ZPL's WYSIWYG[*] Performance Model[†]

Bradford L. Chamberlain Sung-Eun Choi E Christopher Lewis
Calvin Lin[‡] Lawrence Snyder W. Derrick Weathersby

University of Washington, Seattle, WA 98195-2350 USA
[‡]University of Texas, Austin, TX 78712 USA
zpl-info@cs.washington.edu

Abstract

ZPL is an array language designed for high performance scientific and engineering computations. Unlike other parallel languages, ZPL is founded on a machine model (CTA) distinct from any implementing hardware. The machine model, which abstracts contemporary parallel computers, makes it possible to correlate ZPL programs with machine behavior. Using this association, programmers can know approximately how code will perform on a typical parallel machine, allowing them to make informed decisions between alternative programming solutions.

This paper describes ZPL's syntactic cues to the programmer which convey performance information. The *what-you-see-is-what-you-get* (WYSIWYG) characteristics of ZPL operations are illustrated on four machines: the Cray T3E, IBM SP-2, SGI Power Challenge and Intel Paragon. Additionally, the WYSIWYG performance model is used to evaluate two algorithms for matrix multiplication, one of which is considered to be the most scalable of portable parallel solutions. Experiments show that the performance model correctly predicts the faster solution on all four platforms for a range of problem sizes.

1 Introduction

High-level programming languages offer an expressive and portable means of implementing algorithms. They spare the programmer the burden of coding in assembly language and allow the resulting program to be effortlessly recompiled on different compilers and architectures. Yet without a well-defined performance model that indicates how language constructs are mapped to the target machine, the advantages of a high-level programming language are diminished. With no

[*]What You See Is What You Get

[†]This research was supported by DARPA Grant N00014-92-J-1824, AFOSR Grant E30602-97-1-0152, and a grant of HPC time from the Arctic Region Supercomputing Center.

guidelines as to the relative costs of language features, programmers have little basis on which to make implementation choices. The lack of a performance model also means that a program which executes efficiently on one platform may suffer significant performance degradation on other platforms, since there are no guarantees as to how the language will be implemented.

Performance models are well-understood for popular sequential languages such as C and Fortran. In the parallel realm, there is a need for similar models to account for the complex issues related to running on multiple processors. Yet performance models for parallel languages have received little attention. In this paper, we present the performance model for ZPL, a portable data-parallel language whose design goals included presenting users with a clear picture of the costs involved in their algorithms. This is in stark contrast with languages such as High-Performance Fortran whose source code gives very little indication of how a program will perform on any particular machine or compiler.

During program design, programmers often use asymptotic analysis to decide between various algorithmic choices. However, even after a specific algorithm is selected, second-order implementation details must still be considered to achieve optimal performance. For instance, consider the following C code fragments that add a pair of 2D arrays, as an example of how a language's performance model helps the programmer make such implementation decisions.

```
const int N = 1000;
double A[N][N], B[N][N], C[N][N];
int i, j;
```

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

Implementation Choice 1

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

Implementation Choice 2

Although these fragments are algorithmically and asymptotically equivalent, they perform differently due to the way C's row-major array allocation policy interacts with the caching mechanisms of contemporary architectures. Thus, C's performance model dictates that the first implementation choice is preferable because it will traverse the arrays in an order that respects the memory hierarchy. One might argue that a sophisticated optimizing compiler might reorder the loops of the second fragment. However, relying on such an optimization makes the code's performance less portable. Thus, the first implementation choice remains the *correct* choice for C and has therefore become the de facto standard. In contrast, Fortran lays arrays out in column major order, and Fortran programmers therefore choose the second style in their programs. Other aspects of both languages are subject to similar evaluation: parameter passing mechanisms, procedure call overhead, library routines, etc.

It should be noted that these tradeoffs are determined by the virtual machine model used by a language's compilers and, to a lesser degree, the actual machine on which the program is running. To a large extent, the successes of Fortran and C are due to the clear mapping that exists between the languages and the von Neumann machine model, which forms a reasonable description of modern uniprocessors. This ability to "see" an accurate picture of the machine through the language is the most crucial characteristic of a good performance model. Note that although the model will not specify an exact cost for its operators and cannot be used to determine the running time of a program, it nevertheless helps programmers by giving them a rough sense of the consequences of their implementation choices.

In the realm of parallel languages, programmers would like similar performance models on which to base algorithmic decisions. In addition to the standard concerns inherited from the sequential domain, parallel language performance models need to emphasize the cost of interprocessor data transfer, since communication can have a significant impact on the performance of parallel computations. ZPL achieves this by using the CTA parallel machine model [14] as the basis for its performance model. The CTA empahsizes data locality and accurately models contemporary parallel machines. ZPL's performance model has the additional benefit of ensuring that every operation which requires communication is visible to the programmer at the source level. We affectionately refer to this property as ZPL's *WYSIWYG* performance model.

The rest of the paper is organized as follows. In the next section, we summarize the performance models of various parallel languages. In Section 3 we provide a brief introduction to ZPL, and in Section 4 we describe its performance model. Section 5 contains experiments designed to validate our performance model. We conclude in Section 6.

2 Related Work

One method of parallel programming is to use a scalar language such as C or Fortran, combined with message passing libraries such as PVM or MPI. This approach has an implicit performance model in the sense that programmers are aware of where communication takes place, since they must specify it explicitly. However, coding at this low level is tedious and error-prone, motivating the existence of higher-level parallel programming languages.

In the arena of higher-level parallel languages, NESL is a language with a well-defined performance model [1]. It uses a work/depth model to calculate asymptotic bounds for the execution time of NESL programs on parallel computers. Although this model fits NESL's functional paradigm well and allows users to make coarse-grained algorithm decisions, it reveals very little about the lower-level impact of one's implementation choices and how they will be implemented on the target machine. As an example, the cost of interprocessor communication is considered negligible in the NESL model and is therefore ignored entirely.

The most prevalent parallel language, High Performance Fortran [5], suffers a complete lack of a performance model. As a result, programmers must re-tune their program for each compiler and platform that they use, neutralizing any notion of portable performance. Ngo demonstrates that this lack of a performance model results in erratic execution times when compiling HPF programs using different compilers on the IBM SP-2 [11]. One of the biggest causes of ambiguity in the performance of HPF programs is the fact that communication is completely hidden from the user, making it difficult to evaluate different implementation options [4]. As an example, Ngo compares matrix multiply algorithms written in HPF, demonstrating that there is neither any source-level indication of how they will perform, nor does either algorithm consistently outperform the other. [10]. By defining a formal performance model to which all HPF compilers must adhere, this problem could be alleviated.

In response to HPF's hidden and underspecified communication model, F^{--} was developed to make communication explicit and highly visible to the programmer using a simple and natural syntax extension to Fortran 90 [12]. This results in a better performance model than HPF, but not without some cost. The user is forced to program at a local per-processor level, thereby forfeiting some of the benefits of higher-level languages, such as sequential semantics and deterministic execution. Furthermore, by explicitly specifying interprocessor data transfers, programmers are not as protected from nondeterminism, race conditions, and deadlock as they might be in a higher-level language. Thus, although F^{--} is more convenient to use than scalar languages with message passing, it does not raise the level of abstraction to a sufficiently convenient level.

These examples illustrate a tension between providing the benefits of a high-level language and giving the user a low-level view of the execution costs of their algorithm. In ZPL, we strive to achieve the best of both worlds by providing a powerful and expressive language in which low-level operations such as communication are directly visible to programmers through the language's operators.

3 Introduction to ZPL

ZPL is a portable data-parallel language that has been developed at the University of Washington. Its syntax is array-based and includes operators and constructs designed to expressively describe common programming paradigms and computations. ZPL has sequential semantics that allow programs to be written and debugged on sequential workstations and then effortlessly recompiled for execution on parallel architectures. ZPL generally outperforms HPF and has proven to be competitive with hand-coded C and message passing [9, 7]. Applications from a variety of disciplines have been written using ZPL [6, 3, 13], and the language was released for widespread use in July 1997. Supported platforms include the Cray T3D/T3E, Intel Paragon, IBM SP-2, SGI Power Challenge, clusters of workstations using PVM or MPI, and sequential workstations.

In this section, we give a brief introduction to ZPL concepts that are required to understand this paper. A more complete presentation of the language is available in the ZPL Programmer's Guide and Reference Manual [15, 8].

3.1 Regions and Arrays

The *region* is ZPL's most fundamental concept. Regions are index sets through which a program's parallelism is expressed. In their most basic form, regions are simply dense rectangular set of indices similar to those used to define arrays in traditional languages. Region definitions can be inlined directly into a ZPL program, or given names as follows:

```
region R    = [1..N   ,1..N  ];
       BigR = [0..N+1,0..N+1];                (1)
       Top  = [0      ,1..N  ];
```

These declarations define three regions: R is an $N \times N$ index set; BigR is equal to R extended by an extra row and column in each direction; Top describes the row just above R.

Regions have two main roles in ZPL. The first is to declare parallel arrays. This is done by referring to the region in a variable's type specifier as follows:

```
var A: [BigR] double;                         (2)
    B: [R]    integer;
```

These declarations define two arrays: A, an array of doubles whose size is defined by BigR; and B, an $N \times N$ integer array. The second use of a region is to specify the indices over which an array operation should execute. For example, the following statement increments each element of A by its corresponding value of B over the index range specified by R:

```
[R] A := A + B;                               (3)
```

Regions are ZPL's fundamental source of parallelism. Each region is partitioned across the processor set, resulting in the distribution of every array and operation that is defined in terms of that region. The distribution of regions is discussed more fully in Section 4.1.

3.2 The @ Operator

Since regions replace explicit array indexing, ZPL provides the @ *operator* to allow elements with different indices to interact with one another. The @ operator takes an array and an offset vector called a *direction* and shifts the references to the array by the offset. For example, to replace each element of A with the average of its left and right neighbors, one would use:

```
[R] A := A@[0,–1] + A@[0,1];                  (4)
```

Directions are typically named in order to improve a program's readability. For example, the previous example could have been written:

```
direction left  = [0,–1];
          right = [0,1  ];                                    (5)

[R] A := A@left + A@right;
```

Directions are typically reused throughout a program, so this approach also reduces careless indexing mistakes through the selection of meaningful names.

3.3 Reductions and Floods

Reductions and *floods* are ZPL's operators for combining and replicating array values. The reduction operator (*op*<<) uses a binary operator to combine array elements along one or more dimensions, resulting in an array slice or scalar value. For example, consider the following lines of code:

```
[Top] A    := +<<[R] A;                                       (6)
[R]   bigA := max<< A;
```

In the first statement, we use a *partial reduction* to replace each element in the top row of A with the sum of the values in its corresponding column. Note that the region scope at the beginning of the statement (Top) specifies the indices to be assigned, while the one supplied to the reduction operator (R) specifies which elements are to be combined. The two regions are compared statically to determine which dimension(s) should be collapsed. The second statement uses a *complete reduction* to merge all the elements of A into a single scalar using the "max" operator. Complete reductions require only a single region scope since assignment to a scalar does not require a region.

The flood operator (>>) is the dual of a partial reduction. It replicates the values of an array slice across an array. Consider:

```
[R] begin
      A := >>[Top] A;                                         (7)
      B := >>[1,1]  B;
    end;
```

This code demonstrates the application of a single region scope (R) to a block of statements. In the first flood, the top row of A is replicated across all the rows of A in region R. As with partial reductions, two regions are needed to specify the flood operation: one to indicate the source indices of the flood (Top) and the second to specify the destination (R). In the second statement, the value of the first element of B is replicated across all elements of B in region R.

3.4 Gather and Scatter

The *gather* (<##) and *scatter* (>##) operators are a means of arbitrarily rearranging data in ZPL. As arguments, they take a list of arrays that are used to index randomly into the source or destination array (for gather and scatter, respectively). For example, the following code uses the gather operator to perform a matrix transpose of B, assigning the result to A:

```
var I,J: [R] integer;
[R] begin
        I  := Index2;
        J  := Index1;
        A := <##[I,J] B;
    end;
```
(8)

This code makes use of the standard ZPL arrays Index1 and Index2. Index$i$ is a constant array in which every element's value is equal to its index in the i^{th} dimension. Thus, this gather will replace each element of A with the element of B whose index is specified by the corresponding values of I and J. Although we have set I and J to perform a transpose in this example, in general they can be used to specify any permutation or rearrangement of an array's values.

These operators form a good sampling of the various types available to the ZPL programmer. In the next section we will reason about their implementation costs and WYSIWYG performance.

4 ZPL's Performance Model

The performance of a ZPL program is dependent on three criteria: its scalar performance, its concurrency, and its interprocessor communication. ZPL programs are compiled to C as an intermediate format, so its scalar performance is dictated heavily by C's performance model. Concurrency and interprocessor communication are both dependent on how ZPL regions, arrays, and scalars are distributed across the processor set.

4.1 ZPL's Data Distribution Scheme

The key to ZPL's WYSIWYG performance model lies in its region distribution invariant, which constrains how *interacting regions* are partitioned across the processor mesh. Two regions are considered to be interacting if they are both used within a single statement, either directly (using a region scope) or indirectly (by referring to an array declared over one of the regions). For example, in the code fragments of Section 3, BigR and R would be considered to be interacting due to the use of A (declared over BigR) within the scope of R in code fragment 3. In addition, Top and R interact due to their uses in the partial reduction and flood statements in fragments 6 and 7. Thus all three regions are interacting.

The ZPL language dictates that interacting regions must be distributed in a *mesh-aligned* fashion. Being mesh-aligned means that if two n-dimensional regions are partitioned across a logical n-dimensional processor mesh, both regions' slices with index i in dimension d will be mapped to the same processor mesh slice p in dimension d. For example, since R and Top are interacting, they must be mesh-aligned, and therefore column i of Top must be distributed across the same processor column as column i of R. Moreover, mesh-alignment implies that element (i, j) of two interacting regions will be located on the same processor. Thus, each element of R will be located on the same processor as its corresponding element in BigR. Note that using a blocked, cyclic, or block-cyclic partitioning scheme for the bounding region of a set of interacting regions causes the regions to be mesh-aligned. Our ZPL compiler uses a blocked partitioning scheme by default, and for simplicity we will assume that scheme for the remainder of this paper.

Once regions are partitioned among the processors, each array is allocated using the same distribution as its defining region. Array operations are computed on the processors containing the elements in the relevant region scopes. Thus, region partitioning results directly in the concurrency aspect of ZPL's performance model.

One final characteristic of ZPL's data distribution scheme is that scalar variables are replicated across processors. Coherency is maintained either through redundant computation when possible, or interprocessor communication when not.

It might be argued that ZPL's data distribution scheme is too restrictive, forcing programmers to formulate their problems in terms that are amenable to the mesh-alignment principle. Alternatively, ZPL could allow arbitrary array alignment and indexing. In this scenario, the communication cost of a statement would be a function of its data access pattern and the alignment of its arrays. This model is complicated by the fact that a single source-level array (*e.g.*, a formal parameter) may refer to a number of distinct arrays during execution, each with its own alignment scheme. Estimating performance in such a scheme is complex because communication is not manifest in the source code, and the analysis required to locate and evaluate it requires looking more globally than the statement level. In contrast, ZPL's communication costs are dependent only on the operations within a statement. These costs are evaluated qualitatively in the next section and experimentally in Section 5.

4.2 Qualitative Evaluation of Operators

Once ZPL's data distribution scheme is understood, the relative cost of its operators becomes readily apparent. For example, in the element-wise addition and assignment of code fragment 3, we know that corresponding elements of A and B are assigned to the same processor and therefore no communication is required to complete this operation. By this same reasoning all ZPL statements that use only assignment, traditional operators, and function calls will similarly be communication-free. Communication can only be incurred when operators specific to ZPL are used. Furthermore, the cost of these communications can be estimated based on what we know about the partitioning scheme.

The @ operator. Since the @ operator is used to shift an array's references, interacting array values are no longer guaranteed to reside on the same processor. Therefore, point-to-point communication is required to transfer remote values to a processor's local memory. For example, in the case of a blocked decomposition, the statement in code fragment 4 would require each processor to exchange a column of data with its neighboring processors in its row. Since the @ operator generally requires such communication, the programmer can expect that array references with @'s will tend to be more expensive than normal array references.

Floods and Reductions. Flooding replicates values along one or more dimensions of an array. Since the region distribution invariant guarantees that array slices will map to processor slices, this implies that flooding can be achieved by broadcasting values to the processors within the appropriate slice. For example, the first flood in code fragment 7 requires that each processor owning a block of Top to broadcast its relevant values of A to the processors in its column. Similarly, the second statement requires the processor with the first element of B to broadcast the value to all other processors. Once the data is received, it can be replicated across the processor's local block of values. Due to the fact that broadcasts become more expensive as the number of processors grows, we can expect the cost of flooding to increase similarly.

Partial reductions are the dual of flooding, so they will be implemented by combining values along a processor slice, placing the result at the appropriate processor (*e.g.*, using a binary combining tree). Full reductions are similar, but also require a broadcast to replicate the resulting scalar value across all processors. Since reductions have communication patterns that are similar to flooding, we expect them to scale in similar ways, but to be more expensive due to the additional operations required to combine values.

Gathers and Scatters. Scatters and gathers are used to express arbitrary data movement and therefore tend to move larger volumes of data in less regular communication patterns. They will tend to require more communication due to the fact that the source, target, and indexing arrays are all distributed across processors. Performance is further impacted by the cache contention resulting from the number of arrays needed to express the operation and the random access of data required in the source or destination array. As a result of all of these factors, the programmer can expect gathers and scatters to be the most expensive operation described in this paper.

Other Operators. ZPL contains additional operators not described herein (*e.g.*, wraps, reflects, and full and partial scans). Although it could be enlightening to discuss each of them in turn, the more important point is this: *Knowing what an operator does and being familiar with ZPL's data distribution scheme, it is possible for a programmer to qualitatively assess the communication style required by any operator as well as to roughly estimate its performance impact.* In this way, the communication implicit in a ZPL program is visible to programmers without burdening them with the task of explicitly specifying the data transfer. What they see is what they get.

5 Experiments

In this section, we experimentally demonstrate the effectiveness of the ZPL performance model. In the first experiment, we measure the performance of a number of ZPL operations and compare the results to our qualitative analysis. In the second experiment, we show that the source-level evaluations of two matrix multiply algorithms can accurately predict their relative performance.

Both experiments were run on four different parallel machines: the Cray T3E, the SGI Power Challenge, the IBM SP-2, and the Intel Paragon. All interprocessor communication was efficiently implemented using the communication libraries of each machine: SHMEM on the T3E, MPI on the Power Challenge and the SP-2, and NX on the Paragon.

5.1 Performance of ZPL Operations

Figure 5.1 shows the measured performance of selected ZPL array operations: array copy, array addition, translation using the @ operator, flooding, partial reduction, full reduction, and permutation. Each graph shows the execution times of the operations on three processor configurations of different sizes. Each column of graphs represents a particular machine, while each row represents the number of elements of R assigned to *each* processor. R is scaled in this way to maintain similar cache effects and data transfer sizes as the number of processors increases. Note that the running time of a program that scales perfectly will therefore be constant across different configurations. By comparing values within a graph, along a column, or along a row, one can evaluate how ZPL's operators scale with the number of processors and problem size, as well as how portable their performance is across architectures.

Although countless observations could be made from these graphs, we will give just a few to highlight performance issues for each operator that corroborate our analysis from the previous section. To begin with, the WYSIWYG model indicates that the first two statements require no communication and should therefore scale perfectly as the number of processors increases. Looking over the graphs, one can see that this is true in all cases.

Comparing the statement with the @ operation to the array assignment, we see that it tends to be more expensive as expected, due to the required communication. As the problem set size grows, the

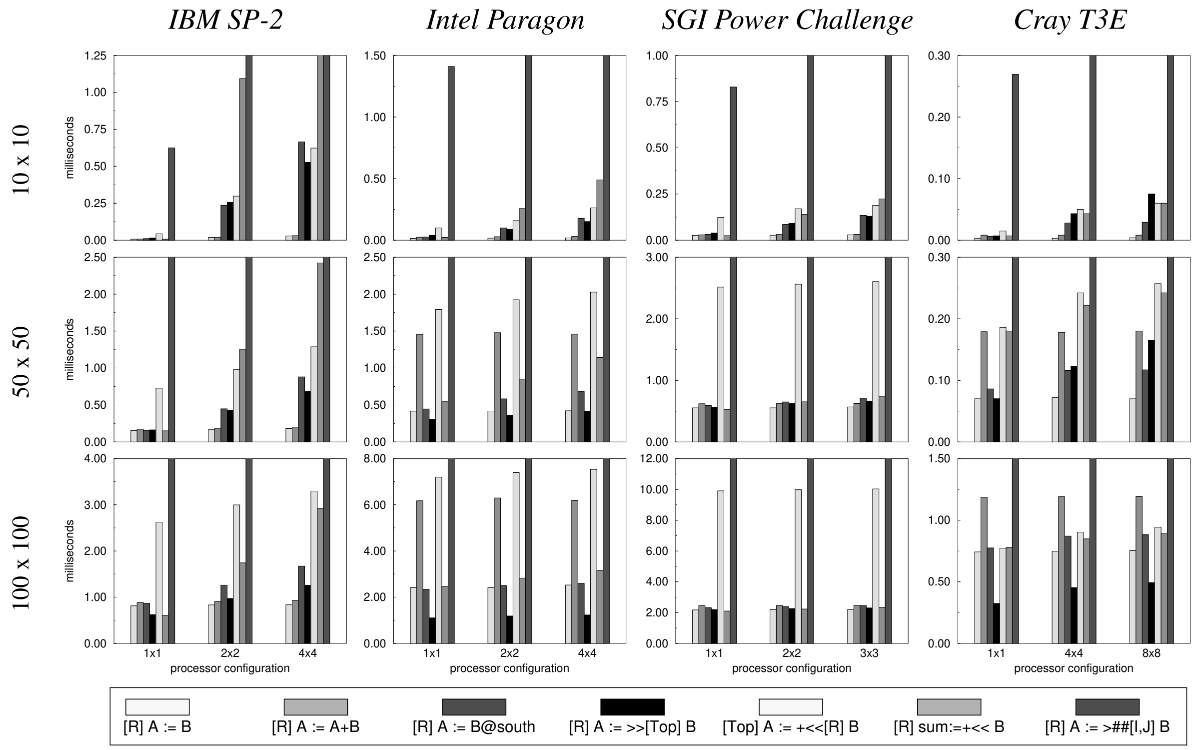

Figure 1: Measured performance of ZPL operations. Each graph shows the execution times on three processor configurations. Each column of graphs represents a machine, and each row represents a per-processor problem size.

time required to perform the N^2 assignments tends to dominate the time spent in communication, reducing the performance gap between the two assignment statements.

As predicted, the flood operator's performance also becomes slower as the number of processors increases. On the T3E, where 64 processors are available, note that the @ operator's performance levels off at 16 processors, while the flood continues to become more expensive. This fits our predictions perfectly since implementing an @ requires a constant amount of point-to-point communication per processor while the broadcasts required to implement flood continue to get more expensive as the number of processors grows.

Comparing partial and full reductions, we see that the more expensive operator is dependent on the problem size and number of processors. On the smaller problem sizes, communication is the dominant factor and the full reduction is generally more expensive since it requires a broadcast. However, on larger problem sizes, the number of additions required to perform a partial reduction

```
region R = [1..N,1..N];
direction right = [0,1];
          below = [1,0];
var A,B,C:[R] double;
```

Declarations for N×N Matrix Multiply

```
[R] begin
      C := 0.0;
      for i := 1 to N do
        C += (>>[1..N,i] A) * (>>[i,1..N] B);
      end;
   end;
```

SUMMA Matrix Multiply

```
[R] begin
      for i := 2 to N do
        [right of R] wrap A;
        [i..N,1..N] A := A@right;
        [below of R] wrap B;
        [1..N,i..N] B := B@below;
      end;
      C := A * B;
      for i := 2 to N do
        [right of R] wrap A;
        A := A@right;
        [below of R] wrap B;
        B := B@below;
        C += A * B;
      end;
   end;
```

Cannon's Matrix Multiply

Figure 2: Two algorithms for Matrix Multiplication in ZPL.

tends to outweigh the extra communication of the full reduction, making it the more expensive operator.

Finally, as predicted, the scatter operation consistently proves to be significantly more expensive than the other operators, generally costing at least an order of magnitude more than the next most expensive operator.

5.2 Matrix Multiply

Although evaluating the performance of individual ZPL statements is instructive, the real test of the WYSIWYG performance model is in evaluating entire algorithms. In Figure 2, we show two algorithms for matrix-matrix multiplication, SUMMA [16] and Cannon's Algorithm [2]. SUMMA is considered to be the most scalable of portable parallel matrix multiplication algorithms. It iteratively floods a column of matrix A and a row of matrix B, accumulating their product in C. Cannon's algorithm skews the A and B matrices as an initialization step and then iteratively performs cyclic shifts of A and B, multiplying them and accumulating into the C matrix. The skewing and cyclic shifts are achieved using ZPL's *wrap* operator within an *of region* — another

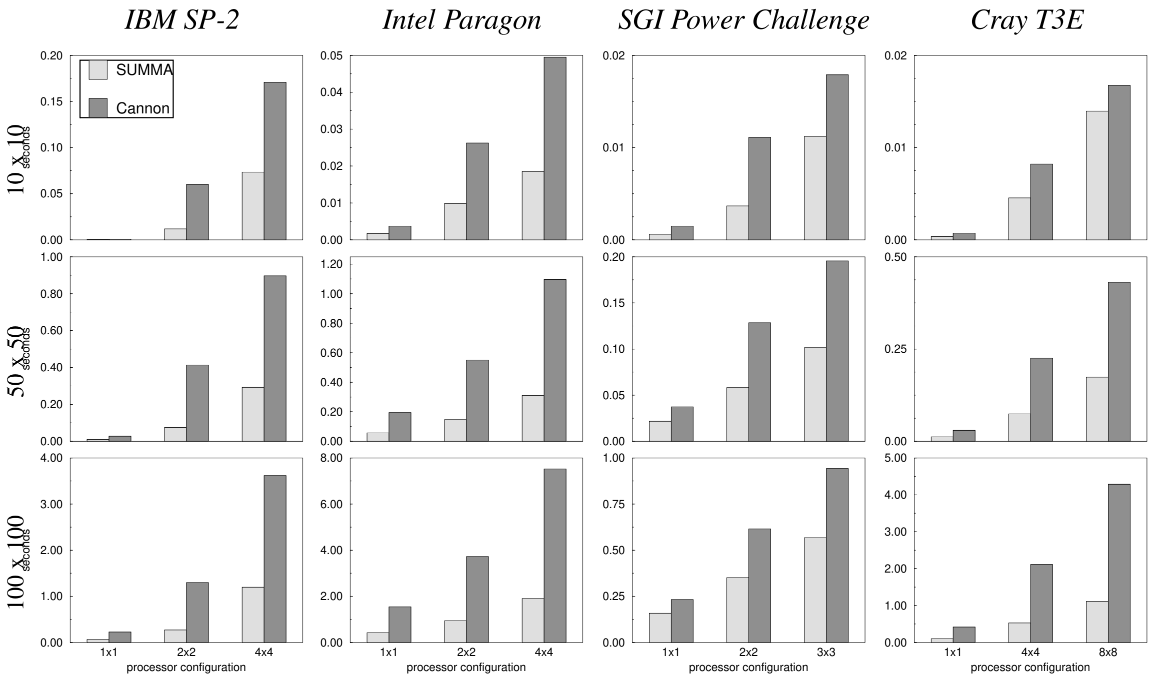

Figure 3: Peformance of SUMMA and Cannon's algorithm for matrix multiplication in ZPL. Each graph shows the execution times on three processor configurations. Each column of graphs represents a machine, and each row represents a per-processor problem size.

form of point-to-point communication in ZPL.

Analyzing these algorithms asymptotically, we see that they both perform $O(N^3)$ computation and $O(N)$ communication. However, looking at the communication incurred by each program, we can determine that they are not equal. The SUMMA algorithm is realized using $2N$ floods while Cannon's algorithm requires $4N$ cyclic shifts. Given that floods and shifts are similar in cost, we can guess that SUMMA should be the better algorithm.

To test our hypothesis, we ran both algorithms on the same four machines for a variety of problem sizes (once again scaling the problem to maintain a constant amount of data per processor). Figure 5.2 shows our results and verifies that SUMMA is faster than Cannon's algorithm in all cases. Performing the same experiment in HPF, Ngo demonstrated that not only is it virtually impossible to predict the performance of these algorithms by looking at the HPF source, but also that neither algorithm consistently outperforms the other across all compilers [10]. ZPL's WYSIWYG performance model makes both source-level evaluation and portable performance a reality.

5.3 Discussion

It should be noted that, as in the sequential realm, ZPL's performance model does not yield exact information about a program's running time. However, it does allow a programmer to be aware of the implications of their implementation decisions by making the mapping of their code to a parallel machine clear. As with sequential languages, a programmer's intuition can be undermined by the complexity of modern machines and the impact of compiler optimizations (*e.g.*, pipelining communication, removing redundant communications). However, as in the sequential world, we expect that ZPL's cues will be invaluable to programmers by allowing them to see the machine through the language.

6 Conclusions and Future Work

A language's performance model gives programmers a rough understanding of a code's performance, facilitating the selection between alternative implementations. Though particularly important in the parallel domain – where the cost of language features may vary greatly, *e.g.*, local versus remote memory access – ZPL is the first parallel programming language to present a performance model distinct from an implementing machine. Not only can programmers evaluate a code's approximate cost, they may do it simply, for they have a clear understanding of how each language feature is mapped to the underlying hardware via the CTA machine model. This we call ZPL's WYSIWYG performance model. We have given a qualitative argument of how the language realizes this and experimentally verified that a diverse collection of parallel machines respect it.

In future work we will extend the ZPL language to handle irregular and sparse problems. The challenge will be to do so while preserving ZPL's WYSIWYG performance model.

References

[1] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[2] L. F. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[3] Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *Ninth International Conference on Supercomputing*, 1995.

[4] Richard Friedman, John Levesque, and Gene Wagenbreth. *Fortran Parallelization Handbook, Preliminary Edition*. Applied Parallel Research, April 1995.

[5] High Performance Fortran Forum. *High Performance Fortran Specification Version 2.0*. January 1997.

[6] E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.

[7] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1995.

[8] Calvin Lin. ZPL language reference manual. Technical Report 94–10–06, Department of Computer Science and Engineering, University of Washington, 1994.

[9] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[10] Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.

[11] Ton A. Ngo, Lawrence Snyder, and Bradford L. Chamberlain. Portable performance of data parallel languages. *to appear in Supercomputing 1997*, November 1997.

[12] Robert W. Numrich and Jon L. Steidel. Simple parallel extensions to Fortran 90. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[13] Wilkey Richardson, Mary Bailey, and William H. Sanders. Using ZPL to develop a parallel Chaos router simulator. In *1996 Winter Simulation Conference*, December 1996.

[14] Lawrence Snyder. Experimental validation of models of parallel computation. In A. Hofmann and J. van Leeuwen, editors, *Lecture Notes in Computer Science, Special Volume 1000*, pages 78–100. Springer-Verlag, 1995.

[15] Lawrence Snyder. *The ZPL Programmer's Guide*. May 1996.

[16] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, University of Texas, Austin, Texas, April 1995.