IT – A Simple Parallel Language for Hierarchical Parallel Architectures

Muhammad Nur Yanhaona, Andrew Grimshaw University of Virginia
Email: {mny9md, grimshaw}@virginia.edu

***Abstract*—After several decades of continuous research and development of hundreds of parallel programming languages; the dominant mechanism of parallelism, unfortunately, remains to be low level threading or message passing libraries attached to sequential language cores. We are investigating an alternative parallel programming paradigm that strives to strike a balance between low-level, platform-specific programming such as in MPI, Pthreads, or CUDA; and high-level, hardware-agnostic language based approach such as in X10 or Chapel. The result is the IT programming language. IT is a language for high performance scientific computing where expression of parallelism in a program is inseparable from reasoning about the capabilities of its execution platform; but the reasoning is done over an abstract machine model that enables portable high performance without losing programmer's productivity. This report describes IT's programming model, syntax, core features, and results of some early performance experiments with IT sample programs on NVIDIA GPGPU platform.**

***Keywords*—Parallel programming Language, type architecture, programming model.**

I. Introduction

'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 [1].' CPU clock speeds reached their peak in the early 2000s. It is widely acknowledged that future non-algorithmic performance improvements will primarily come through concurrency and parallelism. Unfortunately, parallel programming remains hard despite several decades and millions of dollars spent on parallel programming research. Since the 1960's, dozens of architectures and literally hundreds of parallel programming languages have been developed (almost 100 parallel variants were developed for C++ alone). In spite of those hundreds of languages, most parallel programs are still written using low level message passing or threading libraries attached to an existing sequential programming language such as C or FORTRAN. The most common of these is some combination of C or FORTRAN with MPI [2].

It has been argued that the failure of proposed languages is partially due to the community's reluctance to switch to an alternative programming approach. Achieving performance that is on par with that of MPI, where a programmer has absolute control over all aspects of computation and communication, has always been a major challenge for these languages too. That said, we hold that these languages have not and are not seeing success largely because they fail to propose a simple framework for writing parallel programs. Particularly, the programming languages of recent time such as Chapel [3] or X10 [4] are extremely bulky and it is difficult to see any underlying model of computing in any of them that could guide the programmer in writing an efficient, clean, parallel program for his problem.

We are not against message passing (MPI) and other low level parallelizing primitives (such as threads and annotations) for some ideological reason. Rather, we argue that they present an unproductive abstraction with which to write parallel programs. Writing an efficient MPI program for even the simplest problem can be difficult; so much so that the underlying logic may be lost in the plethora of communication and synchronization specific optimizations. This difficulty often dissuades people from learning MPI programming and turns them into mere users of library codes.

We believe that most high level parallel programming languages overshoot the target of providing the right abstraction by hiding too much detail from the programmer. It is often easy to write a parallel program in these languages. Getting good performance on the other hand, or understanding why the performance is good or bad is often very difficult. Further, if the performance is not adequate then programmer is utterly helpless as he does not know how his program actually functions on the target platform. We believe the objective of providing easy parallelism that resonates in all these languages is a lost cause as parallel programming is never an easy task to begin with.

The process of writing a parallel program involves designing an algorithm exhibiting adequate parallelism, determining how independent pieces of the program should communicate with each other, the granularity of those pieces, and finally their mapping to physical processing units (the four-step process, known as the Foster methodology of parallel programming [5]). Good decisions in all these steps depend on the characteristics of the physical platform on which the program will eventually run. This indicates that any approach to parallel programming that ignores the capabilities of underlying execution platforms is unlikely to be successful across the board.

In early 2013, out of our dissatisfaction over present situation of parallel computing, we start investigating on an alternative paradigm that can bring the benefits of both high and low level parallel programming together. Specifically, we look for the answers to the following questions.

1) How to ideally express parallelisms in an algorithm combined with reasoning about hardware's capabilities in a parallel program?

2) What machine abstraction can describes the key features of present day heterogeneous architectures in a uniform way?
3) Where should we draw the line between a programmer's and the compiler's responsibilities in writing an efficient parallel program?
4) What language syntax should be used for portable, high performance?

The result of our investigation is the *PCubeS* 'type architecture' and the *IT* programming language. PCubeS describes a parallel architecture as a hierarchy of processing spaces. It is capable of modeling most contemporary hardware such as accelerators, multi-core processors, distributed memory MPI environments, and hybrid supercomputers. IT exposes an abstract machine model of a hierarchy of logical processing spaces to the programmer that is similar in nature to that of PCubeS's. Execution of a parallel program is viewed as a flow (or flows) of computations along these spaces with components of a space executing parts of computations in parallel and individual parts may be parallel executables on their own. Listing 1 presents a simple IT source code for matrix multiplication as an example.

```
Task 'Matrix Multiply':
    Define:
        a, b, c: array dimension 2: Real Precision Single
    Environment:
        a, b: link
        c: create
    Initialize:
        c.dimension1 = a.dimension1
        c.dimension2 = b.dimension2
    Compute:
        (Space A) {
            do { c[i][j] += a[i][k] * b[k][j]
            } for i, j in c; k in a
        }
    Partition (k, l):
        Space A <2D> {
            c: block—size (k, l)
            a: block—size (k), replicated
            b: replicated, block—size (l)
        }
```

Listing 1. A single-space IT code for matrix multiplication

The matrix multiply task description in Listing 1 illustrates how tasks are broken down into orthogonal specifications: variable declarations, initialization, the actual computation, and a specification of how the data will be partitioned.

IT is designed for high performance scientific computing. Data structure support in IT is minimal – the principal concern is efficient operations on multi-dimensional arrays. Apart from intermixing algorithm design with execution environment modeling, separation of concerns and a declarative syntax are IT's two defining characteristics that distinguish it from contemporary parallel programming languages.

Note that IT does not offer any magic solution for good performance; rather, it provides the necessary tools to write parallel programs with the programmer's control over the aspects that makes a program run well or poorly in different environments.

Our project is relatively new and we are still in the process of enhancing the syntax and finalizing the core features. Nonetheless, the underlying philosophy, programming paradigm, modeling abstraction, and basic language syntax are all been already settled. Currently, we are working on pieces of IT compiler. We divide the compilation process into a platform independent and platform dependent parts. The design of the former is done and we are in the process of developing the latter for NVIDIA GPGPU platform. In the meantime, we are doing performance experiments with sample IT codes by emulating the expected outcome of the second compiler through hand-compiled C+CUDA programs. The results are very promising.

This report serves as an introduction to our new language. Here we explain how and why programming in IT is radically different from other forms of parallel programming; and why we believe IT can be the solution that brings together the portability and ease of expression of high level languages, and the performance of low level architecture-centric computing.

The rest of the report is organized as follows. Section II discusses the philosophy behind IT programming language, Section III briefly introduces PCubeS that is the foundation for abstracting the execution environment in IT, Section IV describes the programming paradigm, Section V presents the core language features, Section VI briefly discusses our ongoing compiler development effort, Section VII discusses the results of two performance experiments on NVIDIA GPGPU platform with sample IT codes, Section VIII summarizes the paper, and Section IX ends it with some related work.

II. Design Rationale

We believe, before we engage in a parallel programming language development project, we should have a clear understanding of the demands of our target population – our understanding does not have to match others' but we should have one to be able to make sense of current situation and establish an appropriate programming model.

That means we should have a specific target population to begin with as a language cannot be everything for everyone. Our target from the beginning is the high performance scientific computing community. We need a language that specifically and exclusively addresses the needs of this community. This suggests that we need a new language and, in addition, that we should not build it on top of some other languages such as Java. Without going into an argument about the merit of OOP in scientific computing, the mere fact that one can write many other applications in Java, rules out that possibility. We believe in the classical opinion that unnecessary features in a language should be strictly avoided [6] [7].

In our opinion, for scientific computing, the most important aspects of programming – in their order of appearance – are performance, programmer productivity, readability, and portability. Among these, performance is so important that despite being severely lacking in other three aspects MPI, Pthreads, CUDA, and similar other low level parallel programming paradigms are so successful. These programming techniques are so attuned to their respective execution platforms that it is extremely difficult, if not futile, for any high level language to top their performance.

Therefore in IT, we set the goal for performance that approximates that of low-level programming techniques. We try to achieve that goal by retaining most of the flexibility of these techniques regarding the choice of size and frequency of communication, granularity of computation units, and distribution of data structures without littering a program with corresponding infrastructure management codes.

The foundation for a clean reasoning interface lies on an idealized machine model that we develop for representing current parallel architectures. The model's responsibility is to faithfully expose the salient programming facilities of any execution platform with their associated costs. Such a model is called a *Type Architecture* [8]. We call our type architecture, the Partitioned Parallel Processing Spaces (PCubeS). PCubeS abstracts away the differences of different architectures by describing them in terms of parallel processing widths, memory capacities, and communication supports.

To allow a programmer to reason about an execution platform's facilities in a productive way, we adopt the principal of *separation of concerns*. In IT, data partitioning and mapping tasks into processing units are clearly demarcated from the core parallel algorithm of a program. Furthermore, the programmer only decides about the nature and frequency of communication and the compiler generates code for that based on the facilities of the target platform.

Our standard for readability is not the 'lines of code'; rather it is the ease of comprehension of a program's runtime behavior and, consequently, the ease of expressing the expected behavior when writing the program. We believe, separation of concerns provides answers to a large part of readability. Moreover we choose a declarative syntax to manifest the underlying parallel algorithm.

The choice of a declarative syntax gives rise to the question, 'where do we draw the boundary between the compiler and programmer's responsibilities regarding writing an efficient program?' Our answer to that is the compiler makes decisions regarding all and only those aspects of programming for which the right or best choice can be identified deterministically. More or less, everything that requires a probabilistic reasoning or too costly to deterministically compute is left for the programmer to specify. We are confident that generating efficient parallel codes for different execution platforms from a declarative source syntax would not be difficult given IT's lean language core and restricted data structure support.

Finally, for portability, we believe aforementioned choices will come as aids. Nevertheless, we accept that portable good performance often may not be achievable due to stark differences in the facilities of parallel architectures. However, to minimize program changes due to hardware differences, we make data partitioning and mapping configurable. Given the program is written appropriately, just by changing how data partitioning is done and parts of a computation are mapped into processing units; the programmer can achieve good performance across execution platforms.

III. PCubeS Type Architecture

To understand IT's programming model, one needs to comprehend PCubeS's hardware abstraction mechanism first. Therefore, this section provides a brief introduction to PCubeS type architecture. A detail discussion of PCubeS with example models of different parallel architectures is a target for a future paper.

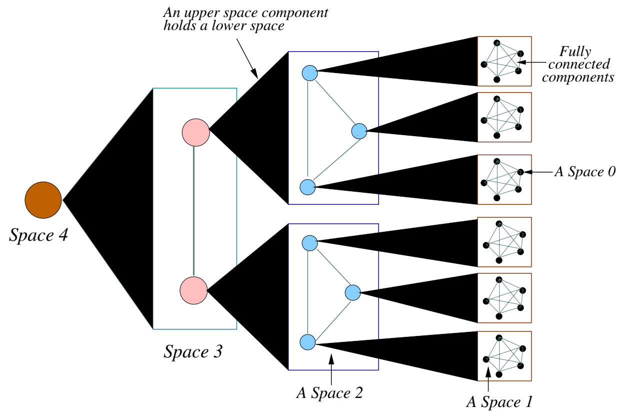
An upper space component holds a lower space
Fully connected components
A Space 0
Space 4
Space 3
A Space 2
A Space 1

Fig. 1. A PCubeS instance with 5 spaces

PCubeS views a parallel architecture as a finite hierarchy of Parallel Processing Spaces (PPS). Each space is characterized by the following four attributes

1) A fixed aggregate parallel processing capacity in terms of concurrent execution units whether it measures in cores, vector length, or SIMD width.
2) A finite memory with a fixed latency, bandwidth and transaction width.
3) A fixed number of uniform, independent partitions or components (PPU).
4) A fixed communication speed and bandwidth across components.

These four characteristics form a four-tuple <processing capacity, <memory size, bandwidth, latency, width>, partitions, <communication bandwidth, latency>>. Here we describe all latencies relative to the fastest unit to simplify programmer's reasoning. Figure 1 gives of an example of a type architecture instance having five spaces.

Note that the memory of a space is not necessarily a physical shared memory. To understand why, assume an execution environment having a CPU controlling two accelerators with the support that each accelerator has access to the CPU main memory alongside its own. Then the CPU memory is the memory of the space that comprises those two accelerators.

Furthermore, if there is a group of workstations each having such a CPU controlling two accelerators and are connected through Ethernet. Then the space comprises by the CPUs does not own the aggregate CPU memory of all workstations either. That is, the memory in a space, say in Space I, is the memory accessible to all its components. The memory local to a component is not part of that Space. Rather a Space I component's memory is a Space $I-1$ memory as the component is the owner of a lower space. From a programming perspective, a PPU has access to two memories (given that they exist): its own private memory and the memory of the space it is in.

Thus, as we go up in the space hierarchy the amount of memory available does not necessarily increase; neither does the lowest space account for the entire memory available in the hardware. Parallel processing capacity, however, monotonically increases as we go up as an upper space component can utilize its immediate lower space to maximize parallelism.

For an example PCubeS instance, consider a typical massively parallel system of the last decade. The computer itself might consist of 32 racks connected by a 64 port Infiniband, each rack consisting of 64 nodes connected by Infiniband with dual uplinks to the switch connecting the racks. Each node has 32 GB main memory and two, four core CPUs with 8 MB shared L3 cache per CPU and 512 KB L2 cache per core, operating at 2.3 GHz. The cache line size and data-path to memory is 4 64 bit words (265 bits wide). Main memory is DDR at 1.66 GHz. Each core is capable of a peak execution rate of 4 FLOPS/clock.

We could model this as a 5-space PCubeS instance. In that model; a Space 0 is a single CPU core with 512 KB memory and 4 words as the memory transaction bandwidth; a Space 1 is a CPU with 4 parallel Space 0 components, a relatively higher latency 8 MB shared memory, and its components' communication characteristics defined in terms of the bandwidth and latency of L3 cache read/write; a Space 2 is a node with two independent Space 1 components giving an aggregate processing capacity of $2 * 4$ and an on-board 32 GB memory of bandwidth $1.6 * 2 * 4$ GW/s; a Space 3 is a collection of 64 Space 2 units within a rack, there is no Space 3 memory and components' communication latency and bandwidth is defined by the characteristics of the Infiniband interconnect; and, finally, the supercomputer as a whole forms the single Space 4 encompassing 32 Space 3 components.

IV. The Programming Model

The abstract machine model of IT closely corresponds to the PCubeS architecture. An IT program is composed of a set of parallelizable tasks. Each task can be viewed as computation happening in multiple spaces and, within each space, in multiple partitions. The distinction between spaces (PPS) and partitions (PPU) of PCubeS, and that of IT is that the latter's are unrestricted. A task can have more or less partitions and spaces than that may be available in its target execution platform. In other words, a task operates on logical spaces and partitions, and we call them Parallel Computation Spaces (PCS) and Parallel Computation Units (PCU) respectively. The programmer has to explicitly map PCSs to PPSs and define granularity of the PCUs individually for each task. The efficiency of his mapping and partitioning schemes is critical for good performance of a task in a target platform.

It is important to grasp the notion of a task and the interplay of different tasks in a program. The program is by itself not a parallel computation – rather each task is. The program acts as a dispatcher and coordinator of tasks. IT has a notion of program environment, which is basically the collection of data structures the programmer is interested in as part of the execution of the program. A data structure in the environment may reside as a whole in a single PPU or may be spread in different PPSs and PPUs. Each task execution accesses and modifies a part of that environment, called the task environment; and the execution order of tasks is determined by their environmental dependencies. The notion of a dependent task further modifying the environment instead of getting input from its predecessor and generating new output is critical for optimizing data movements.

A. *Behavior of a Task*

A task is the basic unit of parallelism in IT. A task is independent and isolated from all other tasks that may be executing at the same time within the program. The execution of a task can be viewed as a flow of computations across spaces. Listing 2 provides a simple example of a dual-space, finite difference code to illustrate various aspects of a task.

```
 1 Task 'Five Point Stencil':
 2     Define:
 3         plate: array dimension 2: Real Precision Single
 4         t: Epoch
 5         total_iterations: Integer
 6     Environment:
 7         plate: link
 8     Initialize (iterations):
 9         total_iterations = iterations
10         t.beginAt(1)
11     Compute:
12         'Synchronize' (Space A) {
13             'Refine Estimate' (Space B) {
14                 localRows = plate.local.dimension1.range
15                 localCols = plate.local.dimension2.range
16                 do {
17                     do { plate[i][j] at (t)
18                             = 1/4 * (plate[i-1][j]
19                             + plate[i+1][j]
20                             + plate[i][j-1]
21                             + plate[i][j+1]) at (t-1)
22                     } for i, j in plate
23                       and (i > localRows.MIN and i < localRows.MAX)
24                       and (j > localCols.MIN and j < localCols.MAX)
25                 } while t % partition.n != 0
26             }
27             Repeat: from 'Refine Estimate' while t % partition.m != 0
28         }
29         Repeat: from 'Synchronize' while t < total_iterations
30     Partition (p, k, l, m, n):
31         Space A <1D> {
32             plate: block-size(p) padding(m)
33         }
34         Space B <2D> divides Space A partitions {
35             plate: block-size(k, l) padding(n, n)
36         }
```

Listing 2. A dual-space finite difference IT code

The task has five distinct sections. In the *Define* section, data structures used in computation are defined. Note that all arrays are dynamic: only their dimensionality and types are specified here, not the dimension lengths. The *Environment* section lists data structures that compose the task environment. Only the data structures listed in Environment persist once the task finishes execution. The code in the *Initialize* section is invoked when the task begin execution. It runs sequentially and initializes some global variables and settles sizes of defined arrays. Then the *Compute* and *Partition* sections are for user computation and data partitioning respectively.

The flow of computations is specified in the *Compute* section of a task, as in line 11 to 29 in the above. In this specific example, there is only one high-level computation at Space A, called *Synchronize*. The flow enters into this computation at line 12 and comes out of it after executing line 28. Then in line 29, it hits the flow control instruction to repeat the computation if iterative refinement is not done up to the specified number of times. If the condition holds true, the flow re-enter into Space 2 to execute Synchronize again.

A computation specifies either the code to execute or a sequence of sub-computations and flow-control instructions. In case of Synchronize, it is the latter. The actual code for iterative refinement is done in the Space B computation *Refine Estimate* spanning from line 13 to 26. So the overall flow for this task is that the flow enters into a Space A computation, goes in and out of a Space B computation a number of times before leaving Space A, and then either reaches its end or returns to the Space A computation.

Although the task can be explained as a single sequence (or a chain) of computations in spaces (PCSs), at runtime it unfolds and executes as a tree whose branching factors are dictated by the instructions given in the *Partition* section, from line 30 to 36. Here the plate is partitioned as slabs of P rows in Space A. Each of those slabs is partitioned as $2D$ blocks of dimension $K - by - L$ in Space B. An independent computation unit (PCU) takes care of a partition in corresponding space. Therefore, assuming there are R rows in the plate, the aforementioned flow will be executed independently in $\lceil R/P \rceil$ PCUs in Space A.

The execution of a PCU follows the owner-compute rule, that is, it operates over only those parts of data structures that are assigned within its partition. Therefore, the for loop in line 22 iterates over the local columns and rows of the plate in individual PCUs.

The significance of space boundaries and movements of the flow of control across those boundaries can be fully understood if we analyse how PCUs are related to one another. *PCUs are related exclusively by their data dependencies, and data-synchronization only happens across space boundaries and as needed basis*. Data dependencies can take one of the following three forms: overlapping regions as paddings, changes to a replicated data structure, and PCUs of a lower space updating data used in a upper space (and vice versa).

Since PCUs execute independently except in case of data dependencies, the overall flow of the task can proceed at different rates in different branches of the execution tree. At a particular instance, different PCUs may be executing different iterations of a computation or even executing altogether different computations.

In the aforementioned task, padding for Space A PCUs is M rows and for Space B PCUs is N in each dimension. That justifies both the while loop in line 25 and the repeat instruction in line 27. Note that the actual values of partition and initialization parameters are passed as arguments when the task is invoked by the coordinator program.

A more involved, nonetheless clear, example of IT task is the LU decomposition task presented in Listing 3.

```
1  Task 'LU Factorization':
2      Define:
3          a, u, l: array dimension 2: Real Precision Single
4          p: array dimension 1: Integer
5          pivot: Integer
6          l_column: array dimension 1: Integer
7          t: Epoch
8      Environment:
9          a: link
10         u, l, p: create
11     Initialize:
12         u.dimension = l.dimension = a.dimension
13         l_column.dimension = l.dimension1
14         p.dimension = a.dimension1
15     Compute:
16         'Prepare' (Space B) {
17             do { u[i][j] = a[i][j] } for i, j in a
18             do { l[i][i] = 1 } for i in l
19         }
20         'Select Pivot' (Space A)
21         Activate if k in local-range of u.dimension2 {
22             do {
23                 pivot maxEntry= u[i][k]
24             } for i in u and i >= k
25         }
26         'Store Pivot' (Space C) {
27             p[k] = pivot
28         }
29         'Interchange Rows' (Space B) {
30             if (k != pivot) {
31                 do {    u[k][j] at (t) = u[pivot][j] at (t-1)
32                         u[pivot][j] at (t) = u[k][j] at (t-1)
33                 } for j in u and j >= k
34                 do {    l[k][j] at (t) = l[pivot][j] at (t-1)
35                         l[pivot][j] at (t) = l[k][j] at (t-1)
36                 } for j in l and j < k
37             }
38         }
39         'Update Lower' (Space A)
40         Activate if k in local-range of l.dimension2 {
41             do { l[i][k] = u[i][k] / u[k][k]
42             } for i in l and i > k
43             l_column = l[...][k]
44         }
45         'Update Upper' (Space B) {
46             do { u[i][j] = u[i][j] - l_column[i] * u[k][j]
47             } for i, j in u and i > k and j >= k
48         }
49         Repeat: from 'Select Pivot' for k in a.dimension1.range
50     Partition:
51         Space C <un-partitioned> { p }
52         Space B <1D> {
53             a<dim2>, u<dim2>, l<dim2>: strided
54             l_column: replicated
55         }
56         Space A <1D> <dynmaic> divides Space B partitions {
57             u<dim2>, l<dim2>: block-size (1)
58             l_column: replicated
59         }
```

Listing 3. LU decomposition with partial pivoting

In the above, the *Select Pivot* and *Update Lower* computation stages are protected with *Activate* conditions. This is the mechanism for avoiding concurrent updates of shared data structures (or shared regions of a data structure) in IT. At a particular iteration of the flow of control, only those PCUs that have respective activating conditions evaluated to $True$ can enter a critical region. Any update made by them on the shared data is propagated to the rests before the data is been read again.

Partitions of a data structure in different spaces may be independent or hierarchically related to one another. That is, we have a partial ordering of spaces and data structures – not a total ordering. Relationships between partitions may be specified either at the space level using the $divides$ instruction as in line 56 of Listing 3 or individual data structure by data structure basis. The sparse matrix dense vector multiplication task in Listing 4 illustrates the use of independent data partitioning.

```
Tuple ValueCoordinatePair:
    value: Real Precision Single
    row, column: Integer

Task 'Sparse Matrix Dense Vector Multiplication':
    Define:
        m: array dimension 1: ValueCoordinatePair
        v, w: array dimension 1: Real Precision Single
        w_local: array dimension 2: Real Precision Single
    Environment:
        m, v: link
        w: create
    Initialize:
        w.dimension = m.dimension1
        w_local.dimension1 = partition.p
        w_local.dimension2 = w.dimension
    Compute:
        'Local Multiplications' (Space A) {
            do {
                index_value_pair = m[i]
                value = index_value_pair.value
                row = index_value_pair.row
                column = index_value_pair.column
                w_local[p][row] += value * Vvcolumn]
            } for k in m; p in w_local
        }
        'Accumulate Parts' (Space B) {
            do {
                w[i] += w_local[k][i]
            } for i in w; k in w_local
        }
    Partition(p, r):
        Space A <1D> {
            w_Local<dim1>: block-size(1)
            m: block-count(p)
            v: replicated
        }
        Space B <1D> {
            w, w_local<dim2>: block-size(r)
        }
```

Listing 4. Sparse matrix dense vector multiplication

In the above code, the dense vector v is replicated in and the sparse matrix m is uniformly distributed among p Space A PCUs. Results produced in individual Space A PCUs in *Local Multiplications* stage is been accumulated by Space B PCUs into w in *Accumulate Parts* stage. As a single Space A PCU may need to contribute its results to all Space B PCUs, w_local's partitioning in Space A and B are independent of each other.

B. Logical to Physical Spaces Mapping

The previous discussion of a task's spaces and partitions is applicable at the logical level. The mapping from logical to physical spaces and from logical to physical partitions is done once a mapping configuration file is given. The compiler needs this file to generate the target code that can run in the physical infrastructure.

Assume we want to run the code in Listing 2 in an MPI-Pthread hybrid environment having two workstations each having one 16-core CPU. Figure 2 depicts a possible mapping for such an environment.

As the figure illustrates, the Space A of the Task is mapped to individual CPUs; and within each CPU, Space B PCUs are mapped to individual cores. The mapping configuration file for this environment should be as follows. (Here the Model parameter identifies the PCubeS model to be used when an architecture has multiple of them.)

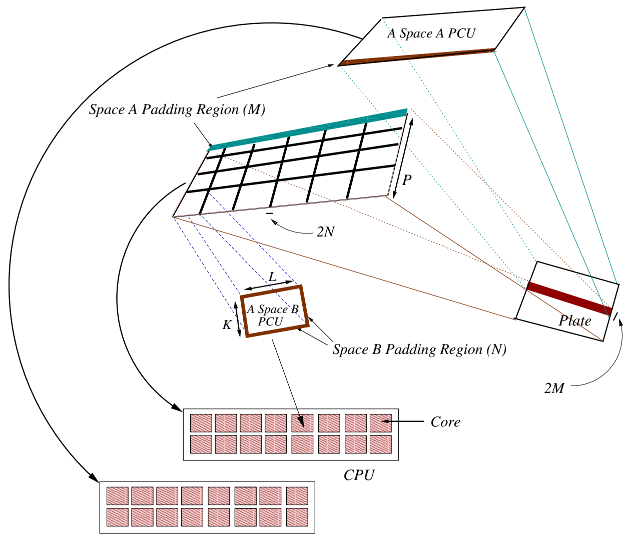

Fig. 2. Mapping logical spaces to physical spaces

```
A Mapping Configuration File
Mapping Configuration:
                    Task: 5 Point Stencil
                    Model: default
                    Space A: CPU
                    Space B: CPU Core
```

Given the above configuration, a reasonable runtime implementation will implement the Space A PCUs as MPI processes and data synchronization in Space A should be done using MPI send-receive primitives. Space B PCUs, on the other hand, can be implemented as Pthreads, where data synchronization should be equivalent to copying updates from one buffer to another within the memory shared by the cores of a CPU.

If we rather want to run the code in Listing 2 in an NVIDIA GPU, we may not want to have a dual-space executable with paddings in both spaces as in the previous case. This is due to the small shared memory of its symmetric multiprocessors (SMs) – the programmable shared memory is typically only 48 KB – that should be used mostly for useful computations. Then the programmer can effectively converts the same code into a single-space computation by using the following mapping configuration file.

```
Alternative Mapping Configuration File
Mapping Configuration:
                    Task: 5 Point Stencil
                    Model: default
                    Space A: un-partitioned
                    Space B: SM
```

With this new mapping, the target code will ignore any partition instruction and argument destined for Space A, but it will execute the computation of Space A in Space B to coordinate the latter.

This mapping example reveals how IT relieves a programmer from dealing with the nitty-gritty details of an execution platform. As in the aforementioned example, he understands communication and synchronization from a logical level. The tedious details of data-exchange in a real environment is understood by the compiler, taken care of by the runtime engine, and the programmer is not bothered about that. This distribution of responsibilities is central to our dual goals of program's efficiency and programmer's productivity in IT.

V. Language Features

This section briefly describes some key features of IT programming language.

A. Data Types

Data types support in IT is minimal. As the language is designed for scientific computing, it has support for only those types that are most useful in that domain. All basic primitive types are supported. Strings are limited to literal strings of characters with no additional behavior, they are useful for invoking tasks only. List and array are the two supported collection types. List is not partition-able and mainly useful in the coordinator program for environment management. It has the conventional methods for append, remove, iterate, and index-based element access; but modification to a list is only allowed inside the coordinator program.

1) Array: Array is the most important and only partition-able data type in IT. All array variables are dynamically allocated. There is one overall and one for each $dimension$ attribute in an array that can be accessed as $dimension\$index$. Each dimension has a $range$ and an $index$ attribute and a flag indicating if it is CIRCULAR or LINEAR. The index has a $stride$ attribute that is used for non-unit striding. Modifications to all these attributes should take place either in the coordinator program or in the Initialize section of a task. There is a local version of all these attributes that can be referred inside a space computation using the syntax $local.attributeName$.

2) Tuple: Tuple is the mechanism for user-defined data types in IT. The grammar for tuple is as follows.

```
Tuple =>  tuple Identifier : Element+
Element => Name : Type
```

The $Type$ in the above can be either a primitive, another tuple type, or a fixed length array – no list or dynamic array is supported. Therefore, a tuple is similar to a struct in C without pointers. An array can hold objects of a tuple type and can be partitioned, but the content of a tuple cannot be broken into pieces. This combined with prohibition on list modification within a task ensure that before a task starts executing its Compute section, the maximum memory to be consumed by each PCU is predetermined.

3) Epoch: Epoch is a convenience data type to aid management of iteration (or time) dependent variables. Assume the value of a variable, say v, at a particular iteration is computed by averaging its values from the last two iterations. Then the computation can be expressed in terms of epoch t as follows.

```
v at (t) =  1/2 * (v at (t − 1) + v at (t − 2))
```

The target code for the above will have three versions of v and maintain their runtime relationship appropriately. The value of t is incremented by 1 each time such an expression is executed. For explicit control of the value of an epoch three methods – *beginAt, advanceBy, and reset* – are supported. The first two takes an integer constant as the sole argument.

B. The Type System

IT is a strongly and statically typed programming language. There is no automatic type conversion or coercion. That said, types of data structures are only specified in the Define section. The type of any undefined variable is determined from its use in conjunction with some defined variable. This can be done at compile time because there is no type inheritance.

Note that any undeclared variable used in a space computation is local to that computation in each PCU and retains any value assigned to it across iterations.

C. Partitioning Support

We realize that the policy for partitioning an array can vary widely depending on applications and infrastructures. Hence in IT we take a flexible approach to data partitioning. Instead of a fixed set of library methods, IT provides an interface for implementing new partitioning policies as libraries. Any runtime implementation of IT has to provide implementations for the four most commonly used partitioning policies: *block-size, block-count, strided, and block-strided.*

D. Input/Output

We plan to implement a mechanism for I/O handling that is geared towards spaces and partitions. If we refer back to the Environment section of Listing 2, we see the $plate$ is mentioned to be linked in the task environment. How it actually becomes available for computation is determined at runtime. The same is true for the A, B operand matrices in Listing 1. This is the sole mechanism for specifying environmental inputs to a task.

In case the input needs to be read from a storage system, the coordinator program issues the read command. Two convenience methods are provided for reading an array and a list of arrays respectively that have the following type signatures (there are two similar methods for write).

```
load_array(fileName: String, dimensions: Integer)
    returns array: T
load_list_of_arrays(fileName: String, dimension: Integer)
    returns list: array: T
```

Here the base type T is resolved by the compiler. The more important fact in the above, however, is that a read in the coordinator program only results in reading of array meta-data such as dimension lengths and locations of data files. Actual data reading takes place inside a task once it begins execution.

We take this approach to I/O handling to take benefits of parallel file-systems such as Lustre [9] or GPFS [10] that are common in current parallel architectures. Given the support for parallel I/O is available, read takes place in the I/O capable PPS nearest to the PPS that need the data first during task execution. In environments lacking parallel I/O, the read happens before actual computation for the task begins. The compiler inserts appropriate codes for I/O in between user's computations using its knowledge of the execution platform. Writes receive the same treatment of reads. Any instruction to write a data structure in the program is translated by the compiler into codes that initiate writes from inside a task as the data becomes available.

VI. A Brief Note on Compilation

Although IT compiler is a topic for a future paper, we briefly discuss the nature of compilation process here.

We are working on a 2-step compilation process. The first, platform independent compiler generates abstract PCU descriptions, flow control commands, data synchronization and communication directives, etc. in an intermediate language form. This compiler does not translate the content of a space computation.

The second, platform dependent compiler takes the mapping configuration file and intermediate code as input and generates the target code. This compiler has intimate understanding of the PCubeS model of its underlying execution platform and translates the abstract description of data partition, synchronization, communication, etc. present in the intermediate form into concrete instructions based on the hardware and programming features available. It uses message passing, threading, SIMD instruction streams, and whatever other facilities available in the execution platform to generate the target code.

Therefore, our plan in IT is to use MPI and other forms of message passing or threading features as infrastructure supports – not to replace them. We believe, this is the right place for low-level programming primitives unless extreme fine-tuning of a program is required.

In short, the first compiler is responsible for generating the correct program and the second is for generating an efficient executable. In fact, in the absence of a mapping configuration file and the knowledge of the hardware, the first compiler can do only a little about code optimization.

The compiler makes several passes over the source code and generates an intermediate representation as an information flow. Initially the flow description has only computation stages as nodes and arcs among them. Through several passes the compiler adds nodes for memory allocation, synchronization, data read-write, etc. in between the computation nodes in appropriate places. To be able to do so, it employs well-known static analysis techniques to determine how different data structures are used in individual computation stages. To determine synchronization needs and to characterize communication arcs it combines data partitioning information with the analysis results.

Finally, it does some architecture independent optimizations such as avoiding memory allocation for variables that are merely references to other variables. The final flow description is generated based on a topological ordering of nodes and needs not match the original ordering the programmer specified in the source code.

We are working on the second compiler for our first target execution platform: a single NVIDIA GPGPU controlled by a CPU. We are investigating how the intermediate information flow can be translated into an efficient C + CUDA program. We choose this platform first due to the shared memory nature of GPUs and an unified address-space provided by the CUDA programming paradigm. Both tremendously simplify the task of communication and synchronization. For examples, replication is implemented as mere reference sharing and data synchronization among SMs are ensured just by choosing appropriate CUDA kernel boundaries.

VII. Early Results

To get an understanding of how IT programs may work in practice, we emulated the output of the second compiler for some sample IT tasks. This section discusses the findings of two such experiments.

For these experiments our test platform was an NVIDIA Tesla C2050. It has 14 symmetric multiprocessors (SMs) with warps operating within as groups of 32 lock-step threads. Its on-board global memory is 3 GB, and shared memory per SM is 64 KB but only 48 KB of that is programmable. Table I provides a tabular description of the PCubeS model of this GPU.

TABLE I. PCubeS Description of Tesla C2050

Space	Processing Capacity	Memory	Partitions	Communication
0: Warp	32	None	0	N/A
1: SM	512	<48 KB, 16, 1, 16>	16	<2, 16>
2: Card	7168	<3 GB, 32, 300, 32>	14	<600, 32>

The graph in Figure 3 shows the performance of two variations of IT matrix multiplication code with respect to the best implementation in the GPU. All versions of IT code are compiled with NVIDIA NVCC compiler 5.5 with compatibility mode set to 2.0 and -O3 compiler optimization flags enabled. The results are average running times in seconds out of three sample runs. The input matrices are square and of equal size. The CUBLAS Sgemm library code [11] that is used as a reference implementation for comparison is a highly tuned, closed source assembly code.

IT source for the first version is the code in Listing 1. As we can see, this is a single space task. We map the space to physical Space 0, that is, to warps. With this configuration, given the large matrix sizes, it is impossible to use the fast shared memory available within the SMs and computation is done directly on the data stored on the slow, on-board global memory; and we have the corresponding curve when runtime configuration for each PCU is $K = L = 1$.

The second version, that is a dual-space code implementing block matrix multiplication algorithm as shown in Listing 5, does not suffer from the aforementioned problem and comes quite close to the assembly code when Space A is mapped to SMs and Space B to warps, and partitioning arguments are chosen appropriately to maximize memory access alignment and thread utilization.

```
Task 'Matrix Multiply':
    Define:
        a, b, c: array dimension 2: Real Precision Single
    Environment:
        a, b: link
```

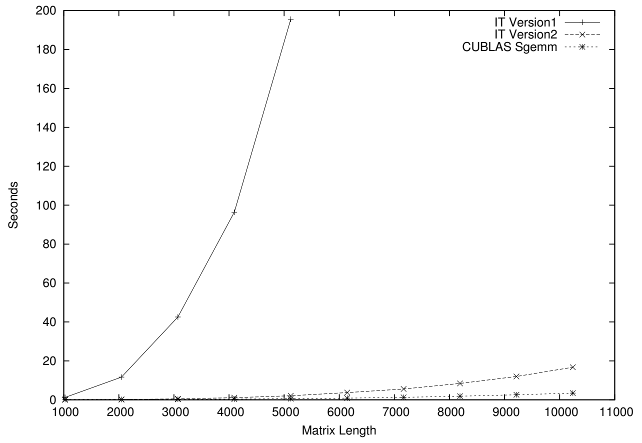

Fig. 3. Performance of two IT matrix multiplication tasks compared to CUBLAS Sgemm

```
        c: create
    Initialize:
        c.dimension1 = a.dimension1
        c.dimension2 = b.dimension2
    Compute:
        'Block multiply matrices' (Space B) {
            do { c[i][j] += a[i][k] * b[k][j]
            } for i, j in c; k in a
        }
        Repeat: from 'Block multiply matrices' foreach Space A sub-partition
    Partition (k, l, q, m, n):
        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)
            }
        }
        Space B <2D> divides Space A sub-partitions {
            c: block-size(m, n)
            a: block-size(m), replicated
            b: replicated, block-size(n)
        }
```

Listing 5. Block matrix multiplication

The actual computation for block matrix multiplication and the straightforward algorithms are the same. The block algorithm only computes the output incrementally from partial results to have better memory reuse. The nature of this change is reflected in the modifications done in the corresponding IT task above. The principal difference between Listing 1 and Listing 5 is, thereby, in the partition section.

Figure 4 compares the performance of different IT LU factorization codes with that of two reference implementations. All these codes are compiled and run with the same settings used in the previous experiment.

Like the first matrix multiplication code, the first IT implementation of LU factorization as shown in Listing 3 does not perform well. This is due to the choice of strided column partitioning in Space B that results in severe efficiency loss in read/write to global memory when Space B is mapped to SMs. Only by storing the L and U matrices in transposed (i.e., column major) order and choosing strided or block-strided row partitioning that problem can be eliminated. Consequently

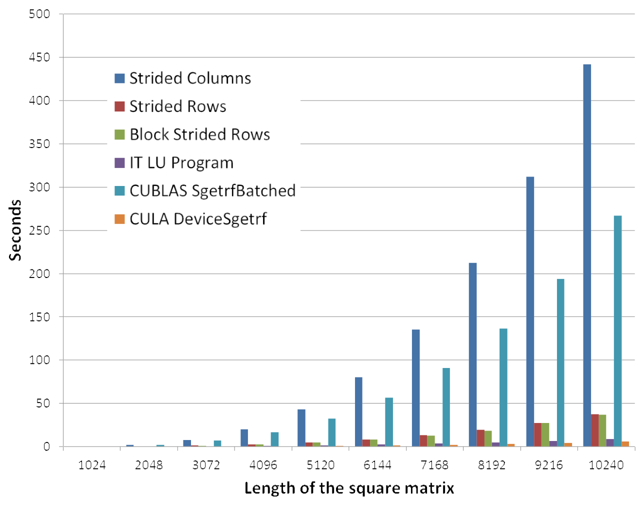

Fig. 4. Performance of several CUDA implementations of LU decomposition

the next two implementations show remarkable performance improvement. This is almost as good as one can get with classic LU decomposition algorithm.

More efficient implementations for shared memory environments use a block LU decomposition algorithm that combines several BLAS level 2 and level 3 routines. CULA DeviceSgetrf [12] is one such standard implementation for NVIDIA GPUs. We implemented the block LU decomposition algorithm as an IT program that combines a slightly modified LU decomposition task with matrix multiplication task (we do not present the program here for space limitation). Our program's efficiency remains within 68% of the reference implementation even for the largest matrix. Given that our implementation of matrix multiplication has significant performance gap with CUBLAS Sgemm that is used in CULA, the performance of LU program looks very promising.

The bad performance of CUBLAS SgetrfBatched appears quite surprising. May be the reason behind this is that the library function is optimized for batched decomposition of many small matrices and not intended to be used for decomposition of one large matrix. We present its outcome not to show how good we are compared to a standard implementation; rather to prove the point that blind use of vendor provided library functions is not an appropriate approach to parallel programming.

VIII. Conclusion

In this report we introduced the PCubeS type architecture and the IT programming language. PCubeS builds on Synder's notion of type architectures and the need to accurately expose key architectural features with their associated costs to the programmer via the programming language.

PCubeS presents a parallel machine as a finite hierarchy of nested parallel processing spaces (PPS), each with its own aggregate parallel processing capacity; finite memory with fixed latency, bandwidth, and transaction width; a fixed set of uniform, independent partitions (PPU); and a fixed communication speed between components.

Not all possible architectural styles can be represented by PCubeS. However many contemporary and envisioned architectures can be represented cleanly: large scale clusters of multicore nodes, single nodes with one or more accelerators, clusters of multicore nodes with accelerators, and even computational grids with similar supercomputers at multiple sites, e.g., many of the top 100 machines.

The IT language builds on PCubeS and reflects the hierarchy and nature of the architectural model in the notion of nested computing spaces (PCS). The amount of memory available in each PPS is reflected in the need to partition the data so as to ensure that there is sufficient memory to hold the computing spaces and their partitions (PCU). Similarly, the costs of moving up and down the hierarchy, and moving data between spaces is captured in a task's flow of control.

The goal with IT is to keep it slim and simple, while also providing the programmer the tools to manage the truly critical data layout, alignment, communication, and synchronization as first class language elements.

IT tasks consist of several sections that describe different aspects of the tasks: the data structures, in particular arrays; linkage of local variables and structures to the global environment; initialization; the actual execution statements defined in multiple compute spaces; and the partitioning directives. These are further bound before runtime by a mapping configuration that maps the different spaces in a task to particular hardware resources.

The separation of the computing aspects of a code from its data placement, we believe, is both crucial and novel, and results in codes for which the compute (algorithmic) section need not change as one goes from platform to platform. Instead, porting will often involve only changes in partitioning and mapping (assuming the IT runtime system has been ported).

Early results 'hand compiling' to the IT runtime system we are developing for CUDA look very promising. We showed how use of the type architecture description in the development of a code led to better and better performance. Performance of the final versions of IT matrix multiplication and LU factorization tasks were not too far off of what can be considered the best hand-tuned implementations.

IX. Related Work

To resolve the tension between high and low level programming techniques and combine the best of both worlds, Snyder first proposed to adopt an idealized machine model that should serve as the standard hardware-programming language interface. His seminal work 'Type Architectures, shared memory, and the corollary of a modest potential [8]' thus provides the foundation for the machine abstraction in any parallel programming language. He names the interface the *Type Architecture:* a description of the facilities of hardware.

Snyder's candidate type architecture (CTA) is a finite set of sequential computers connected in a fixed, bounded degree graph, with a global controller. CTA is not suitable for present day parallel architectures that are characterized by heterogeneity and often have hierarchies. After Snyder, only few type architectures have been proposed for parallel machines. Type architectures such as LogP [13] and LogGP [14] serve as more of an analysis tool and, to the best of our knowledge, never beget any language. The parallel memory hierarchy (PMH) model [15] comes closest to PCubeS. In PMH, a hardware platform is viewed as layers of memory with increasing capacities as one proceeds bottom up. Attention has neither been given to processing capability nor been to inter-component communication cost of individual layers.

DARPA funded parallel programming language initiatives of recent years claim to be directed towards high productivity [16]. Chapel [3], X10 [4], and Fortress [17] – the three DARPA funded languages – have manifold features for expressing, grouping, and mapping parallelisms. Fortress even has all its expressions treated to be parallel by default. Benchmark programs written in these languages often run on par with equivalent MPI programs, but the languages have dubious prospects regarding being widely adopted in the future. We believe a wide assortment of features overburdens these languages. In many cases, the desire to support everything leads to poor performance. Fortress is a glaring example of this problem. The project itself has been terminated due to lack of efficient implementations of proposed features.

Recent partitioned global address space (PGAS) languages such as Co-array Fortran [18], Titanium [19], and UPC [20] maintain a shared memory view of the environment. Here the shared memory is, however, visibly partitioned among available processors. The programmer distinguishes between local and non-local memory references and responsible for explicit synchronizations. These languages are closer to IT than DARPA languages due to their focus on a modeling framework. We believe PGAS model hides too much of the underlying architecture; thereby inappropriate for high performance computing.

Despite significant research on high level languages, dominant modes of parallel computing are still low level, library based approaches. The message passing interface (MPI) [2] is the standard for parallel programming for over a decade. The OpenMP [21] directive based parallelization is also popular for easy parallelization in shared memory environments. A central reason for MPI and OpenMP's success is that they require minimal learning over a programmer's existing knowledge of C or FORTRAN. To parallelize a code using OpenMP, the programmer just adds few pragma directives on top of loops. Although one can do only so much with pragmas, often-times that is all what he needs. MPI that extends C and FORTRAN with explicit communication facilities, on the other hand, provides the programmer with absolute control regarding exploitation of parallelism.

NVIDIA's CUDA [22] programming model has become quite popular with the advent of GPGPUs in high performance computing. CUDA follows in the footsteps of earlier SIMD/SPMD languages such as C*, C**, pC^{++}, and many others [23]. It is another low level programming technique and only applicable in NVIDIA GPUs. Just like in MPI, efficient CUDA programs are difficult to write and lack readability.

So far few languages have been developed with an underlying type architecture in mind. Snyder's own Poker [24] programming language suffers from difficulties in mapping many algorithms to the CTA type architecture. Space Limited Procedures [25] and Sequoia [26] [27] are two languages developed with PMH as the type architecture. These languages are overly restrictive as they provide no support for communication between parallel task units that rules out many common computational problems.

In PCubeS, we slightly move away from Snyder's notion of type architecture. In his work, Snyder focuses on describing the bare hardware only, but we treat low level programming features combined with hardware features as facilities. This modeling shift provides the flexibility critical for viewing diverse architectures as PCubeS instances.

Although elements of IT were there for a long time, their presentation and treatment in IT are quite different from the previous approaches. This combined with PCubeS is an interesting new combination that we hope will be able to provide answers to many current problems of parallel programming.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, M. J. Demmel, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from berkeley," TECHNICAL REPORT, UC BERKELEY, Tech. Rep., 2006.

[2] D. W. Walker, D. W. Walker, J. J. Dongarra, and J. J. Dongarra, "Mpi: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[3] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[5] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] N. Wirth, "On the design of programming languages," in *IFIP Congress*, 1974, pp. 386–393.

[7] C. A. R. Hoare, "Hints on programming language design." Stanford, CA, USA, Tech. Rep., 1973.

[8] L. Snyder, "Annual review of computer science vol. 1, 1986," J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, Eds. Palo Alto, CA, USA: Annual Reviews Inc., 1986, ch. Type Architectures, Shared Memory, and the Corollary of Modest Potential, pp. 289–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=17814.17826

[9] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *PROCEEDINGS OF THE LINUX SYMPOSIUM*, 2003, p. 9.

[10] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083349

[11] nVidia, *CUBLAS Library User Guide*, v5.0 ed., http://docs.nvidia.com/cublas/index.html, nVidia, Oct. 2012. [Online]. Available: http://docs.nvidia.com/cublas/index.html

[12] E. Photonics, *CULA Programmer's Guide*, r17 ed., EM Photonics, May 2013. [Online]. Available: http://www.culatools.com/dense/

[13] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/155332.155333

[14] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation," 1995.

[15] B. Alpern, L. Carter, and J. Ferrante, "Modeling parallel computers as memory hierarchies," in *Programming Models for Massively Parallel Computers, 1993. Proceedings*, 1993, pp. 116–123.

[16] E. LUSK and K. YELICK, "Languages for high-productivity computing: The darpa hpcs language project," *Parallel Processing Letters*, vol. 17, no. 01, pp. 89–102, 2007. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626407002892

[17] J. Steele, GuyL., E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, and S. Ryu, "Fortress (sun hpcs language)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 718–735. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_190

[18] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[19] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," in *In ACM*, 1998, pp. 10–11.

[20] T. El-Ghazawi and L. Smith, "Upc: Unified parallel c," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188483

[21] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, p. 40–53, 2008.

[23] G. V. Wilson and P. Lu, Eds., *Parallel Programming Using C++*, ser. Scientific and Engineering Computation. pub-MIT:adr: MIT Press, 1996, foreword by Bjarne Stroustrup. Describes fifteen parallel programming systems based on C++. [Online]. Available: http://www.mitpress.com/book-home.tcl?isbn=0262731185

[24] D. Notkin, L. Snyder, D. Socha, M. L. Bailey, B. Forstall, K. Gates, R. Greenlaw, W. G. Griswold, T. J. Holman, R. Korry, G. Lasswell, R. Mitchell, and P. A. Nelson, "Experiences with poker," in *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*, ser. PPEALS '88. New York, NY, USA: ACM, 1988, pp. 10–20. [Online]. Available: http://doi.acm.org/10.1145/62115.62118

[25] B. Alpern, L. Carter, and J. Ferrante, "Space-limited procedures: a methodology for portable high-performance," in *Programming Models for Massively Parallel Computers, 1995*, 1995, pp. 10–17.

[26] M. Bauer, J. Clark, E. Schkufza, and A. Aiken, "Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia," *SIGPLAN Not.*, vol. 46, no. 8, pp. 13–24, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/2038037.1941558

[27] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, 2006, pp. 4–4.