ZPL's WYSIWYG Performance Model*

Bradford L. Chamberlain, Calvin Lin†, Sung-Eun Choi, Lawrence Snyder, E Christopher Lewis, W. Derrick Weathersby

Dept. of Computer Science and Eng.
University of Washington
Box 352350
Seattle, WA 98195-2350
zpl-info@cs.washington.edu

†Department of Computer Sciences
The University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188
lin@cs.utexas.edu

Abstract

ZPL is a parallel array language designed for high performance scientific and engineering computations. Unlike other parallel languages, ZPL is founded on a machine model (the CTA) that accurately abstracts contemporary MIMD parallel computers. This makes it possible to correlate ZPL programs with machine behavior. As a result, programmers can reason about how code will perform on a typical parallel machine and thereby make informed decisions between alternative programming solutions. This paper describes ZPL's performance model and its syntactic cues for conveying operation cost. The what-you-see-is-what-you-get *(WYSIWYG) nature of ZPL operations is demonstrated on the IBM SP-2, Intel Paragon, SGI Power Challenge, and Cray T3E. Additionally, the model is used to evaluate two algorithms for matrix multiplication. 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 programmers the burden of coding in assembly language and simplify the task of porting programs to new machines. However, 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. Without any guidelines as to how language constructs are implemented, performance-conscious programmers have little basis on which to make implementation choices. In addition, programs that execute efficiently on one platform may suffer significant performance degradation on other platforms because there are no guarantees as to how a compiler will implement the language's features.

Performance models are well-understood for popular sequential languages such as C and Fortran, because there is a clear mapping between their constructs and the von Neumann machine model, which reasonably approximates contemporary 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 does not specify an exact cost for language operators and cannot be used to determine the precise running time of a program, it nevertheless aids programmers by giving them a rough sense of the consequences of their implementation choices.

As a simple example, analysis of the following two loops shows them to be algorithmically and asymptotically equivalent. However, C programmers use the first implementation because it accesses the elements in the same order that C's language definition requires them to be laid out in memory. This results in an implementation that respects the memory hierarchy of contemporary machines.

```
const int m = 1000, n = 2000;
double A[m][n], B[m][n], C[m][n];
int i, j;
```

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

Implementation 1

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

Implementation 2

*This research was supported in part by DARPA Grants N00014-92-J-1824 and E30602-97-1-0152, NSF Grant CCR-9710284, and a grant of HPC time from the Arctic Region Supercomputing Center.

This example illustrates that even after asymptotic analysis and algorithmic design, second-order implementation details are still a factor in obtaining good performance. Although a sophisticated compiler might transform the second implementation into the first, C's performance model does not guarantee this, and therefore programmers concerned with portable performance will not rely on it. As a result, the first implementation is the *right* choice in C. Conversely, Fortran uses column-major order, so Fortran programmers will use the second loop ordering. Other aspects of both languages are subject to similar evaluation, including parameter passing mechanisms, procedure call overheads, library routines, and system calls. Consideration of a language's performance model in this way enables high quality machine-independent programming.

In the realm of parallel programming, there is a similar need for language performance models that account for the costs associated with running on multiple processors in addition to those inherited from the sequential domain. In particular, these parallel models should emphasize the cost of interprocessor data movement since communication often significantly impacts application performance.

ZPL was the first parallel language to provide an explicit performance model distinct from an implementing machine. The effectiveness of its performance model is the result of an early design decision to preserve machine visibility rather than to rely on sophisticated compiler analysis and optimization. This is in stark contrast with parallel languages such as High Performance Fortran (HPF) that utilize directives to specify parallelism. Because HPF's directives are optional and because the compiler is free to ignore them, it is difficult for a programmer to reason about an algorithm's performance without detailed knowledge of the compiler. Ngo has shown that this lack of a performance model leads to unpredictable, inconsistent, and poor performance [11].

In this paper we describe the performance model of the ZPL parallel array language. We describe the straightforward mapping of ZPL constructs to the *Candidate Type Architecture* (CTA), a parallel analog of the von Neumann machine model that accurately abstracts contemporary MIMD parallel computers [15]. This allows programmers to reason about the behavior of their ZPL programs on parallel machines. In addition, we demonstrate that ZPL's syntax inherently identifies operations that induce communication. These visual cues simplify the first-order evaluation of a parallel program's cost and motivate our description of the performance model as "what-you-see-is-what-you-get" (WYSIWYG).

We demonstrate the use of ZPL's WYSIWYG performance model in two experiments. The first verifies the accuracy of its syntactic cues across several problem sizes, architectures, and numbers of processors. The second illustrates the use of ZPL's performance model to accurately select the better of two matrix multiplication codes written in ZPL.

The remainder of the paper is organized as follows. In the next section, we summarize related work. 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

A common method of parallel programming is to use a scalar language such as C or Fortran, in combination with message passing libraries such as PVM or MPI. This approach has an implicit performance model formed by the performance model inherited from the sequential language in combination with the explicit interprocessor communication specified by the programmer. However, coding at this per-processor level is tedious and error-prone, motivating the need for higher-level parallel programming languages.

NESL is an example of a higher-level parallel language that includes a well-defined performance model [2]. It uses a work/depth scheme to calculate asymptotic bounds for the execution time of NESL programs on parallel computers. Although this model matches NESL's functional paradigm well and allows users to make coarse-grained algorithmic decisions, it reveals very little about the lower-level impact of one's implementation choices and how they will be mapped to the target machine. For 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 [6], suffers from the complete lack of a performance model. As a result, programmers must re-tune their programs for each compiler and platform that they use, neutralizing any notion of portable performance. Ngo *et al.* demonstrate that this lack of a performance model results in erratic execution times when compiling HPF programs using different compilers on the IBM SP-2 [12]. 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 [5]. As an example, Ngo compares matrix multiplication algorithms written in HPF, demonstrating that there is neither any source-level indication of how they will perform, nor any consistency in the relative performance of the algorithms [11]. By defining a performance model to which all HPF compilers must adhere, this problem could have been alleviated. Most compilers compensate for HPF's lack of a performance model by providing tools that give source-level feedback about the compilation process and/or program execution. The dPablo toolkit [1] is one such example. The problem with this approach is that such tools are tightly coupled to a particular compiler's compilation model, and therefore do not aid in the creation of portably performing programs.

In contrast with HPF's hidden and unspecified 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 [13]. This results in a clearer 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 shielded from race conditions and deadlock as they would 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 demonstrate a tension between providing the benefits of a high-level language and giving the programmer 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 constructs and operators 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 ported to parallel architectures in a single recompilation.

ZPL generally outperforms HPF and has proven to be competitive with hand-coded C and message passing [10, 9]. Applications from a variety of disciplines have been written using ZPL [4, 7, 14], 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/Origin, clusters of workstations using PVM and MPI, and sequential workstations.

In this section, we give a brief introduction to ZPL concepts that are required to understand this paper. More complete presentations of the language are available in the ZPL Programmer's Guide and Reference Manual [8, 16].

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 sets 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   ];
       Top  = [0      ,1..n   ];        (1)
       BigR = [0..n+1,0..n+1];
```

These declarations define three regions: R is an $n \times n$ index set; Top describes the row just above R; BigR is an extension of R by an extra row and column in each direction. Diagrams of these regions are shown in Figure 1.

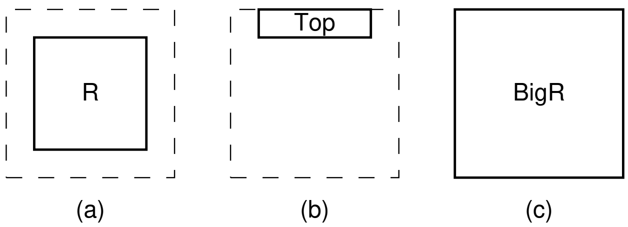

Figure 1. Diagrams of the region declarations in code fragment 1: (a) region *R*, (b) region *Top*, and (c) region *BigR*. The dashed boxes indicate the index space [0..*n*+1,0..*n*+1].

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

These declarations define two arrays: A, an $n \times n$ array of doubles, and B, an integer array defined over BigR. Figure 2 shows diagrams of these arrays.

The second use of regions is to open a *region scope* that specifies 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)
```

Figure 3 illustrates this statement.

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

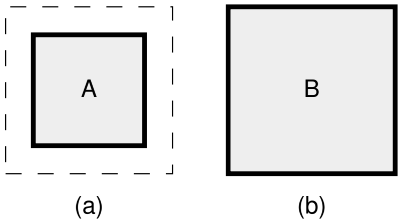

Figure 2. Diagrams of the array declarations in code fragment 2: (a) array *A*, declared to be of size *R*; (b) array *B*, declared to be of size *BigR*. The shading indicates that, unlike regions, arrays have data associated with each index.

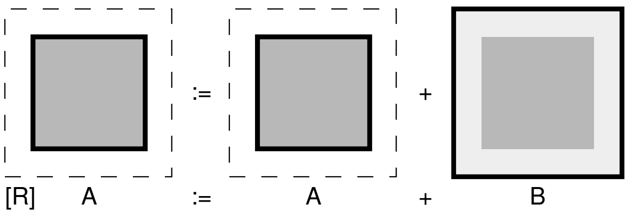

Figure 3. Illustration of the array addition in code fragment 3. The darker shading indicates the array elements referenced in each expression.

3.2. The @ operator

Since regions eliminate explicit array indexing, ZPL provides the @ *operator* to allow translated references to arrays. The @ operator takes an array and an offset vector called a *direction* as operands and shifts references to the array by the offset. For example, to replace each element of B with the sum of its left and right neighbors, one would write:

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

Directions are generally named in order to improve a program's readability. For example, line (4) could have been written:

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

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

Figure 4 shows a picture of this operation. Directions are typically reused throughout a program, so naming them meaningfully also reduces careless indexing mistakes.

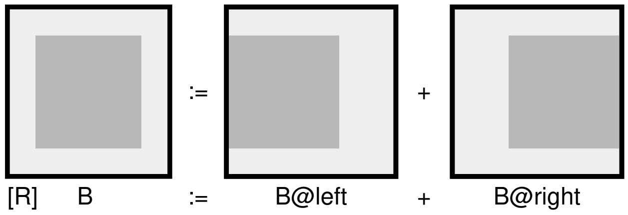

Figure 4. Illustration of the neighbor summation expressed in code fragment 5. Note that the @ operator translates references to *B*.

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:

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

In the first statement, we use a *partial reduction* to replace each element in the top row of B with the sum of the values in its corresponding column (illustrated in Figure 5). The region scope at the beginning of the statement (Top) specifies the indices to be assigned, while the one supplied with the reduction operator (R) specifies which elements are to be combined. The two regions are compared to determine which dimension(s) should be collapsed. The second statement uses a *full reduction* to merge all the elements of A into a single scalar, biggestA, using the "max" operator. Full reductions require only a single region scope since assignment to a scalar does not require a region.

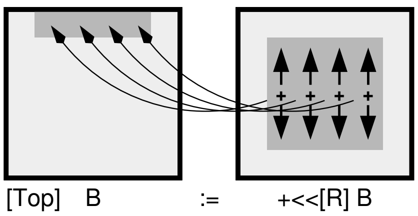

Figure 5. Illustration of the partial reduction in code fragment 6. Values of *B* are added column-wise over *R* and the sums are assigned to the corresponding elements in *Top*.

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

Consider:

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

This code demonstrates the application of a single region scope (R) to a block of statements. In the first assignment statement, the top row of B is replicated across all the rows of B in region R (shown in Figure 6). 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 A is flooded across all elements of A in region R.

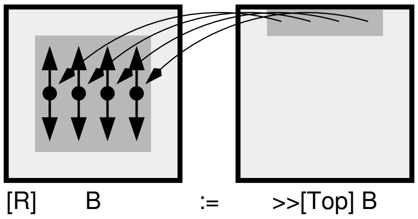

Figure 6. Illustration of the first flood statement in code fragment 7. Each element of *B* in *Top* is replicated across its corresponding column in *R*.

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 into the source or destination array (for gather and scatter, respectively). For example, the following code uses the scatter operator to perform a matrix transpose of B, assigning the result to A:

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

This code makes use of the built-in 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 scatter will replace each element of A with the element of B whose index is specified by the corresponding values of I and J. This is illustrated in Figure 7. Although we have set

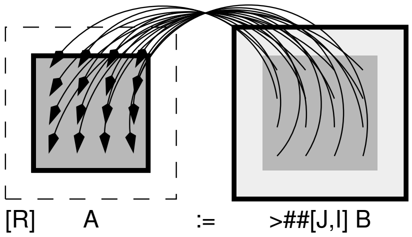

Figure 7. Illustration of the scatter operation in code fragment 8. Each element in *B* is assigned to the element in *A* specified by its corresponding values of *J* and *I*.

I and J to perform a transpose in this example, any permutation or rearrangement of an array's values is possible.

These operators form a representative sampling of the features 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 accuracy of ZPL's performance model depends both on the mapping of ZPL constructs to the CTA parallel machine model and on the CTA's ability to model real parallel computers. If both of these mappings are straightforward, programmers will be able to accurately reason about the performance of ZPL programs. In the context of this paper, the two significant features of the CTA are that it emphasizes data locality and that it neither specifies nor places importance on the processor interconnect topology. ZPL reflects the CTA's emphasis on locality by its syntactic identification of operators that induce communication. The lack of emphasis on interconnect topology allows ZPL to compute using a virtual hypergrid of processors.

The performance of ZPL code depends on three criteria: scalar performance, concurrency, and interprocessor communication. ZPL programs are compiled to C as an intermediate format, so their scalar performance is dictated heavily by C's performance model. Concurrency and interprocessor communication are both determined by the distribution of ZPL regions, arrays, and scalars across the processor grid.

4.1. ZPL's data distribution scheme

The key to ZPL's WYSIWYG performance model lies in its *region distribution invariant*, which constrains how regions' index sets are partitioned across the virtual processor grid. ZPL dictates that all regions must be partitioned in a *grid-aligned* fashion. This implies that each dimension of the region is mapped independently to its corresponding dimension in the virtual processor grid. For example, in two dimensions each region would require two mappings: one between its rows and the processor rows, the second between its columns and the processor columns. This notion generalizes to r-dimensional regions on r-dimensional virtual processor grids (where some dimensions may be degenerate). Note that partitioning each dimension of a region using a blocked, cyclic, or block-cyclic scheme results in grid-alignment. Our ZPL compiler uses blocked partitioning by default, and for simplicity we will use this scheme for the remainder of the paper.

ZPL places additional constraints on *interacting regions*. Two regions are considered to be interacting if they are both referenced within a single statement. These references can either be explicit (by referring to the region within a region scope) or implicit (by referring to an array that was declared over the region). For example, in the code fragments of Section 3, R and BigR are considered to be interacting due to the use of B (declared over BigR) within the scope of R in code fragment 3. Furthermore, Top and R interact due to their uses in the partial reduction and flood statements of code fragments 6 and 7. Thus, all three regions are interacting.

ZPL requires the mapping functions used to partition interacting regions to be identical for all indices common to both regions. For example, since R and Top are interacting, column i of Top must be mapped to the same processor column as column i of R for all i, $1 \leq i \leq n$. Since this requirement applies to every dimension, any index common to a pair of interacting regions must necessarily be located on the same processor for both regions. Thus, every index (i, j) of R will be mapped to the same processor as the corresponding index (i, j) in BigR.

Once regions are partitioned across the processors, each array is allocated using the same distribution as the region over which it was declared. Array operations are computed on the processors that own the elements in the enclosing region scopes. Thus, region partitioning determines the concurrency of a ZPL program.

One final characteristic of ZPL's data distribution scheme is that scalar variables are replicated across processors. Coherency is maintained either through redundant computation or interprocessor communication.

It might be argued that ZPL's data distribution scheme is overly restrictive, forcing programmers to formulate their problems in terms that are amenable to the region distribution invariant. Alternatively, ZPL could support arbitrary array alignment and indexing, thereby providing greater flexibility to the programmer. The problem with this approach is that the communication cost of a statement would be determined by the degree to which its arrays are aligned, something that would not be apparent in the source code. Thus, estimating performance in such a scheme would require programmers to look at their code more globally than the statement level. In contrast, ZPL's communication costs are dependent only on the operations within a statement and can therefore be trivially identified. These costs are evaluated qualitatively in the next section.

4.2. Qualitative evaluation of operators

Once ZPL's data distribution scheme is defined, the relative costs of its operators become 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 any ZPL statement that only uses assignment, traditional operators, and function calls will also be communication-free. Thus, communication is only induced when ZPL's nontraditional operators are used, allowing programmers to readily identify it. Furthermore, the cost of these communications can be estimated based on an understanding of the data distribution 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 5 would require each processor to exchange a column of B with both of the 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 reside on processor slices, flooding can be achieved by broadcasting values to processors within the appropriate slice. For example, the first flood in code fragment 7 requires that each processor owning a section of Top broadcast its relevant values of B to the processors in its column. Similarly, the second statement requires the processor with the first element of A 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 need to combine values along a processor slice, placing the result at the appropriate processor (*e.g.*, using a combining

operator	communication paradigm	communication complexity	communication volume	flops
@-reference	point-to-point	$O(1)$	$O(n)$	0
flood	broadcast	$O(\log p)$	$O(n)$	0
partial reduce	reduction	$O(\log p)$	$O(n)$	$O(n^2)$
full reduce	reduction	$O(\log p)$	$O(1)$	$O(n^2)$
gather/scatter	permutation	$O(p^2)$	$O(n^2)$	0

Table 1. Summary of the expected cost per processor of selected ZPL operators, assuming a $p \times p$ processor grid with a distribution of $n \times n$ array elements per processor. *Communication paradigm* indicates the induced style of data transfer; *communication complexity* signifies the depth of the communication schedule; *communication volume* indicates the number of elements transferred at each step in the schedule; *flops* indicates the number of floating point operations required to perform the operation.

tree). Full reductions are similar, but require a final 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 similarly, but to be more expensive due to the operations required to combine array values.

Gathers and scatters. Gathers and scatters 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 the processor grid. Performance is further impacted by the cache contention resulting from the number of arrays in use as well as the random data access required for 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. Table 1 summarizes this analysis of ZPL operators, estimating their expected costs asymptotically in terms of problem and processor grid size. ZPL contains additional operators not described in this paper such as *wraps*, *reflects*, and *partial* and *full 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 in a ZPL program is directly visible to programmers without burdening them with the task of explicitly specifying 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 execution time of a number of ZPL statements and compare the results to our expectations from the qualitative analysis of the previous section. In the second experiment, we show that the source-level evaluations of two matrix multiplication algorithms can accurately predict their relative performance.

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

5.1. Performance of ZPL operations

Figure 8 shows the measured execution times of several ZPL operations performed on arrays of doubles: array copy ([R] A := B), array addition ([R] A := A+B), array translation ([R] A := B@south), flooding ([R] A := >>[Top] B), partial reduction ([Top] B := +<<[R] A), full reduction ([R] sum := +<< B), and matrix transpose using scatter ([R] A := >##[J,I] B). Each graph shows the statements' execution times on three processor grids of varying size. Each row of graphs represents a particular machine, while each column represents a specific problem size. The problem size indicates the number of elements of R assigned to *each* processor. R is scaled in this way to maintain similar cache effects and data transfer volumes across all processor grids for a platform. Note that statements which scale perfectly will have consistent running times within a graph. By comparing bars within a graph, across a row of graphs, or along a column of graphs, one can evaluate how ZPL's op-

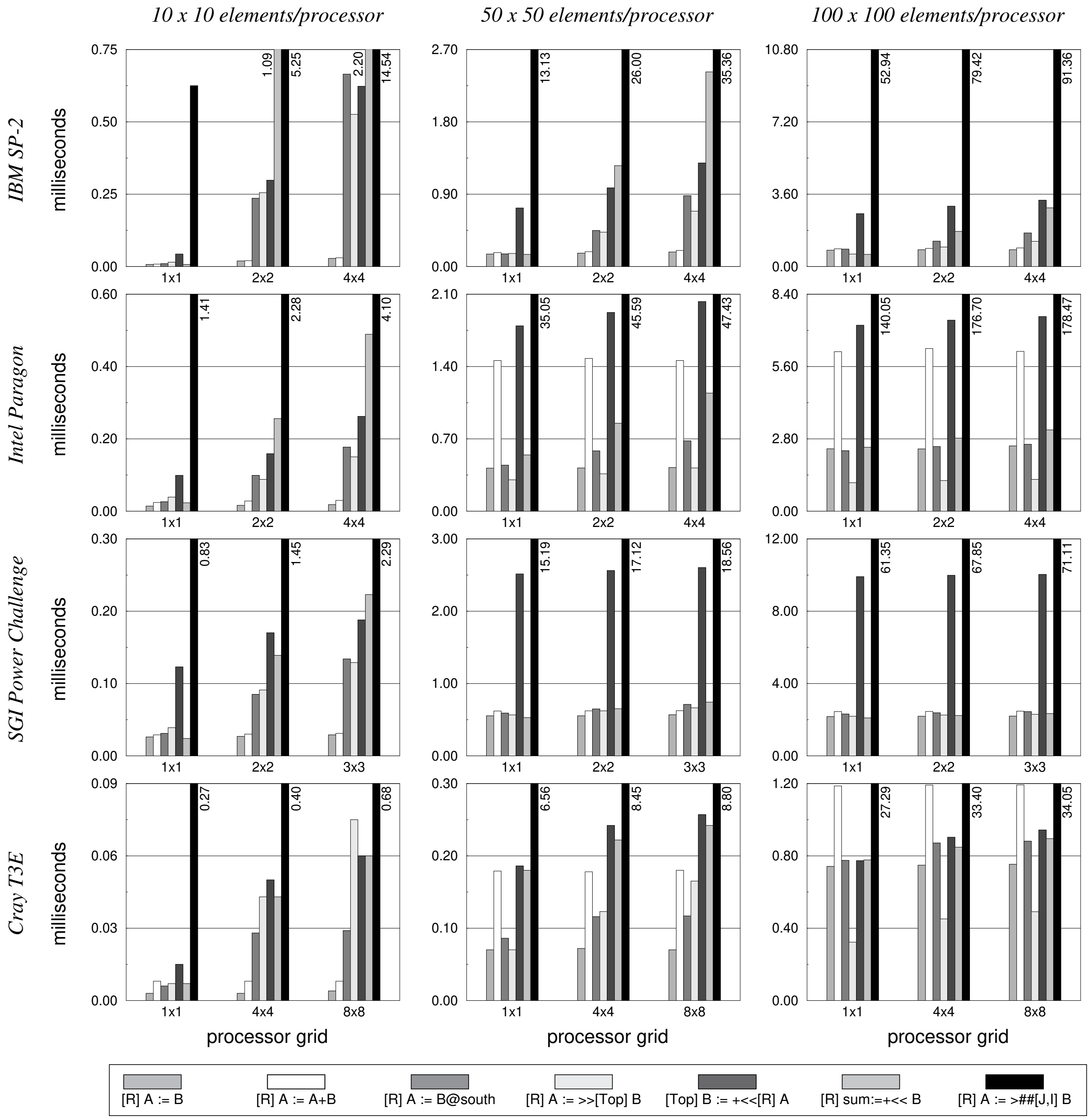

Figure 8. Measured performance of ZPL operations. Each graph shows the execution times on three processor configurations. Each column of graphs represents a per-processor problem size, and each row represents a machine.

erators scale with the number of processors, scale with the problem size, and perform across architectures.

Although numerous observations can be made from these graphs, we list just a few to highlight performance issues related to our analysis in the previous section. To begin with, the WYSIWYG model indicates that the array copy and array addition statements should require no communication and therefore scale perfectly as the number of processors increases. This is demonstrated to be true by the consistent execution times of the first two bars within each graph.

Comparing the array translation (bar 3) with the array

statement	communication complexity	communication volume	flops	elements referenced
[R] A := B	0	0	0	$2n^2$
[R] A := A+B	0	0	n^2	$2n^2$
[R] A := B@south	1	n	0	$2n^2$
[R] A := >>[Top] B	$\log p$	n	0	$n^2 + n$
[Top] B := +<<[R] A	$\log p$	n	$n^2 - n$	$n^2 + n$
[R] sum := +<< B	$2 \log p$	1	n^2	n^2
[R] A := >##[J,I] B	p^2	n^2	0	$4n^2$

Table 2. Summary of expected statement costs per processor. The first three columns are as in Table 1. *Elements referenced* gives the number of distinct array references required by each statement.

copy (bar 1), we see that it tends to be more expensive as expected, due to the required communication. The difference is least significant for the single processor cases where there is no communication. Note that for the SP-2, Paragon, and Power Challenge, where only a modest number of processors were available, increasing the processor grid size results in an increased execution time in spite of the expected $O(1)$ communication complexity. This is due to the fact that each processor in a 2×2 grid is either a source or a destination of communication, while larger grids require some processors to do both. On the T3E where more processors are available, it can be seen that the time to perform the translation levels off, confirming the predicted $O(1)$ communication complexity. On all platforms, the time required to perform the n^2 assignments tends to dominate the $O(n)$ communication as the problem size grows, reducing the performance gap between the two types of assignment.

As predicted, the flood operator's performance (bar 4) becomes slower as the number of processors increases. Looking at how the flood operator scales with the number of processors, note that on the Cray T3E where large processor sets were available, the flood does not level off as the @ operator did. This is consistent with its $O(\log p)$ communication complexity as predicted in Table 1.

Examining the partial and full reductions (bars 5 and 6), we see that they similarly match their predicted communication complexity, becoming more expensive as the the number of processors increases. In addition, note that on smaller problem sizes where communication is less dominated by the $O(n^2)$ computation, the full reduction does not scale with the number of processors as well as the partial reduction. This is evidence of the fact that the full reduce requires a broadcast in addition to the reduction over *all* the processors, whereas the partial reduction simply requires a reduction over a single column of processors.

Finally, as predicted, the scatter-based matrix transposition (bar 7) consistently proves to be significantly more expensive than the other operators, generally costing an order of magnitude more than the next most expensive statement.

There are a few results that may seem surprising at first glance. For instance, why are the floods so much cheaper than other statements on so many configurations? And why do full reductions often outperform partial reductions when they require more communication? The proper response to these questions is to notice that our analysis up to this point has been concerned with parallel performance issues to the exclusion of the scalar component of the performance model—in particular, the memory hierarchy.

To rectify this problem, let us consider the statements more thoroughly. Table 2 presents a summary of the statements' costs, using an analysis similar to Table 1. We dispense with the big-O notation in order to display constants that may be relevant for this analysis. In addition, we inspect the regions and arrays used in each statement to determine the number of distinct elements that it references. For example, the array copy statement accesses all elements in A and B over R and therefore references $2n^2$ elements.

With this more complete analysis, it becomes apparent that the memory hierarchy accounts for these seeming anomalies. For example, the flood is shown to reference fewer array elements than all other statements except the reductions (which require $O(n^2)$ additions). Thus, its modest memory requirements and lack of floating point operations most likely account for its better relative performance. Memory requirements can also explain the disparity between the partial and full reductions since the full reduction can accumulate its values into a scalar rather than an array, resulting in greater locality. The fact that the observed behaviors are amplified on larger problem sizes and the Paragon (which has a smaller cache) serves as further confirmation that the memory hierarchy is responsible.

It is important to realize that the contents of Table 2 can be generated simply by examining the statements individually, reasoning about the type of communication they require, and calculating the number of array references, all within the context of the ZPL performance model. No fur-

```
direction right = [0,1];
          below = [1,0];

region R = [1..n,1..n];

var A,B,C:[R] double;
```

(a) Common declarations for $n \times n$ matrix multiplication

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

(b) SUMMA matrix multiplication

```
[R] begin
      /* initialize matrices by skewing */
      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;
      /* compute first product and iterate */
      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;
```

(c) Cannon's matrix multiplication

Figure 9. Two algorithms for $n \times n$ matrix multiplication in ZPL and their common declarations.

ther information is needed to produce these estimates, and yet they can accurately characterize performance. In the next section we use this identical technique to evaluate matrix multiplication algorithms.

5.2. Matrix multiplication

Although analyzing the performance of individual ZPL statements is instructive, the real test of the WYSIWYG performance model is in evaluating whole algorithms. In Figure 9, we give two ZPL implementations for dense matrix-matrix multiplication: SUMMA [17] and Cannon's Algorithm [3]. 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 and accumulating them into the C matrix. The skewing and cyclic shifts are achieved using ZPL's *wrap* operator within an *of region*—another form of point-to-point communication in ZPL.

Analyzing these algorithms asymptotically reveals that they both perform $O(n^3)$ computation and $O(n)$ communications. However, using the WYSIWYG performance model as we did in the previous section, we can perform a more precise evaluation.

Table 3 summarizes the results of this analysis. The first column shows that the initialization step of Cannon's algorithm requires it to perform twice as many communications than the SUMMA algorithm overall. Although the communication in Cannon's algorithm is more scalable due to its $O(1)$ communication complexity, it is not obvious that this will be sufficient to make up for the factor of two difference in the number of communications. Looking at the memory footprint of each algorithm, we see that the implementation of Cannon's algorithm touches far more memory than SUMMA. This is due not only to its initialization step, but also because it accesses every element of all three matrices while performing the shifts in its main loop. In contrast, each iteration of SUMMA only has to reference n elements of the A and B matrices to perform the floods.

Based on this analysis, we can hypothesize that SUMMA will tend to outperform Cannon's algorithm, especially on the larger problem sizes where memory is expected to become the bottleneck.

To test our hypothesis, we ran both programs 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 10 shows our results and verifies that SUMMA outperforms Cannon's algorithm in all cases. Performing the equivalent experiment in HPF, Ngo demonstrated that not only is it virtually impossible to predict the relative performance of these algorithms by looking at the HPF source, but also that neither algorithm consistently outperforms the other across all compilers [11]. ZPL's WYSIWYG performance model makes both source-level evaluation and portable performance a reality.

algorithm	number of communications	communication complexity	communication volume	flops	elements referenced
Cannon	$4n$	1	n	$2n^3 - n^2$	$n \cdot (2\frac{n^2}{2} + 3n^2)$
SUMMA	$2n$	$\log p$	n	$2n^3$	$n \cdot (n^2 + 2n)$

Table 3. Summary of the expected costs of the matrix multiplication algorithms per processor. *Number of communications* indicates the number of times a communication (cyclic shift or flood) is used by the algorithm. The other communication statistics are reported per communication. All other columns are as in Table 2.

5.3. Summary

In our experiments, we see that ZPL's WYSIWYG performance model allows us to reason about the execution of a program without having a specific machine in mind. It should be noted that, as in the sequential domain, ZPL's performance model does not yield exact information about a program's running time. This would be impossible. However, it does allow programmers to be aware of the implications of their implementation decisions by making the mapping of their code to a parallel machine explicit. As with sequential languages, a programmer's intuition may be inaccurate due to the complexity of modern machines or the impact of compiler optimizations (*e.g.*, pipelining communication or removing redundant communications). However, we expect that by revealing the mapping of ZPL to parallel machines, both through its performance model and its syntactic cues, the programmer will be better equipped to confront these challenges.

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. Such models are particularly crucial in the parallel domain where the cost of language features may vary greatly in magnitude (*e.g.*, local versus remote memory access). Yet, ZPL is the first high-level parallel programming language to present a performance model that allows users to see the target machine through their code. This is done by cleanly mapping the language to the hardware via the CTA machine model, and it gives programmers the ability to reason about a code's relative performance. Moreover, operators that induce communication are clearly visible in ZPL syntax. We refer to this as ZPL's WYSIWYG performance model. We have given an explanation of how the language achieves it, demonstrated how programmers can use it, and experimentally verified that a diverse collection of parallel machines respect it.

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

References

[1] V. S. Adve, J.-C. Wang, J. Mellor-Crummey, D. A. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95*, December 1995.

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

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

[4] M. D. Dikaiakos, C. Lin, D. Manoussaki, and D. E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *Ninth International Conference on Supercomputing*, pages 365–374, July 1995.

[5] R. Friedman, J. Levesque, and G. Wagenbreth. *Fortran Parallelization Handbook, Preliminary Edition*. Applied Parallel Research, April 1995.

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

[7] E. C. Lewis, C. Lin, L. Snyder, and G. 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.

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

[9] C. Lin and L. Snyder. SIMPLE performance results in ZPL. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[10] 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.

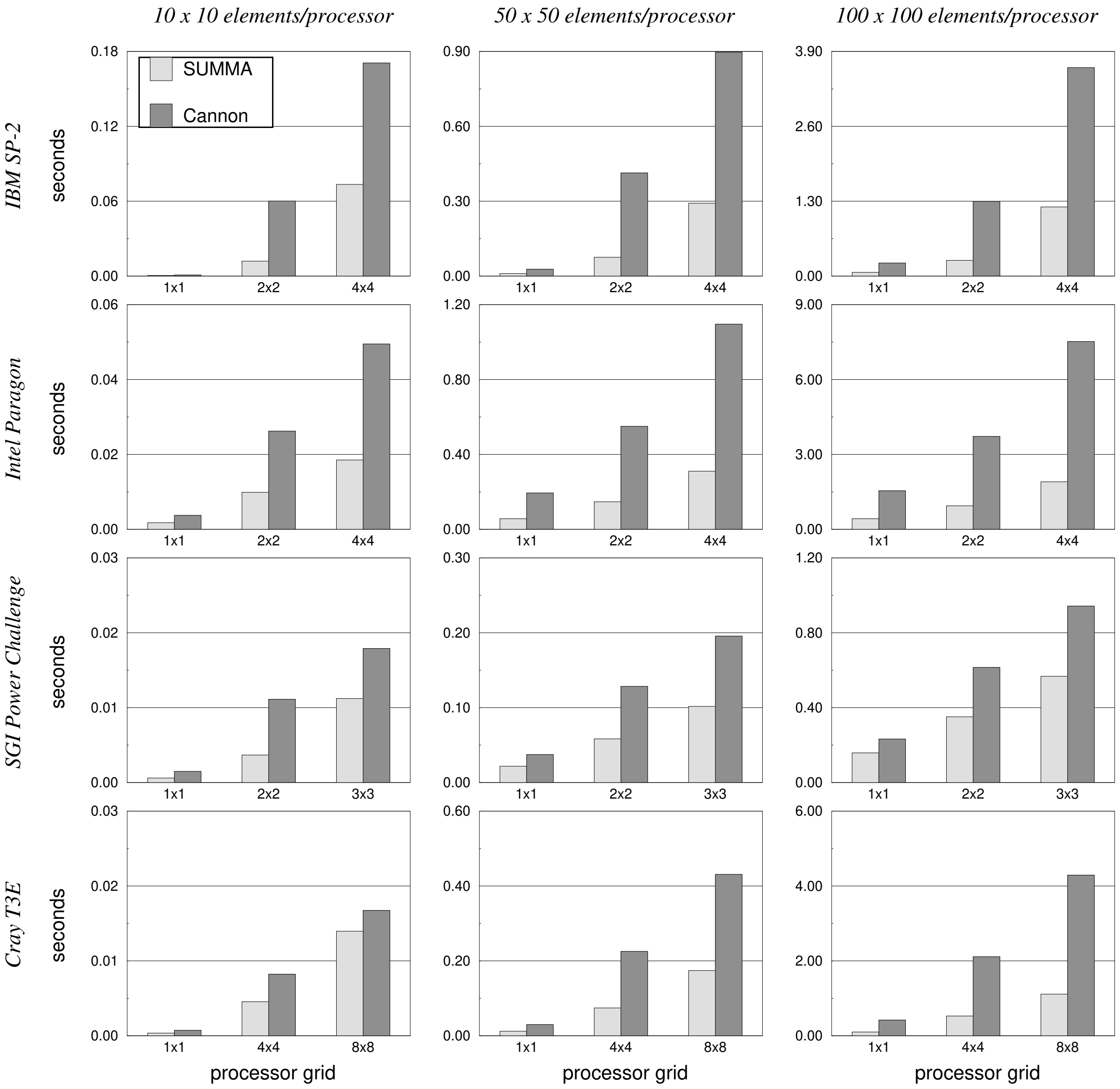

Figure 10. Performance of SUMMA and Cannon's algorithm for matrix multiplication in ZPL. Each graph shows the execution times on three processor configurations. Each column of graphs represents a per-processor problem size, and each row represents a machine.

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

[12] T. A. Ngo, L. Snyder, and B. L. Chamberlain. Portable performance of data parallel languages. In *SC97: High Performance Networking and Computing*, November 1997.

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

[14] W. Richardson, M. Bailey, and W. H. Sanders. Using ZPL to develop a parallel Chaos router simulator. In *1996 Winter Simulation Conference*, December 1996.

[15] L. 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.

[16] L. Snyder. *The ZPL Programmer's Guide*. MIT Press (in press—available at publication date at ftp://ftp.cs.washington.edu/pub/orca/docs/zpl_guide.ps), 1998.

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