
earning_list = compute_earnings()
average = get_avg(earning_list)
Variable Definitions:
average, median: Real double-precision
earning_list: List of Integer
Space B
Variable Assignments:
median, earning_lis
Instructions Stream:
...
median = get_median(earning_list)

Figure 3: Demonstration of a 2 Space Model

There are further additions to the notion of a space in IT. First, an IT space or LPS has a dimensionality attribute. That is an LPS is not a characterless vacuum, rather it is more like a geometric coordinate space. Second, an LPS can be partitioned into independent units called Logical Processing Units (LPU). The same instructions stream executes in all LPUs but on different parts of data structures. A group of LPUs may, however, share a particular part of a data structure. Then only one can update that part at a time and afterwards the remaining LPUs receive the update.

Execution

In the absence of data dependencies on shared/overlapped parts instruction streams in individual LPUs execute independently. Figure 4 depicts the breakup of a single LPS for the matrix-matrix multiplication task of Listing 1.

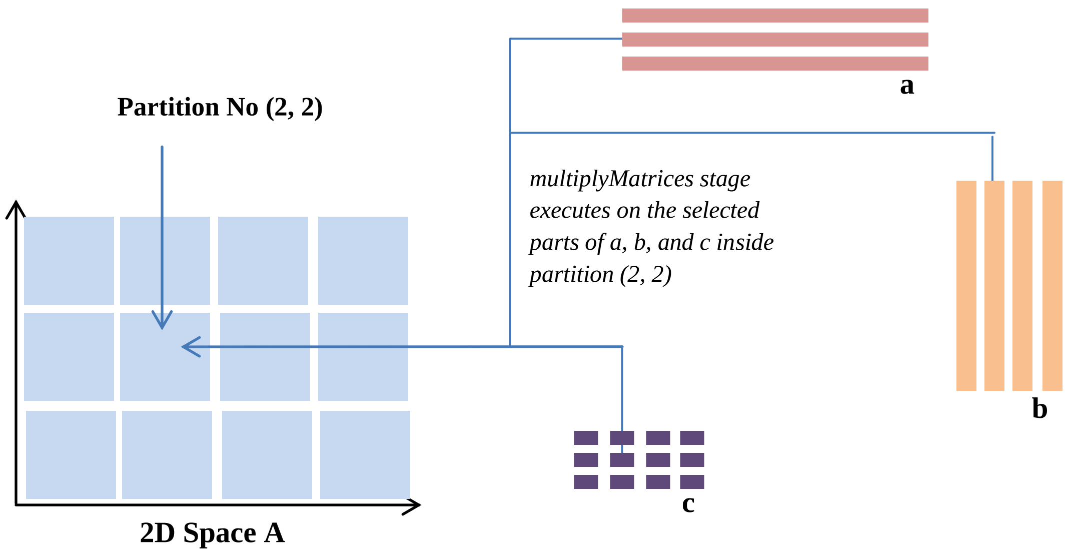

Partition No (2, 2)
a
multiplyMatrices stage executes on the selected parts of a, b, and c inside partition (2, 2)
b
c
2D Space A

Figure 4: An Illustration of Space Partitioning for a Small Matrix-Matrix Multiply Problem. A block of rows of *a*, a block of columns of *b*, and a block of *c* are contained in the LPU corresponding to partition (2, 2).

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

For example, this facility can be used to convert a traditional matrix-matrix multiplication program into a block matrix-matrix multiplication program as done in Listing 1. Programmatically this is achieved using the Sub-partition construct (Line 43 to 45 of Listing 1). Figure 5 below illustrates the effect of sub-partitioning for the discussed problem.

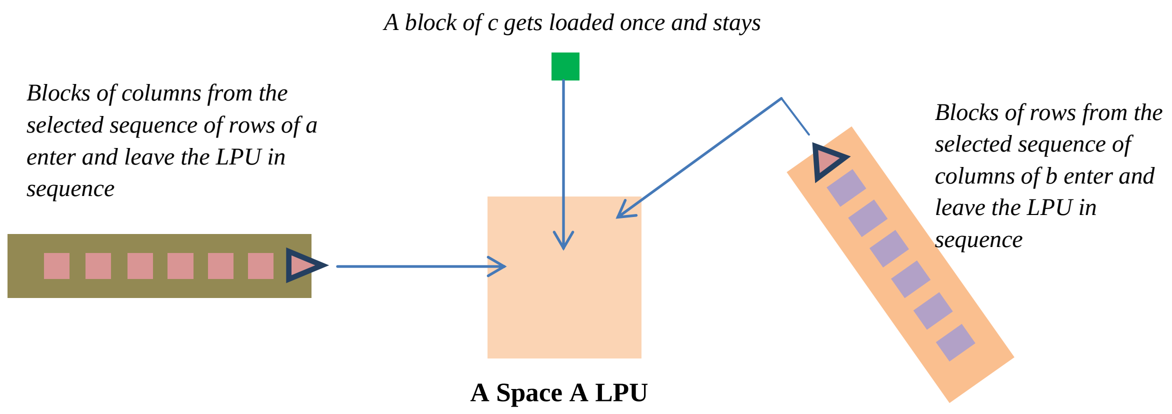

A block of c gets loaded once and stays
Blocks of columns from the selected sequence of rows of a enter and leave the LPU in sequence
Blocks of rows from the selected sequence of columns of b enter and leave the LPU in sequence
A Space A LPU

Figure 5: Incremental Data Loading in an LPU

Finally, the power of LPS goes further in IT with the support of LPS hierarchies. In IT each LPU can be treated as an LPS and can be further partitioned into lower level LPUs. In fact, there may be multiple lower level LPSes for an upper level LPU that may differ in their dimensionality and partition counts. Figure 6 illustrates the idea of the LPS hierarchy for a Monte-Carlo area estimation program.

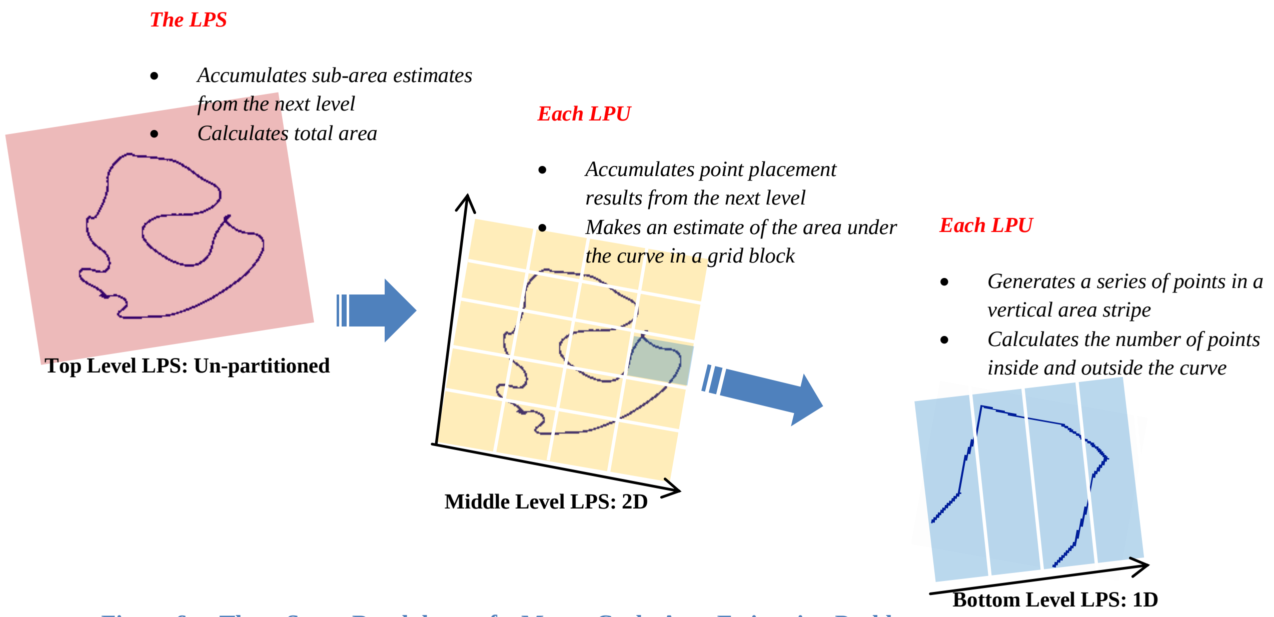

The LPS
Accumulates sub-area estimates from the next level
Calculates total area
Top Level LPS: Un-partitioned
Each LPU
Accumulates point placement results from the next level
Makes an estimate of the area under the curve in a grid block
Middle Level LPS: 2D
Each LPU
Generates a series of points in a vertical area stripe
Calculates the number of points inside and outside the curve
Bottom Level LPS: 1D

Figure 6: a Three Space Breakdown of a Monte-Carlo Area Estimation Problem

Regardless of how many LPSes share a single data structure and what the relationships among those LPSes are, IT ensures that the update in any data part is exclusive and all instruction streams operate over the most up-to-date data. It is the responsibility of the IT compiler to establish strong data consistency through whatever mechanism appropriate in the target execution platform. The programmer is only responsible for applying appropriate conditions in the code to ensure that only one LPU updates the shared data part at a time.

Let us now discuss the individual sections of an IT task in more detail.

Task Definition:

An IT task is defined in terms of six distinct sections as follows.

```
Task "Name of the Task":
        Define:
                // list of variable definitions
        Environment:
                // instructions regarding how environmental variables of the task are related to rest of the program
        Initialize <(optional initialization parameters)>:
                // variable initialization instructions
        Stages:
                // list of parallel procedures needed for the logic of the algorithm the task implements
        Computation:
                // a flow of computation stages in LPSes representing the computation
        Partition <(optional partition parameters)> :
                // specification of LPSes, their relationship, and distribution of data structures in them
```

This sequence has to be maintained for a task definition to be valid. The Initialize Section, however, can be skipped if the task does not require it. Note that IT only supports C-style single line comments. As in C, a comment in IT starts with '//' and includes the rest of the line it is placed in. In addition, note that IT uses newlines as statements separator. If the programmer needs to break a long statement into multiple lines then he or she should place a '\' at the end of each line except the last to indicate a statement continuation.

Define Section

Variables for the task are defined in this section using the following syntax

```
<comma separated variable names: common Type name>+
```

Variables defined in this section persist for the lifetime of the task. In IT terminology, these variables are called *task-global* variables. Local variables used inside compute-stages need not and should not be defined here. It is the compiler's responsibility to generate code that ensures that any read on a task-global variable reads its most up-to-date value.

Environment Section

Only the task-global variables listed in the Environment section outlive the execution of a task – all other variables are automatically garbage collected as the task finishes execution. There are three ways an environmental variable can be associated to a task. Below are examples of each of these.

```
a: link
b: create
c: link-or-create
```

In the above, the variable *a* must be already present in the program environment for the task to start executing. This can happen in two ways: *a* is an environmental variable of another task that has already completed or *a* has an input binding instruction and would be read from a file. On the contrary, *b* will be created and added to the environment as a bi-product of current task execution. If *b* already exists before the task starts executing then the older copy of it will be replaced by the newer one. The variable *c* on the other hand, is first searched in the environment for existence and gets created only if it is not there already.

It is important to understand that the three variables above will survive after the completion of the underlying task in the manner it left them. To give an example, assume a finite difference stencil code estimating heat propagation within a *plate* variable on a cluster of GPGPUs. Then after the task ends, each participating GPU will retain the part of the *plate* assigned to them for computation. The parts will be accumulated to a central location or re-distributed only if deemed appropriate for the subsequent logic of the program.

Initialize Section

The optional Initialize section is for setting up task global variables for subsequent parallel execution. The syntax for this section is as follows.

```
Initialize <(comma separated parameter names)>:
        statements+
```

If the Initialize Section has parameters then only parameter names are needed – not their types. Types of the parameters are inferred from their use in the statements inside. Further, if the name of a parameter matches a task-global variable then the former is directly assigned to the latter. Thus no explicit assignment statement is needed.

The Initialize section executes sequentially at the beginning of a task. The most common use of the Initialize section is to set up the dimension ranges of dynamic arrays and capacities of task-global lists, as done in Listing 1. Except for linked environmental arrays and lists, that information is unknown at task launch for all lists and dynamic arrays. Setting up dimensions is important for arrays in particular as without the dimension ranges, the task launcher cannot determine what array part should go where. If any array dimension or list capacity is undefined after the execution of the Initialize section then the program fails with a runtime exception.

Note that IT arrays, like Fortran arrays, can have a non-zero starting index for any dimension. Furthermore, a dimension range can be inverted. That is, it can decrease from a max to a min index as opposed to increase from a min to max index.

Stages Section

The list of parallel procedures – called *compute-stages* in IT terminology – needed to implement the logic of a task is defined in this 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 supplied as arguments for them during a stage invocation in the subsequent Computation section. The types of parameters and any local variable used inside a stage is inferred from the invocation context. This technique allows the stages to be type-polymorphic just like C++ template functions. Listing 3 shows four compute stages taken from an LU factorization task. Note that the *do {…} for* loops in different stages of Listing 3 are all parallel for loops. If a stage requires a sequential loop then the following alternative syntax can be used.

```
do in sequence {statement+} for $index in Range-Expression step Step-Expression
do in sequence {statement+} while Boolean-Expression
```

```
selectPivot(pivot, u, k) {
        do { pivot = reduce ("maxEntry", u[i][k]) } for i in u and i >= k
}
interchangeRows(u, l, k, pivot) {
        do {    u[k][j] at (current) = u[pivot][j] at (current − 1)
                u[pivot][j] at (current) = u[k][j] at (current − 1)
        } for j in u and j >= k
        do {    l[k][j] at (current) = l[pivot][j] at (current − 1)
                l[pivot][j] at (current) = l[k][j] at (current − 1)
        } for j in l and j < k
}
updateLower(u, l, l_column, k) {
        do { l[i][k] = u[i][k] / u[k][k] } for i in l and i > k
        l_column = l[...][k]
}
updateUpper(u, l_column, k) {
        do { u[i][j] = u[i][j] − l_column[i] ∗ u[k][j] } for i, j in u and i > k and j >= k
}
```

Listing 3: Core Computation Stages of an IT LU Factorization Task

A Note on the Declarative Syntax

IT uses declarative syntax for both the compute-stages and the flow definition in the Computation section (will be described shortly after). The syntax is declarative as it instructs what computation to be done but not how to do it. For example, the *do…for* loop of the *updateUpper* stage in Listing 3 iterates over two indices *i*, *j*. This should result in a doubly nested loop in conventional imperative programming languages. In IT, however, the ordering of two index traversals is left for the compiler to decide. For the flow definition, this philosophy extends further at a coarser level where requirements for synchronization and communication in-between compute-stages' transitions are determined based on the context in the flow.

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

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

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

In addition, a declarative syntax simplifies static analysis of the source code, which an IT compiler heavily depends on for proper code generation. Moreover, in some situations, a declarative syntax makes it easy to express a parallel algorithm that would be difficult to express otherwise. For example, Listing 4 shows a compute stage from an iterative stencil computation that uses Jacobi iterations to calculate the next state of a heated plate based on the current state. Without the declarative syntax for calculating the current plate cell values from previous values of neighboring cells, this parallel code would be difficult to express cleanly. The problem with an imperative syntax is that sometimes the difficulty in expression leads to the choice of a sequential algorithm as opposed to a parallel one.

```
refineEstimates(plate) {
        localRows = plate.dimension1.range
        localCols = plate.dimension2.range
        do { plate[i][j] at (current)                                   \
                = 1/4 * (plate[i-1][j]                                  \
                + plate[i+1][j]                                         \
                + plate[i][j-1]                                         \
                + plate[i][j+1]) at (current - 1)
        } for i, j in plate                                             \
          and (i > localRows.min and i < localRows.max)                 \
          and (j > localCols.min and j < localCols.max)
}
```

Listing 4: A Compute Stage for an Iterative Refinement Stencil Task (note the use of *at (current)* and *at (current-1)*)

Computation Section

The Compute Section describes the logic of the task as a flow of compute-stages in LPSes. Listing 5 depicts the computation section of a block LU factorization task. (Ideally block LU factorization in IT should rather be implemented using multiple tasks, as the coordinator program of Listing 2 shows, to minimize data movements.)

```
Space A {
        Space B {
                prepareLU(a, u, l)
        }
        Repeat for r in a.dimension1.range step block_size {
                calculateRowRange(a, row_range, block_size)
                Repeat for k in row_range {
                        Space B {
                                Where k in u.local.dimension1.range { selectPivot(pivot, u, k) }
                        }
                        storePivot(p, k, pivot)
                        Space B {
                                Where k != pivot {
                                        Epoch { interchangeRows(pivot, k, u, l) }
                                }
                                Where k in l.local.dimension1.range  { updateL(l, k, l_row) }
                                updateURowsBlock(u, l_row, k, row_range)
                                collectLColParts(l_column, l, k, row_range)
                        }
                        generatePivotColumn(p_column, l_column, row_range, k)
                        Space B {
                                updateUColsBlock(u, p_column, k, row_range)
                        }
                }
                Space B {
                        Where r in l.local.dimension1.range { copyUpdatedLBlock(l_block, row_range, l) }
                        Space C {
                                Repeat foreach sub-partition {
                                        saxpy(u, u_block, l_block, row_range)
                                }
                        }
                }
        }
}
```

Listing 5: Computation Section of a Block LU Factorization Task

For the computation of Listing 5, the flow of control operates entirely in the confinement of Space-A. Within it, the control first enters Space-B to prepare *l* and *u* from the argument matrix *a*. Then it comes out of Space-B and repeats the rest of the flow for *block_size* sequence of rows in *a*. The current *row_range* is determined first for each upper level iteration. Then a lower level repeat loop executes stages of classic LU factorization such as *selectPivot*, *interchangeRows*, *and updateLower*. Only a block of rows and columns of *u* get updated within the nested repeat loop. After the flow of control exits the first nested repeat loop, it enters a second loop to update the rest of u using a *SAXPY* operation. The final computation happens in Space-C. Note that an LPS can be placed inside another LPS in the flow definition (as done for Space-B within Space-A and Space-C within Space-B) when the former divides LPUs from the latter. This hierarchy of LPSes is defined in the Partition Section.

LU factorization of Listing 5 involves several updates of shared variables inside compute-stages. For example, the *pivot* that is selected in the *selectPivot* stage (Line 9) is needed during execution of *storePivot* (Line 11) and *interchangeRows* (Line 14). The programmer ensures that only one LPU of Space-B update the pivot in an iteration of the repeat loop of Line 7 to Line 24 using the *Where* condition for the *selectPivot's* invocation. The *selectPivot* stage executes only on the LPU that has the associated condition true. Notice the use of the keyword *local* in the *Where* condition. It indicates that the associate range to be compared is for the part of the array an LPU has – not for the entire array. The compiler injects appropriate synchronization/communication in between the transition from *selectPivot* to subsequent stages to ensure that the up-to-date value of the *pivot* is available wherever needed.

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

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

Repeat Boolean-expression { nested sub-flow }
Where Boolean-expression { nested sub-flow }
Epoch {nested stages accessing version dependent data structures }

The *Repeat* instruction results in the nested sub-flow to iterate for the time-being the associated condition remains valid. The *Where* condition restricts the execution of the nested sub-flow in the LPUs the associated condition evaluates to true. The *Epoch* boundary is used to dictate when the version number of any version-dependent data structure should be updated. For example, the stencil computation of Listing 4 needs the computation to be done on the updated version after each invocation of the *refineEstimate* stage. Hence, the corresponding flow definition should put the stage within an Epoch block.

Partition Section

The Partition Section specifies the configuration of the LPSes used in the computation flow, their relationships, and how data structure parts are distributed among LPUs of an LPS. The grammar for the Partition Section is as follows.

partition := ***Partition*** <([arguments,]+)>?: space-config+
space-config := ***Space*** id <dimensionality> <parent>? <attribute>* { var-list-config+ sub-partition-config? }
dimensionality := *un-partitioned* | [1-9]+d
parent := *divides* **Space** Id partition-type
partition-type := *partitions* | *sub-partitions*
attribute := *dynamic*
sub-partition-config := *Sub-partition* <dimensionality> <order> { var-list-config+ }
var-list-config := [Variable_Name,]+ | var-list-partition-config
var-list-parition-config := [name-with-dimension,]+ : [partition-instruction,]+
name-with-dimension := Variable_Name<[dim[0-9]+,]+> | Variable_Name
partition-instruction: partition-function([argument,]*) <padding-instruction>? | *replicated*
padding-instruction := *padding*([argument,]+)
Id := [A-Z]
argument := Variable_Name
order := *increasing* | *decreasing* | *unordered*

Listing 6 illustrates the Partition Section for the block LU factorization task of Listing 5. Here, the upper-most LPS, Space-A, is un-partitioned; Space-B is 1 dimensional and divides Space-A; Space-C further divides Space-B and furthermore contains a sub-partition configuration.

```
Partition(b):
        Space A <un-partitioned> {
                a, p, l_column, l_row, p_column, l_block, u_block
        }
        Space B <1d> divides Space A partitions {
                a<dim2>, u<dim1>, u_block<dim1>, l<dim1>, l_column: block_stride(b)
                l_row, p_column, l_block: replicated
        }
        Space C <2d> divides Space B partitions {
                u: block_size(b, b)
                u_block: block_size(b), replicated
                l_block: replicated, block_size(b)
                Sub-partition <1d> <unordered> {
                        u_block<dim2>, l_block<dim1>: block_size(b)
                }
        }
```

Listing 6: Partition Section of a Block LU Factorization Task

The array *p* keeping track of pivot row indexes from different iterations only exists in Space-A. So there will be a single undivided copy of it for the entire task and, hence, *storePivot* stage of Listing 5 (Line 11) that updates the array has been placed to execute in Space-A. The number of LPUs in Space-B, on the other hand, will depend on the number of strided blocks of *a, u, l* generated at task launch time. That is, the Space-B LPUs count depends on the input size and the partition parameter *b*. Some other data structures such as *l_row* will be shared among all those LPUs. Any of the replicated data structures can be updated by one Space-B LPU at a time only. Therefore, stages that update those structures are constrained with a *Where* condition in Listing 5. Finally, Space-C is 2d and it uses *block_size* partition function to further divide the structures of Space-B into 2d blocks. It also uses a sub-partition configuration to ensure incremental access to sub-sections of *u_block* and *l_block* during the execution of saxpy computation-stage (Listing 5, Line 28 to 30). Note that scalar variables and lists need not be specified in the Partition Section of a task – only array partitions should and must be specified. Scalars and lists are replicated in all LPSes they are found in invoked compute-stages.

The LPSes of a task have a partial ordering. Therefore, in the general case, the LPS hierarchy may look like a tree rather than a path as in Listing 6. There is an implicit default LPS, called the *Root*, that is the top-most LPS in a task's partition. If the same data structure is present in independent LPSes then a complete reordering/copying of data may ensue during a transition from one LPS to another LPS in the computation flow. The compiler and runtime system takes care of such data movements as in all other cases.

Furthermore, IT supports overlapping of boundary regions between data parts in the form of padding, as shown in Listing 7 in the partition configuration of a 2-Space iterative stencil computation. This form of sharing is frequently used for iterative refinements and multi-grid methods.

```
Partition (k, l, m, n, p1, p2):
        Space A <2d> {
                plate: block_size(k, l) padding(p1)
        }
        Space B <2d> divides Space A partitions {
                plate: block_size(m, n) padding(p2)
        }
```

Listing 7: Partition Section of a 2-Space Iterative Stencil Task

In the partition configuration of Listing 7, each *k*-by-*l* size plate partition of Space-A is padded in all sides by *p1* rows or columns: ('*plate: block_size(k, l) padding(p1)*' is only a syntactic sugar for '*plate<dim1, dim2>: block_size(k) padding(p1), block_size(l) padding(p1)*'). Each of those partitions is then divided in Space-B into smaller *m*-by-*n* blocks that are also padded, by p2 rows or columns. When a boundary overlapping is present in a task's partition, the compiler synchronizes the overlapped regions each time the computation flow exits the underlying LPS.

Currently, IT provides four standard partitioning functions as library support. These are as follows.

```
block_size(s): divides an array dimension into partitions of size s
block_count(c): divides an array dimension into c parts
stride(): distributes the indices of a dimension into processing units using strides of size 1
block_stride(s): distributes the indices of a dimension into processing units using blocked strides of size s
```

There is a future plan for incorporating user-defined partition functions too. Note that, the latter two partition functions in the above reorder the indices of a dimension but the former two do not. Regardless of the choice of the partition function, the definition of compute-stages and their subsequent usage in the computation flow of a task do not change. It is the responsibility of the compiler to generate appropriate back-end code based on the partition specification in the source code. This is another reason a declarative syntax was needed for the compute-stages.

Finally, note that the arguments of the Partition Section, if they exist, can be used in the Initialize and Computation sections using the *partition.${argument-name}* syntax.

Mapping Logical Spaces to Physical Spaces

Mapping of LPSes of an IT program to PPSes of the target backend is straightforward. Each LPS of each task of the program needs to be mapped separately to the PPSes the programmer deems appropriate. The mapping configuration is passed as an additional input file to the compiler along with the source program and has the following syntax.

```
mapping := task-mapping+
task-mapping := "Task-Name" { lps-mapping+ }
lps-mapping := Space Id : pps-no
Id := [A-Z] | Root
pps-no := [0-9]+
```

For example, a mapping file for the block LU factorization program of Listing 2 to run it on the Hermes cluster of Figure 1 may be as follows.

```
"Initiate_LU" {
        Space A: 3      // NUMA-node
}
"LU_Factorization" {
        Space A: 6      // Cluster
        Space B: 3      // NUMA-node
}
"SAXPY" {
        Space A: 3      // NUMA-node
        Space B: 2      // Core-pair
}
```

Listing 8: A Mapping File for the Block LU Factorization Program

Note that in Listing 8 above, the LPS-PPS mappings of individual tasks are independent of one another. For example, Space-A in the three tasks might have different data structures, or different partitioning for some common data structures, or even the exact same data partitions despite the LPS being mapped to the same PPS in all three cases. What happens to data organization as the program makes transition from one task to the other, as instructed in Listing 2, depends upon the actual runtime relationship between these tasks' partition hierarchies. The compiler is responsible for generating appropriate data movement instructions, if need may be, by comparing task partition hierarchies and tasks' LPS-PPS mappings.

The only restriction in the LPS-PPS mapping is that the partial ordering of LPSes in a task's partition hierarchy must be respected when mapping them to PPSes. To elaborate, if Space-A is an ancestor of Space-B in a task's partition then the latter cannot be mapped to a PPS situated higher than that the former has been mapped to. Both LPSes can be mapped to the same PPS though.

Degrees of Parallelism in an IT Program

The degree of parallelism in IT varies with tasks and within a task in execution of compute-stages assigned to different LPSes. When an LPS is mapped to a PPS of the target hardware, the LPUs correspond to that LPS get multiplexed to the PPUs of that PPS. Consider the mapping for LU Factorization task in Listing 8. Space-A has been mapped to Space-6 of the Hermes cluster. There is only one PPU in Space-6 in that machine, as apparent from the PCubeS description of Figure 1. Thus LU Factorization compute-stages that are assigned to Space-A will all execute in that single PPU. This raises the associated question how a code can execute in Space-6, which according to Figure 1 has no compute and memory capacity.

When an LPS is assigned to a PPS lacking compute/memory capacity, the PPUs of the latter execute the LPUs of the former using the compute/memory capacity of the closest ancestor/descendant PPS according to the following rule.

1. If an LPS is assigned to a PPS having no compute capacity then computation for all LPUs of that LPS is done by a single PPU in the nearest lower/upper level PPS capable of computation.
2. If a PPU has insufficient memory to hold the data for LPUs it operates on then it will use the memory of the nearest ancestor PPU that can hold that data and stage data in/out as needed.
3. Reduction operations assigned to any higher level PPU are collectively carried out by all PPUs in the sub-tree rooted at the former PPU in machine's PCubeS hierarchy.

In the LU Factorization example, space-B has been mapped to Space-3 of Hermes cluster. An inspection of Figure 1 reveals that there are total 32 Space-3 PPUs in the cluster (1 PPU for each NUMA node, 8 NUMA nodes per CPU, and 4 CPUs in the cluster). Therefore, Space-B compute-stages will execute independently in parallel in 32 PPUs.

Note that the multiplexing of LPUs to PPUs respects logical hierarchies of LPSes in the source task. Assume for example, Space-B divides Space-A in the SAXPY task of Listing 8. According to the mapping specification, Space-A LPUs will be multiplexed to the NUMA nodes. Then Space-B LPUs that divides a particular Space-A LPU assigned to a specific NUMA node will be multiplexed within the core-pairs of that NUMA node only.

The programmer can restrict the execution of a task to be confined within a smaller section of the target machine instead of consuming it entirely by specifying a mapping for the special *Root LPS*. Consider the single space Matrix-Matrix Multiplication task of Listing 1 and assume that the programmer wants to map Space-A to the cores of Hermes but intends to use only one Hermes node from the cluster. Then the mapping file for the program will be as follows.

```
"Block_Matrix-Matrix_Multiply" {
        Space Root: 5    // Socket
        Space A: 1       // Core
}
```

Listing 9: Mapping File for Block Matrix-Matrix Multiplication

Space-Root is by-default un-partitioned and all LPSes of a task descends from it. Therefore according to the mapping of Listing 9, only one Hermes node will be utilized for the task and the overall degree of parallelism for the sole Space-A compute-stage will be 64.

IT Compilers

At the time of this writing, two IT compilers have been developed for two backend architecture types. The first compiler is for multicore CPUs, and the second is for cluster of multicore CPUs or distributed memory supercomputers built from multicore CPU building blocks. Executables generated by the second compiler can run in hybrid supercomputers that have both accelerators and CPUs in their nodes but then the computing power of the accelerators will remain un-utilized. This paper is about the language – not the compilation process – regardless this section gives a broad overview of the working of these compilers and the nature of executable they generate to illustrate how the concepts of the proposed programming paradigm work in practice. (Details of these two compilers are the subject of two upcoming papers.)

Compiler for Multicore CPUs

The backend compiler for multicore CPUs generates C++ code parallelized with POSIX Threads (Pthreads) from IT source code. There is one thread for each execution core, unless the number of threads is further restricted by the specification in the mapping file. A thread's affinity to a core does not change throughout the execution of a task and each task in the program generates and cleans up its own set of threads.

Each thread works as a composite PPU controller and gets assigned PPU IDs for different PPSes depending its location. For example, the thread handling the first core of a NUMA node of a Hermes node (Figure 1) gets a valid Space-3 ID as well as a valid Space-1 ID. The remaining seven threads in that NUMA node get valid Space-1 ID but no Space-3 ID. They are in the former's Space-3 group and maintain a Group ID for that.

At the task launch, each thread is given the same *computation flow template* generated by the compiler and the thread executes or skips part of it depending on its PPU and Group IDs. The flow template embodies the programmer specified compute-stages coupled with compiler stages for data movements and synchronization instructions. The execution of a thread happens independently of the other threads unless there is some synchronization where multiple threads of a group need to participate.

This multicore compiler allocates contiguous memory for entire arrays. Different threads operate on different parts of the allocation depending on what LPUs they execute. Un-partitioned scalar variables are also maintained as properties of a single data structure shared among all threads. Therefore, LPU-LPU data dependency resolution, in most cases, can be done at the PPU-PPU level through Pthread synchronization primitives.

One limitation exists in the current compiler regarding memory management. PCubeS treats caches as memory. Further, the programming model of IT demands precise control over the movement of data in and out of caches. Programmatic control of the caches, however, has not been implemented yet in the compiler's LPU management libraries. Rather the compiler relies on the hardware's default caching machinery to do the compiler's job. This lack of control over the caches sometimes skews the performance of the executable.

An elaborate memory management scheme with programmatic cache control will be addressed in the next version of the compiler.

Compiler for Distributed + Shared Memory Hybrid Machines

This compiler generates MPI + Pthreads hybrid C++ code. There is one MPI process (called a *segment controller* in IT terminology) per multicore CPU that controls all the threads that operate on its different cores. That is, the MPI layer works as an overlay for the threads that do the actual computations.

Data dependencies between the threads of a segment controller are resolved using Pthreads synchronizations as done in code generated by the multicore compiler. Cross-CPU data dependencies, on the other hand, are resolved using MPI communications. Note that data movement among MPI processes is not hierarchical. Rather, the shared data directly goes from the sources of update to the destinations where the data is needed next. Furthermore, each process receives exactly the amount of data it needs, nothing more nothing less.

Unlike in the case of the previous compiler, memory allocation for arrays in this compiler happens as part-by-part basis. A segment controller allocates and maintains all data parts needed for LPU computations done by various threads it contains. Similar to memory, I/O handling is also done at the segment controller level with the MPI process loading/storing only the portion of data relevant to its threads.

Note that there is a one-to-one correspondence between different kinds of data dependencies possible within an IT task and different modes of MPI communications. For example, a data dependency from a list of lower space LPUs to a higher space LPU is equivalent to an MPI gather. The compiler exploits these kinds of similarities to generate executable that uses the pair-to-pair or collective MPI communication that is most efficient in a particular context.

Conclusion

The need for programming languages that map efficiently to contemporary parallel hardware is more important than ever. Modern machines are increasingly heterogeneous and have ever deeper memory & communication hierarchies. Papering over the gaps between our programming language models and the underlying machines is difficult and error prone already - continuing failure to align the programming models and the machines will lead to increasingly complex and non-portable applications.

IT, and the associated PCubeS type architecture, directly address the disconnect between machine architectures and language. We believe strongly in Synder's – "the medium is the message" - that the community badly needs languages in which they can efficiently express their algorithms on diverse, heterogeneous, and very deep architectures.

Firstly, IT Logical Processing Spaces map to PCubeS spaces, which in turn correspond to the layers in the physical machine. Data partitioning to ensure data locality for computation is under the control of the programmer who understands the algorithm. The complexity of actually partitioning the data and moving the data between stages is not the programmer's responsibility; rather it is the compiler and run-time systems responsibility. The cost of restructuring and moving data is made clear in the language constructs: within a stage data access is local, but stage boundaries may incur significant cost when data re-partitioning is required.

Second, IT clearly separates the specification of the computation, the "what I want to compute," from the specification of how the computation will be performed, how the data will be partitioned, and how the resulting partitions (LPUs) will be mapped to the physical resources of the machine. This leads to portable codes – it is not the computation after all that needs to change when moving between machines - it is the implementation of the computation that needs to change.

IT is not a design study or vapor-ware. Two IT compilers and their associated run-time systems have been developed: the first for multi-core architectures and the second for distributed memory machines in which the nodes may be multi-core nodes. Preliminary results on a 64-core AMD-based multi-core machine (from the Hermes cluster described in Figure 1) and a cluster having Cray Cluster Solutions nodes connected by FDR Infiniband are available at [web site]. In these preliminary results, we compare IT implementations of matrix multiply, LU factorization, conjugant gradient, and integer sort against hand coded sequential C, hand coded OpenMP/C, hand-coded MPI/C. The results are quite promising.

We are continuing our evaluation of IT in three dimensions. First, we are developing a larger set of application kernels and beginning to develop real applications. These will be used on a larger variety of machines to test portability, and on larger machines to test scalability. Second, we are continuing to optimize the generated code.

During the early compiler development we were using the “get it right, then make it fast” philosophy. Now it is time to make it fast. Third, we are working on extending the compilers to support other PCubeS physical layers. Specifically, we intend to target NVIDEA GPUs next. Once complete, IT applications will be able to address large scale heterogeneous machines available today.

Appendix A: IT Grammar –Version 1

Program Definition

program	:= component+
component	:= class | task | function | coordinator

Class Definition

class	:= *Class* Type_Name ':' element_def+
Type_Name	:= [A-Z]+([a-z][A-Z][0-9]_)+
element_def	:= names ':' static_type
names	:= Variable_Name | names ',' Variable_Name
Variable_Name	:= [a-z]+([a-z][A-Z][0-9]_)*
static_type	:= scalar_type | static_array
scalar_type	:= *Integer*
	| *Real single'-'precison*
	| *Real double'-'precision*
	| *Character*
	| *Boolean*
	| *String*
	| Type_Name
static_array	:= T_Array static_dim+ of scalar_type
static_dim	:= '[' [0-9]+ ']'

Task Definition

task	:= *Task* Task_Name ':' define environment initialize? stages computation partition
Task_Name	:= String

Define Section

define	:= *Define* ':' definition+
definition	:= names ':' type
type	:= static_type | dynamic_type
dynamic_type	:= list | dynamic_array
list	:= *List of* type
dynamic_array	:= dimensionality *Array of* type
dimensionality	:= [0-9]+*d*

Environment Section

environment	:= *Environment* ':' linkage+
linkage	:= names ':' mode
mode	:= *link* | *create* | *link_or_create*

Initialize Section

initialize	:= *Initialize* arguments ':' code
arguments	:= | '(' names ')'

Stages Section

stages	:= *Stages* ':' stage+
stage	:= Stage_Name '(' names ')' '{' code '}'
Stage_Name	:= Variable_Name

Computation Section

computation	:= *Computation* ':' flow_def+
flow_def	:= space_contr '{' flow_def+ '}' | exec_contr '{' flow_def+ '}' | stage_invoke
space_contr	:= *Space* Space_Id
Space_Id	:= [A-Z]
exec_contr	:= repeat | epoch | where
repeat	:= *Repeat for* range_expr | *Repeat foreach sub-partition*
range_expr	:= Variable_Name *in* range
range	:= Variable_Name | property | '[' expr "..." expr ']'
where	:= *Where* bool_expr
epoch	:= *Epoch*
stage_invoke	:= Stage_Name '(' args ')'
args	:= arg | args ',' arg
arg	:= expr

Partition Section

partition	:= *Partition* <([arguments,]+)>?: space-config+
space-config	:= *Space* Space_Id <dimensionality> <parent>? <attribute>* { var-list-config+ } sub-partition-config
dimensionality	:= *un-partitioned* | [1-9]+*d*
parent	:= *divides Space* Space_Id partition-type
partition-type	:= *partitions* | *sub-partitions*
attribute	:= *dynamic*
sub-partition-config	:= *Sub-partition* <dimensionality> <order> { var-list-config+ }
var-list-config	:= [Variable_Name,]+ | var-list-partition-config
var-list-parition-config	:= [name-with-dimension,]+ : [partition-instruction,]+
name-with-dimension	:= Variable_Name<[dim[0-9]+,]+> | Variable_Name
partition-instruction	:= partition-function([argument,]*) <padding-instruction>? | *replicated*

```
padding-instruction            := padding([argument,]+)
argument                       := Variable_Name
order                          := increasing | decreasing | unordered
```

Expressions & Statements

```
code             := stmt+
stmt             := parallel_loop | sequencial_loop | if_else_block | expr
parallel_loop    := do '{' stmt+ '}' for index_ranges
sequencial_loop  := do in sequence '{' stmt+ '}' for Variable_Name in expr step_expr
                         | do in sequence '{' stmt+ '}' while expr
index_ranges     := index_range | index_ranges ';' index_range
index_range      := names in Variable_Name restrictions
                         | names in Variable_Name '.' Dimension_No restrictions
Dimension_No     := "dimension"[0-9]+
restrictions     := | and expr
if_else_block    := if '(' expr ')' '{' stmt+ '}' else_block
else_block       := | else | else_if else_block
else             := else '{' stmt+ '}'
else_if          := else if '(' expr ')' '{' stmt+ '}'
step_expr        := | step expr | restrictions
expr             := arithmatic_expr | logical_expr  | epoch_expr
                         | reduction_expr | field_access | array_access
                         | function_call | constant | assignment_expr
epoch_expr       := expr at (current lag)
lag              := | '-' [0-9]+
reduction_expr   := reduce '(' Reduction_Fn ',' expr ')'
Reduction_Fn     := Variable_Name
function_call    := Variable_Name '(' args ')'
```

Coordinator Program Definition

```
coordinator      := Program '(' Variable_Name ')' '{' meta_code '}'
meta_code        := meta_stmt+
meta_stmt        := stmt | special_stmt
special_stmt     := create_obj | task_invoke
create_obj       := new dynamic_type | New static_type '(' obj_args ')'
obj_args         := | obj_arg | obj_args ',' obj_arg
obj_arg          := Variable_Name ':' expr
task_invoke      := execute '(' task ':' Task_Name ';' environment ':' env optional_secs)
env              := Variable_Name
optional_secs    := | ';' optional_sec | ';' optional_sec ';' optional_sec
optional_sec     := section ':' args
section          := initialize | partition
```

Function Definition

```
function         := Function Variable_Name ':' in_out function_body
```

in_out	:= output | input output
input	:= *Arguments* ':' definitions
output	:= *Results* ':' definitions
function_body	:= *Compute* ':' code

Appendix B: IT Data File Formats

Currently IT supports both binary and text data files for input/output of lists and arrays. Layout of content in both file formats is the same. The only difference is data items are separated by a space in a text file while they do not have any gaps in-between in a binary file. The metadata describing the content layout and other information lies at the end of the file after the content – except for the collection type, which is specified at the beginning. Therefore, the general structure of an input/output file is as follows.

```
Collection Type
Text or binary content
Metadata describing the content
```

A file's format is distinguished by its extension. A binary file ends with the '*.bin*' and a text file with the '*.txt*' extensions. The content is laid out in the row major order in both cases.

Since arrays can hold lists and vice versa, there are four possible values for collection types: *Array*, *List*, *Array of Lists*, and *List of Arrays*. Note that the type of collection elements is not needed as that is determined from the context of the program where file reading/writing is done.

For a regular array holding elements of static types, the terminal metadata just gives the dimension lengths in the format "dimension-0-length*dimension-1-length*dimension-2-length*…" and for a regular list the metadata gives its length. Thus a 10-by-10 matrix and a list of 100 points should have the following data file structures.

```
Array
Text or binary content
10*10
```

```
List
Text or binary content
100
```

For both *Array of Lists* and *List of Arrays*, the next to last metadata element holds a directory. Each entry of the directory provides the length information of the individual elements of the collection and the locations they begin from in the format '*beginning:length*.' The directory is laid out, again, in the row major order. So a list holding three matrices and a 2-by-2 matrix holding 4 lists should have the following data layout.

```
List of Arrays
matrix-0-content matrix-1-content matrix-2-content
0:10*10 100:3*3 110:100*5000
3
```

```
Array of Lists
list-00-content list-01-content list-10-content list-11-content
0:15 15:100 115:2 117:1000
2*2
```

Appendix C: Program Examples

LU Factorization

The traditional LU Factorization algorithm suffers from bad memory/cache reuse characteristics. It proceeds diagonally by adding one more column in the lower and one more row in the upper triangular matrices respectively. At each step of the algorithm, however, the entire remaining portion of the upper triangular matrix needs to be updated based on the most recently calculated row and column. Updating the remaining part of the upper triangular matrix is the most time consuming part of the algorithm and dominates its asymptotic running time. Unfortunately, for bigger matrices this step results in a lot of cache thrashing too.

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

In IT the modified algorithm can be easily represented using two tasks, one for the factorization part and another for the SAXPY part. A third task is used to initialize the upper and lower triangular matrices from the input argument matrix.

```
Program (args) {
        blockSize = args.block_size
        initEnv = new TaskEnvironment(name: "Initiate LU")
        bind_input(initEnv, "a", args.argument_matrix_file)

        // a straightforward initializer task prepares the initial state of upper and lower triangular matrix from the argument matrix
        execute(task: "Initiate LU"; environment: initEnv)

        luEnv = new TaskEnvironment(name: "Transposed LU Factorization")
        luEnv.u = initEnv.u
        luEnv.l = initEnv.l
        rows = initEnv.a.dimension1.range
        max1 = rows.max
        max2 = initEnv.a.dimension2.range.max

        do in sequence {
                lastRow = k + blockSize - 1
                if (lastRow > max1) { lastRow = max1 }
                range = new Range(min: k, max: lastRow)

                // a modified LU factorization tasks execute all stages of a traditional LU factorization with the exception that instead of
                // updating the entire remaining portion of the upper triangular matrix in each iteration of the algorithm it only updates a
                // block of rows going to be used in the remaining iterations
                execute(task: "Transposed LU Factorization"; environment: luEnv; initialize: range)

                if (lastRow < max1) {
                        saxpyEnv = new TaskEnvironment(name: "SAXPY")
                        saxpyEnv.a = luEnv.u[(lastRow + 1)...max1][k...lastRow]
                        saxpyEnv.b = luEnv.l[k...lastRow][(lastRow + 1)...max2]
                        saxpyEnv.c = luEnv.u[(lastRow + 1)...max1][(lastRow + 1)...max2]

                        // after the modified LU factorization finishes a SAXPY task updates the stale contents of the upper triangular
                        // matrix using the recently updated block of lower and upper triangular matrices
                        execute(task: "SAXPY"; environment: saxpyEnv; partition: args.k, args.l, args.q)
                }
        } for k in rows step blockSize

        bind_output(luEnv, "u", args.upper_matrix_file)
        bind_output(luEnv, "l", args.lower_matrix_file)
        bind_output(luEnv, "p", args.pivot_matrix_file)
}

Task "Initiate LU":
        Define:
                a, u, l: 2d Array of Real double-precision
        Environment:
                a: link
                u, l: create
        Initialize:
```

```
        u.dimension1 = l.dimension1 = a.dimension2
        u.dimension2 = l.dimension2 = a.dimension1
    Stages:
        prepare(a, u, l) {
            do { u[j][i] = a[i][j] } for i, j in a
            do { l[i][i] = 1 } for i in l
        }
    Computation:
        Space A { prepare(a, u, l) }
    Partition:
        Space A <1d> { a<dim2>, u<dim1>, l<dim1>: stride() }

Task "Transposed LU Factorization":
    Define:
        u, l: 2d Array of Real double-precision
        p: 1d Array of Integer
        pivot, k: Integer
        l_row, p_column: 1d Array of Real double-precision
        k_range: Range
    Environment:
        u, l: link
        p: link-or-create
    Initialize (k_range):
        p.dimension = p_column.dimension = u.dimension1
        l_row.dimension = l.dimension2
    Stages:
        selectPivot(pivot, u, k) { do { pivot = reduce ("maxEntry", u[k][j]) } for j in u and j >= k }
        storePivot(p, k, pivot) { p[k] = pivot }
        interchangeRows(u, l, pivot, k) {
            do {    pivot_entry = u[i][k]
                    u[i][k] = u[i][pivot]
                    u[i][pivot] = pivot_entry
            } for i in u and i >= k
            do {    pivot_entry = l[i][k]
                    l[i][k] = l[i][pivot]
                    l[i][pivot] = pivot_entry
            } for i in l and i < k
        }
        updateLower(k, u, l, l_row) {
            do {    l[k][j] = u[k][j] / u[k][k]
                    u[k][j] = 0
                    l_row[j] = l[k][j]
            } for j in l and j > k
        }
        updateUpperRows(u, l_row, k, k_range) {
            do { u[i][j] = u[i][j] - l_row[j] * u[i][k]  } for i, j in u and i > k and i <= k_range.max and j > k
        }
        retrieveLowerPart(p_column, l, k, k_range) {
            do { p_column[i] = l[i][k] } for i in l and i >= k_range.min and i < k
        }
        updateUpperColumns(u, p_column, k, k_range) {
            do {    u[i][k] = u[i][k] - u[i][j] * p_column[j]
            } for i, j in u and i > k_range.max and j >= k_range.min and j < k
        }
    Computation:
        Space A {
            Repeat for k in k_range {
                Space B {
                    Where k in u.local.dimension1.range { selectPivot(pivot, u, k) }
                }
                storePivot(p, k, pivot)
```

```
                    Space B {
                              interchangeRows(u, l, pivot, k)
                              Where k in l.local.dimension1.range { updateLower(k, u, l, l_row) }
                              updateUpperRows(u, l_row, k, k_range)
                    }
                    retrieveLowerPart(p_column, l, k, k_range)
                    Space B {
                              updateUpperColumns(u, p_column, k, k_range)
                    }
          }
     }
Partition:
     Space A <un-partitioned> { p, p_column, l }
     Space B <1d> divides Space A partitions {
          u<dim1>, l<dim1>: stride()
          l_row, p_column: replicated
     }

Task "SAXPY":
     Define:
          a, b, c: 2d Array of Real double-precision
     Environment:
          a, b, c: link
     Stages:
          multiplySubtract(c, a, b) {
               do {
                    do { total = reduce ("sum", a[i][k] * b[k][j]) } for k in a
                    c[i][j] = c[i][j] - total
               } for i, j in c
          }
     Computation:
          Space A {
               Repeat foreach sub-partition {
                    multiplySubtract(c, a, b)
               }
          }
     Partition (k, l, q):
          Space A <2d> {
               c: block_size(k, l)
               a: block_size(k), replicated
               b: replicated, block_size(l)
               Sub-partition <1d> <unordered> {
                    a<dim2>, b<dim1>: block_size(q)
               }
          }
```

Iterative Stencil

In an iterative stencil program, the fixed point state of an object of interest is computed by iteratively updating the values of its element cells from the values of their neighboring cells. This update is done a maximum number of times, as done in the algorithm implemented here, or until the state change of the object in successive iterations becomes negligible. There are several different methods for calculating a cell value from its neighbor. A common technique is the Jacobi iteration where a cell's value for the next iteration is the average of the current values of its neighbors. This technique is used to compute heat propagation on a 2D heated plate in the IT program given here.

A parallel implementation of the heated plate problem divides the plates into slabs/chunks and let different processors to update states of individual slabs. The nature of this computation requires that boundary regions of adjacent slabs overlap; otherwise the boundary cells of each slab cannot be updated properly. Consequently, the slab boundaries needed to be synchronized every so often, depending on the size of the overlapping.

The IT implementation presented here uses two levels of partition for the plate. First, the plate is divided into bigger 2D slabs with a larger boundary overlapping among them. Then individual slabs are further divided into smaller 2D slabs with a relatively smaller boundary overlapping among the latter. This allows to restrict the synchronization of boundaries within the confinement of each upper level slabs for a certain number of iterations. After that, the upper level slabs can be synchronized.

This kind of breakdown comes in handy when mapping the task into hierarchical architectures where there are great differences in the communication overhead at different hardware levels. For example, MPI communication overhead compared to copying data from one location to another of the memory in a distributed shared memory machine.

```
Program (args) {
    stencilEnv = new TaskEnvironment(name: "Five Points Stencil")
    bind_input(stencilEnv, "plate", args.input_file)
    bind_output(stencilEnv, "plate", args.output_file)

    iterations = args.iterations
    p = args.p
    k = args.k
    l = args.l
    m = args.m
    n = args.n
    execute(task: "Five Points Stencil"; environment: stencilEnv; initialize: iterations; partition: p, k, l, m, n)
}

Task "Five Points Stencil":
        Define:
                plate: 2d Array of Real single-precision
                max_iterations: Integer
                counter_1, counter_2, counter_3: Integer
        Environment:
                plate: link
        Initialize (max_iterations):
        Stages:
                refineEstimates(plate) {
                        localRows = plate.dimension1.range
                        localCols = plate.dimension2.range
                        do { plate[i][j] at (current)                                   \
                                = 1/4 * (plate[i-1][j]                                  \
                                + plate[i+1][j]                                         \
                                + plate[i][j-1]                                         \
                                + plate[i][j+1]) at (current - 1)
                        } for i, j in plate                                             \
                         and (i > localRows.min and i < localRows.max)                  \
                         and (j > localCols.min and j < localCols.max)
                }
        Computation:
                // the whole computation should iterate for max_iterations number of times
                Repeat for counter_1 in [1...max_iterations] {
                        // after partition.p1 / partition.p2 upper level iterations the flow should exit for upper level padding
                        // synchronization
                        Space A {
                                Repeat for counter_2 in [1...partition.p1] step partition.p2 {
                                        // after partition.p2 iterations the flow should exit Space B for lower level
                                        // padding synchronization
                                        Space B {
                                                Repeat for counter_3 in [1...partition.p2] {
                                                        // epoch needs to be advanced after each refinement step
                                                        Epoch {
                                                                refineEstimates(plate)
                                        }
                                    }
                                }
                            }
                        }
                    }
                Partition (k, l, m, n, p1, p2):
                    Space A <2d> {
                        plate: block_size(k, l) padding(p1)
                    }
                    Space B <2d> divides Space A partitions {
                        plate: block_size(m, n) padding(p2)
                    }
```

Radix Sort

The parallel radix sort algorithm presented here is a simplified version of efficient GPGPU radix sort described in [20]. The IT implementation gets rid of GPGPU specific instructions and portrays the core idea behind the algorithm. In a parallel architecture, a radix sort is preferable to a quick sort due to the former's good scaling characteristics and the algorithm of [20] displays massive performance gain in GPGPU architectures.

The algorithm works for different radix values and implements the radix sort algorithm as multiple rounds of distribution sort. Each round of the distribution sort has three steps. First, each partition calculates the number of keys it has with different values (0 to radix - 1) at the current digit place of interest. Then those counts get summed up in a single location that gives the starting indexes for keys in individual partitions. Using that information, each partition can calculate the exact relocation offset for the keys it has. Finally, the keys are relocated to those offsets.

The algorithm maps quite well in IT model as a dual space computation as given below.

```
Program (args) {
    rdSortEnv = new TaskEnvironment(name: "Radix Sort")
    bind_input(rdSortEnv, "keys", args.input_file)
    bind_output(rdSortEnv, "keys", args.output_file)

    digits = args.digits
    key_size = args.key_size
    p = args.p
    execute(task: "Radix Sort"; environment: rdSortEnv; initialize: digits, key_size; partition: p)
}

Task "Radix Sort":
    Define:
        keys, scatter_offsets, l: 1d Array of Integer
        f: 1d Array of Boolean
        g: 2d Array of Integer
        radix, digits, round, r: Integer
        sorting_rounds: Range
    Environment:
        keys: link
    Initialize (digits, key_size):
        radix = 2 ** digits
        scatter_offsets.dimension = f.dimension = l.dimension = keys.dimension
        g.dimension1.range.min = 0
        g.dimension1.range.max = partition.p - 1
        g.dimension2.range.min = 0
        g.dimension2.range.max = radix - 1
        sorting_rounds.min = 0
        sorting_rounds.max = key_size / digits - 1
    Stages:
        digitDecoding(radix, r, keys, flags) {
                  max = radix - 1
                  ones = max << (round * digits)
                  r_value = r << (round * digits)
                  do {
                            if ((keys[i] & ones) == r_value) { flags[i] = 1 } else { flags[i] = 0 }
                  } for i in keys
        }
        reduction(result, flags, operator) {
                  do { total = reduce(operator, flags[i]) } for i in flags
                  result = total
        }
        multiLevelScan(g) {
                  current_sum = 0
                  do in sequence {
                            do in sequence {
                                      current_value = g[i][rd]
                                      g[i][rd] = current_sum
                                      current_sum = current_sum + current_value
                            } for i in g.dimension1.range
                  } for rd in g.dimension2.range
        }
        determineScatterOffsets(g, f, scatter_offsets, r) {
                  current_sum = g[g.local.dimension1.range.min][r]
                  do in sequence {
                            if (f[i]) {
                                      scatter_offsets[i] = current_sum
                                      current_sum = current_sum + 1
                            }
                  } for i in f.dimension1.range
        }
        rearrangeKeys(keys, scatter_offsets) {
                  do { keys[scatter_offsets[i]] at (current) = keys[i] at (current - 1) } for i in keys
        }
    Computation:
        Space A {
                  // repeat from the beginning to have as many rounds of distribution sort as required by the size of the keys
                  Repeat for round in sorting_rounds {
                            // calculate the number of keys that have different values for the current digit place in each partition
                            Space B {
                                      Repeat for r in g.dimension2.range {
                                                // determine which keys in the array have r in the current digit place
                                                digitDecoding(radix, r, keys, l)
                                                // determine the total number of local keys that have 1 in the current digit
                                                // place for value r
                                                reduction(g[g.local.dimension.range.min][r], l, "sum")
                                      }
                            }
                            // determine the starting offsets for the entries belonging to different partitions for various values for
                            // the current digit place
                            multiLevelScan(g)
                            Space B {
                                      // do the final offset calculation for keys in the current round for all values in the current
                                      // digit place
                                      Repeat for r in g.dimension2.range {
                                                // again, determine which keys in the array have r in the current digit place
                                                digitDecoding(radix, r, keys, f)
                                                // determine the distribution offsets of keys respective to the first key with r
                                                // at the digit place
                                                determineScatterOffsets(g, f, scatter_offsets, r)
                                      }
                            }
                            // finally shuffle the keys based on the calculated scatter offsets to finish one round of the
                            // distribution sort
                            Epoch {
                                      rearrangeKeys(keys, scatter_offsets)
                            }
                  }
        }
    Partition (p):
        Space A <un-partitioned> {
                  keys, g, scatter_offsets
        }
        Space B <1D> divides Space A partitions {
                  keys, scatter_offsets, f, l: block_count(p)
                  g<dim1>: block_size(1)
        }
```

Bibliography

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[3] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. *SIGPLAN Not.*, 46(8):13–24, February 2011.

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 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, 2012. IEEE Computer Society Press.

[5] Filip Blagojevic, Paul Hargrove, Costin Iancu, and Katherine Yelick. Hybrid pgas runtime support for multicore nodes. Ín *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[7] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[9] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[10] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[11] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Zpl: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211, March 2000.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[13] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.

[14] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, May 2015.

[15] Tarek El-Ghazawi and Lauren Smith. Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC ’06, New York, NY, USA, 2006. ACM.

[16] Karl Furlinger, Colin Glass, Jose Gracia, Andreas Knupfer, Jie Tao, Denis HÃ¼nich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. Dash: Data structures and algorithms with support for hierarchical locality. In LuÃs Lopes, Julius Å½ilinskas, Alexandru Costan, RobertoG. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, JosÃ© Gracia, Sascha Hunold, StephenL. Scott, Stefan Lankes, Christian Lengauer, JesÃºs Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 542–552. Springer International Publishing, 2014.

[17] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992.

[18] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[19] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330 – 354, 2000.

[20] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’10, pages 545–546, New York, NY, USA, 2010. ACM.

[21] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[22] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[23] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[24] Kunle Olukotun. Beyond parallel programming with domain specific languages. *SIGPLAN Not.*, 49(8):179–180, February 2014.

[25] Lawrence Snyder. Annual review of computer science vol. 1, 1986. chapter Type Architectures, Shared Memory, and the Corollary of Modest Potential, pages 289–317. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[26] Jr. Steele, GuyL., Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (sun hpcs language). In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 718–735. Springer US, 2011.

[27] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[28] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. Magma library version 0.1, 2009.

[29] David W. Walker, David W. Walker, Jack J. Dongarra, and Jack J. Dongarra. Mpi: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[30] Muhammad N. Yanhaona and Andrew S. Grimshaw. The partitioned parallel processing spaces (pcubes) type architecture. Technical report, Technical Report, UVA Computer Science, 2014.

[31] Muhammad N. Yanhaona and Andrew S. Grimshaw. A roadmap for a type architecture based parallel programming language. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, COSMIC '15, pages 7:1–7:10, New York, NY, USA, 2015. ACM.

[32] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.