Chapter 1

INTRODUCTION TO PARALLEL PROGRAMMING

The past few decades have seen large fluctuations in the perceived value of parallel computing. At times, parallel computation has optimistically been viewed as the solution to all of our computational limitations. At other times, many have argued that it is a waste of effort given the rate at which processor speeds and memory prices continue to improve. Perceptions continue to vacillate between these two extremes due to a number of factors, among them: the continual changes in the "hot" problems being solved, the programming environments available to users, the supercomputing market, the vendors involved in building these supercomputers, and the academic community's focus at any given point and time. The result is a somewhat muddied picture from which it is difficult to objectively judge the value and promise of parallel computing.

In spite of the rapid advances in sequential computing technology, the promise of parallel computing is the same now as it was at its inception. Namely, if users can buy fast sequential computers with gigabytes of memory, imagine how much faster their programs could run if p of these machines were working in cooperation! Or, imagine how much larger a problem they could solve if the memories of p of these machines were used cooperatively!

The challenges to realizing this potential can be grouped into two main problems: the hardware problem and the software problem. The former asks, "how do I build a parallel machine that will allow these p processors and memories to cooperate efficiently?" The software problem asks, "given such a platform, how do I express my computation such that it will utilize these p processors and memories effectively?"

In recent years, there has been a growing awareness that while the parallel community can build machines that are reasonably efficient and/or cheap, most programmers and scientists are incapable of programming them effectively. Moreover, even the best parallel programmers cannot do so without significant effort. The implication is that the software problem is currently lacking in satisfactory solutions. This dissertation focuses on one approach designed to solve that problem.

In particular, this work describes an effort to improve a programmer's ability to utilize parallel computers effectively using the ZPL parallel programming language. ZPL is a language whose parallelism stems from operations applied to its arrays' elements. ZPL derives from the description of Orca C in Calvin Lin's dissertation of 1992 [Lin92]. Since that time, Orca C has evolved to the point that it is hardly recognizable, although the foundational ideas have remained intact. ZPL has proven to be successful in that it allows parallel programs to be written at a high level, without sacrificing portability or performance. This dissertation will also describe aspects of Advanced ZPL (A-ZPL), ZPL's successor language which is currently under development.

One of the fundamental concepts that was introduced to Orca C during ZPL's inception was the concept of the *region*. A region is simply a user-specified set of indices, a concept which may seem trivially uninteresting at first glance. However, the use of regions in ZPL has had a pervasive effect on the language's appearance, semantics, compilation, and runtime management, resulting in much of ZPL's success. This dissertation defines the region in greater depth and documents its role in defining and implementing the ZPL language.

This dissertation's study of regions begins in the next chapter. The rest of this chapter provides a general overview of parallel programming, summarizing the challenges inherent in writing parallel programs, the techniques that can be used to create them, and the metrics used to evaluate these techniques. The next section begins by providing a rough overview of parallel architectures.

1.1 Parallel Architectures

1.1.1 Parallel Architecture Classifications

This dissertation categorizes parallel platforms as being one of three rough types: *distributed memory*, *shared memory*, or *shared address space*. This taxonomy is somewhat coarse given the wide variety of parallel architectures that have been developed, but it provides a useful characterization of current architectures for the purposes of this dissertation.

Distributed memory machines are considered to be those in which each processor has a local memory with its own address space. A processor's memory cannot be accessed directly by another processor, requiring both processors to be involved when communicating values from one memory to another. Examples of distributed memory machines include commodity Linux clusters.

Shared memory machines are those in which a single address space and global memory are shared between multiple processors. Each processor owns a local cache, and its values are kept coherent with the global memory by the operating system. Data can be exchanged between processors simply by placing the values, or pointers to values, in a predefined location and synchronizing appropriately. Examples of shared memory machines include the SGI Origin series and the Sun Enterprise.

Shared address space architectures are those in which each processor has its own local memory, but a single shared address space is mapped across the distinct memories. Such architectures allow a processor to access the memories of other processors without their direct involvement, but they differ from shared memory machines in that there is no implicit caching of values located on remote machines. The primary example of a shared address machine is Cray's T3D/T3E line.

Many modern machines are also built using a combination of these technologies in a hierarchical fashion, known as a *cluster*. Most clusters consist of a number of shared memory machines connected by a network, resulting in a hybrid of shared and distributed memory characteristics. IBM's large-scale SP machines are an example of this design.

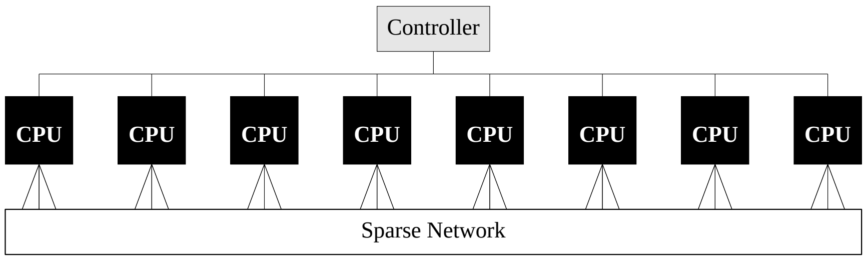

Figure 1.1: The Candidate Type Architecture (CTA)

1.1.2 The CTA Machine Model

ZPL supports compilation and execution on these diverse architectures by describing them using a single machine model known as the *Candidate Type Architecture* (CTA) [Sny86]. The CTA is a reasonably vague model, and deliberately so. It characterizes parallel machines as a group of von Neumann processors, connected by a sparse network of unspecified topology. Each processor has a local memory that it can access at unit cost. Processors can also access other processors' values at a cost significantly higher than unit cost by communicating over the network. The CTA also specifies a controller used for global communications and synchronization, though that will not be of concern in this discussion. See Figure 1.1 for a simple diagram of the CTA.

Why use such an abstract model? The reason is that parallel machines vary so widely in design that it is difficult to develop a more specific model that describes them all. The CTA successfully abstracts the vast majority of parallel machines by emphasizing the importance of locality and the relatively high cost of interprocessor communication. This is in direct contrast to the overly idealized PRAM [FW78] model or the extremely parameterized LogP model [CKP$^+$93], neither of which form a useful foundation for a compiler concerned with portable performance. For more details on the CTA, please refer to the literature [Sny86, Sny95, Lin92].

1.2 Challenges to Parallel Programming

Writing parallel programs is strictly more difficult than writing sequential ones. In sequential programming, the programmer must design an algorithm and then express it to the computer in some manner that is correct, clear, and efficient to execute. Parallel programming involves these same issues, but also adds a number of additional challenges that complicate development and have no counterpart in the sequential realm. These challenges include: finding and expressing concurrency, managing data distributions, managing interprocessor communication, balancing the computational load, and simply implementing the parallel algorithm correctly. This section considers each of these challenges in turn.

1.2.1 Concurrency

Concurrency is crucial if a parallel computer's resources are to be used effectively. If an algorithm cannot be divided into groups of operations that can execute concurrently, performance improvements due to parallelism cannot be achieved, and any processors after the first will be of limited use in accelerating the algorithm. To a large extent, different problems inherently have differing amounts of concurrency. For most problems, developing an algorithm that achieves its maximal concurrency requires a combination of cleverness and experience from the programmer.

As motivating examples, consider matrix addition and matrix multiplication. Mathematically, we might express these operations as follows:

Matrix addition: Given matrices A and B $(m \times n)$,
$A + B = C\ (m \times n),$ where $C_{i,j} = A_{i,j} + B_{i,j}.$

Matrix multiplication: Given matrices A $(m \times n)$ and B $(n \times o)$,
$A \times B = C\ (m \times o),$ where $C_{i,k} = \sum_{j=1}^{n} A_{i,j} \cdot B_{j,k}$

Consider the component operations that are required to implement these definitions. Matrix addition requires $m \cdot n$ pairwise sums to be computed. Matrix multiplication requires the evaluation of $m \cdot n \cdot o$ pairwise products and $\Omega(m \cdot \log n \cdot o)$ pairwise sums. In considering the parallel implementation of either of these algorithms, programmers must ask themselves, "can all of the component operations be performed simultaneously?" Looking at matrix addition, a wise parallel programmer would conclude that they can be computed concurrently—each sum is independent from the others, and therefore they can all be computed simultaneously. For matrix multiplication, the programmer would similarly conclude that all of the products could be computed simultaneously. However, each sum is dependent on values obtained from previous computations, and therefore they cannot be computed completely in parallel with the products or one another.

As a result of this analysis, a programmer might conclude that matrix addition is inherently more concurrent than matrix multiplication. As a second observation, the programmer should note that for matrices of a given size, matrix multiplication tends to involve more operations than matrix addition.

If the programmer is designing an algorithm to run on p processors where $p \ll m, n, o$, a related question is "are there better and worse ways to divide the component operations into p distinct sets?" It seems likely that there are, although the relevant factors may not be immediately obvious. The rest of this section describe some of the most important ones.

1.2.2 Data Distribution

Another challenge in parallel programming is the distribution of a problem's data. Most conventional parallel computers have a notion of *data locality*. This implies that some data will be stored in memory that is "closer" to a particular processor and can therefore be accessed much more quickly. Data locality may occur due to each processor having its own distinct local memory—as in a distributed memory machine—or due to processor-specific caches as in a shared memory system.

Due to the impact of data locality, a parallel programmer must pay attention to where data is stored in relation to the processors that will be accessing it. The more local the values are, the quicker the processor will be able to access them and complete its work. It should be evident that distributing work and distributing data are tightly coupled, and that an optimal design will consider both aspects together.

For example, assuming that the $m \times n$ sums in a matrix addition have been divided between a set of p processors, it would be ideal if the values of A, B, and C were distributed in a corresponding manner so that each processor's sums could be computed using local values. Since there is a one-to-one correspondence between sums and matrix values, this can easily be achieved. For example, each processor p_k could be assigned matrix values $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$, $\forall i \equiv k \pmod{p}, \forall j \in 1 \ldots n$.

Similarly, the implementor of a parallel matrix multiplication algorithm would like to distribute the matrix values, sums, and products among the processors such that each node only needs to access local data. Unfortunately, due to the data interactions inherently required by matrix multiplication, this turns out to be possible only when matrix values are explicitly replicated on multiple processors. While this replication may be an option for certain applications, it runs counter to the general goal of running problems that are p times bigger than their sequential counterparts. Such algorithms that rely on replication to avoid communication are not considered *scalable*. Furthermore, replication does not solve the problem since matrix products are often used in subsequent multiplications and would therefore require communication to replicate their values across the processor set after being computed.

To create a scalable matrix multiplication algorithm, there is no choice but to transfer data values between the local memories of the processors. Unfortunately, the reality is that most interesting parallel algorithms require such communication, making it the next parallel programming challenge.

1.2.3 Communication

Assuming that all the data that a processor needs to access cannot be made exclusively local to that processor, some form of data transfer must be used to move remote values to a processor's local memory or cache. On distributed memory machines, this communication typically takes the form of explicit calls to a library designed to move values from one processor's memory to another. For shared memory machines, communication involves cache coherence protocols to ensure that a processor's locally cached values are kept consistent with the main memory. In either case, communication constitutes work that is time-consuming and which was not present in the sequential implementation. Therefore, communication overheads must be minimized in order to maximize the benefits of parallelism.

Over time, a number of algorithms have been developed for parallel matrix multiplication, each of which has unique concurrency, data distribution, and communication characteristics. A few of these algorithms will be introduced and analyzed during the course of the next few chapters. For now, we return to our final parallel computing challenges.

1.2.4 Load Balancing

The execution time of a parallel algorithm on a given processor is determined by the time required to perform its portion of the computation plus the overhead of any time spent performing communication or waiting for remote data values to arrive. The execution time of the algorithm as a whole is determined by the longest execution time of any of the processors. For this reason, it is desirable to balance the total computation and communication between processors in such a way that the maximum per-processor execution time is minimized. This is referred to as *load balancing*, since the conventional wisdom is that dividing work between the processors as evenly as possible will minimize idle time on each processor, thereby reducing the total execution time.

Load balancing a matrix addition algorithm is fairly simple due to the fact that it can be implemented without communication. The key is simply to give each processor approximately the same number of matrix values. Similarly, matrix multiplication algorithms are typically load balanced by dividing the elements of C among the processors as evenly as possible and trying to minimize the communication overheads required to bring remote A and B values into the processors' local memories.

1.2.5 Implementation and Debugging

Once all of the parallel design decisions above have been made, the nontrivial matter of implementing and debugging the parallel program still remains. Programmers often implement parallel algorithms by creating a single executable that will execute on each processor. The program is designed to perform different computations and communications based on the processor's unique ID to ensure that the work is divided between instances of the executable. This is referred to as the *Single Program, Multiple Data* (SPMD) model, and its attractiveness stems from the fact that only one program must be written (albeit a nontrivial one). The alternative is to use the *Multiple Program, Multiple Data* (MPMD) model, in which several cooperating programs are created for execution on the processor set. In either case, the executables must be written to cooperatively perform the computation while managing data locality and communication. They must also maintain a reasonably balanced load across the processor set. It should be clear that implementing such a program will inherently require greater programmer effort than writing the equivalent sequential program.

As with any program, bugs are likely to creep into the implementation, and the effects of these bugs can be disastrous. A simple off-by-one error can cause data to be exchanged with the wrong processor, or for a program to deadlock, waiting for a message that was never sent. Incorrect synchronization can result in data values being accessed prematurely, or for race conditions to occur. Bugs related to parallel issues can be nondeterministic and show up infrequently. Or, they may occur only when using large processor sets, forcing the programmer to sift through a large number of execution contexts to determine the cause. In short, parallel debugging involves issues not present in the sequential world, and it can often be a huge headache.

1.2.6 Summary

Computing effectively with a single processor is a challenging task. The programmer must be concerned with creating programs that perform correctly and well. Computing with multiple processors involves the same effort, yet adds a number of new challenges related to the cooperation of multiple processors. None of these new factors are trivial, giving a good indication of why programmers and scientists find parallel computing so challenging.

The design of the ZPL language strives to relieve programmers from most of the burdens of correctly implementing a parallel program. Yet, rather than making them blind to these details, ZPL's regions expose the crucial parallel issues of concurrency, data distribution, communication, and load balancing to programmers, should they care to reason about such issues. These benefits of regions will be described in subsequent chapters. For now, we shift our attention to the spectrum of techniques that one might consider when approaching the task of parallel programming.

1.3 Approaches to Parallel Programming

Techniques for programming parallel computers can be divided into three rough categories: parallelizing compilers, parallel programming languages, and parallel libraries. This section considers each approach in turn.

1.3.1 Parallelizing Compilers

The concept of a parallelizing compiler is an attractive one. The idea is that programmers will write their programs using a traditional language such as C or Fortran, and the compiler will be responsible for managing the parallel programming challenges described in the previous section. Such a tool is ideal because it allows programmers to express code in a familiar, traditional manner, leaving the challenges related to parallelism to the compiler. Examples of parallelizing compilers include SUIF, KAP, and the Cray MTA compiler [HAA$^+$96, KLS94, Ter99].

Listing 1.1: Sequential C Matrix Multiplication

```
for (i=0; i<m; i++) {
  for (k=0; k<o; k++) {
    C[i][k] = 0;
  }
}
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<o; k++) {
      C[i][k] += A[i][j] * B[j][k];
    }
  }
}
```

The primary challenge to automatic parallelization is that converting sequential programs to parallel ones is an entirely non-trivial task. As motivation, let us return to the example of matrix multiplication. Written in C, a sequential version of this computation might appear as in Listing 1.1.

Well-designed parallel implementations of matrix multiplication tend to appear very different than this sequential algorithm, in order to maximize data locality and minimize communication. For example, one of the most scalable algorithms, the SUMMA algorithm [vdGW95], bears little resemblance to the sequential triply nested loop. SUMMA consists of n iterations. On the i^{th} iteration, `A`'s i^{th} column is broadcast across the processor columns and `B`'s i^{th} row is broadcast across processor rows. Each processor then calculates the cross product of its local portion of these values, producing the i^{th} term in the sum for each of `C`'s elements. Figure 1.2 shows an illustration of the SUMMA algorithm.

The point here is that effective parallel algorithms often differ significantly from their sequential counterparts. While having an effective parallel compiler would be a godsend, expecting a compiler to automatically understand an arbitrary sequential algorithm well enough to create an efficient parallel equivalent seems a bit naive. The continuing lack of such a compiler serves as evidence to reinforce this claim.

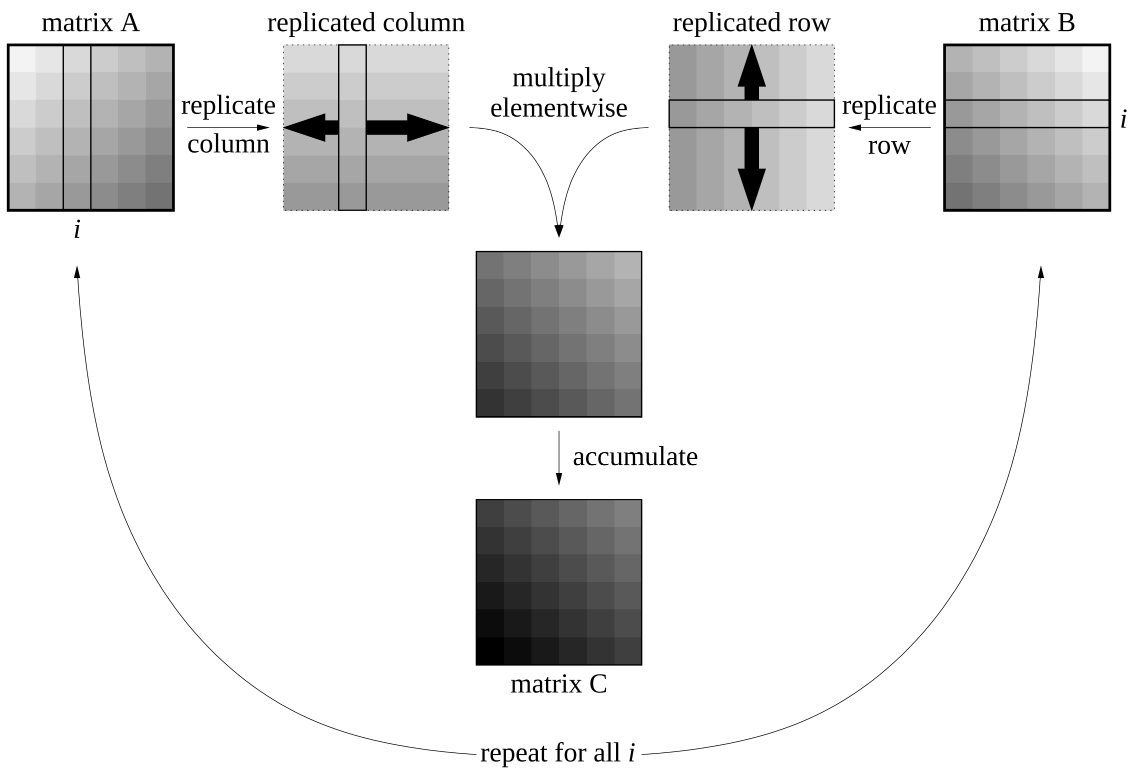

Figure 1.2: The SUMMA Algorithm For Matrix Multiplication

Many parallelizing compilers, including those named above, take an intermediate approach in which programmers add directives to their codes to provide the compiler with information to aid it in the task of parallelizing the code. The more of these that need to be relied upon, the more this approach resembles a new programming language rather than a parallelizing compiler, so further discussion of this approach is deferred to the next section.

1.3.2 Parallel Programming Languages

A second approach to parallel programming is the design and implementation of parallel programming languages. These are languages designed to support parallel computing better than sequential languages, though many of them are based on traditional languages in the hope that existing code bases can be reused. This dissertation categorizes parallel languages as being either *global-view* or *local-view*.

Global-view Languages

Global-view languages are those in which the programmer specifies the behavior of their algorithm as a whole, largely ignoring the fact that multiple processors will be used to implement the program. The compiler is therefore responsible for managing all of the parallel implementation details, including data distribution and communication.

Many global-view languages are rather unique, providing language-level concepts that are tailored specifically for parallel computing. The ZPL language and its regions form one such example. Other global-view languages include the directive-based variations of traditional programming languages used by parallelizing compilers, since the annotated sequential programs are global descriptions of the algorithm with no reference to individual processors. As a simple example of a directive-based global-view language, consider the pseudocode implementation of the SUMMA algorithm in Listing 1.2. This is essentially a sequential description of the SUMMA algorithm with some comments (directives) that indicate how each array should be distributed between processors.

Listing 1.2: Pseudo-Code for SUMMA Using a Global View

```
double A[m][n];
double B[n][o];
double C[m][o];
double ColA[m];
double RowB[o];

// distribute C [block,block]
// align A[:,:]  with C[:,:]
// align B[:,:]  with C[:,:]
// align ColA[:] with C[:,*]
// align RowB[:] with C[*,:]

for (i=0; i<m ; i++) {
  for (k=0; k<o; k++) {
    C[i][k] = 0;
  }
}

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    ColA[i] = A[i][j];
  }
  for (k=0; k<o; k++) {
    RowB[k] = B[j][k];
  }

  for (i=0; i<m ;i++) {
    for (k=0; k<o; k++) {
      C[i][k] += ColA[i] * RowB[k];
    }
  }
}
```

The primary advantage to global-view languages is that they allow the programmer to focus on the algorithm at hand rather than the details of the parallel implementation. For example, in the code above, the programmer writes the loops using the array's global bounds. The task of transforming them into loops that will cause each processor to iterate over its local data is left to the compiler.

This convenience can also be a liability for global-view languages. If a language or compiler does not provide sufficient feedback about how programs will be implemented, knowledgeable programmers may be unable to achieve the parallel implementations that they desire. For example, in the SUMMA code of Listing 1.2, programmers might like to be assured that an efficient broadcast mechanism will be used to implement the assignments to `ColA` and `RowB`, so that the assignment to `C` will be completely local. Whether or not they have such assurance depends on the definition of the global language being used.

Local-view Languages

Local-view languages are those in which the implementor is responsible for specifying the program's behavior on a per-processor basis. Thus, details such as communication, data distribution, and load balancing must be handled explicitly by the programmer. A local-view implementation of the SUMMA algorithm might appear as shown in Listing 1.3.

The chief advantage of local-view languages is that users have complete control over the parallel implementation of their programs, allowing them to implement any parallel algorithm that they can imagine. The drawback to these approaches is that managing the details of a parallel program can become a painstaking venture very quickly. This contrast can be seen even in short programs such as the implementation of SUMMA in Listing 1.3, especially considering that the implementations of its `Broadcast...()`, `IOwn...()`, and `GlobToLoc...()` routines have been omitted for brevity. The magnitude of these details are such that they tend to make programs written in local-view languages much more difficult to maintain and debug.

Listing 1.3: Pseudo-Code for SUMMA Using a Local View

```
int m_loc = m/proc_rows;
int o_loc = o/proc_cols;
int n_loc_col = n/proc_cols;
int n_loc_row = n/proc_rows;

double A[m_loc][n_loc_col];
double B[n_loc_row][o_loc];
double C[m_loc][o_loc];
double ColA[m_loc];
double RowB[o_loc];

for (i=0; i<m_loc ; i++) {
  for (k=0; k<o_loc; k++) {
    C[i][k] = 0;
  }
}

for (j=0; j<n; j++) {
  if (IOwnCol(j)) {
    BroadcastColSend(A,GlobToLocCol(j));
    for (i=0; i<m_loc; i++) {
      ColA[i] = A[i][j];
    }
  } else {
    BroadcastColRecv(ColA);
  }
  if (IOwnRow(j)) {
    BroadcastRowSend(B,GlobToLocRow(j));
    for (k=0; k<o_loc; k++) {
      RowB[k] = B[j][k];
    }
  } else {
    BroadcastRowRecv(RowB);
  }

  for (i=0; i<m_loc ;i++) {
    for (k=0; k<o_loc; k++) {
      C[i][k] += ColA[i] * RowB[k];
    }
  }
}
```

1.3.3 Parallel Libraries

Parallel libraries are the third approach to parallel computing considered here. These are simply libraries designed to ease the task of utilizing a parallel computer. Once again, we categorize these as global-view or local-view approaches.

Global-view Libraries

Global-view libraries, like their language counterparts, are those in which the programmer is largely kept blissfully unaware of the fact that multiple processors are involved. As a result, the vast majority of these libraries tend to support high-level numerical operations such as matrix multiplications or solving linear equations. The number of these libraries is overwhelming, but a few notable examples include the NAG Parallel Library, ScaLAPACK, and PLAPACK [NAG00, BCC$^+$97, vdG97].

The advantage to using a global-view library is that the supported routines are typically well-tuned to take full advantage of a parallel machine's processing power. To achieve similar performance using a parallel language tends to require more effort than most programmers are willing to make.

The disadvantages to global-view libraries are standard ones for any library-based approach to computation. Libraries support a fixed interface, limiting their generality as compared to programming languages. Libraries can either be small and extremely special-purpose or they can be *wide*, either in terms of the number of routines exported or the number of parameters passed to each routine [GL00]. For these reasons, libraries are a useful tool, but often not as satisfying for expressing general computation as a programming language.

Local-view Libraries

Like languages, libraries may also be local-view. For our purposes, local-view libraries are those that aid in the support of processor-level operations such as communication between processors. Local-view libraries can be evaluated much like local-view languages: they give the programmer a great deal of explicit low-level control over a parallel machine, but by nature this requires the explicit management of many painstaking details. Notable examples include the MPI and SHMEM libraries [Mes94, BK94].

1.3.4 Summary

This section has described a number of different ways of programming parallel computers. To summarize, general parallelizing compilers seem fairly intractable, leaving languages and libraries as the most attractive alternatives. In each of these approaches, the tradeoff between supporting global- and local-view approaches is often one of high-level clarity versus low-level control. The goal of the ZPL programming language is to take advantage of the clarity offered by a global-view language without sacrificing the programmer's ability to understand the low-level implementation and tune their code accordingly. Further chapters will develop this point and also provide a more comprehensive survey of parallel programming languages and libraries.

1.4 Evaluating Parallel Programs

For any of the parallel programming approaches described in the previous section, there are a number of metrics that can be used to evaluate its effectiveness. This section describes five of the most important metrics that will be used to evaluate parallel programming in this dissertation: performance, clarity, portability, generality, and a programmer's ability to reason about the implementation.

1.4.1 Performance

Performance is typically viewed as the bottom line in parallel computing. Since improved performance is often the primary motivation for using parallel computers, failing to achieve good performance reflects poorly on a language, library, or compiler.

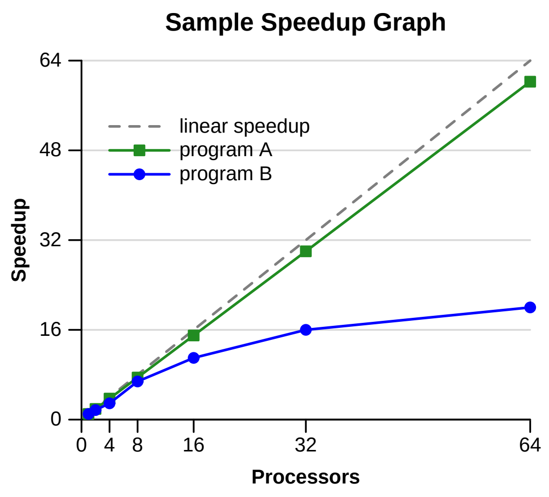

Figure 1.3: A Sample Speedup Graph. The dotted line indicates linear speedup ($speedup_p = p$), which represents ideal parallel performance. The “program A” line represents an algorithm that scales quite well as the processor set size increases. The “program B” line indicates an algorithm that does not scale nearly as well, presumably due to parallel overheads like communication. Note that these numbers are completely fabricated for demonstration purposes.

This dissertation will typically measure performance in terms of *speedup*, defined to be the fastest single-processor execution time (using *any* approach) divided by the execution time on p processors:

$$speedup_p = T_{1_{\min}}/T_p$$

If the original motivating goal of running a program p times faster using p processors is met, then $speedup_p = p$. This is known as *linear speedup*. In practice, this is challenging to achieve since the parallel implementation of most interesting programs requires work beyond that which was required for the sequential algorithm: in particular, communication and synchronization between processors. Thus, the amount of work per processor in a parallel implementation will typically be more than $1/p$ of the work of the sequential algorithm.

On the other hand, note that the parallelization of many algorithms requires allocating approximately $1/p$ of the sequential program's memory on each processor. This causes the working set of each processor to decrease as p increases, allowing it to make better use of the memory hierarchy. This effect can often offset the overhead of communication, making linear, or even *superlinear* speedups possible.

Parallel performance is typically reported using a graph showing speedup versus the number of processors. Figure 1.3 shows a sample graph that displays fictional results for a pair of programs. The speedup of program A resembles a parallel algorithm like matrix addition that requires no communication between processors and therefore achieves nearly linear speedup. In contrast, program B's speedup falls away from the ideal as the number of processors increases, as might occur in a matrix multiplication algorithm that requires communication.

1.4.2 Clarity

For the purposes of this dissertation, the clarity of a parallel program will refer to how clearly it represents the overall algorithm being expressed. For example, given that listings 1.2 and 1.3 both implement the SUMMA algorithm for matrix multiplication, how clear is each representation? Conversely, how much do the details of the parallel implementation interfere with a reader's ability to understand an algorithm?

The importance of clarity is often brushed aside in favor of the all-consuming pursuit of performance. However, this is a mistake that should not be made. Clarity is perhaps the single most important factor that prevents more scientists and programmers from utilizing parallel computers today. Local-view libraries continue to be the predominant approach to parallel programming, yet their syntactic overheads are such that clarity is greatly compromised. This requires programmers to focus most of their attention on making the program work correctly rather than spending time implementing and improving their original algorithm. Ideally, parallel programming approaches should result in clear programs that can be readily understood.

1.4.3 Portability

A program's portability is practically assured in the sequential computing world, primarily due to the universality of C and Fortran compilers. In the parallel world, portability is not as prevalent due to the extreme differences that exist between platforms. Parallel architectures vary widely not only between distinct machines, but also from one generation of a machine to the next. Memory may be organized as a single shared address space, a single distributed address space, or multiple distributed address spaces. Networks may be composed of buses, tori, hypercubes, sparse networks, or hierarchical combinations of these options. Communication paradigms may involve message passing, single-sided data transfers, or synchronization primitives over shared memory.

This multitude of architectural possibilities may be exposed by local-view approaches, making it difficult to implement a program that will run efficiently, if at all, from one machine to the next. Architectural differences also complicate the implementation of global-view compilers and libraries since they must run correctly and efficiently on all current parallel architectures, as well as those that may exist in the future.

Ideally, portability implies that a given program will behave consistently on all machines, regardless of their architectural features.

1.4.4 Generality

Generality simply refers to the ability of a parallel programming approach to express algorithms for varying types of problems. For example, a library which only supports matrix multiplication operations is not very general, and would not be very helpful for writing a parallel quicksort algorithm. Conversely, a global-view functional language might make it simple to write a parallel quicksort algorithm, but difficult to express the SUMMA matrix multiplication algorithm efficiently. Ideally, a parallel programming approach should be as general as possible.

Listing 1.4: Two matrix additions in C. Which one is better?

```
double A[m][n];
double B[m][n];
double C[m][n];

for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    C[i][j] = A[i][j] + B[i][j];
  }
}

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    C[i][j] = A[i][j] + B[i][j];
  }
}
```

1.4.5 Performance Model

This dissertation defines a *performance model* as the means by which programmers understand the implementations of their programs. In this context, the performance model need not be a precise tool, but simply a means of weighing different implementation alternatives against one another.

As an example, C's performance model indicates that the two loop nests in Listing 1.4 may perform differently in spite of the fact that they are semantically equivalent. C specifies that two-dimensional arrays are laid out in row-major order, and the memory models of modern machines indicate that accessing memory sequentially tends to be faster than accessing it in a strided manner. Using this information, a savvy C programmer will always choose to implement matrix addition using the first loop nest.

Note that C does not say how much slower the second loop nest will be. In fact, it does not even guarantee that the second loop nest *will* be slower. An optimizing compiler may reorder the loops to make them equivalent to the first loop nest. Or, hardware prefetching may detect the memory access pattern and successfully hide the memory latency normally associated with strided array accesses. In the presence of these uncertainties, experienced C programmers will recognize that the first loop nest should be no worse than the second. Given the choice between the two approaches, they will choose the first implementation every time.

C's performance model gives the programmer some idea of how C code will be compiled down to a machine's hardware, even if the programmer is unfamiliar with specific details like the machine's assembly language, its cache size, or its number of registers. In the same way, a parallel programmer should have some sense of how their code is being implemented on a parallel machine—for example, how the data and work are distributed between the processors, when communication takes place, what kind of communication it is, *etc*. Note that users of local-view languages and libraries have access to this information, because they specify it manually. Ideally, global-view languages and libraries should also give their users a parallel performance model with which different implementation alternatives can be compared and evaluated.

1.5 This Dissertation

This dissertation was designed to serve many different purposes. Naturally, its most important role is to describe the contributions that make up my doctoral research. With this goal in mind, I have worked to create a document that examines the complete range of effects that regions have had on the ZPL language, from their syntactic benefits to their implementation, and from their parallel implications to their ability to support advanced parallel computations. I also designed this dissertation to serve as documentation for many of my contributions to the ZPL compiler for use by future collaborators in the project. As such, some sections contain low-level implementation details that may not be of interest to those outside the ZPL community. Throughout the process of writing, my unifying concept has been to tell the story of regions as completely and accurately as I could in the time and space available.

In telling such a broad story, some of this dissertation's contributions have been made as a joint effort between myself and other members of the ZPL project—most notably Sung-Eun Choi, Steven Deitz, E Christopher Lewis, Calvin Lin, Ton Ngo, and my advisor, Lawrence Snyder. In describing aspects of the project that were developed as a team, my intent is not to take credit for work that others have been involved in, but rather to make this treatment of regions as complete and seamless as possible.

The novel contributions of this dissertation include:

- A formal description and analysis of the region concept for expressing array computation, including support for replicated and privatized dimensions.

- A parallel interpretation of regions that admits syntax-based evaluation of a program's communication requirements and concurrency.

- The design and implementation of a runtime representation of regions which enables parallel performance that compares favorably with hand-coded parallel programs.

- The design of the Ironman philosophy for supporting efficient paradigm-neutral communications, and an instantiation of the philosophy in the form of a point-to-point data transfer library.

- A means of parameterizing regions that supports the concise and efficient expression of hierarchical index sets and algorithms.

- Region-based support for sparse computation that permits the expression of sparse algorithms using dense syntax, and an implementation that supports general array operations, yet can be optimized to a compact form.

The chapters of this dissertation have a consistent organization. The bulk of each chapter describes its contributions. Most chapters contain an experimental evaluation of their ideas along with a summary of previous work that is related to their contents. Each chapter concludes with a discussion section that addresses the strengths and weaknesses of its contributions, mentions side issues not covered in the chapter proper, and outlines possibilities for future work.

This dissertation is organized as follows. The next three chapters define and analyze the fundamental region concept. First, Chapter 2 describes the role of the region as a syntactic mechanism for sequential array-based programming, using ZPL as its context. Then, Chapter 3 explains the parallel implications of regions, detailing their use in defining ZPL's performance model. The implementation of regions and of ZPL's runtime libraries is covered in Chapter 4. The two chapters that follow each describe an extension to the basic region concept designed to support more advanced parallel algorithms. The notion of a parameterized region is defined in Chapter 5 and its use in implementing multigrid-style computations is detailed. Chapter 6 extends the region to support sparse sets of indices, and demonstrates its effectiveness in a number of sparse benchmarks. Finally, Chapter 7 presents my concluding remarks and summarizes opportunities for future work.