The High-Level Parallel Language ZPL Improves Productivity and Performance

Bradford L. Chamberlain*† Sung-Eun Choi‡ Steven J. Deitz* Lawrence Snyder*

*University of Washington
Seattle, WA 98195
{brad,deitz,snyder}@cs.washington.edu

†Cray Inc.
Seattle, WA 98104
bradc@cray.com

‡Los Alamos National Laboratory
Los Alamos, NM 87545
sungeun@lanl.gov

Abstract

In this paper, we qualitatively address how high-level parallel languages improve productivity and performance. Using ZPL as a case study, we discuss advantages that stem from a language having both a global (rather than a per-processor) view of the computation and an underlying performance model that statically identifies communication in code. We also candidly discuss several disadvantages to ZPL.

1. Introduction

In the spring of 2003, we encountered a curious bug in one of the NAS parallel benchmarks. To evaluate the scalability of ZPL, we were comparing our ZPL implementation of the NAS CG benchmark against the provided Fortran+MPI implementation on an increasing power-of-two number of processors of a new 1024-node cluster at Los Alamos National Laboratory (LANL). Both implementations ran flawlessly on up to 512 processors but, on our first 1024 processor run, the Fortran+MPI failed to verify correctly even as the ZPL worked. A day after we reported the failed verification to NAS, they were able to produce identical erroneous results on an IBM SP.[1] It wasn't a strange interaction between LANL's experimental cluster and ZPL, but rather a bug in the long-standing Fortran+MPI benchmark...

* * *

ZPL is a high-level parallel programming language developed at the University of Washington. Our implementation is based on a compiler that translates ZPL programs to C code with calls to MPI, PVM, or SHMEM, as the user chooses. Since the first release of this compiler in 1997, there have been significant improvements as we have evolved the language. This paper discusses some of the lessons we have learned over this time.

Like Co-array Fortran, High Performance Fortran, Titanium, Unified Parallel C and other parallel languages, ZPL offers scientists who are frustrated by MPI a much improved parallel programming experience. The anecdote above, which we will come back to later in this paper, illustrates this point and is the sort of issue we will discuss in this paper. The point of this anecdote is not that the provided Fortran+MPI benchmark was poorly written. Indeed, the NAS benchmarks are well-known for being well-written and highly-optimized. The point, as we will see later, is that the high-level nature of ZPL virtually eliminates a wide class of parallel programming bugs, thus making parallel programming easier.

Focusing on ZPL, this paper addresses how high-level parallel languages improve both productivity and performance. Throughout this paper, we will present anecdotes, code segments, and qualitative arguments as evidence of this improvement. The purpose of this paper is not to advertise ZPL but rather to encourage researchers to explore the space of language abstractions which ZPL champions.

This paper is organized as follows. In the next section, we characterize the design space of ZPL. No introduction to the language is offered; the interested reader is instead referred to the literature [4, 21]. In Section 3, we examine aspects of ZPL that increase productivity and performance. In Section 4, we discuss limitations of ZPL and, in Section 5, we conclude.

2. Characterizing ZPL

Figure 1 shows C+MPI and ZPL implementations of a trivial benchmark. The idea behind the benchmark is to iteratively replace each element in a 1D array with the average of its two neighboring elements until the change between the values in the array on successive iterations is small. Though admittedly contrived, the codes effectively illustrate two important characteristics of ZPL.

First, ZPL is a global-view parallel language. The programmer writes code that largely disregards the processors that will execute it. Thus array `A` is declared based on the

[1] Personal Communication. Rob F. Van der Wijngaart. April 9, 2003.

```c
  #include <stdio.h>
  #include <stdlib.h>
| #include "mpi.h"

  int n;
  double *A, *Tmp;
  const double epsilon = 0.000001;

  int main(int argc, char* argv[]) {
    int i, iters;
    double delta;
|   int numprocs, rank, mysize;
|   double sum;

|   MPI_Init(&argc, &argv);
|   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
|   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (argc != 2) {
      printf("usage: line n\n");
      exit(1);
    }
    n = atoi(argv[1]);
|   mysize = n * (rank + 1) / numprocs -
|            n * rank / numprocs;
    A = malloc((mysize+2)*sizeof(double));
    for (i = 0; i <= mysize; i++)
      A[i] = 0.0;
|   if (rank == numprocs - 1)
      A[mysize+1] = n + 1.0;
    Tmp = malloc((mysize+2)*sizeof(double));
    iters = 0;
    do {
      iters++;
|     if (rank < numprocs-1)
|       MPI_Send(&(A[mysize]), 1, MPI_DOUBLE, rank + 1,
|                1, MPI_COMM_WORLD);
|     if (rank > 0)
|       MPI_Recv(&(A[0]), 1, MPI_DOUBLE, rank - 1,
|                1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
|     if (rank > 0)
|       MPI_Send(&(A[1]), 1, MPI_DOUBLE, rank - 1,
|                1, MPI_COMM_WORLD);
|     if (rank < numprocs-1)
|       MPI_Recv(&(A[mysize+1]), 1, MPI_DOUBLE, rank + 1,
|                1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      for (i = 1; i <= mysize; i++)
        Tmp[i] = (A[i-1] + A[i+1]) / 2.0;
      delta = 0.0;
      for (i = 1; i <= mysize; i++)
        delta += fabs(A[i] - Tmp[i]);
|     MPI_Allreduce(&delta, &sum, 1, MPI_DOUBLE,
|                   MPI_SUM, MPI_COMM_WORLD);
|     delta = sum;
      for (i = 1; i <= mysize; i++)
        A[i] = Tmp[i];
    } while (delta > epsilon);
|   if (rank == 0)
      printf("Iterations: %d\n", iters);
|   MPI_Finalize();
  }
```

(a)

```
program line;

config var
  n : integer = 6;

region
  R = [1..n];
  BigR = [0..n+1];

direction
  east = [1];
  west = [-1];

var
  A, Tmp : [BigR] double;

constant
  epsilon : double = 0.000001;

procedure line();
var
  iters : integer;
  delta : double;
begin
  [BigR] A := 0;
  [n+1] A := n + 1;
  iters := 0;
  [R] repeat
    iters += 1;
    Tmp := (A@east + A@west) / 2.0;
    delta := +<< abs(A - Tmp);
    A := Tmp;
  until delta <= epsilon;
  writeln("Iterations: %d":iters);
end;
```

(b)

Figure 1. A trivial benchmark written to compare (a) C+MPI and (b) ZPL. This benchmark measures how many iterations are needed for an array to reach a fixed point. The user sets `n`, the size of the problem, at the command line. The program starts by initializing the array to zero with left and right borders set to `0` and `n+1` respectively. On each iteration, the elements in the array are replaced by the average of their two neighbors. The program terminates when the sum of the changes is less than the fixed constant `epsilon`. The number of iterations is reported. The use of the italics in the C+MPI code indicates the changes that are necessary to make when parallelizing the sequential language C using MPI. The vertical bars on the left indicate new lines of code; in addition, `mysize` replaces occurrences of `n`.

global bound of `n + 1`. In contrast, C+MPI is a local-view parallel language, and array `A` is declared based on per-processor bounds of `mysize + 2`. The highlighted parts of the C+MPI code show the changes that needed to be made from a sequential C code and the burden that is placed on the local-view programmer. In addition to using local bounds to size arrays on a per-processor basis, interprocessor communication must be explicitly managed in tedious detail.

It is important to note the difference between global-view languages such as HPF [14] and ZPL and local-view languages with global address spaces such as Co-array Fortran [19], Titanium [24], and UPC [3]. The latter are sometimes referred to as fragmented languages because they require programmers to divide the expression of their computation between the processors in an SPMD style of programming. They are significantly easier to use than C+MPI because of their global address space but, unlike global-view languages, they require the user to manage low-level synchronization.

Second, despite its global view, communication is explicit in ZPL. The details of communication are managed by the compiler, but the ZPL programmer is readily aware of where communication is induced. This provides a simple, but powerful performance model called the what-you-see-is-what-you-get (WYSIWYG) performance model [5]. The only communication in the simple ZPL program in Figure 1 is induced by the *at operator* (`@`), which shifts data across processor boundaries, and the *reduce operator* (*op*<<), which determines the sum of values distributed across all the processors.

ZPL is the only language to offer both properties. In the most well-known global-view language, High Performance Fortran, the programmer achieves parallelism by supplying directives of distribution and parallel computation. As a parallel extension to Fortran 90, its easy to reason about what is computed. However, communication requirements for a given statement are invisible in the syntax, thus making it a challenge for both programmers and compilers to optimize communication. On the other hand, local-view languages tend to make communication explicit but at the expense of the global view of computation.

3. Advantages of ZPL

This section is composed of seven parts, each of which addresses some aspect of the advantage of high-level parallel programming languages. The first two parts look at advantages of having a global view of computation; it makes parallel programming easier and provides for more general parallel programs. The next two parts focus on language abstractions; structural abstractions improve programmability while orthogonal abstractions make it easy to tune parallel codes. The fifth part discusses how a high-level performance model makes it easy to maintain fast code, and the sixth part discusses how high-level languages stop programmers from over-specifying code and keeping the compiler from making effective optimizations. In the final part, we show some performance results which suggest that the bottom line of high-performance is still achievable.

3.1. Global-view languages make parallel programming easier

As a case in point, we will elaborate on our introductory story. The NAS Parallel Benchmarks (NPB) [1] have long served as a way for us to evaluate the performance of ZPL. These benchmarks were designed to assist in evaluating the performance of parallel supercomputers. Derived from Computation Fluid Dynamics (CFD) applications and implemented in Fortran or C and MPI, they are "intended to be run with little or no tuning, [and] approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer." [1] This statement is actually too modest. These benchmarks are highly-tuned and represent the upper end of achievable performance with a message-passing library. The benchmarks are well-written, stable programs that garner a substantial degree of respect in the high-end computing community.

The NAS CG benchmark estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The bug we encountered on 1024 processors was all the more curious because of the relative age of CG. Not only did the CG benchmark run flawlessly on up to 512 processors, but it had also been used for years in evaluating parallel systems. The 1024-processor bug was found to be in the initialization and was fixed by the NAS team within a week.[2] What happened was that an array used later in the computation and treated as scratch in the initialization was sized based on the problem size divided by the number of processors. As the number of processors grew, it became too small to fit the initialization data.

Because of the local view of computation, the array was sized based on local per-processor bounds. Given a global-view language like ZPL, that same array would be sized based on the global bounds. Had the programmer made a similar array sizing mistake in ZPL, the benchmark would have failed on any number of processors, not just when the number became large. Thus global-view languages make the development of working parallel programs easier.

Global-view languages make parallel programming easier for many other reasons too. Here is another story of using ZPL.[3] A professional programmer at HP with over "5 years of experience ... doing regular product development

[2] Personal Communication. Haoqiang Jin. April 10, 2003.
[3] Personal Communication. George Forman. December 5, 2003.

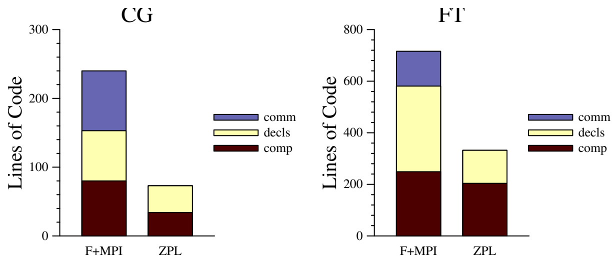

Figure 2. Charts showing line counts of the Fortran+MPI and ZPL implementations of the NAS CG and FT benchmarks. The counts are subdivided into lines used for communication, declarations, and computation. These counts are being reprinted from a previous paper which can be consulted for more detailed information on the ZPL implementations of these benchmarks [11].

work" wrote a code to test clustering algorithms. "It was a code he cared about and had cultivated for many research experiments over the course of a year+ for testing different kinds of clustering. It was tuned for performance, because he had to do many runs for research significance." The core of the computation was 355 lines but, by describing it in Matlab, the programmer was able to explain it to a second HP programmer easily.

This second programmer, having worked on ZPL in the past, was eager to try writing the code in ZPL. Sequential runs took copious amounts of time, and both researchers expected they could achieve near linear speedup. In roughly 6 hours, the second programmer had the ZPL code working. It was 73 lines. Not only did it show nearly linear speedup [25] but, to the C programmer's surprise, its sequential performance was better than that of the optimized C code. This application helped the HP researchers demonstrate that clustering code across continents, even with bad network latency, is better than shipping data to local clusters [13].

Though lines of code is not an ideal metric for evaluating a parallel programming language, it does provide some quantitative measure of programmability. Figure 2 counts the number of lines of code in the timed portions of the NAS CG and FT benchmarks. The ZPL codes require less than half the number of lines used to write the equivalent Fortran+MPI. Inspecting the codes reveals similar complexities and simplifications as in this paper's examples, yet on a larger scale. This is a testament to how much easier it is to use ZPL.

3.2. Global-view languages provide for more general parallel programs

To keep MPI programs from requiring even more lines of code, they are often written with assumptions about the problem size or the number of implementing processors. For example, these may both be required to be powers of two. For the provided Fortran+MPI implementations of the NAS CG, FT, and MG benchmarks, the number of processors is required to be a power of two. In contrast to this restriction to the Fortran+MPI versions, the ZPL program can run on a non-power-of-two number of processors.

These assumptions in MPI are not surprising. They greatly simplify the implementation and permit optimizations that would not be possible in the more general case. However, there are times when one wants to run on a non-power of two number of processors. For example, given a 64 processor machine, scientists might not want to wait for a 16 processor job to finish if their programs could run on the 48 available processors sooner. Also, due to budget constraints, machines are often not composed of a power-of-two number of processors.

Modifying an MPI code to introduce such flexibility can often impact all aspects of the code. Moreover, as illustrated in Figure 6 and seen in the literature [7], running the more general ZPL version of the CG, FT, and MG benchmarks on a non-power-of-two number of processors results in improvements over the next smallest power-of-two number of processors for the Fortran+MPI benchmark.

3.3. Structural abstractions improve programmability

Sparse problems comprise a challenging and crucial class of computation in high-end computing. Yet it is important to remember that the sparsity of an array or matrix relates to its potential for optimized implementation rather than the fundamental operations it supports. As an example, matrix-vector multiplication is a mathematical operation whose definition is independent of whether the matrix operand is sparse or dense; its sparsity merely provides an opportunity for reducing the computational and storage overheads of the operation.

Most languages fail to support abstractions for sparse data structures, placing the effort of exploiting sparsity on the programmer rather than the tools. Programmers must build their own data structures to represent sparse arrays and this change in representation forces a corresponding change to the computation itself. As an example, consider the Fortran codes in Figure 3(a) which implement matrix-vector multiplication for a dense array and for a sparse array using compressed row storage. Note that the change from sparse to dense is pervasive in the code. The 2D array a

DENSE | SPARSE

(a)

```
real p(n), w(n)
real a(n,n)

do j = 1,numrows
  sum = 0.d0
  do k = 1,numcols
    sum = sum + a(j,k)*p(k)
  enddo
  w(j) = sum
enddo
```

```
real p(n), w(n)
real a(nnz)
integer colidx(nnz)
integer rowstr(n)

do j = 1,numrows
  sum = 0.d0
  do k = rowstr(j),rowstr(j+1)-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  w(j) = sum
enddo
```

(b)

```
region R = [1..n, 1..n];
       RowVect = [*, 1..n];
       ColVect = [1..m, *];

var M: [R] double;
    V: [RowVect] double;
    S: [ColVect] double;

[ColVect] S := +<<[R] (M * V);
```

```
region R = [1..n, 1..n] where /* pattern */;
       RowVect = [*, 1..n];
       ColVect = [1..m, *];

var M: [R] double;
    V: [RowVect] double;
    S: [ColVect] double;

[ColVect] S := +<<[R] (M * V);
```

Figure 3. Dense and sparse implementations of matrix-vector transpose in (a) Fortran+MPI and (b) ZPL

becomes a 1D array of values with two integer vectors to provide directory information. This forces the inner loop to be restructured to iterate properly over the directory, index a, and index into p using an indirect index. This represents a substantial modification to the code considering that the mathematical operation being expressed has not changed. The problem is exacerbated in parallel codes where communication code must also be rewritten to deal with sparse structures.

In contrast, ZPL supports sparse arrays and matrices as a fundamental concept, allowing programmers to specify an array's sparsity as part of the declaration of its size and shape [8]. This results in minimal impact on the computation itself. Consider the ZPL implementations of sparse and dense matrix-vector multiplication in Figure 3(b). By isolating the impact that such a simple conceptual change has on the code, the programmer can easily switch between sparse and dense representations with little penalty. For example, in the NAS MG benchmark, the input array V is truly sparse, containing only 20 non-zeroes in its 512^3 elements for class C. Using a sparse representation can improve the space and computational costs associated with V, yet making this change requires significant effort in most languages and as a result, most implementations do not bother. In ZPL the change is trivial, reducing the overall memory footprint of the program by $1/3$. As a second example, the NAS FT benchmark checks its results by taking a sparse walk through a dense array. Representing this subset of values directly using a sparse region is a simple change in ZPL and improves performance by making the parallelism more explicit.

By separating the specification of sparsity from its use in computation, the compiler is also given increased flexibility in its choice of sparse data structure implementations. The ZPL compiler automatically tunes its sparse representation based on the requirements dictated by its usage in the code [4]. One could furthermore imagine allowing the user to specify a preferred sparse data structure as part of the array's declaration. In conventional languages where the user must manage sparsity explicitly, such changes tend to affect every line of code that refers to the array, violating the general principle of separating data structure from algorithm.

3.4. Orthogonal abstractions make it easy to tune parallel codes

Significant changes in performance can be realized by fine-tuning a parallel code after it is written. For example, a programmer could want to change the ratio between the number of processors in the column and row dimensions of a 2D processor grid. In the NAS CG benchmark, this results in improved performance when the data-to-processor ratio is large. In the Fortran+MPI implementation, this is a difficult change but, in the ZPL implementation, it is trivial.

Figure 4 shows the code involved in transposing a row to a column in Fortran+MPI and ZPL. Because the row and column arrays are replicated across their dimension of the transpose array, it makes sense, for performance of the transpose, to use a 1:1 or 2:1 row-to-column processor layout in the 2D processor grid. The 1:1 ratio is always ideal,

```fortran
if( l2npcols .ne. 0 ) then
  call mpi_irecv(q, exch_recv_length,
                 dp_type, exch_proc, 1,
                 mpi_comm_world, request, ierr)
  call mpi_send(w(send_start), send_len,
                dp_type, exch_proc, 1,
                mpi_comm_world, ierr)
  call mpi_wait( request, status, ierr )
else
  do j=1,exch_recv_length
    q(j) = w(j)
  enddo
endif
```

(a)

```
[Row] W := P#[Index2, srcindex];
```

(b)

Figure 4. NAS CG transpose code in (a) Fortran+MPI and (b) ZPL.

but if we want to run on an odd power-of-two number of processors, e.g. 8, then sometimes we need to use a 2:1 ratio, e.g. a 4×2 processor grid. In these cases, the communication pattern is one-to-one. Each processor needs to send data to only one processor and needs to receive data from only one processor.

The Fortran+MPI code, because of its low-level of abstraction, cannot keep the processor grid orthogonal to the computation. Thus the one-to-one communication pattern is unyielding. In ZPL, on the other hand, if the processor grid has a 2:1 or 1:1 ratio, the one-to-one communication pattern is achieved, but the processor grid is not restricted to having this ratio. This is useful for the part of the code that implements the sparse computation. It turns out, for the sparse computation, that a 1:2 or even a 1:4 ratio, improves the performance of this part of the code. If the data-to-processor ratio is high then the overall performance of the code improves since the sparse computation, rather than the transpose, is the bottleneck.

3.5. A high-level performance model makes it easy to maintain fast code

Wavefront computations are common in scientific applications. A wavefront computation is one in which the value of each data element is dependent on one or more values computed in previous iterations of the loop nest. Though inherently serial, *pipelining* is a well known, but tedious, technique for efficient parallelization of wavefront computations [9, 23, 15].

The Accelerated Strategic Computing Initiative's (ASCI) SWEEP3D benchmark solves a three dimensional neutron transport problem. Figure 5 compares the ASCI Fortran+MPI and ZPL implementations of the core computation. The Fortran version is simplified for improved clarity. As always, the reduction in code size is dramatic (over three times). Here we will focus on the pipelining itself.

The Fortran version assumes that the problem is only distributed over two (i and j) of the three dimensions. Consequently, the k dimension is treated differently than the other two when the computation is actually the same. For example, the first twelve lines in the main loop deal with initializing the *inflows*. Notice a subtlety in the initialization of the i- and j-inflows; they are actually performed within the pipeline loop (`kk`). In other words, there is communication in the *inner loop* of the computation. This has a profound performance implication, yet the code looks nearly the same as a very simple data parallel array operation. To fix this problem, the `kk` loop should just be moved down below the inflow initialization.

The wavefront computation in the ZPL version begins with the `interleave` keyword. This forces the statements within its scope to execute in an interleaved manner by fusing the statements into the same scalarized loops. The encompassed "prime at" references `'@` indicate to the programmer *and the compiler* that the operations within the statement block may require serialization of the computation. The compiler can and does implement pipelining as described above, thus relieving the programmer from worrying about the details of the implementation including the *tile size* to use for pipelining. These special prime at references explicitly indicate that those referenced values are dependent on values computed in previous iterations, resulting in serialization. Notice that in the Fortran version, indexing withstanding, it is not at all clear which references cause the serialization. Moreover, if these references were to change such that no serialization were necessary, the programmer should explicitly move the i- and j-inflow initialization outside the `kk` loop for fully parallel execution. Not doing so would not necessarily result in an incorrect program, just an inefficient one.

3.6. High-level languages stop programmers from over-specifying code

Though the key advantage of a high-level language for productivity is that it frees the programmer from the heavy burden of writing low-level implementing code, there is a further advantage: The compiler is freed from having to use that implementation. That is, low-level code over-specifies

```
do mo = 1, mmo  !outer angles loop (batches of mi angles)
  <initialize K—inflows —— triply nested loop>
  do kk = 1, kb ! outer planes loop (batches of mk—planes)
    if (ew\_rcv .ne. 0) then ! I—inflows for block
      <receive boundary values>
    else
      <initialize I—inflows —— triply nested loop>
    endif
    if (ns\_rcv .ne. 0) then ! J—inflows for block
      <receive boundary values>
    else
      <initialize J—inflows —— triply nested loop>
    endif
    ! JK—diagonals with MMI pipelined angles
    do idiag = 1, jt+nk—1+mi—1
      do jkm = 1, ndiag
        do i = i0, i1, i2 ! I—line recursion
          ci = mu(m)*hi(i)
          dl = ( sigt(i,j,k) + ci + cj + ck )
          dl = 1.0 / dl
          ql = phi(i) + ci*phiir + \
                        cj*phibj(i,lk,mi) + \
                        ck*phikb(i,j,mi)
          phi(i)  = ql * dl
          phiir          = 2.0d+0*phi(i) — phiir
          phii(i)        = phiir
          phijb(i,lk,mi) = 2.0d+0*phi(i) — phijb(i,lk,mi)
          phikb(i,j,mi)  = 2.0d+0*phi(i) — phikb(i,j,mi)
        end do ! i
        phiib(j,lk,mi) = phiir
      end do ! jkm
    end do ! idiag
    <compute and send outflows>
  end do ! kk
end do ! mo
```

(a)

```
[R] begin
[lasti of R] phiib := 0.0;  —— boundary i inflow
[lastj of R] phijb := 0.0;  —— boundary j inflow
[lastk of R] phikb := 0.0;  —— boundary k inflow
      ci := mu * hi;
      cj := eta * hj;
      ck := tsi * hk;
      dl := 1.0 / (Sigt + ci + cj + ck);
      interleave
        ql := phi + ci*phiib'@lasti +
                    cj*phjb'@lastj +
                    ck*phikb'@lastk;
        phi := ql * dl;
        phiib := 2.0*phi — phiib'@lasti;
        phijb := 2.0*phi — phijb'@lastj;
        phikb := 2.0*phi — phikb'@lastk;
      end;
      ——— final i, j, and k outflows
[lasti in R] leakage[1+i3] +=  wmu * phiib * dj * dk;
[lastj in R] leakage[3+j3] += weta * phijb * di * dk;
[lastk in R] leakage[5+k3] += wtsi * phikb * di * dj;
end;
```

(b)

Figure 5. Core computation of the ASCI SWEEP3D benchmark in (a) Fortran+MPI and (b) ZPL.

an implementation, possibly limiting the compiler's ability to optimize.

Many researchers have shown that message passing is often the wrong choice for efficient communications (e.g., [22, 16, 18]). Regardless, most parallel programs written in low-level languages such as C or Fortran use message passing, partly due to the fact that a standard interface exists (MPI) but also to the fact that it is easier to use than other proposed interfaces. In fact, these other interfaces were primarily designed as targets for libraries and compilers for high-level languages rather than programmers.

The reason these alternative communication libraries perform better than message passing libraries is that the exposed interface more closely matches the implementing hardware. For example, Striker et al. [22] showed that the synchronization required of message passing limits performance compared to one-sided communication on the T3D, a machine that provided hardware support for one-sided communication via the SHMEM library [2]. ARMCI [17] generalizes the low-level libraries of modern PC network interface cards (such as Myrinet, Quadrics Elan, and Infiniband) to provide efficient one-sided communication. ARMCI has been shown to perform well on high-performance clusters [18], a platform generally accepted as one for message passing.

Like other high-level parallel languages, programmers do not write interprocessor communication commands in ZPL. Rather, the compiler determines where communication may be required and it inserts into the object code the appropriate calls to the ZPL runtime library. The compiler actually generates calls that describe the non-local data dependences, *not explicit communication calls*. The calls in this interface, called *Ironman* [6], describe four important locations in the object code. Two are for the *destination* (DR/DN), and the other two for the *source* side (SR/SV).

Destination Ready (DR). The locally cached copy of the non-local data will not be read again (until DN). These memory locations on the destination processor are now *ready* to be overwritten with new values.

Source Ready (SR). The values needed on the destination processor have just been written. The source processor is *ready* to transmit the values.

Destination Needed (DN). The locally cached copy of the non-local data will be read. The non-local data is *needed* at the destination.

Source Volatile (SV). The values needed on the destination processor will be overwritten. The values are now *volatile* and must be transmitted by this point.

These Ironman calls are bound to a specific communication library (MPI, SHMEM, etc.) at link-time. This late binding enables the use of the most appropriate communication mechanisms for a given platform *without changing the user's program itself*. For example, for MPI, DR and DN bind directly to `MPI_Irecv` and `MPI_Wait`, and SR and SV bind to `MPI_Isend` and `MPI_Wait`. For a put-based implementation of a one-sided communication library such as SHMEM or ARMCI, SR puts the data from source to destination, DR and DN perform loosely-coupled synchronization with SR, and SV is not needed.

When programmers write MPI message passing code directly, it is the semantics of message passing *not the individual implementations of MPI* that ultimately limit performance. For example, data that is irregularly laid out in memory, must be marshaled and brought together into a contiguous message buffer before it can be sent. However, some libraries (such as SHMEM and ARMCI) and PC networks interfaces (such as Dolphin SCI and Quadrics Elan) expose via their native communication library remote direct memory access (RDMA) to remote addresses. These do not require the extra copy and memory overhead. An implementation of MPI using these facilities has no control over this data marshaling because the code to perform this operation is embedded in the program itself. If a ZPL program is to be run using MPI, then the ZPL libraries for MPI would perform the marshaling; on a machine with efficient RDMA, no marshaling would be performed.

By removing the burden of writing low-level implementing code such as communication calls, the compiler is able to better optimize communication using data dependence information. Moreover, the late binding to the native communication library allows for the most efficient communication library to be used without penalties incurred when using a particular library.

3.7. High-level languages achieve high-performance

Performance is the bottom line of parallel programming. The whole reason to parallelize a code is to make it run faster. If the high-level language hurts either the sequential performance or the program's ability to scale to higher numbers of processors, then it loses its value.

Figure 6 shows the performance of the NAS CG and FT benchmarks in ZPL and Fortran+MPI across three platforms. These platforms are representative of the diversity of machines. The T3E provides a top-of-the-line network for low-latency interprocessor communication whereas the cluster has much higher latency, but faster processors.

Note that the generality of the ZPL implementation is apparent in the 176-processor run on the IBM SP. Here the ZPL code can improve its performance further even though the Fortran+MPI code was not written to run on a non-power-of-two number of processors.

4. Limitations and Evolution

A very reasonable observation to make about ZPL is that for all of its convenient features and abstractions, it does not support arbitrary models of parallel programming. While ZPL's support for parallel computation using sparse and dense multidimensional index sets supports a wide variety of high-end applications, it lacks similar abstractions for other paradigms such as distributed hash tables, graph-based data structures, and nested parallelism. Other desirable features such as user-defined data distributions and task parallelism are only now being added to the language [10, 12].

Our explanation for this lack of generality is one of philosophy. While many languages strive for complete generality from their inception, these languages tend to either provide a very low level of abstraction, to never get all of their features implemented, or to never achieve good performance for more than a narrow range of features. In contrast, our approach has been to start with those facilities we know how to compile well and then add generality to the language as our understanding and experience grow. As a result, we have managed to achieve good performance throughout ZPL's lifetime while keeping the language's concepts elegant and interoperable.

The downside to this approach is that it has taken a long time to acquire the knowledge. At times, ZPL's evolution has been slow and incremental. To reduce the effects of this problem, we are currently in the process of producing an open source release of ZPL in hopes of engaging a broader community in its support and development (previous releases have contained the compiler and runtime binaries without their sources). A private release of the source to colleagues at U. Mass-Lowell has already allowed ZPL to be ported to the unusual Mercury-Race architecture [20].

We also anticipate that the open source release should allow us to support a broader community of parallel programmers, since many potential users in the past have expressed their unwillingness to base their research on a language whose implementation they could not access directly (in part for fear that we would cease to support it in the future). Meanwhile, our evolution of the language progresses as does our enthusiasm for it, particularly as we look beyond the NAS parallel benchmarks to consider more challenging applications that push the limits of what we are currently able to express cleanly and efficiently in ZPL.

5. Conclusions

There is a growing consensus that the bottleneck for productivity in parallel computing lies with the low-level

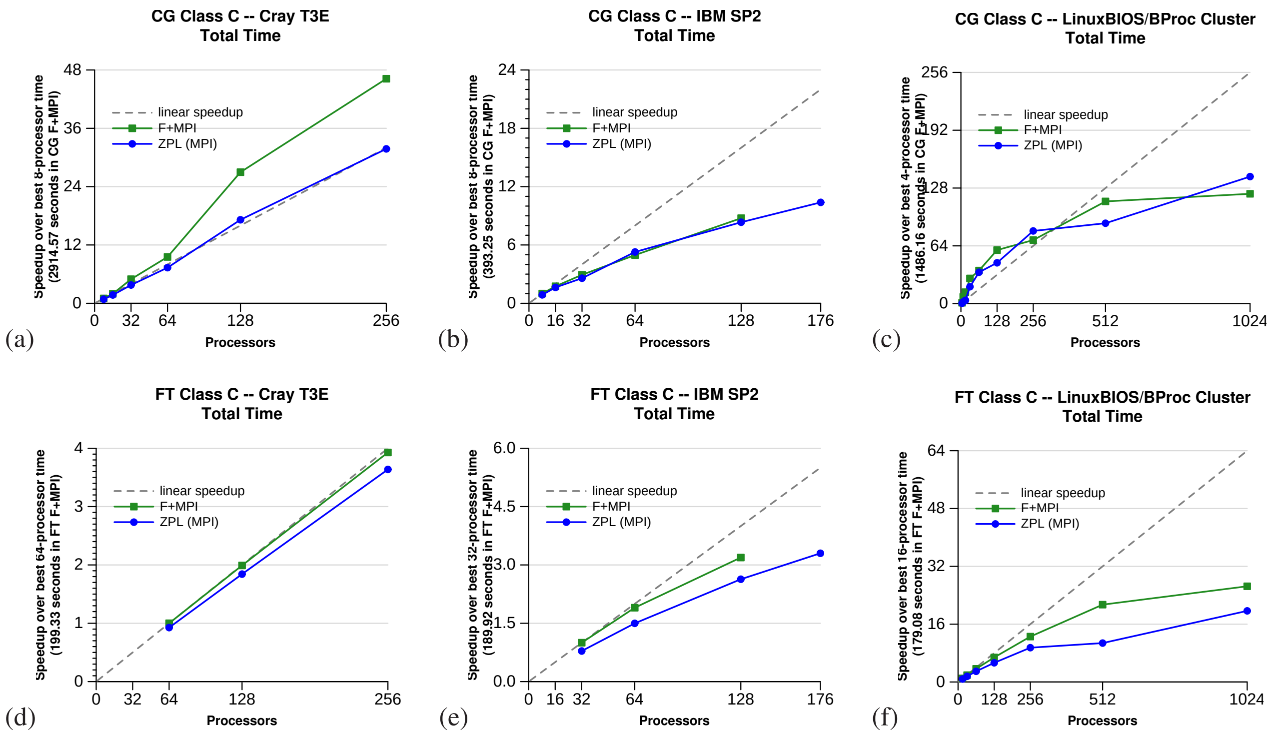

Figure 6. Graphs showing the total speedups of class C of the NAS CG and FT benchmarks across three platforms. The first column shows results on Yukon, a 272 processor Cray T3E with 260 user processors. Each processor is a 450 MHz Alpha processor with 256 MB of memory. The second column shows results on Icehawk, a 200 processor IBM SP with 176 user processors. The SP2 is composed of 44 nodes with 2 GB of memory per node. Each node contains four 375 MHz power3 processors. The third column shows results on up to 1024 processors of Pink, a 2048 processor cluster built with the LinuxBIOS/BProc technology. Pink is composed of 1024 nodes with 2 GB of memory per node. Each node contains two 2.4 GHz Intel Xeon processors. These results are being reprinted from a previous paper which can be consulted for more detailed information on the ZPL implementations of these benchmarks [11].

programming models that users must rely on to express their programs. In this paper, we explored the benefits of languages that provide the programmer with a global view of their computation rather than a local per-processor view. In addition, we discussed why it is beneficial to allow programmers to reason about the implementation of their codes. In ZPL, this is achieved by making all communication requirements visible in the source code, allowing both the programmer and the compiler to reason effortlessly about this bottleneck of parallel computing.

By supporting a global view of computation with communication cues, ZPL provides programmers with a simpler programming model which allows for rapid development, evolution, and tuning. ZPL also makes the programmer's intentions clear to the compiler so that it can implement the code efficiently using a variety of data structures and communication protocols on any modern architecture.

6. Acknowledgments

During the time of this work, the second author was employed by Los Alamos National Laboratory and funded by the Mathematical Information and Computer Sciences (MICS) program of the DOE Office of Science, and the third author was supported by a DOE High Performance Computer Science Graduate Fellowship (HPCSGF). Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. This paper has been recorded under LANL LAUR number LA-UR-03-9521.

The authors would like to thank the many who have contributed to ZPL in the past, and thus made this work possible. This work was supported in part by grants of HPC resources from the Arctic Region Supercomputing Center and

Los Alamos National Laboratory for which we are grateful. Thanks are also due to the anonymous reviewers for their helpful suggestions.

References

[1] D. Bailey, T. Harris, W. Saphir, R. F. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NASA Ames Research Center (NAS-95-020), December 1995.

[2] R. Barriuso and A. Knies. SHMEM user's guide. Technical report, Cray Research Inc., May 1994.

[3] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[4] B. L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[5] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

[6] B. L. Chamberlain, S.-E. Choi, and L. Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

[7] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the ACM Conference on Supercomputing*, 2000.

[8] B. L. Chamberlain and L. Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[9] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–844, 1986.

[10] S. J. Deitz. Renewed hope for data parallelism: Unintegrated support for task parallelism in ZPL. Technical report, University of Washington (2003-12-04), December 2003.

[11] S. J. Deitz, B. L. Chamberlain, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.

[12] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Proceedings of the IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[13] G. Forman and B. Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explorations Newsletter*, 2(2):34–38, 2000.

[14] High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1*, November 1994.

[15] E. C. Lewis and L. Snyder. Pipelining wavefront computations: Experiences and performance. In *Fifth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2000.

[16] S. Lumetta, A. Mainwaring, and D. Culler. Multi-protocol active messages on a cluster of SMP's. In *Supercomputing*, 1997.

[17] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Workshop on Runtime Systems for Parallel Programming (RTSPP)*, April 1999.

[18] J. Nieplocha, J. Ju, and E. Apra. One-sided communication on the Myrinet-based SMP clusters using the GM message-passing library. In *CAC*, 2001.

[19] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[20] D. Rey, J. Stubblefield, and J. Canning. Porting the parallel array programming language ZPL to an embedded multicomputing system. In *Proceedings of the 2002 conference on APL*, pages 168–175. ACM Press, 2002.

[21] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[22] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the ACM International Conference on Supercomputing*, 1995.

[23] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[24] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[25] B. Zhang, M. Hsu, and G. Forman. Accurate recasting of parameter estimation algorithms using sufficient statistics for efficient parallel speed-up demonstrated for center-based data clustering algorithms. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2000.