Problem Space Promotion and Its Evaluation as a Technique for Efficient Parallel Computation*

Bradford L. Chamberlain E Christopher Lewis Lawrence Snyder

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
{*brad,echris,snyder*}@*cs.washington.edu*

Abstract

In this paper we describe a parallel programming paradigm called problem space promotion *(PSP), a technique that increases parallelism by reducing communication and synchronization. We present four algorithms that exploit PSP and evaluate their communication characteristics relative non-PSP solutions. Our analysis is aided by the use of parallel algorithm notation that is concise, yet accurately reflects parallelism and communication costs. Our analysis illustrates circumstances under which the use of PSP is beneficial and detrimental to performance, and experiments on the Cray T3E attest to the validity of the analysis. We find that PSP can significantly improve the performance and scaling behavior of certain computations, even when compared to existing high quality parallel algorithms.*

1 Introduction

Problem space promotion (PSP) is a parallel solution technique for problems involving combinatorial interactions of array data. PSP reformulates algorithms that operate over d-dimensional data as a computation in a d'-dimensional problem space, where $d' > d$. The goal of PSP is to increase the parallelism of the solution by reducing the algorithm's communication and synchronization requirements. For instance, consider an algorithm in which all n elements of a 1-dimensional array require pairwise interactions. Figure 1(a) illustrates a straightforward parallel solution that cyclically shifts a copy of the array, so that after n shifts, all n^2 interactions have been considered. In contrast, the PSP solution illustrated by Figure 1(b) promotes the problem space to a 2-dimensional $n \times n$ space and uses the 1-dimensional array as the rows and columns of this space, thereby implicitly representing all n^2 comparisons without an iterative loop.

As a concrete example of problem space promotion, consider the following PSP algorithm for sorting a vector, V, of n unique numbers.

*This research was supported in part by DARPA Grant E30602-97-1-0152 and the Intel Corporation.

To appear in the 13th ACM SIGARCH International Conference on Supercomputing, June 1999, Rhodes, Greece.

$C \leftarrow V^T \leq V$		compare all pairs, (1 if true, 0 if false)
$P \leftarrow \text{Sum\_cols}(C)$		column sums give sorted index position
$V \leftarrow V[P]$		permute elements of V into order

This solution, which requires quadratic work, is an instance of problem space promotion, because it performs all n^2 comparisons simultaneously. Though it accepts 1-dimensional input, it performs operations in terms of 2-dimensional arrays, *e.g.*, array C. This is in contrast to an algorithm which iteratively transforms the 1-dimensional input directly. The question of interest here is, *Can problem space promotion be an effective technique for writing efficient parallel programs?*

The chief advantage of PSP in the sorting example, above, is that it specifies that all comparisons can be performed simultaneously, and that all columns can be summed independently. Specifically, a parallel implementation of this algorithm broadcasts the data and its transpose to the processors, which are conceptually arranged in a 2-dimensional mesh, allowing them to work independently of each other with little or no synchronization. This yields much greater concurrency than, say, the odd-even transposition sort [11], which requires the processors to synchronize repeatedly throughout the computation. It is this increase in parallelism with the simultaneous relaxation of synchronization constraints that motivates interest in PSP algorithms. The fact that PSP algorithms are frequently specified in a clean and elegant form is a significant added advantage.

There are apparent disadvantages, too. First, it appears that storage requirements grow to the size of the promoted problem space, *e.g.*, the C array is $n \times n$. In fact, this storage can be eliminated, restoring the storage requirements to the order of the problem input. Second, there is a potential increase in the amount of work required to compute the result. This problem has two forms. One form concerns non-asymptotic complexity, such as missed opportunities to exploit problem symmetry, *e.g.*, the sort above could perform $n^2/2$ comparisons. The other form concerns asymptotic complexity, *e.g.*, for sorting, $\Theta(n \log n)$ comparisons suffice. Though this may be a potential shortcoming with PSP solutions, a suitable complexity model for practical portable parallel computations remains to be worked out, leaving it unclear whether facts like "$\Theta(n \log n)$ comparisons suffice" lead to significantly more efficient parallel solutions. Issues of synchronization and communication complexity may dominate work complexity for certain algorithms.

In this paper, we make the following contributions.

- We introduce a new parallel solution technique called problem space promotion for increasing parallelism in certain

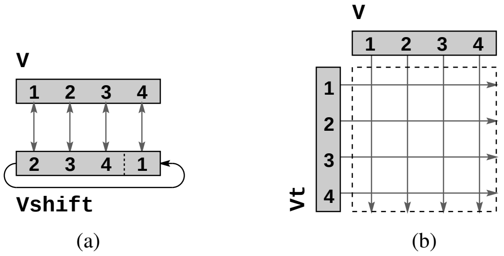

Figure 1: Illustration of problem space promotion. (a) Conventional solution versus (b) problem space promotion.

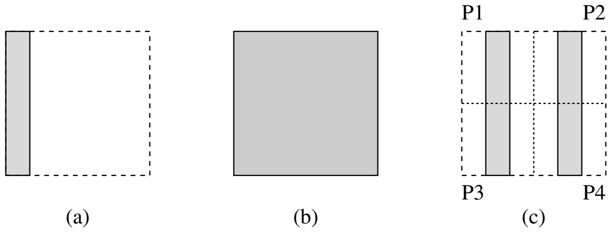

Figure 2: Illustration of flood operator. (a) A column of an array, (b) a replicated form of the column across the problem space, and (c) the actual allocation of memory to represent the flooded data across 4 processors.

computations.

- We analytically and experimentally show its potential performance improvement.
- We demonstrate the use of the ZPL parallel programming language as an algorithmic notation to assist in evaluating the parallel performance of algorithms.

The remainder of this paper is organized as follows. The next section introduces the ZPL-based notation we use to describe algorithms. Section 3 describes problem space promotion by way of four computations. We evaluate the performance implications of PSP both analytically and experimentally in Section 4. Section 5 considers related work, and the final section gives conclusions.

2 Brief Summary of ZPL

In order to describe and study problem space promotion we adopt an algorithmic notation based on the ZPL parallel, array-based programming language [20, 23]. The notation has the advantage that it is succinct for presentational purposes, it can be executed on parallel computers with excellent performance [6, 17], and it exposes issues of parallelism and communication overhead without requiring the programmer to manage complex details. This section briefly introduces the core components of the ZPL language and its performance model. Complete treatments are available in the literature [20, 5].

Basics. ZPL is an imperative programming language supporting all the usual (*i.e.*, Fortran, C or Pascal) data types, scalar operators, and control structures. Like Fortran 90 and APL, it is an array language, permitting atomic operation on whole arrays or subarrays. ZPL is distinguished from other array languages by its use of *regions* in describing parallel, array computations [7].

Regions. *Regions* represent index sets. The region `[1..n,1..n]` is a 1-origin $n \times n$ index set, while `[1..n,1]` is a subregion describing its first column, and `[1,1..n]` is a subregion describing its first row. Regions can be named and referred to symbolically as follows: `region R = [1..n,1..n]`. Regions are used in declarations to specify the size and shape of arrays as follows: `var A,B,C: [R] float`. These arrays will have an element for each index in region `R`.

Regions are also used to determine the indices of arrays involved in an array statement's computation. Specifically, a region preceding a statement establishes the extent of all array computations in its dynamic scope. For example, the following statement operates only over the first row of arrays `B` and `C`.

```
[1,1..n] C := 2*B;
```

When a new region scope is entered, it overrides the enclosing region of the same rank. If a dimension is left blank, its indices are inherited from the corresponding dimension of the enclosing region. For example, region `[1,]` in the context of region `[1..n,1..n]` has the indices of region `[1,1..n]`.

ZPL supports a number of operations on regions and arrays. The latter provide array-level computations and permit the indices for a particular array reference to be adjusted with respect to the enclosing region scope. We summarize several array operators below.

Parallel Prefix. ZPL includes the parallel prefix operations, reduction and scan. For example, `[R] s:=+<<B` is the sum reduction of the elements of array `B` described by the indices in region `R`. There are versions of reduction and scan operators for other arithmetic and logic operations (*e.g.*, product, max, logical or). For arrays of rank greater than one, it is possible to perform partial reductions on subarrays. Thus, assuming `B` and `C` are 2-dimensional arrays and `R` is a 2-dimensional region as defined above, the following statement reduces each row to its sum and stores the result in the third column of `C`.

```
[1..n, 3] C := +<<[R] B; -- add rows
```

The two regions serve the following purposes: The $n \times n$ *source* region, `R`, encoded in the reduction operator, describes which elements of the operand will participate in the reduction, while the column *result* region, `[1..n,3]`, preceding the statement indicates where the result is to be stored. In ZPL, the result region's degenerate dimensions (*i.e.*, dimensions representing a single index) with respect to the source region indicate which dimensions are reduced.

Replication. Flooding is the logical dual of reduction, providing an abstraction for replication. Thus, the following expression, illustrated by Figures 2(a) and (b), replicates the first column of `B` across the second dimension using the flooding operator (>>).

```
>>[1..n,1] B
```

The region associated with a flood operator, in this case `[1..n,1]`, specifies what portion of the array operand is to be replicated. The effect of the expression, above, is to create a logical 2-dimensional array with a conceptually infinite number of n element columns. Floods and reductions can be combined to express computations such as the construction of the permutation vector, `P`, from the sort computation in the introduction. Assume that `V` and `Vt` are row and column vectors, respectively, and that region `R` is as defined above.

```
[1,1..n] P := +<<[R](>>[,1] Vt) <= (>>[1,] V);
```

Recall that empty ranges in a region inherit the range of the enclosing region, thus `[,1]` in the context of region `R([1..n,1..n])` is equivalent to `[1..n,1]`. The statement above assigns to `P` the column-wise reduction of the n^2 elements created by comparing (using <=) the result of flooding `V` transpose (`Vt`) and `V`. A key property of flooding that is exploited in this computation is that only a single copy of a flood's defining values are stored on each processor, as illustrated by Figure 2(c). Thus, this computation need not create an n^2 temporary array, despite the fact that it computes over such a logical array. This is a key source of efficiency for PSP computations in ZPL.

Shifting. To reference a shifted form of an array, the shift operator (`@`) is used with an integer vector that gives the direction and magnitude of the translation in each dimension. Thus, the following statement assigns the elements at indices 0 through $n-1$ of array `Y` to indices 1 through n of array `X`.

```
[1..n] X := Y@(-1);
```

Arbitrary data movement. ZPL provides a gather operator (`#`), permitting arbitrary data movement. Encoded in brackets following the gather operator is an integer array for each dimension of the result, specifying the source of each element. The only gather operation needed in this paper is the transpose, which uses ZPL's constant arrays `_1` and `_2` to specify dimensional interchange. In general, the value of array `_k` at index $i_1, \ldots, i_d$ is i_k. Thus, the following statement performs the transpose on `V` needed to compute `Vt`: `[1..n,1] Vt := V#[_2,_1]`.

Performance Model. ZPL supports a performance model that permits programmers to reason about the parallelism and communication overhead in their codes [5]. In ZPL, all arrays are *aligned*.[1] There are two main implications of this: (i) the data elements of two different arrays at index (i, j) are guaranteed to reside on the same processor, and (ii) the data elements, *e.g.*, at indices (i, j) and (i, k) reside in the same processor row when the arrays are distributed across a virtual processor grid.

Given this alignment, each array operator induces a particular form of interprocessor communication. The shift operator (`@`) results in simple point-to-point communication between neighboring processors in the grid. The flood operator (>>) results in a broadcast often along one dimension of the grid. The gather operator (`#`) potentially results in an arbitrary redistribution of data. Statements that contain no array operators require no communication. Despite the high-level nature of the ZPL notation, the programmer can reason about the implementation of their code on a parallel machine. We will exploit this fact in our discussion of problem space promotion.

3 Algorithms Exploiting Problem Space Promotion

Below, we present PSP solutions to common computations in preparation for analyzing them in the next section. First, we abstractly describe the structure common to all PSP algorithms. Then, we present four computations given in both a conventional and PSP form. We use the ZPL form from the previous section to augment the algorithm descriptions. In these codes, declarations are omitted, because the type and size of the arrays is clear from context; and by convention, array variables are capitalized, while scalars are not.

[1] In fact, not *all* arrays need be aligned, only those that are used together, but the simplification is sufficient for this presentation.

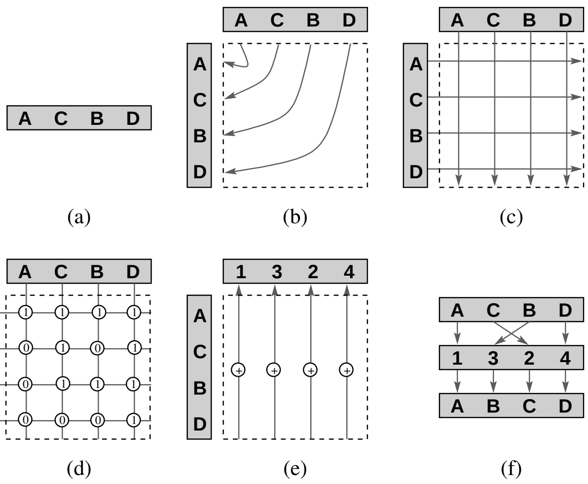

Figure 3: Summary of the phases of problem space promotion. (a) 1-dimensional input, (b) orientation phase, (c) replication phase, (d) computation phase, (e) collapse stage, and (f) permutation step performed by PSP sorting algorithm.

3.1 The Structure of PSP Algorithms

There are four stages common to all PSP algorithms: (i) data orientation, (ii) data replication, (iii) computation, and (iv) data collapse. For this discussion we assume that the indices of the promoted d-dimensional problem space are block distributed across a conceptual d-dimensional mesh of processors, and the input data initially occupies a lower dimensional slice of the d-dimensional problem space. The *data orientation stage* orients the input data with respect to the problem space. For example, in the sorting example of the introduction, the input vector is initially a row of the problem space (Figure 3(a)), so it must be copied and transposed to occupy a column (Figure 3(b)). On a parallel machine, data orientation requires a nontrivial, yet regular, amount of interprocessor communication. In ZPL we realize orientation via the gather operator (`#`).

Next, the properly oriented data is *replicated* so that data fills the entire problem space (Figure 3(c)). In ZPL we realize replication via the flood operator (>>). Note that on a given processor the replication is only conceptual, for only the defining values of an array need to be represented, as illustrated by Figure 2(c).

After replication, the *computation stage* begins, during which each processor computes entirely on local data (Figure 3(d)). No communication or synchronization is required during this stage. Depending on how the programmer phrases the computation phase, problem space sized arrays may be created (see the n-body code, below). We have previously developed compiler techniques by which this storage requirement is eliminated via statement fusion and array contraction [16]. The compiler can consistently eliminate the higher-dimensional storage requirement in PSP codes, because it folds the local accumulation portion of the collapse stage (below) into the computation stage.

Finally, the *collapse stage* performs a reduction along one or more dimensions of the problem space in order to gather the result of the computation stage into a lower dimensional structure (Figure 3(e)). In ZPL we use the reduction operators (*e.g.*, `+<<` or `max<<`) to collapse. Some algorithms may perform additional operations at this point to compute the final result. For example, the sorting code permutes the input array according to the result of the collapse stage (Figure 3(f)). A closer inspection of Figure 3 reveals how the input sequence (A, C, B, D) is sorted according to

```
[1..n] begin                -- odd stage
         -- swap if I@right value greater
         if (_1 % 2) then   -- odd indices
           if (I > I@right) then
             I := I@right;
           end;
         else               -- even indices
           if (I < I@left) then
             I := I@left;
           end;
         end;
       end;
```

(a)

```
[1,1..n] begin     -- assume R = [1..n,1..n]
           -- assume row 1 of V contains input
           -- orient data (transpose row into col)
[1..n,1]   Vt := V#[_2,_1];
           -- replicate, compute and collapse
           P := +<<[R] (>>[1,]V <= >>[,1]Vt);
           -- permute V data according to array P
           V := V#[_1,P];
         end;
```

(b)

Figure 4: Sorting. (a) Conventional (odd stage from odd-even transposition sort) and (b) PSP implementations.

```
[1..n] begin
         S := 0;
         for i := 1 to n do
[i..n]     S += ((>>[i] V) = V);
         end;
         -- frequency of mode
         count := max<< S;
         -- get actual mode value
         mode := max<< ((count = S) * V);
       end;
```

(a)

```
[1,1..n] begin     -- assume R = [1..n,1..n]
           -- assume row 1 of V contains input
           -- orient data (transpose row into col)
[1..n,1]   Vt := V#[_2,_1];
           -- replicate, compute and collapse
           S := +<<[R] (>>[1,]V = >>[,1]Vt);
           -- frequency of mode
           count := max<< S;
           -- get actual mode value
           mode := max<< ((count = S) * V);
         end;
```

(b)

Figure 5: Mode calculation. (a) Conventional and (b) PSP implementations.

the algorithm from the introduction.

In the following sections, we examine conventional and PSP solutions to various computations. We restrict our analysis to giving *work complexity*, the time complexity of the computation executing on a single processor. The bulk of the discussion of communication complexity is left for Section 4.

3.2 Sorting

We begin by considering sorting. Given a sequence of n numbers, $A = \langle a_1, a_2, \ldots, a_n \rangle$, the permuted sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of A is a sorted form of A when $a'_1 \le a'_2 \le \ldots \le a'_n$.

Odd-Even Transposition Sort is a parallel sorting algorithm due to Demuth [11]. It iterates for $\lceil n/2 \rceil$ steps, each of which is comprised of two stages. The first stage, which appears in Figure 4(a), compares in parallel each odd element to its right neighbor. If the former is greater than the latter, the two values are exchanged. This code exploits *shattered control flow*, *i.e.*, array-based control flow, in order to selectively compute on elements of array `I`. The second stage performs the same comparison/exchange for the even elements. Each of the $n/2$ iterations performs $\Theta(n)$ comparisons resulting in $\Theta(n^2)$ work complexity.

A PSP sorting algorithm appears appears in Figure 4(b).[2] After a transpose of the input data from a row to a column, the algorithm uses the flood operator to broadcast the n input elements across the rows and columns of the 2-dimensional problem space. Index (i, j) of the problem space contains the ith and jth elements of the input. As a result, all n^2 comparisons for the sort my be performed completely in parallel. The results of each comparison (a 1 or a 0) are collapsed by summing along each column, producing a single row, `P`. The array `P` is then used to permute the input array resulting in a sorted form of array `V`. Because the algorithm computes over a 2-dimensional problem space, it has $\Theta(n^2)$ work complexity. Similar algorithms have been described as constant time sorting algorithms for unrealizable CRCW machines [3, 22].

[2] For simplicity we assume that the input sequence contains no duplicates. The structure of the algorithm is unchanged when extended to handle duplicates.

3.3 Mode Calculation

Next, we consider the mode computation. Given a set of n samples, $A = \{a_1, a_2, \ldots, a_n\}$, the mode, m, is the element of A that occurs most frequently.

The algorithm in Figure 5(a) extends the obvious serial solution to the parallel domain. The algorithm broadcasts the elements of the input array, one at a time, to all the other processors. Each processor compares the broadcast value with its local values. If they are the same, a counter associated with the local value is incremented. After iterating over all n elements, a maximum reduction is performed to find out how many times the most frequently occurring element occurs. Finally a maximum reduction is used to find the value that actually occurs most frequently. There are n iterations, each containing array operations of cost $\Theta(n)$, resulting in $\Theta(n^2)$ work complexity.

A PSP mode calculation algorithm appears in Figure 5(b). This algorithm is similar to the sort code, except that equality is used for comparison. A sum reduction in the column i thus indicates the number of occurrences of the value in input position i. The computation over the 2-dimensional problem space results in $\Theta(n^2)$ work complexity.

3.4 Matrix Product

Matrix product is a fundamental operation from linear algebra. The product of an $m \times s$ matrix A and an $s \times n$ matrix B is an $m \times n$ matrix C whose elements are

$$c_{ij} \quad = \quad \sum_{k=1}^{s} a_{ik} \times b_{kj}, \quad 1 \le i \le m, 1 \le j \le n. \qquad (1)$$

For simplicity, we assume that $n = m = s$. The SUMMA1 algorithm, in Figure 6(a), and its variants have been been shown to give excellent parallel performance [21]. The algorithm contains `n` iterations. In iteration `i`, column `i` of array `A` and row `i` of array `B` are replicated across the rows and columns, respectively, of the problem space. The resulting array, `C`, is incremented by the element-wise product of the two replicated arrays. After `n` iterations, array

```
[R begin     -- assume R = [1..n,1..n]
    C := 0.0;
    -- for each column in A and row in B
    for i := 1 to n do
       C += ((>>[,i] A) * (>>[i,] B));
    end;
  end;
```

(a)

```
  -- assume IK  = [1..n,1   ,1..n]
  --         KJ  = [1   ,1..n,1..n]
  --         IJ  = [1..n,1..n,1   ]
  --         IJK = [1..n,1..n,1..n]

  -- orient the A and B matrices
[IK] At := A#[_1,_3,_2];
[KJ] Bt := B#[_3,_2,_1];
  -- replicate, compute and collapse
[IJ] C := +<<[IJK] ((>>[IK] At) * (>>[KJ] Bt));
```

(b)

Figure 6: Matrix product. (a) Conventional (SUMMA1) and (b) PSP implementations.

C contains the matrix product of A and B. The work complexity of this algorithm is $\Theta(n^3)$.

The PSP solution to matrix product appears in Figure 6(b). It adds another dimension to the problem space so that all n^3 products are implicitly represented in the 3-dimensional problem space. Assume that the input matrices, A and B, initially occupy a 2-dimensional slice across the first two dimensions of the problem space. First, they must be oriented so that A occupies a 2-dimensional slice across the first and third dimensions and B occupies a slice across the second and third dimensions. The oriented copies of the data are stored in arrays At and Bt. Next, arrays At and Bt are replicated across the second and first dimensions, respectively. An element-wise product is taken in the 3-dimensional space, and a sum reduction along the third dimension is assigned to array C, resulting in the matrix product of A and B. The PSP algorithm in ZPL has the additional benefit that the code very closely resembles the textbook definition of matrix product, equation (1). The work complexity of this code is $\Theta(n^3)$ due to the scalar product performed in the 3-dimensional problem space.

3.5 N-Body Simulation

An n-body computation simulates the motion of masses—such as astronomical bodies—over time given an initial configuration describing the position, velocity and mass of each body. At each time step, the gravitational attraction of each body on every other body is calculated to determine the subsequent configuration of the system. Though algorithms exist that ignore interactions between distant bodies, certain contexts requires that all n^2 interactions be considered.

Figure 7(a) contains the core of a conventional n-body code that simulates the motion of bodies in a 3-dimensional space. It calculates the acceleration (Acc) in each dimension imparted on each body by all the other bodies. The body positions and masses are copied into temporary arrays (RollPos and RollMass) which are cyclically shifted $n-1$ times. After each shift, the acceleration imparted by the bodies in the Roll arrays are calculated on the bodies in the input array, and the result is accumulated into Acc. Each array operation computes over n elements, and there are $n-1$ iterations, resulting in $\Theta(n^2)$ work complexity.

Figure 7(b) contains a PSP equivalent of the code in Figure 7(a).

```
[R]         begin     -- assume R = [1..n]
              Acc[] := 0.;

              RollPos[]  := Pos[];
              RollMass := Mass;
              for iter := 1 to (n-1) do
[next of R]     wrap RollPos[], RollMass;

                RollPos[] :=  RollPos@next[];
                RollMass := RollMass@next;
                Delta[] := Pos[] - RollPos[];
                DistSqr := sqr(Delta[X])+sqr(Delta[Y])+
                           sqr(Delta[Z]);
                Dist := sqrt(DistSqr);
                DistInv := 1.0/(DistSqr*Dist);
                Acc[] -= RollMass*Delta[]*DistInv;
              end;
            end;
```

(a)

```
[R]         begin      -- assume R = [1..n,1..n]
[1..n,*]      begin    -- orient and replicate
                AccCol[] := AccRow[]#[_2,_1];
                VelCol[] := VelRow[]#[_2,_1];
                PosCol[] := PosRow[]#[_2,_1];
              end;

              -- compute
              Delta[] := PosCol[] - PosRow[];
              DistSqr := sqr(Delta[X])+sqr(Delta[Y])+
                         sqr(Delta[Z]);
              -- special case the diagonal
              if (_1 = _2) then
                DistInv := 0.0;
              else
                Dist    := sqrt(DistSqr);
                DistInv := 1.0 / (DistSqr * Dist);
              end;

              -- collapse
[*,1..n]      AccRow[] := +<<[R] (MassCol * Delta[] *
                                   DistInv);
            end;
```

(b)

Figure 7: N-body simulation. (a) Conventional and (b) PSP implementations.

Arrays PosRow, VelRow, and AccRow contain the initial configuration. Column oriented copies of these arrays are created. This code exploits a feature of ZPL where arrays can be defined in such a way that assignments to them are implicitly replicated across a dimension, thus achieving the replication stage. The distance between all pairs of bodies is found and used to calculate the accelerations due to each pair of bodies, which are then reduced into AccRow. This algorithm performs a constant number of operations over a 2-dimensional problem space, thus it has $\Theta(n^2)$ work complexity. Note that a number of arrays in this code, such as Delta, DistSqr, Dist and DistInv, are full 2-dimensional arrays. The ZPL compiler generates a single loop nest to implement all the statements that contain these references and contracts each of these arrays to a scalar value [16]. As a result, this code only requires memory linear in the number of bodies.

4 Analysis

We begin this section with a discussion of the performance implications of problem space promotion. Next, we experimentally evaluate its performance impact on the computations described in the previous section.

4.1 Discussion

The previous section presented two algorithms to solve each problem, one conventional—which we call the *base* solution—and one PSP. Below, we discuss the expected performance of the two codes by considering the issues of communication complexity, the nature of the communication, and early termination.

Communication complexity. Because each pair of algorithms from the previous section has equivalent work complexity, the base and PSP solutions are distinguished by their communication complexity. All the base algorithms contain $\Theta(n)$ communication operations. Specifically, sort and n-body use $\Theta(n)$ point-to-point nearest neighbor communication operations (`@`s), and mode and matrix multiplication, use $\Theta(n)$ broadcast operations (`>>`). The algorithms that exploit PSP, on the other hand, require a constant number of communication operations. The orientation phase transposes a row or a plane in the higher dimensional problem space via the gather operator (`#`). The replication phase broadcasts the data across the higher dimensional problem space via the flood operator (`>>`). The collapse phase performs reduction(s) across a dimension of the problem. Some of the algorithms, such as sort, also require an additional constant number of miscellaneous communication operations. In all four cases, PSP potentially improves parallel performance by decreasing the communication complexity without adversely affecting work complexity. Furthermore, because communication synchronizes the processors, PSP reduces the number of times that the processors must synchronize, permitting the entire computation phase to proceed in parallel.

Nature of the Communication. In addition to the number of communication operations, the type of communication operation must also be considered. Specifically, nearest neighbor shifts are $\Theta(1)$ operations, while for p processors, broadcasts and reductions are generally regarded as $\Theta(\log p)$ operations, though specialized hardware and p small relative to n render them nearly constant as well. Despite the fact that PSP algorithms use more expensive communication operations (*e.g.*, gathers, broadcasts and reductions), it is unlikely on any modern parallel architecture that $\Theta(n)$ nearest neighbor operations will be less costly than $\Theta(1)$ broadcast or reduction operations.

Furthermore, PSP often increases the volume of communicated data. For example, the base mode computation only broadcasts n words. The PSP mode computation, on the other hand, transposes n words, broadcasts $2n$ words, and performs a reduction of n words. Despite this increased volume, we do not expect this to limit PSP performance, because PSP algorithms require fewer communication operations, and we expect the synchronization and startup cost of initiating the communication to be the dominant costs.

Early Termination. There are certainly circumstances where problem space promotion may be inappropriate. Problem space promotion transforms an iteration space into a dimension of the problem space. As a result, we must know *a priori* how many iterations the loop contains. Moreover, some base algorithms may be able to terminate the loop early, which cannot be trivially accomplished in PSP since the loop has been transformed into part of the problem space. For example, the odd-even transposition sorting algorithm can be modified to terminate early if it finds that the data is already sorted. Similarly, the base mode algorithm can terminate early if there are j values left to be tested and a mode of at least j occurrences has already been found. For both examples, there is a tradeoff to be considered: *Will the expected input allow the algorithm to terminate suitably early to compensate for the algorithm's inferior communication complexity?*

Problem space promotion can be extended to allow early termination in certain cases by repeatedly going through all four phases, wherein the added dimension to the problem space contains only a portion of the problem. For example, in the mode calculation, the orientation phase could place only half the values across the extra dimension, thus finding the mode of half the data. If there are more occurrences of the mode than unconsidered remaining data, the algorithm terminates. Otherwise, it considers half of the remaining data, accumulating the results with the previous iteration. The process continues for at most $\log n$ steps. Naturally, the communication complexity is worse than a pure PSP approach, but it allows for early drop-out. Computations such as n-body simulation or matrix product cannot exploit this technique, but the base solutions cannot benefit from early termination, either.

4.2 Experimentation

In order to verify the analysis of the previous section, we measure the performance of the codes from Section 3 on the Cray T3E. All codes are written in ZPL and compiled with the ZPL compiler [23]. Prior work has demonstrated that this is a high quality compiler. The code it produces has performance comparable to C with MPI [18, 6] and generally outperforms HPF [19, 17]. The ZPL compiler aggressively optimizes communication for overlap with computation [9]. Figure 8 contains two speedup curves for each problem introduced in Section 3. The two curves give speedup for the base and PSP solutions. Both speedup curves are calculated with respect to the same sequential execution time. As a result, greater speedup implies faster performance, allowing the two curves to be compared directly.

The PSP versions of the codes all have significantly better scaling behavior than the base solutions. This confirms our prediction that the improved communication complexity due to PSP improves overall performance. The mode and sort codes are dominated by communication (*i.e.*, they contain relatively few floating point operations). As a result, the speedup of the base solutions suffer greatly from the many communication operations they contain. PSP significantly improves these codes. The n-body code spends a relatively larger portion of the total execution time executing floating point code, so both the base and PSP solutions scale better than mode and sort. We discuss the matrix product codes below. Also note that as the number of processors increases, the relative cost of communication versus computation increases, and the benefit of PSP increases.

We cannot conclude from the graphs in Figure 8 that the PSP mode calculation and sorting algorithms are optimal for parallel machines. Instead, we conclude that by applying problem space promotion to an existing solution we can significantly improve parallel performance by increasing the degree of parallelism. The matrix product codes are particularly noteworthy, because the PSP solution outperforms the SUMMA1 algorithm [21], which is regarded as a very high quality parallel algorithm. The relatively poor scaling behavior of SUMMA1 led to variations on this algorithm that broadcast several rows and columns at a time, achieving some of the benefits of PSP.

These graphs do not highlight the fact PSP algorithms have different cache behavior because of how they traverse the input arrays. It would appear that this is a second order effect, so we leave it for future work. We expect to find that PSP algorithms have inferior cache behavior but that they will benefit from blocking techniques.

5 Related Work

The problem space promotion technique can be exploited in programming languages and systems other than ZPL, for example, C

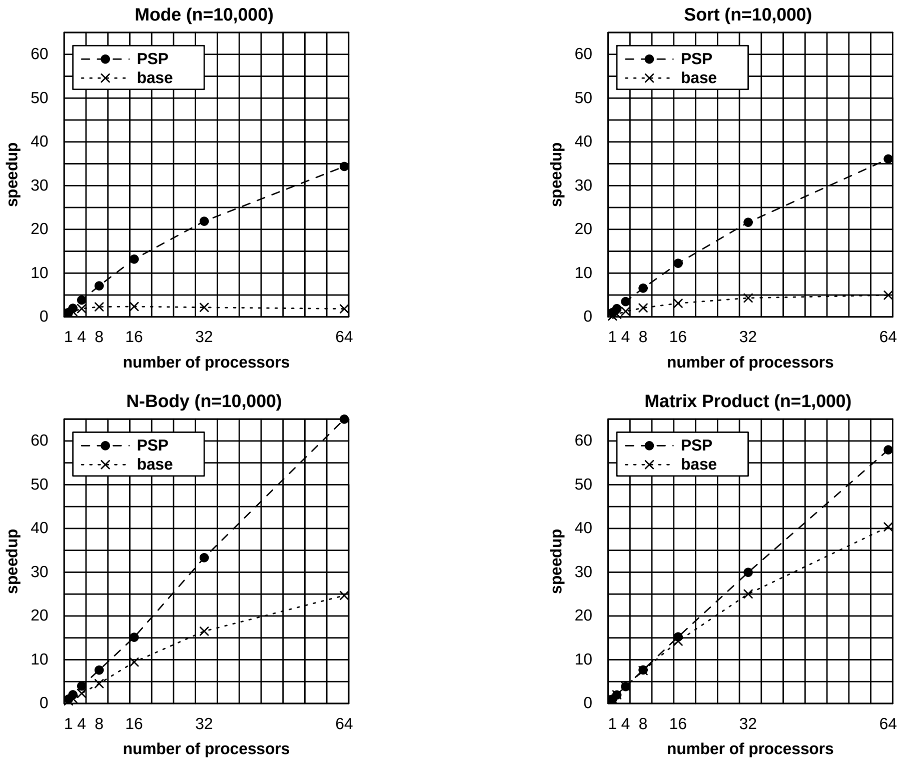

Figure 8: Speedup of base and PSP codes on the Cray T3E.

with MPI (message passing interface). In this sense, PSP is a language neutral parallel solution technique. On the other hand. not all languages permit programmers to reason about the communication costs of their applications. Presumably, a compiler for HPF [14] or a parallel implementation of APL [4, 13, 8] could exploit the benefits of PSP, but they do not offer a performance model. Ngo has evaluated the role of performance models in ZPL and HPF [19]. He found that a performance model is essential in producing consistently high quality programs across different machines and compilers. ZPL provides an explicit performance model, facilitating the analysis in this paper. Previously, we have described how the ZPL performance model is used to evaluate implementation alternatives [5].

Problem space promotion is related to a standard technique used by APL programmers to write "one-liners" that perform computation without control flow [15]. In APL, outer product is used to convert the logical iteration space to an array. Thus, for example, the sorting operation given in the introduction is expressed as

$$V\texttt{[+/[1]}V\circ\texttt{.}\leq V\texttt{]}$$

where the outer product ($V\circ\texttt{.}\leq V$) creates a square array of 0s and 1s that is then sum reduced (+/[1]) in one dimension to produce the permutation vector to index the array. Historically, approaches similar to PSP have been used to avoid expensive-to-interpret iterative solutions, and we are not aware that any APL interpreter or compiler takes advantage of the special structure of this computation.

PSP-like solutions to specific problems have appeared in the literature. For example, Aggarwal *et al.* analyze the communication complexity of a 3-dimensional matrix multiple algorithm in the context of the LPRAM (local-memory parallel random access machine) model [2], and Agarwal *et al.* evaluate a 3-dimensional matrix multiple algorithm on the IBM SP2 [1]. They do not explore the work as a general solution technique.

Researchers have proposed parallel machine models in an effort to understand and predict performance. The LogP machine model is a well studied model intended to accurately represent real parallel computers and serve as a basis for algorithm design [10]. It has been applied, for example, to sorting algorithms on the CM-5 [12], but the model's highly parameterized nature casts doubt on its portability. For example, in a LogP analysis of a PSP code, the underlying implementation of the collapse stage (a reduction) on a particular machine would need to be modeled. A ZPL analysis (and implementation) abstracts these machine specific details.

6 Conclusion

Careful consideration of the costs of computation and communication on real parallel computers facilitates the development of unique and beneficial algorithmic solution techniques. In this vein, we have developed a technique for increasing parallelism, called problem space promotion (PSP). We have described the technique abstractly and in terms of four specific computations with the ZPL parallel programming language, and we have analyzed these programs' expected performance on a real parallel machine using ZPL. Experiments on the Cray T3E have confirmed the analysis, demonstrating that PSP is a promising technique for increasing parallelism. Though we have represented and implemented the sample applications in ZPL, PSP is a general solution technique that may be applied in other parallel programming contexts. In the future, we will formalize our ZPL evaluation and analysis techniques, taking steps toward the goal of developing a practical and useful theory for parallel complexity analysis.

Acknowledgments. We thank Sung-Eun Choi for her comments on drafts of this paper. This research was supported by a grant of HPC time from the Arctic Region Supercomputing Center.

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palker. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–82, September 1995.

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.

[3] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[4] Timothy A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[5] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, March 1998.

[6] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–85, July–September 1998.

[7] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computations. Technical Report UW-CSE-98-10-02, Department of Computer Science and Engineering, University of Washington, October 1998.

[8] Wai-Mee Ching and Alex Katz. An experimental APL compier for a distributed memory parallel machine. In *Supercomputing '94*, pages 59–68, November 1994.

[9] Sung-Eun Choi and Lawrence Snyder. Quantifying the effect of communication optimizations. In *International Conference on Parallel Processing*, August 1997.

[10] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, May 1993.

[11] H. B. Demuth. *Electronic Data Sorting*. PhD thesis, Stanford University, October 1956.

[12] Andrea C. Dusseau, David Culler, Klaus Erik Schauser, and Richard P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.

[13] R. Greenlaw and L. Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111–127, April 1990.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*. January 1997.

[15] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[16] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

[17] 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.

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

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

[20] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, Cambridge, Massachusetts, 1999.

[21] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[22] Biing-Feng Wang, Gen-Huey Chen, and Ferng-Ching Lin. Constant time sorting on a processor array with a reconfigurable bus system. *Information Processing Letters*, 34(4):187–192, April 1990.

[23] ZPL Project. ZPL project homepage. *http:/www.cs.washington.edu/research/zpl*.