A Survey of how Parallel Programming Models Address Computation Distribution

Andrew Stone
PhD Research Examination
Colorado State University
Department of Computer Science
stonea@cs.colostate.edu

April 27, 2010

Abstract

The power wall is driving parallelism to play an increasingly important role in software development. However, parallel software development is complicated by several challenges that are absent in serial software development. We argue that these challenges emerge from three factors that influence the design and performance of parallel software. These factors are: the distribution of computation, the distribution of data, and the communication that occurs to transfer data and coordinate computation. Industry and the research community have development several programming models to address these factors. In this paper we survey how five of parallel programming models address the factor of computation distribution. Our survey is structured in terms of four interpretive criteria, and we discuss a number of evaluative criteria can be used to judge programming models. After our survey, we discuss the impact the other two factors (distribution of data and the use of communication) have on computation distribution and hypothesize on what other challenges programmers face.

1 Introduction

Since the 1970s Moore's Law has resulted in a trend of exponentially growing processor performance [33]. There are commercial and research interests motivating a continuation of this trend [5, 8]. Additional computing performance drives the sale of new hardware; new software can leverage such hardware. For example, additional computing resources enable quicker and more accurate results in scientific simulation software.

Historically, faster clock speeds have driven this performance trend. However, clock speeds have approached a number of physical barriers [33, 5]. One barrier is the power wall: the unsustainable exponential power growth that occurs with increases in processor operating frequency. Increased power usage is not only costly but also results in additional emitted heat. Dissipating excessive amounts of heat requires sophisticated and expensive cooling systems. One way of increasing computing performance without hitting the power wall is to concurrently utilize multiple processing units. Given this fact, it seems likely that as the role of clock speeds subsides the role of parallelism will rise. This is supported by the rising trend of multicore processor production. The increasing availability of multicore machines, along with improvements in network technology, has also influenced the development of cloud computing services and the production of ever-larger compute clusters.

Although these trends indicate a future abundance of parallel computing resources, these resources are useless in the absence of software to properly utilize them. Parallel software faces several challenges unseen in serial software. These challenges are made evident upon reflection of the architectural differences between serial and parallel machines. The Von Neumann model has long been used to illustrate the architecture of a serial computer. In this model a computer consists of a single processing unit (that contains a control unit and an arithmetic logic unit) tied to a single memory store. Parallel machines, of course, have multiple processing units, and sometimes these machines have multiple stores. Parallel machines also have some type of interconnect present between multiple processing units. The addition of multiple processing units implies that software must distribute itself across these units in or-

der to effectively utilize them[1]. The addition of multiple memory units implies that parallel software must also distribute its data across these units. A communication interconnect is necessary to coordinate computation across multiple units and to transfer data. These three factors of parallel programming: the distribution of computation, the distribution of data, and the use of communication, influence the design and performance of parallel software. In this paper we choose to focus specifically on one: the distribution of computation.

Several programming models exist to help programmers and compiler writers face the challenges of parallel software development. In this paper we compare how these models address the issue of computation distribution and discuss how these models can be evaluated and compared. In Section 2 we outline criteria for this comparison and evaluation. In Section 3, we apply the criteria against the surveyed approaches. We summarize our analysis in the tables listed in Appendix A. The structure of these tables reflects the structure of our survey. After our survey, in Section 4, we discuss how computation distribution interacts with the two other factors (data distribution and communication). In Section 5 we end this paper with summarizing and concluding remarks.

2 Interpretive and evaluative criteria

When it comes to the distribution of computation, different programming models have different assumptions, limitations, and strengths. To aid in understanding these assumptions, limitations, and strengths we compare these models along various criteria. There are two types of criteria we consider: *interpretive criteria* and *evaluative criteria*. Interpretive criteria are used to describe a model, evaluative criteria are used to judge it. In this paper we focus our interpretive criteria on how computation distribution is addressed.

2.1 Interpretive criteria

Specifically, the interpretive criteria we consider are: 1) how the distribution of computation across processing elements is specified in a programming model, 2) how the distribution of computation across time is specified in a programming model, 3) how computation is structured in a programming model, and 4) how compilers or runtimes of a programming model realize a distribution of computation. In this subsection we discuss these criteria, and in Section 3 we will use them to survey five existing programming models.

2.1.1 The specification of how computation is distributed across processing elements

Our first criterion is the specification of how computation is distributed across processing elements. Some programming models have users explicitly specify how to distribute computation across processing elements; others have users define some strategy to guide a runtime to realize a distribution. Still, in others, computation distribution is automatically determined by a compile-time analysis. Sometimes the implicit distribution of computation is guided by an explicit definition of the distribution of data. In our surveyed models we see all of these approaches taken.

2.1.2 The specification of how computation is distributed across time

For parallel architectures, computation must not only be distributed across processing elements, but it must also be distributed across time. A single processing element can only be scheduled to execute one instruction at a time[2]. Thus, it is not enough to simply have a task distributed for execution on a processing element, it is also necessary that this task be executed at the proper time. Often dependences exist between tasks – the result of one task may be a prerequisite for the execution of another. Improperly scheduled tasks can lead to programs that have hard to debug race-conditions.

As with the specification of computation distribution across processing elements, the specification of computation distribution across time can either be done implicitly or explicitly. An example of an implicitly specified distribution across time might be a task graph where edges show dependences – several schedules can follow from analyzing this graph but no specific schedule is explicitly specified.

2.1.3 The structure of computation

Regardless of whether computation distribution is specified explicitly or implicitly programming models expect computation to have some type of structure. Models often have associated compilers or libraries to facili-

[1] Software may not always be able to utilize all processing units at peak efficiency. Indeed, if the inherently sequential portion of software greatly exceeds the parallelizable portion, the overhead incurred by parallelism may outweigh the improved performance of the parallelizable part. This limitation indicates a need to increase the parallelizability of software. Gustafson's law suggests that parallelizability can be increased through an increase in problem size [18].

[2] Vector processors can apply operations to multiple pieces of data concurrently, and many processors employ instruction level parallelism, however the stream of instructions is still fed-in one at a time.

tate with structuring computation. For example, in the map-reduce [12] model, users split computation into two types of tasks: mappers and reducers. Some map-reduce libraries, such as Hadoop [3], aid in structuring computation by including abstract mapper and reducer classes that programmers subclass to specify the mapping and reducing tasks.

2.1.4 The realization of distributed computation

Although programming models help to structure and specify computation, they are without practical benefit if they do not include some mechanism that ensures that the modeled computation is realized as specified when executed. This mechanism can come in the form of a compiler, a runtime system, or both. Generated code from a compiler sometimes builds on-top of other programming models. For example, some code generators output code in the MPI model. When this is the case multiple mechanisms need are employed to realize computation.

2.2 Evaluative criteria

The four interpretive criteria we identify can be used to gain an understanding of how parallel programming models address computation distribution. Another type of criteria can be used to judge these approaches. We call this type of criteria evaluative criteria. In this subsection we list several evaluative criteria for parallel programming models. Although these identified criteria are used evaluate programming models as a whole, they are all affected by how programming models address the factor of computation distribution. Each of the criteria we identify can best be described using a question they aim to answer. We list these questions and the criteria they associate with:

- *Portability* — How easily can a program written for one platform be retargeted to work with another? There is a diversity of architectures parallel programs can execute on (clusters, multicore systems, GPGPUs, etc.). If the structure or specification of computation is closely modeled after a single architecture it is unlikely that the model is portable.

- *Expressability* — How many different types of computation can be expressed in the model? The specification of computation distribution and the structure of computation impact the expressability of a model.

- *Performance* — How quickly do programs implemented in the model execute? Ultimately parallelization is motivated by the need for performance so this is an especially important criterion.

- *Testability* — How easy is to verify that programs written in the model are implemented correctly? Non-deterministic behavior negatively impacts testability. Also, it is often easier to test a serial implementation than a parallel implementation, and a common way of testing parallel applications is to compare results with a known working serial implementation. Some models improve testability by maintaining serial semantics, meaning they can be used to model a computation that can be understood and work as either a serial or parallel program.

- *Fault-tolerance* — How well can programs in the model address failure of components? For example, is the application able to recover in case of network or server failure?

- *Composability* — How well can the model be used within another model? Perhaps it is the case that one model is best at describing the overall structure of a computation while another is best at describing a particular aspect of that overall structure.

- *Interoperability* — How well can the modeled program interact with computations written in a different model? Interoperability is closely related to composability. Conceptually, the difference between composability and interoperability is that two composed models are combined to synthesize a new computation, whereas two interoperating models are represented as two discrete, albeit interacting, computations.

- *Clarity* — Is it easy to read and understand computations specified by the model? The opposite of clarity is obfuscation. Clarity is desirable because it impacts the maintainability of an application. The tangling of an algorithm with optimization and parallelization details negatively impacts clarity.

- *User-control* — How much control do developers have in addressing the realization of modeled computations? Models that heavily hide implementation details from developers have low user-control.

- *User-responsibility* — How much are developers required to specify before a computation can be realized? All other things being equal, models that have computation distribution specified implicitly or determined automatically have lower user-responsibility than models that require an explicit specification of computation distribution.

```
def rowOrder(n) {
    for row in 1..n {
        forall col in 1..n {
            yield (row, col);
        }
    }
}

def colOrder(n) {
    for col in 1..n {
        forall row in 1..n {
            yield (row, col);
        }
    }
}

// To change to column order simply change
// the iterator referenced by the for-loop:
for (row,col) in rowOrder(10) {
    writeln("at index: ", row, " ", col);
}
```

Figure 1: Row and column parallel iterators in Chapel

3 Surveyed models

To gain an understanding of the various ways computation distribution can be addressed we use the interpretive criteria outlined in Section 2 to analyze five existing parallel programming models. We also evaluate the impact these approaches have on portability. Our results are summarized in Appendix A. The five models we survey are: 1) iterated loops in the Chapel programming language, 2) Sequoia, 3) the map-reduce model in Granules, 4) a language called Id Nouveau, and 5) Tang and Xue's model for distributing tiled computations. In the following subsections we examine each of these systems.

3.1 Chapel iterators

Chapel [10] is one of the DARPA High Productivity Computing System (HPCS) languages [25]. The HPCS languages include features that aim to improve programmer productivity in the domain of parallel application development. One such feature is Chapel's *iterator* [21]. Iterators address the temporal issue of computation-distribution by making the traversal of elements orthogonal from the loop body that operates on those elements.

Iterators are referenced at the top of *for* or *forall* loops. Internally, they are written in a fashion syntactically similar to functions; as such they have the same expressibility as function. However, instead of returning values like a function, they *yield* values to loop-bodies. Conceptually control-flow of an iterated loop enters the iterator and executes the loop-body at the specified yield points. If a yield is within a parallel section of code (such as nested inside of a forall loop), the parallel yields will result in parallel instances of the loop-body being executed. Yield's may be embedded in Chapel's *on* statements to explicitly specify what processor a loop-body should execute on.

Iterators improve program clarity by enabling a separation of the specification of how to iterate over values from the specification of what operations are performed on each iteration. The visitation order of loop indices can impact performance. For example, loop-optimizations, such as tiling, reorder the iterations of loops to improve cache locality. The best iteration order for a computation is often influenced by the architecture it executes on. Thus simplifying the process of modifying an algorithm's schedule will also improve program portability. In [21] Joyner et. al demonstrate how the scheduling of a Smith-Watermann algorithm can be switched from a non-tiled schedule to a tiled schedule through the use of iterators.

In Figure 1 we show our own example of two iterators. The `rowOrder` iterator will sequentially step through each row of an n by n index space and operate on the elements of each row in parallel. The `colOrder` iterator will sequentially step through each column of the n by n space and operate on the elements of each column in parallel. The loop in our example references the `rowOrder` iterator, but it could easily be modified to use the `colOrder` iterator.

The use of iterators can have an undesirable impact on performance. Iterator's may require an amount of computation that wouldn't be seen in a loop nest with hard-coded boundaries; this is especially the case for iterators written in a generic fashion. For example, consider a generic, recursively defined, n-dimensional tiling iterator. The recursive nature of such an iterator would incur overhead from call-stack manipulation. The recursive use of iterators, however, does demonstrate that they are composable entities with other iterators.

Currently, it is also the responsibility of programmers to decide on what iterator is best suited for a given computation/architecture. One potential area for further research is to study how iterators can be chosen automatically. This could be done either at compile or runtime. However, dynamically switching iterators at runtime is complicated by the fact that iterators are not expressible as first class entities in the Chapel language. However, this limitation can be overcome by wrapping iterators within objects, which are first class entities.

3.2 Sequoia

Sequoia [16] is a programming language designed for the development of memory-hierarchy aware programs. Memory-hierarchies are arrangements of memory pools ordered in terms of response-time. For example, multi-

core processors typically have a fast access to L1 cache, a slower access to L2 cache, and an even slower access to main memory. Programs that frequently access values outside of cache will spend significant portions of time idling [13]. Additionally some architectures, such as the Sony/Toshiba/IBM Cell Broadband Engine [30], have memory pools that are local to individual processing elements (synergistic processing elements in the Cell). In architectures that have local memory pools it is commonly the responsibility of programs to transfer of data to and from the local memory pools as needed. This is the case in the Cell and compute clusters. Thus proper use of the memory hierarchy is important for both program performance and portability.

The memory hierarchy's importance motivates several constructs in the Sequoia language. These constructs enable programmers to have control over the movement and placement of data in the hierarchy. Sequoia's blocking primitives are used to partition arrays into smaller chunks. In Sequoia, when tasks are called and passed partitioned data, they will be compiled to perform communication within the memory hierarchy as needed. Recursively called tasks are meant to compute on data in a successive level of the memory hierarchy.

The unit of computation in the Sequoia language is the task. Tasks are defined as having two variants: an inner variant and a leaf variant. The inner variant decomposes input data into chunks amenable to a successive level of the memory hierarchy. In the inner variant, tasks decompose data for a successive memory-hierarchy level and recursively call themselves to operate on that data. The leaf variant is performed when computation should no longer be decomposed.

The inner variant of a task is expressed using a C-like language that has access to a tunable parameters. Tunable parameters are used to determine how to partition data for a successive layer of the memory hierarchy. The values of these parameters are architecture dependent and specified inside of a Sequoia mapping specification files. Sequoia's mapping specification files contain a number of *instance structures*. Each successive instance structure is meant to correspond to each successive level of the memory hierarchy. Given n instance structures a Sequoia computation will recursively decompose its input data n times, using the tunable parameters each instance structure defines to specify how to perform this decomposition.

Sequoia's mapping specification files simplify the porting of programs from one architecture to another. A new architecture may include a different memory hierarchy, which can be abstracted by a new mapping specification. When porting from one architecture to another a Sequoia program will have to be recompiled using a new mapping specification and a compiler targeted for the new architecture, however, the algorithm code should not require modification.

In [16] Fatahalian et al., evaluate the portability of Sequoia by implementing, and comparing the scalability of several numerical benchmarks on both a machine with a Cell processor and on a 16 node cluster of Intel P4 Xeons. The scalability of most benchmarks was comparable across the two architectures. However, the SAXPY and SGEMV benchmarks performed poorly on the Cell. Fatahalian et al. argue this is due to these computations being bandwidth-bound.

One current limitation of the Sequoia system is that its blocking primitives only operate on arrays of data, and they can only produce rectangular blocks. Computation in Sequoia also appears limited to working with algorithms that expressed in a divide and conquer manner on data. Some task parallel programs, such as master-worker computations, may not easily be expressible in the Sequoia model. The fact that communication is not exposed to the programmer in Sequoia has advantages and disadvantages. This non-exposure improves code clarity but limits user-control and may make certain optimizations, such as communication aggregation, unexpressible. Nevertheless, Sequoia has a clear strength in its ability to abstractly represent the memory-hierarchy and its separation of architecture-dependent details from task specification improves clarity and portability.

3.3 Map-Reduce and Granules

Granules [28, 29] is a streaming-based runtime for cloud computing. It enables the concurrent execution of tasks on a distributed set of resources. The granules system includes support for the map-reduce model. Map-reduce [12] performs computation on a large dataset by having concurrent mapping tasks operate on independent chunks of the dataset. Mapping tasks emit results that are then combined by one or more reducer tasks.

More generally, Granules models computations as a series of tasks that operate on various datasets. Datasets can come in different forms – files, databases, network packets, etc. Granule's tasks can be connected together as a graph that streams data from one task to another. These graphs have a directed structure and can contain loops.

The nodes of graphs in Granules represent computational tasks. Computational tasks are the fundamental unit of computation in Granules. They operate on one or more datasets and have an associated scheduling strategy. The strategy is described in a 3-dimensional parameter space. The axes of this space are: the counts axis, the data-driven axis, and the periodicity axis. The count axis specifies the number of times a task should

be executed, the data-driven axis specifies whether the task should be executed when data is available from one of its associated datasets, the periodicity axis specifies if the task should be executed on a periodic basis (for example, every 500 milliseconds). Complex scheduling strategies can combine these three. For example: execute some task up to 20 times, every 500 milliseconds, when data is available. Scheduling strategies can also be specified to execute until some termination condition is met.

Resource running the Granules runtime system have a pool of worker threads that manage and interleave the concurrent execution of tasks. The runtime is responsible for dispatching and managing these tasks, as well as establishing and managing streams so tasks can communicate through its NaradaBrokering substrate.

The Granules runtime is meant in facilitating cloud computing applications. Cloud computing services center around delivering resources, such as software, computation, and storage services, in an elastic manner. Availability of elastic resources scale up and down as the application requires them to. The Granules runtime is responsible for managing and allocating tasks along such resources.

In [29] Granules is evaluated by comparing the performance of various benchmarks (k-means, matrix-multiplication, mRNA sequencing, etc.) against Hadoop, Dryad, and MPI. There are several limitations to the map-reduce model, including the fact that mapping tasks are meant to be stateless and pleasantly parallel. However, Granules is more expressible than other map-reduce frameworks in that it can operate on streaming data and does not require a synchronization step between mapping and reducing phases. Most cloud-computing applications are written to be stateless; the statelessness of these applications enables the re-launching of tasks in case of failures. The ability to re-launch tasks in case of failure improves application fault-tolerance.

3.4 Id-Nouveau

Id-Nouveau [31] is a language and compiler that generates parallel code from programs that have serial semantics. It is a functional language, but uses an array construct called an I-structure. I-structures work like arrays in imperative languages but have elements that can only be written to once. In the Id-Nouveau programming model, programmers specify a domain decomposition for these arrays and this decomposition is used to determine a computation distribution. The Id-Nouveau compiler aims to distribute computation so that it has a locality of reference. In other words, computation is close to the data it operates on.

Parallelism in Id-Nouveau is not explicitly stated by the programmer but implicitly determined from the dependencies on data. This is simplified due to Id-Nouveau's single-assignment semantics. If a task accesses an element of an array that has not yet been assigned, it will block until a value of that element has been assigned.

Domain decompositions in Id-Nouveau are defined in terms of *array mappings*. These mappings consist of three functions: *map*, *local*, and *alloc*. The map function specify what processor owns the array index, the local function specifies where in that processors memory the element resides, and the alloc function is used to allocate instances of arrays with the mapping of a given size.

In [31] Rogers and Pingali demonstrate the use of Id-Nouveau by implementing a Gauss-Seidel computation in the language and comparing Id-Nouveau's generated parallel version against a hand-written parallel version. There compiler performs several optimizations, such as communication aggregation, that enable the performance of the generated version to be comparable to that of the hand-written. One striking limitation of the Id-Nouveau model is that computations must operate on data that is structured as arrays. Like Sequoia, it is unclear how well Id-Nouveau's approach works for irregular applications. It is also unclear if data can be reused in the Id-Nouveau system. The write-once semantics of arrays makes the data dependences of the computation evident but these semantics, outside of some mechanism to ensure reuse of data, may be too wasteful to be practical.

3.5 Tang and Xue's model for tiled code generation

Tang and Xue [34] discuss compiler techniques for generating tiled, message-passing, SPMD code. Iteration space tiling (sometimes known as blocking) is an optimization technique that restructures loops by aggregating iteration points into a series of tiles [35]. Processors execute iteration points within a tile atomically, that is a processor executes all of the points within a single before moving on to execute the next. In Tang and Xue's model, tiles have an n-dimensional rectangular shape.

Tang and Xue discuss how to use their described compiler techniques to convert a modeled sequential program into a modeled tiled sequential program, and how to convert that modeled tiled sequential program into a message passing SPMD program. The initial sequential modeled program represents a program as single statement that assigns a value to an array. The statement is nested in a set of for loops whose boundaries are calculated by affine expressions. The assigned value of the array is calculated by an function. All array accesses

in the function must be indexed by affine expressions. Specifically, the program is modeled as such:

for($i_1 = L_1$; $i_1 \leq U_1$; i_1++)
...
for($i_n = L_n$; $i_n \leq U_n$; i_n++)
$A(f(\overrightarrow{i})) = F(A(f(\overrightarrow{i} - \overrightarrow{d_1})), ..., A(f(\overrightarrow{i} - \overrightarrow{d_r})))$

Where:

- A is an array.
- i_1 through i_n are loop index variables.
- L_1 through L_n and U_1 through U_n are affine expressions of the loop index variables above them.
- F is a (side-effect free) function.
- f is an affine expression of loop variables.
- $\overrightarrow{i}$ is a vector of the loop variables i_1 through i_n.
- $\vec{d_1}$ through $\vec{d_r}$ are constant dependence vectors. It is assumed that for all k 1..r $\vec{d_k} \in \mathbb{Z}^n$. It is also assumed that $r \geq n$.

However, the authors do claim that the techniques they describe can be broadened to work with multiple array variables appearing in multiple statements in a loop nest, and that these statements may be guarded by conditionals. From this modeled sequential computation, Tang and Xue proceed to describe how to generate tiled code when given a tile-size vector (an n-dimensional vector of positive integers) that specifies the size of the tiles.

The authors proceed to describe how a computation distribution can be calculated according to a tile allocation function. This function cyclically assigns tiles to processors along m dimensions. It is assumed that processors are modeled as an m-dimensional mesh where m is less-than or equal to the dimensionality of the modeled iteration space.

A novel aspect of Tang and Xue's work is that data is distributed according to a computer owns rule. In some systems computation distribution follows from data-writes according to an owners-compute rule. That is computation is distributed so processors only modify data that is local to them. Tang and Xue's work is done in an opposite fashion: a computation distribution is first applied and a data-distribution follows to ensure that data is allocated to the processor that modifies it.

There are quite a few limitations in this work. Most notably: the restriction that dependences and loop-boundaries be affine expressions, the restriction that tiles be rectangular, the restriction that processors must fit an m-dimensional mesh, and the restriction that the only tile allocation function is one that cyclically allocate tiles to processors. Portability is somewhat addressed in that program can be regenerated using smaller or larger tile sizes to suite the architecture they are ported to. However, to best utilize architectures that have complex memory hierarchies multi-level (or hierarchical tiling [9]) may be necessary.

3.6 Other Models

Although we evaluate five programming models in this paper, there exist several others. The current dominant model of High Performance Computing is the Message Passing Interface (MPI). MPI includes constructs to enable programmers to conduct point-to-point and collective communication on distributed architectures. Due to the explicit nature of communication in the MPI model, some have criticized it as being low level and complex – referring to it as the assembly language of parallel programming [15]. Nevertheless, MPI is one of the most portable and successful parallel programming models to-date [17].

Communication is simplified in parallel programming for shared memory architectures. A popular shared-memory programming model is OpenMP [11]. OpenMP includes a set of compiler directives to mark sections of code for parallel execution. OpenMP's approach of using compiler directives enables it to build ontop of existing languages. There are implementations of OpenMP that operate with C, C++, and Fortran compilers [2]. Although most implementations of OpenMP are limited to working on shared-memory architectures, some researchers have looked into extending it to work on clusters [32, 20].

Other languages, such as Co-Array Fortran [27] attempt to simplify parallel programming by making communication one-sided. In a one-sided communication model a process can access off-processor data without explicitly coding message exchanges on the sending and receiving processes.

Other languages for parallel programming include the DARPA HPCS languages [25]: X10 [14], Fortress [6], and Chapel [10]; the Partitioned Global Address Space (PGAS) Languages: Titanium [36] and Unified Parallel C (UPC); and languages for general purpose GPUs: CUDA [26] and OpenCL [1].

4 Discussion

The models we analyze address the issue of computation distribution in different ways. However, computation distribution is not an isolated parallel programming issue – the other two factors we mentioned in the introduction (data distribution and communication) influ-

ence how computation should be distributed. Likewise, the distribution of computation can have an influence on these factors. In this section we give a broader view of how computation distribution fits into the development of parallel programs and discuss what makes one programming model more amenable to working with existing code base than another.

4.1 Task and data parallelism

In some programming models the distribution of computation is implicitly determined from an explicitly defined data-decomposition. For example, Id Nouveau's array mappings not only explicitly distribute data and computation, but are also used by the Id Nouveau compiler to determine how to distribute computation according to an owner-computes rule. On the other hand, in Xue and Tang's tiling model, the model is first distributed into tiles, and then a distribution of data is determined using a computer-owns rule.

Programming models can be classified according to the emphasis they place on data versus computation distribution. Models that expose parallelism by having threads work on partitions of a data set are called data-parallel [19]. On the other hand, programming models that have parallelism specified via tasks are called task-parallel. Models that explicitly expose data distribution and implicitly determine computation distribution often present a data parallel view to the programmer. Similarly, models that explicitly expose computation distribution and implicitly determine data distribution often expose a task parallel view.

Sometimes, models have aspects that are task parallel and others asspects that are data parallel. The map-reduce model is a good example of a model that is both task and data parallel. In map-reduce computation is defined in terms of two types of tasks: a mapping task and a reducer task. However, the mapping tasks themselves are data-parallel as they operate on a partitioning of some input data-set.

Chapel's iterators could also work in both a task-parallel or data-parallel context depending on what's being iterated over. For example, if values in an array are being iterated over and the loop-body operates on the iterates values then the iterator is being used in a data-parallel manner. If instead, the iterator is being used to iterate over nodes in a task-graph then the iterator is being used in a task-parallel manner.

4.2 Impacts

Computation distribution, data distribution, and communication do not work in isolation. Choices in each of these factors have impacts on the others. Some complicated choices include: how to coordination computation, when and how to replicate of data, when and how to replicate computation, and when and how to reuse data. To gain a better understanding of how these factors impact each other, we explicitly list a few of these impacts.

Data distribution and communication impact each other in the following manners:

- Communication is often necessary to initially distribute data from an original source to the location where it is specified to be distributed to.
- Communication is required for a processor to retrieve a piece of data that is not in its local memory.
- Data may be replicated to avoid excess communication from processors that otherwise wouldn't have local copies of the data. This duplicates data to avoid communication.

Computation distribution and communication impact each other in the following manners:

- Communication is required to synchronize computation running on separate processing elements.
- If two processing elements require the result of some computation, that computation might independently be run on the two processing elements to avoid having a communication of a result from one processing element to the other. This duplicates computation to avoid communication.

Data distribution and computation distribution impact each other in the following manners:

- The re-use of memory requires that computation be scheduled so it will not be dependent on an overwritten values; this is to say computation scheduling must be cognizant of anti-dependences.
- As discussed in Section 4.1, some models can have computation distribution implicitly follow from data-distribution and others can have computation-distribution follow from data-distribution.

4.3 Separation and program portability

Porting an application requires that its architecture dependent portions be modified to operate with the new architecture. If all architecture dependent portions are encapsulated inside of a library, runtime, or compiler, and there exists a copy of that library, runtime, or compiler for the targeted architecture, then programmer's will not have to rewrite code. If this is not the case then code will have to be rewritten. For example, there exist

libraries for the map-reduce model that operate on both multicore machines and clusters, as such, retargetting a map-reduce from one to the other will not require a rewrite.

Some approaches, such as Chapel's Iterators and Sequoia's mapping specification files address program portability by placing architecture dependent parts of an application in an orthogonal specification from the algorithm. Porting the application to a new architecture will require changes, but only in this cleanly separated part.

4.4 Applying models to existing applications

The ease of restructuring an existing code-base to function in a parallel programming model is dependent on how closely the structure of that computation matches the structure the model assumes. For example, many existing computations are written in terms of loop nests. As such, restructuring an existing loop nest to work with Chapel iterators would not be difficult. However, Chapel, like Id-Neouveau and Sequoia, requires that computation be written in the model's own programming languages. As such, porting an application written in a different language will require a transliteration at best and a rewrite at worst.

Some programming models, such as preprocessing tools like OpenMP [11], or libraries such as Hadoop [3], or STAPL [7] (an STL like library for parallel programs) operate on-top of popular language like C or Fortran. When this is the case not all algorithm code will have to be rewritten if that code is in the model's base language. However, as previously stated, how much of a rewrite is needed is proportional to how closely the structure of the existing computation matches the model's assumed structure of computation.

5 Conclusions

In this paper we discuss four interpretive criteria that can be used to describe and compare how programming models address computation distribution. We also discuss several evaluative criteria that can be used to judge and compare these models. The four interpretive criteria we describe are: 1) the specification of computation distribution across processing elements, 2) the specification of computation distribution across time, 3) the assumed structure of computation, and 4) the realization of computation distribution. We apply these criteria to gain understanding of five programming models; our results are summarized in Appendix A. However, computation distribution does not work in isolation. Due to the fact that communication and data distribution also impact computation distribution, a deep understanding of how a model addresses these two other factors is necessary for a deep understanding of how computation distribution fits into the larger picture. We do not explicitly evaluate data-distribution and communication in this paper, however, in Section 4 we do discuss how computation distribution is impacted by these two other factors.

5.1 Where to go from here

Throughout Section 3 we survey the strengths and limitations of several programming models. Future work could, of course, focus on overcoming these limitations. A greater insight into these limitations could be reached by expanding this survey to look at the two other factors of parallel programming: data distribution, and communication. New models could also be developed based off of what is learned from an expanded survey.

Another type of survey that would be useful would be to examine the common types of parallel computations that exist, as well as the common techniques that are employed to structure and implement these computations. Some groups [22, 4] are studying just that, formalizing these techniques as design patterns. In [24] Krieger, Stone, and Strout study how programming models address implementation details that occur when realizing some of these parallel programming patterns. It would be useful to extend this work to evaluate these models in terms of the criteria outlined in this paper.

Acknowledgements

The author thanks Drs. Daniel Massey, Shrideep Pallickara, and Michelle Strout for their service on the research examination committee. The author also thanks Chistopher Krieger for making his research examination available [23]. The author read [23] for insight on the structure and writing style of a successful examination. Additionally, the author thanks Jon Roelofs and Alan Lamielle for their editorial assistance.

References

[1] OpenCL overview. `http://www.khronos.org/opencl/`.

[2] OpenMP website - list of compilers. http://openmp.org/wp/openmp-compilers/.

[3] Apache hadoop website. `http://hadoop.apache.org/`, April 2010.

[4] The our pattern language (opl) wiki. `http://parlab.eecs.berkeley.edu/wiki/patterns/patterns`, April 2010.

[5] T. Agerwala and S. Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, 25(3):58–69, 2005.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The fortress language specification. `http://research.sun.com/projects/plrg/`.

[7] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel programming library for c++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, August 2001.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[9] L. Carter, J. Ferrante, and S. F. Hummel. Efficient parallelism via hierarchical tiling. In *Proc. of SIAM Conference on Parallel Processing for Scientific Computing*, pages 680–685, 1995.

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

[11] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] U. Drepper. What every programmer should know about memory. *Red Hat, Inc.*, 2007. `http://people.redhat.com/drepper/cpumemory.pdf`.

[14] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[15] N. Fang and H. Burkhart. Structured parallel programming using mpi. In *HPCN Europe 1996: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 840–847, London, UK, 1996. Springer-Verlag.

[16] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[17] W. Gropp. Learning from the success of mpi. In *HiPC '01: Proceedings of the 8th International Conference on High Performance Computing*, pages 81–94, London, UK, 2001. Springer-Verlag.

[18] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[19] W. D. Hillis and G. L. S. Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[20] J. Hoeflinger. Extending OpenMP to clusters. White paper, Intel of Cape Town, 2006.

[21] M. Joyner, B. L. Chamberlain, and S. J. Deitz. Iterators in chapel. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, April 2006.

[22] K. Keutzer and T. Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, June 2009.

[23] C. D. Krieger. A critical analysis of abstract data type-centric parallelization strategies. *PhD Qualifier Research Exam, Computer Science Department, Colorado State University*, 2009.

[24] C. D. Krieger, A. Stone, and M. M. Strout. Mechanisms that separate algorithms from implementations for parallel patterns. In *ParaPLoP Workshop on Parallel Programming Patterns*, March 2010.

[25] E. L. Lusk and K. A. Yelick. Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters*, 17(1):89–102, 2007.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[27] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[28] S. Pallickara, J. Ekanayake, and G. Fox. An overview of the granules runtime for cloud computing. In *Proceedings of the IEEE International Conference on e-Science*, December 2008.

[29] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support for map-reduce. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2009.

[30] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference. Digest of Technical Papers. ISSCC.*, San Francisco, CA, USA, 2005. IEEE International.

[31] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–80, New York, NY, USA, 1989. ACM.

[32] Y. suk Kee. ParADE: An OpenMP programming environment for SMP cluster systems. In *Proceedings of ACM/IEEE Supercomputing (SC'03)*, pages 12–15, 2003.

[33] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[34] P. Tang and J. Xue. Generating efficient tiled code for distributed memory machines. *Parallel Computing*, 26(11):1369 – 1410, 2000.

[35] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.

[36] 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*, pages 10–11, 1998.

A Summary of programming models survey

In this appendix we include a table for each of the five surveyed programming models. These tables describe the surveyed models in terms of the interpretive criteria outlined in Section 2. We also evaluate the impact the model's approach to computation distribution has on portability.

Table 1: Interpretation of Chapel iterator model

Specification of computation distribution across processing elements	Explicitly through `on` statements, can also be done in a more implicit manner by following data-distributions (specified with Chapel's distribution mechanism)
Specification of computation distribution across time	Explicitly stated in iterators, this specification is separated from the algorithm code (the specification of what to do on each iteration).
Structure of computation	A single program is written in the imperative Chapel programming language. Iterators are used to specify the scheduling of loops. On each iteration, iterators return some value (a piece of data or an index) that a loop-body operates on.
Realization of distributed computation	Chapel compiler produces code to realize the specified loop.
Impact on program portability	Different computation schedules may be more/less applicable to different platforms. It's possible to switch the scheduling simply by switching the iterator a loop uses.

Table 2: Interpretation of Sequoia

Specification of computation distribution across processing elements	The mapping of computation to processing elements occurs implicitly in the Sequoia model. The runtime system is responsible for realizing this mapping. Users specify how to decompose data for each level of the memory hiearchy.
Specification of computation distribution across time	Scheduling in Sequoia is explicitly addressed by its mapping primitives and calls to subtasks. The mapping details are separated from algorithm code.
Structure of computation	Computation is specified by two variants: an inner variant that recursively splits computation into smaller tasks, and a leaf variant that performs the actual computation.
Realization of distributed computation	The runtime system follows mapping specifications and executes the inner variant of code to partition computation, it is responsible for executing the leaf variant of the partitioned computation on the appropriate processing element.
Impact on program portability	Once written in the Sequoia model, the memory hierarchies of new systems can be addressed by redefining the values in a mapping specification file.

Table 3: Interpretation of Map-Reduce in Granules

Specification of computation distribution across processing elements	The runtime system is responsible for this distribution, no such specification is required. However, users do need to specify how to partition data-sets, and users can explicitly define the arragment of mapping and reducing tasks (which the Granules system will distribute across processing elements).
Specification of computation distribution across time	Users specify a scheduling strategy defined through parameters that state how many times a task should be executed, whether the task should be scheduled as data is available, and whether the computation should be scheduled on a periodic basis. These parameters can be composed and qualified conditionally.
Structure of computation	In the map-reduce model computation is split into a mapper, which operates on data and outputs an intermediate result, and a reducer, which aggregates the intermediate results from the mappers.
Realization of distributed computation	Granule's runtime system is responsible for discovering computation resources and distributing computation to these resources. The runtime system is also responsible for governing the lifetime of an application and scheduling computation as outlined in the specified scheduling strategy.
Impact on program portability	Cloud computing is centered around the notion of scaling up and down as new resources are needed or available. In this sense cloud computing applications dynamically retarget themselves as resources become available.

Table 4: Interpretation of Id Nouveau

Specification of computation distribution across processing elements	Computation distribution is not specified in this model, however data is specified with a domain decomposition. The distribution of computation follows from the decomposition of data and the data-dependencies inherent in the computation.
Specification of computation distribution across time	Programs are written in a serial manner with a serial schedule (often specified with for loops). The compiler is responsible for the scheduling of computation of a parallel version, however whatever schedule is applied it is guaranteed to satisfy all data-dependencies.
Structure of computation	Programs are written as a serial program with data having an applied domain-decomposition, computation distributions follows from the domain-decomposition. Arrays have write-once semantics.
Realization of distributed computation	The Id-Nouveau compiler analyzes the serial program to produce a parallel variant of the program for each distributed machine. The analysis involves "compile-time resolution" and several optimizations.
Impact on program portability	Compilers for different machines could be tuned for other machines. The machine model is fairly general.

Table 5: Interpretation of Tang and Xue's model for tiled code generation

Specification of computation distribution across processing elements	The initial model does not have such specifications.
Specification of computation distribution across time	The initial model has computation scheduled by a series of nested loops. A tiled model of the computation can be calculated that maps iteration points uniquely into tiles and models the dependencies that exist between tiles.
Structure of computation	Computation is specified as an array assignment nested inside a series of loops. The loop-boundries must be affine expressions of proceeding loop indices. The assigned value is computed from a function that only references other array values through affine expressions.
Realization of distributed computation	The system generates message-passing SPMD code that tiles the modeled iteration space. The techniques Tang and Xue discuss can generate code that distributes these tiles across processors in a cyclic manner across m-dimensions. It is assumed that the processors are arranged as an m-dimensional mesh.
Impact on program portability	Different architectures may operate best with different tile sizes. Recompiling a program to have different sized tiles is a matter of changing a tile-size vector within the model and recompiling under the new specification.