Performance Evaluation of the Myrias SPS-2 Computer

Oliver A. McBryan, Roldan Pozo

CU-CS-505-90 December 1990

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430

(303) 492-7514
(303) 492-2844 Fax
mcbryan@boulder.colorado.edu

PERFORMANCE EVALUATION OF THE MYRIAS SPS-2 COMPUTER

Oliver A. McBryan †

Roldan Pozo †

Center for Applied Parallel Processing
Dept. of Computer Science,
University of Colorado,
Boulder, CO 80309, USA.

ABSTRACT

The Myrias SPS-2 is the first massively parallel computer to provide a MIMD shared-memory programming model. Systems have been built with up to 1,024 nodes, although the experiments reported here were all performed on a 64 node system. We describe the system in some detail, particularly the software environment for applications programming. The highlight of the software environment is the virtual shared memory environment.

We have analyzed the performance of the shared memory environment by studying system efficiency as a function of both number of processors in use and of paging activity. We conclude that the system is robust and provides high efficiency for tasks of granularity about 100,000 floating point operations. However there is about a 50% overhead for the luxury of utilizing virtual shared memory. Programming the system is enormously easier than for other local memory MIMD systems.

Our performance studies start with measurements of simple saxpy type numerical processes. We also describe the implementation and performance of Poisson type relaxation kernels in one, two and three dimensions, as well as a complete application from the oceanographic modeling area (the Shallow Water Equations). Efficiency was high as long as task granularity was sufficiently large.

Keywords: parallel, performance, pde

† Research supported in part by AFOSR grant AFOSR-89-0422.

1. MYRIAS SPS-2 OVERVIEW

1.1. SPS-2 Hardware

The Myrias SPS-2 computer, built by Myrias Research Corp. of Edmonton, Alberta, is a massively parallel distributed memory computer with a virtual shared memory software environment. The largest system built to date has 1044 processing elements but the system design is scalable to even larger sizes.

Each processor is a 32-bit Motorola 68020 microprocessor, with a 68882 floating point coprocessor, and 4 Mbytes of local memory. The architecture is a three-level hierarchical bus design, utilizing 33 Mbytes/sec busses to interconnect processors within clusters and clusters to each other. Processors are assembled in groups of four on a board, connected among each other by a bus, along with an I/O port controller. At the second level in the hierarchy is the card cage, containing 16 processor boards and thus 64 processors, as well as an off-cage communication board. A pair of busses connect the 16 boards of a cage. Each communication board supports five off-cage links which can be connected to other cages or to the front end computer, which is currently a SUN 3 work-station. While the host provides user access to the SPS-2, all SPS-2 tasks execute entirely on SPS-2 nodes. The host is used only for compilation, program initiation and access to I/O devices.

1.2. SPS-2 Software and Programming Model

All system software has been developed for an abstract system called the G Machine. This is intended to clarify the programming model and to provide a layer of insulation from hardware details. The SPS-2 supports a global 32-bit virtual address space. Processes achieve parallelism by spawning independent subtasks, each conceptually executing in a *copy* of the parent's memory. To preserve efficiency, pages of the parent's memory are not actually copied until required by a child (*copy−on−demand*). There is no explicit concept of shared access to memory locations by multiple processors. However upon completion of child processes their memories are merged with the parent's memory according to specific rules explained in detail below. The subtask creation and subsequent merging process represent an implicit shared memory concept.

1.2.1. PARDO

A single facility is available to effect subtask creation, execution and merging: the PARDO construct (also see PARBEGIN below). This is implemented as Fortran and C language extensions called *pardo*. The PARDO model used by Myrias is somewhat unusual in that there are no possibilities for multiple processors to *directly* modify shared data (see below). A PARDO is executed by specifying a code segment to be executed and the number of child tasks to be run.

```
      pardo 10, I = 1, N
         <Code to be executed>
10    continue
```

Any Fortran DO loop may be replaced by a PARDO provided the body of the loop is independent across all loop iterations. The C *pardo* is similar to the Fortran version, and is thus syntactically weaker than the C *for* statement:

```
pardo (var=start : end, step) { ... } ,
```

where the *start*, *end* and *step* expressions must be of integer type. Each thread of execution performs completely independently in its own address space, starting with a copy of the parents memory. Execution of a child proceeds in normal sequential mode, except that PARDO's may be nested recursively. On completion of *all* children, the memory states of the children are merged to form the new memory state of the parent. Thus a child can never affect the memory of another child, but can affect the memory state of its parent, but only after all children merge.

A second available facility for parallel task execution is the PARBEGIN which implements a parallel IF-THEN-ELSE or case statement.

```
parbegin
      <code>
parallel
      <code>
...
parallel
      <code>
end par
```

The C version utilizes *parallel{*, *par* and *}* in place of *parbegin*, *parallel* and *end par*. The PARBEGIN may be thought of as a special case of PARDO where the different cases are simply different loop iterations, and so we will ignore PARBEGIN in the sequel.

1.2.2. Task Memory Merging Rules

The rules for merging of child memories, at a parent memory address, on child task completion are:

- If no child stored a value at the address, the location in the parent memory retains its original value.
- If exactly one child changed the value at the address, the location in the parent receives the last value from the child.
- If more than one child stores a value at the address the result is unpredicatable (*undefined*) unless all values stored are the same.

Efficiency is maintained throughout the process by using a copy-on-write approach which ensures that most of the global address space is never really replicated. While the hardware ignores the issue of whether a given memory location is undefined, the Myrias debugger may be used to detect undefined values.

A significant possibility for efficiency in merging occurs in connection with merge timing. Logically, merging occurs upon termination of the last child task and is completed before the parent executes the next instruction. In practice, merging can begin as soon as a child task completes, with various merging activities proceeding on the very processors which would otherwise be idled by the terminated task(s). Furthermore it is not essential that the merge actually complete for a page until that page is referenced in a subsequent instruction by the parent - indeed in many circumstances some pages may never be referenced before the parent terminates, and thus need never be merged!

The memory copying and merging rules at times imply surprising semantics. For example, the code fragment:

```
      pardo 10 j=1,N
         do 10 i=1,N
         A(j,i) = A(i,j)
10    continue
```

will correctly transpose the matrix A.

Another interesting feature of Myrias software is that all statically allocated memory (e.g. static variables and common blocks in C or Fortran) is actually only created by the kernel on first reference.

1.2.3. Use of Recursion

The programming approach to the system is functional in nature and is based on parallel recursion. In fact recursion has been added to Fortran as a Myrias extension, and is strongly recommended to users in the various manuals. As an example, we describe an implementation of a global vector sum, assuming the vector is initially of length N. The parent creates two tasks, each intended to sum half of the elements, storing the results in variables *left* and *right*. Each child task similarly spawns two more children, storing results into two local variables of the principal child, and so on. On completion of a pair of child tasks, its parent has the two partial sums in separate variables, and therefore it proceeds to add these, providing the sum to the next parent above. In this way the result is obtained at the global parent in time $O(log(N))$ without ever sharing memory. All assignment of work to processors is handled automatically.

This programming style is undoubtedly elegant, but because of the recursion involved, the resulting program is not recognizable as a standard Fortran program. Recognizing this problem, Myrias has recently extended the PARDO model by adding the capability to specify that certain variables are to be shared for reduction operations. For example, the summation variable, perhaps *sum*, can be declared to be a candidate for reduction, using a compiler directive, in which case the compiler would recognize the construct

```
        pardo 10 i=1,n
          sum = sum + expr(i)
10      continue
```

as a reduction operation and would generate appropriate code. Myrias supports reduction operations for integer and floating point *sum*, max, min, logical operations and several others.

1.2.4. Comments on Effectiveness of Myrias Software

The ideal parallel software development environment is one where one can take a serial program from a CRAY, SUN or other system, and run it immediately on the parallel system *without change*. The program may run with poor performance: for example no parallelism has been introduced, but at least it is running on the new hardware. Then, one can begin the refinement process of optimizing the software to the parallel hardware, but starting with a working program.

The current Myrias system comes close to attaining this goal - far closer than any other MIMD massively parallel system. Serial programs indeed run immediately on the system. In fact even programs with massive memory needs will run without modification: while the programs run on a single processor, they can access memory stored on other processors. The only requirement is to request sufficient SPS-2 processors (domain size) before running.

To begin the refinement process, one introduces PARDO instead of DO wherever independent iterations are observed in DO loops - the compiler could also be optimized to detect some of these. Performance is generally then greatly enhanced since multiple processors will be working on the code previously handled by one. Unfortunately this is the end of the line on SPS-2 - no further optimizations are available. It is desirable that further levels of refinement be made available. For example, users may in the end want to specify which processors specific tasks should run on, or to lock and share specific memory locations for write access without incurring the overhead of a complete merge.

1.3. SPS-2 Performance

In the following sections 2-5 we describe the performance evaluation of the SPS-2 which we have performed with the 64-processor SPS-2 at the Center for Applied Parallel Processing (CAPP) in Boulder, Colorado. In section 2 we discuss simple scalar performance benchmarks which provide a single-node upper bound on realizable performance from more realistic applications. In section 3 we describe performance for vector operations of the *Saxpy* type which in principle involve no merging interaction between child tasks on completion. Section 4 studies relaxation schemes in 1, 2 and 3 dimensions where we find that merging and non-local paging effects reduce efficiency very substantially. Section 5 discusses a complete application - the solution of the Shallow Water Equations, and provides comparisons between the CRAY-XMP, the Connection Machine CM-2 and the SPS-2.

Throughout the study we quote performance in terms of megaflops (Mflops). Timing was done throughout in terms of wall-clock time (Myrias routine *ertime*). Measurements in tables specify the number of tasks T used, as well as the domain size (number of active processors) D which ranges from 1 up to 64. We have tested values of T that are several (2,4, .. 16) times greater than D, but found little performance gain in most cases. Thus we describe here results for the case $T=D$ in most examples.

To allow relative comparison, the non-parallel single processor performance numbers have sometimes been scaled *linearly* to an "equivalent performance" on a 64 processor machine. We will always mark such scaled quantities in the tables with an asterisk to help identify them.

Several tables also record a quantity called "Efficiency". This will be defined (unless stated otherwise) as the ratio:

$$Efficiency\,(T) = Mflops\,(T)/(\,T\;Mflops\,(1)\,)\;,$$

where $Mflops\,(T)$ is the observed Mflops with $T=D$ tasks, and $Mflops\,(1)$ is the serial performance, with PARDO replaced by DO. Efficiency measurements will generally be performed where possible on scaled problems: i.e. the data set for the efficiency experiment with T tasks should be T times larger than that used to study the single processor speed. In cases where $Mflops\,(1)$ cannot be measured due to memory constraints, we use $Mflops\,(D_{min})$ as the reference point, where D_{min} is the smallest domain on which the benchmark will run.

There are other measures of efficiency used in this study. Generally if an efficiency value of 1.0 appears in a table, then that quantity was used as an efficiency reference for that table. Such efficiency values are supplied primarily to aid in understanding the variations within tables and should not be regarded as measures of absolute efficiency (such as the function given above).

2. SCALAR PERFORMANCE

In order to get a feeling for the likely peak machine rates of the system, we have analyzed performance of several simple scalar benchmarks. All were on a single SPS-2 processor, in a loop sufficiently long (e.g., 10,000,000) to ensure that loop overhead or similar effects are not being measured, and that timing resolution is not causing perturbations. For comparison, results from a SUN Sparc-station 4/110 and from a MIPS workstation have been included. The results are shown in Table 1.

Table 1: SPS-2 Single Node Scalar Performance

Benchmark	1-Node Mflops	64-Node Mflops*	Ext. Prec. Mflops*	SUN Sparc	MIPS
$c = a + c$	.0545	3.488	6.515	2.617	4.939
$c = a*c$	.0510	3.264	6.180	2.589	2.952
$c = a*c$ unrolled	.0874	5.594	8.211	3.256	3.529
$c = c + a*c$	.0753	4.819	8.570	3.254	3.737
$c = c + a*c ; a = a + c*b$	.0874	5.594	8.7296	3.937	3.969
super−unroll	.6288	40.24			

Each algorithm was implemented as a Fortran program and compiled with the -O2 option of the Myrias Fortran compiler, mpfc, giving the highest level of standard optimization. The data types were 32-bit. Slightly higher performance would be obtained with double precision data, and higher still with extended precision. Extended precision is obtained using the -O2f compiler option. For uniformity of results we quote only the single precision values. Double precision data would have decreased the largest problems that we could study in the PDE solution and application benchmarks. For comparison, we include above the extended precision benchmark values. The 64-Node figures and extended precision figures are simply linear extrapolations of the single node values and are useful in comparisons with the real 64-node parallel benchmarks presented later. While extended precision produces close to a doubling of performance in simple scalar benchmarks, we found only a 6%-14% improvement in the performance of the relaxation and shallow water benchmarks described later.

The unrolled version of $c = a*c$ was obtained by unrolling 4 times. The *super−unroll* is a special benchmark designed to make maximum use of the 68881 registers and is totally unrepresentative of any normal computation. Since it is specific to the SPS-2 it was not measured on other systems. The actual code consists of the three lines:

$$fi = fi * fi$$
$$sum = sum * fi$$
$$sum = sum - fi$$

repeated 16 times and surrounded by a DO loop of length 1 million. The super-unroll 64-Node values are actual measured values for a PARDO loop (the rest of those columns are extrapolated).

3. SAXPY TYPE VECTOR KERNELS

We have performed a detailed study of the performance of *saxpy* type operations in order to learn more about the relevant system parameters. Saxpy type operations appear in many situations as the low-level computationally intensive core of vectorizable applications. Examples include direct solvers for systems of equations and conjugate gradient iterative solvers for PDE. As test cases we have used the vector operations:

$$a(i) = a(i) + s^*c(i), \quad i=1,\ldots,N. \qquad (1)$$

$$a(i) = a(i) + b(i)^*c(i), \quad i=1,\ldots,N. \qquad (2)$$

which we refer to as *saxpy* and *vaxpy* respectively. All arrays are floating point and s is a scalar.

We have performed three studies of (1), (2) encompassing both single processor and multiple processor cases. We remark that on a standard distributed processor, such as an Intel iPSC/2, there would be no point in measuring multiprocessor versions of these two benchmarks. Neither involves any interprocessor communication, and so one can predict exact linearity of performance over the single-processor case. Because of interaction with the virtual memory system this is no longer true on the SPS-2.

3.1. Single Node Performance

Single-node performance is measured by ensuring that all of the vectors a, b, c are resident in the node memory. Because of the virtual memory, one can run a program on a single processor (i.e. PARDO of length 1, or simply a DO), but with data spread over a larger number of processors (the domain). We will denote the number of tasks actually computing by T, and the domain size of processors requested to provide virtual memory by D in the sequel. To ensure true single processor behavior, one must therefore run on a single processor with the domain size also set to 1 - i.e. $T=1$ and $D=1$. The performance results, presented in Table 2, are in agreement with the scalar benchmarks presented earlier. The SPS-2 performance was measured for the largest vectors that would fit in a single node: of length 65,536 elements.

Table 2: SPS-2 Single Node Saxpy Performance

Benchmark	1-Node Mflops	64-Node Mflops*	Sun Sparc	MIPS
$a = a+s^*c$	.056	3.584	.9979	.9790
$a = a+b^*c$	.052	3.328	.7266	.6919

We have also studied (1-2) in two more interesting situations:

a) a single processing node where N is so large that the vectors cannot reside in a single node ($T=1, D>1$).

b) the true parallel processing case where we allow more than one processor to perform the work ($T > 1, D > 1$).

Test a) gives a measure of the cost of accessing virtual memory, while test b) involves spawning child tasks (since $T > 1$) and therefore measures the overheads of task creation and merging. The simplest case discussed above with all vectors resident in one node and only one task is useful as a comparison point.

3.2. Single Processor Access to Virtual Memory

In case a) we examine the effectiveness of the *virtual* shared memory environment of the SPS-2 from the viewpoint of a single processor. A remarkable feature of the SPS-2 is that a program computing on only one processor can solve a problem that is too large to fit in a single processor. The ability to do so stems from the SPS-2 virtual memory. For this study we take various vectors in (1-2) and complete the operation using only one processor. Table 3 shows the degradation in performance resulting from this use of virtual memory in the case that the vector length is the largest that will fit in a single node ($D = 1$). Table 4 shows the case where we allow the vector length to scale with the domain size D, but still use only one processor to perform the computation $T = 1$.

Table 3: Single Node Multiple Domain
SPS-2 Saxpy Performance: Fixed Length Vector

Benchmark	Length	Tasks	Domain	Mflops
$a = a + s * c$	64K	1	1	.0566
$a = a + s * c$	64K	1	2	.0419
$a = a + s * c$	64K	1	4	.0415
$a = a + s * c$	64K	1	8	.0363
$a = a + s * c$	64K	1	16	.0363
$a = a + s * c$	64K	1	32	.0362
$a = a + s * c$	64K	1	64	.0361
$a = a + b * c$	64K	1	1	.0545
$a = a + b * c$	64K	1	2	.0376
$a = a + b * c$	64K	1	4	.0375
$a = a + b * c$	64K	1	8	.0317
$a = a + b * c$	64K	1	16	.0318
$a = a + b * c$	64K	1	32	.0317
$a = a + b * c$	64K	1	64	.0317

Table 4: Single Node Multiple Domain
SPS-2 Saxpy Performance: Scaled Vector Length

Benchmark	Length	Tasks	Domain	Mflops	Efficiency
$a = a+s*c$	64K	1	1	.0566	1.000
$a = a+s*c$	128K	1	2	.0417	.737
$a = a+s*c$	256K	1	4	.0390	.689
$a = a+s*c$	512K	1	8	.0324	.572
$a = a+s*c$	1M	1	16	.0312	.551
$a = a+s*c$	2M	1	32	.0320	.565
$a = a+s*c$	4M	1	64	.0303	.535
$a = a+b*c$	64K	1	1	.0545	1.000
$a = a+b*c$	128K	1	2	.0375	.688
$a = a+b*c$	256K	1	4	.0338	.620
$a = a+b*c$	512K	1	8	.0265	.486
$a = a+b*c$	1M	1	16	.0255	.468
$a = a+b*c$	2M	1	32	.0258	.473
$a = a+b*c$	4M	1	64	.0254	.466

The efficiency number given here is the ratio of performance with D processor domain to the performance on 1 processor. Thus we observe about a 40-50% drop in performance due to the overhead of virtual memory access. Of course the overhead would be reduced if more significant work were performed at each vector element.

3.3. Multiprocessor Overheads and Task Granularity

As discussed in the introduction, the SPS-2 copies parent data spaces to child tasks and merges pages of data on task completion that have been written into by multiple processes. Since the indices i in operations (1,2) are all distinct, it follows that if the work is divided among several tasks, no two tasks ever operate on the same element $a(i)$ and thus the merging rules are satisfied trivially in all cases. Basically this means that no merging between children should be required on task completion. This is not quite true since, except in exceptional cases, a division of the vector $a(i)$ among T tasks will generally not provide segments restricted to disjoint sets of pages. Thus some limited inter-child page merging should be expected, except in the special case that a is chosen to be page-aligned and where N is a multiple of T and N/T is a multiple of the page size.

In this study, case b) referred to above, we allocate large vectors a, b, c and we study performance as the number of tasks T is increased. We use two possible interpretations of "large vectors". In the first case, we choose fixed size vectors and vary T. As per-task data granularity decreases with increasing T, performance decreases relatively - i.e less than linear speedup is exhibited. Linear speedup may be restored by increasing the work performed per task. This we do by repeating the operations (1,2) *nit* times in each task. For constant *nit* we find that performance is sub-linear, but the deviation from linear is less the larger *nit* becomes. With the choice $nit = T$ completely linear growth is exhibited. Table 5 exhibits performance measured for several vectors lengths and using from 1 to 64 processors. The three columns correspond to the choices $nit = 1$, $nit = T/4$ and $nit = T$. For each choice we provide both Mflops and computational efficiency. The efficiency base (1.0) is the performance of a single processor on the largest array that will fit in its memory (D=1) - .0565 Mflops. The single processor was measured with $nit = 1$, which is not optimal, and efficiencies above 1.0 in the table occur for this reason.

Table 5: Fixed Vector Length
SPS-2 Saxpy Performance

			nit = 1		**nit = T/4**		**nit = T**	
Length	**Tasks**	**Domain**	**Mflops**	**Effic.**	**Mflops**	**Effic.**	**Mflops**	**Effic.**
4M	64	64	1.383	.386	3.657	1.020	4.088	1.141
256K	4	4	.139	.62	.133	.59	.206	.91
256K	8	8	.207	.45	.183	.40	.457	1.01
256K	16	16	.349	.39	.722	.80	.965	1.07
256K	32	32	.618	.34	1.849	1.02	2.069	1.14
256K	64	64	.779	.21	3.561	1.01	4.039	1.12
64K	1	1	.056	1.000				
64K	2	2	.072	.643				
64K	4	4	.129	.576	.122	.545	.215	.960
64K	8	8	.152	.339	.250	.558	.437	.975
64K	16	16	.231	.258	.589	.657	.936	1.044
64K	32	32	.317	.177	1.031	.575	1.925	1.074
64K	64	64	.309	.086	2.507	.699	3.751	1.047

The data for the case $nit = T$ are essentially linear - i.e. efficiency is 100% to within measurement accuracy. Thus we conclude that the work involved in the case $nit = T$ dominates the PARDO overheads. Within a single processor this work amounts to applying a saxpy operation at each point of a vector of length 262144/64 and repeating this 64 times, or a total of .52 million floating point operations (task granularity). Linearity was not achieved with the choice $nit = T/4$ which corresponds to a task granularity of 130K floating point operations. Our conclusion is that tasks with granularity below 100,000 operations will encounter severe inefficiencies.

In the second interpretation of "large vector" we take the vector length with $T{=}D$ tasks to be T times the vector length with one processor. This is a more appropriate measure of parallel efficiency in that vector size (i.e. work to be performed) is being matched to the domain size. In this case we observe results as in table 6 below. We note that performance is about 35% of linear in scaling from 1 to 64 processors.

Table 6: Variable Vector Length
SPS-2 Saxpy Performance

Benchmark	Length	Tasks	Domain	Mflops	Efficiency
$a = a+s*c$	64K	1	1	.056	1.000
$a = a+s*c$	128K	2	2	.071	.634
$a = a+s*c$	256K	4	4	.136	.607
$a = a+s*c$	256K	8	8	.204	.455
$a = a+s*c$	1M	16	16	.390	.435
$a = a+s*c$	2M	32	32	.757	.422
$a = a+s*c$	4M	64	64	1.389	.388

4. POISSON TYPE RELAXATION PERFORMANCE

In this section we study simple relaxation processes for 1D, 2D and 3D Poisson equations. These are typical of processes occurring in many applications codes involving either elliptic PDE solution or time evolution equations. The most direct applicability of these measurements is to performance of standard "fast solvers" for the Poisson equation. The code kernels we will describe are essentially those used in relaxation, multigrid and conjugate gradient solution of the Poisson equation. Because the Poisson equation has constant coefficients, the ratio of computational work per grid point to memory traffic is severe, and it is fair to say that while typical, these are very hard PDE to solve efficiently on a distributed memory system. To give a more realistic problem we discuss an actual time evolution system in the following section, where the same general performance is exhibited.

The three relaxation processes we study are:

$$b(i) = 2a(i) + a(i-1) + a(i+1), \tag{1}$$

$$b(j,i) = 4a(j,i) + a(j,i-1) + a(j,i+1) + a(j-1,i) + a(j+1,i), \tag{2}$$

$$\begin{aligned} b(k,j,i) = 6a(k,j,i) + a(k,j,i-1) + a(k,j,i+1) + \\ a(k,j-1,i) + a(k,j+1,i) + a(k-1,j,i) + a(k+1,j,i). \end{aligned} \tag{3}$$

Here the arrays are respectively of dimensions n_1, $n_1{\times}n_2$, $n_1{\times}n_2{\times}n_3$. The equations above are to be applied at each point of the *interior* of the corresponding 1D, 2D or 3D rectangular grid, which we will denote generically as G. Were the equations to be applied at the boundary of G, then they would index undefined points on the right hand side. This choice of relaxation scheme corresponds to imposition of Dirichlet boundary conditions in a PDE solver.

To maintain brevity we will describe only the 2D case in detail. We implement the above algorithm serially by enclosing the expression in a nested set of DO loops, one for each grid direction. For the 2D case the code looks like:

```
        do 10 j = 2,n1-1
        do 10 i = 2,n2-1
           b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1) + a(j-1,i) + a(j+1,i)
   10   continue
```

To parallelize the code with T tasks, we would like to replace each DO with a PARDO, but this in general generates too many tasks - a number equal to the grid size. Instead we will decompose the grid G into T contiguous rectangular subgrids and each of T tasks will be assigned to process a different subgrid.

The partitioning scheme used is simple. Let $T{=}T_1T_2$ be a factorization of T. Then we divide the index interval $[2,n_1{-}1]$ into T_1 essentially equal pieces and similarly we divide $[2,n_2{-}1]$ into T_2 pieces. The tensor product of the interval decompositions defines the 2D subgrid decomposition.

In case T_1 does not divide n_1-2 evenly, we can write:

$$n_1-2 = h_1T_1 + r_1, \quad 0 \le r_1 < T_1.$$

We then make the first r_1 intervals of length h_1+1 and the remaining T_1-r_1 intervals of length h_1, and similarly in the other dimension(s). This is conveniently done with a procedure

decompose $(a,b,t,istart,iend)$

which decomposes an interval $[a,b]$ into t near-equal length subintervals as above, and which initializes arrays $istart(t)$, $iend(t)$ with the start and end indices of each subinterval.

Thus the complete code to parallelize the above loop takes the form:

```
      decompose(2,n1-1,t1,istart1,iend1)
      decompose(2,n2-1,t2,istart2,iend2)
      pardo 10 q1=1,t1
      pardo 10 q2=1,t2
        do 10 i= istart1(q1),iend1(q1)
        do 10 j= istart2(q2),iend2(q2)
          b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1) + a(j-1,i) + a(j+1,i)
10    continue
```

In practice we collapse the double PARDO into a single PARDO of product length $T=T_1T_2$ in order to minimize possible recursive PARDO overhead.

The above procedure provides many different parallelizations of a given problem, one for each possible factorization of the number of tasks T. At one extreme are decompositions by rows (case T_1=1), and at the other extreme are decompositions by columns (T_2=1), with intermediate values representing decompositions by subrectangles. As we will see, performance is strongly influenced by which of these choices is made. We have in all cases found that decomposition by columns gives maximum performance. This is not a priori obvious, as in fact area-perimeter considerations would suggest that virtual memory communication would be minimized with a decomposition where $T_1 = T_2$. Two competing effects are at work: the communication bandwidth requirements are determined by the perimeter of subgrids, whereas communication overhead costs (including merging) are determined additionally by a factor proportional to the total number of data requests. The latter quantity is minimized by a column division. Row-division is unfavorable because of the Fortran rules for data storage.

In Table 7 we present the results for 1D, 2D and 3D relaxation processes on arrays of varying size and using various number of processors. We have presented the best performance measured for each task value T, which in all cases occurred with $T_1=T$. We have found only slight variations from using multiple tasks per processor (i.e. $T>D$) and have presented only results for the cases $T=D$ as a result. For each grid size, the 16-processor result is regarded as efficiency 1.0, and the higher processor number efficiency is computed relative to the 16-processor case (the problems are too large to fit on a single processor with $T=D=1$).

The measurements described here were performed by repeating the PARDO loop for the relaxation process 10 times. However only the last 9 iterations were actually timed. The first iteration plays the role of an initializer - pages of data that are randomly scattered in memory are brought to their most desirable locations during this first iteration of the PARDO. This seems to be the appropriate measurement approach to give a feeling for likely performance of iterative solvers or time evolution equations, both of which involve perhaps hundreds of iterations of a basic step similar to the one studied here. For a "one-shot" relaxation operation performance will be of order 20% lower due to overheads related to bringing in initial copies of pages. Times varied by only a percent or so if 1 iteration was timed rather than 9.

Table 7: Relaxation Performance

Dim.	Grid	D	T1	T2	T3	Mflops	Efficiency
1	1M	16	16			.587	1.000
1	1M	32	32			1.033	.880
1	1M	48	48			1.692	.961
1	1M	64	64			1.880	.801
2	$1K\times1K$	16	16	1		.889	1.000
2	$1K\times1K$	32	32	1		1.516	.853
2	$1K\times1K$	48	48	1		2.400	.900
2	$1K\times1K$	64	64	1		2.796	.786
3	128×128×64	16	16	1	1	1.063	1.000
3	128×128×64	32	32	1	1	1.955	.919
3	128×128×64	48	48	1	1	2.593	.813
3	128×128×64	64	64	1	1	3.359	.790

It is instructive to study the variation in performance for a given task number T as the task decomposition varies - we call this varying the subgrid aspect ratio, although in fact it is the *task subgrid* aspect ratio. We present sample results for 2D and 3D relaxations in Tables 8 and 9. The efficiency measures the deviation from the optimal case. Not all aspect ratios would in fact run. For heavily row-oriented ratios (e.g. T_1=1, T_2=T) the system runs out of virtual memory and kills the program unless the grid size is very small. There is no way to predict under what circumstances this will occur, nor to recover gracefully when it does occur.

Table 8: 2D Effect of Subgrid Aspect Ratio

Dim.	Grid	D	T1	T2	Mflops	Efficiency
2	$1K \times 1K$	64	8	8	.537	.191
2	$1K \times 1K$	64	16	4	1.207	.429
2	$1K \times 1K$	64	32	2	1.997	.710
2	$1K \times 1K$	64	64	1	2.811	1.000
2	512×512	64	1	64	.036	.022
2	512×512	64	2	32	.076	.047
2	512×512	64	4	16	.217	.134
2	512×512	64	8	8	.502	.310
2	512×512	64	16	4	.946	.584
2	512×512	64	32	2	1.336	.825
2	512×512	64	64	1	1.619	1.000

Table 9: 3D Effect of Subgrid Aspect Ratio

Dim.	Grid	D	T1	T2	T3	Mflops	Efficiency
3	128×128×64	64	4	4	4	1.424	.411
3	128×128×64	64	8	4	2	2.292	.661
3	128×128×64	64	16	2	2	2.286	.660
3	128×128×64	64	8	8	1	2.723	.786
3	128×128×64	64	16	4	1	3.082	.889
3	128×128×64	64	32	2	1	3.292	.950
3	128×128×64	64	64	1	1	3.465	1.000
3	66×66×66	64	1	1	64	.089	.045
3	66×66×66	64	4	4	4	.908	.463
3	66×66×66	64	64	1	1	1.963	1.000

Finally, we demonstrate for the case of 2D relaxation, that choosing task numbers T larger than the domain size D does not improve performance. Table 10 illustrates the variation in Mflops on a $1K \times 1K$ grid when the number of tasks ranges over 64, 128, 512 and 1024 on a 64 processor domain.

Table 10: 2D Effect of Many Tasks per Processor

Dim.	Grid	D	T1	T2	Mflops	Efficiency
2	$1K\times1K$	64	64	1	2.811	1.000
2	$1K\times1K$	64	128	1	2.736	.973
2	$1K\times1K$	64	512	1	2.508	.892
2	$1K\times1K$	64	1024	1	2.243	.800

4.1. Implementations of Block Ordering

We have studied two data layouts and three computational methods for implementing block-ordering of matrices.

The first data layout is the one described above where we declare array data as a 2D array, in which case Fortran allocates the data contiguously by columns. Each task processes a subarray, and clearly the columns in subarrays are also then contiguous, although successive columns in any subarray are not contiguous, and involve a substantial stride, except in the case $T_1=T$. Of course the virtual memory system plays havoc with the concept of contiguity at the word level, but it will certainly preserve the concept at the page level, and with a reasonable implementation, contiguous columns should mostly fall in contiguous pages.

The second data layout is one where we store the data for each *subgrid* contiguously: i.e. the sub-columns are stored contiguously and successive sub-columns *are* contiguous within each grid block. The second layout is equivalent to the first in case $T_1=T$ but not otherwise. The data storage in this case may be implemented as a single large linear array or as a linear array of 2D sub-arrays. Since individual tasks work almost entirely within a single block, one might expect better performance with this layout due to the improved locality of data.

The three computational methods used are:

1) the "normal" method computes the start and end indices of grid-blocks within the large 2D data arrays (layout 1) as described above, using multiple calls to the procedure *decompose*. Standard DO loops are then performed with bounds determined by *decompose*.

2) the "index" method uses an explicit address calculation to retrieve data elements from a large 1D array, ordered as in layout 2.

3) the "subroutine" method, treats the large one-dimensional array (layout 2) as a 1D array of smaller 2D arrays, and passes a pointer to the appropriate 2D array to a subroutine. Then subroutine declarations are used to equivalence the 1D input parameter array to a 2D array.

Table 11 shows the relative performance of these three methods in the case $T_1=T$, where they all involve the same data layout. The index method (recommended in the PARDO Cookbook) actually degrades performance. This is due in part to the fact that the address calculation for each array reference involves 13 integer operations and hence overshadows savings in page referencing/merging. The other two approaches are more or less comparable, with an advantage for the subroutine method in the case of small task number (i.e. large subgrids).

Table 11: Block 2D Relaxation

			Performance in Mflops		
Grid	**D**	**T**	**Normal**	**Index**	**Subroutine**
512×512	12	12	0.27	0.14	0.51
512×512	15	15	0.36	0.16	0.60
512×512	18	18	0.44	0.19	0.72
512×512	21	21	0.52	0.22	0.80
512×512	24	24	0.54	0.24	0.79
512×512	27	27	0.86	0.27	0.85
512×512	30	30	0.89	0.28	1.03
512×512	33	33	0.90	0.30	0.97
512×512	36	36	0.99	0.31	1.14
512×512	39	39	0.98	0.31	1.18
512×512	42	42	1.05	0.32	1.22
512×512	45	45	1.11	0.33	1.24
512×512	48	48	1.15	0.33	1.23
512×512	51	51	1.24	0.35	1.24
512×512	54	54	1.24	0.34	1.26
512×512	57	57	1.27	0.33	1.16
512×512	60	60	1.29	0.33	1.19
512×512	63	63	1.30	0.35	1.36

In Table 12 we present some examples of performance of the three methods in cases where $T_1 \neq T$, i.e. for non full-column decompositions. As expected, the subroutine method shows a definite advantage the further we move from the case $T_1 = T$.

Table 12: Block 2D Relaxation Comparisons

Grid	D	Mflops T	T1	T2	Normal	Index	Subroutine
512×512	15	15	15	1	0.36	0.16	0.60
512×512	15	15	5	3	0.21	0.18	0.27
512×512	15	15	3	5	0.14	0.16	0.32
512×512	30	30	30	1	0.89	0.28	1.03
512×512	30	30	15	2	0.59	0.33	1.02
512×512	30	30	10	3	0.48	0.33	0.83
512×512	30	30	6	5	0.36	0.33	0.87
512×512	30	30	5	6	0.34	0.32	0.78
512×512	45	45	45	1	1.11	0.33	1.24
512×512	45	45	15	3	0.62	0.44	1.16
512×512	45	45	9	5	0.49	0.43	1.11
512×512	60	60	60	1	1.3	0.34	1.20
512×512	60	60	30	2	1.10	0.48	1.40
512×512	60	60	15	4	0.81	0.54	1.36
512×512	60	60	12	5	0.67	0.53	1.39
512×512	60	60	10	6	0.56	0.54	1.28
512×512	63	63	63	1	1.29	0.34	1.26
512×512	63	63	21	3	0.81	0.56	1.25
512×512	63	63	9	7	0.51	0.54	1.18
512×512	63	63	7	9	0.43	0.55	1.17
512×512	63	315	315	1	1.12	0.03	1.04
512×512	63	315	105	3	0.73	0.33	1.07
512×512	63	315	63	5	0.60	0.44	1.11
512×512	63	315	45	7	0.48	0.48	1.14
512×512	63	315	35	9	0.39	0.50	1.09

5. A COMPLETE APPLICATION - THE SHALLOW WATER EQUATIONS

As another example of the current capabilities of massively parallel architectures, we describe the implementation of a standard two-dimensional atmospheric model - the Shallow Water Equations - on the SPS-2. These equations provide a primitive but useful model of the dynamics of the atmosphere or of certain ocean systems. Because the model is simple, yet captures features typical of more complex codes, the model is frequently used in the atmospheric sciences community to benchmark computers[2]. Furthermore, the model has been extensively analyzed mathematically and numerically[3,4]. We have recently implemented the Shallow Water Equations model on a range of parallel machines (including Connection Machine CM-2, Intel iPSC/860, SUPRENUM-1, ES-1) using both explicit[5-7] and spectral[8] solution methods for the equations.

The Shallow Water Equations, without a Coriolis force term, take the form

$$\frac{\partial u}{\partial t} - \zeta v + \frac{\partial H}{\partial x} = 0\,,$$
$$\frac{\partial v}{\partial t} - \zeta u + \frac{\partial H}{\partial y} = 0\,,$$
$$\frac{\partial P}{\partial t} + \frac{\partial Pu}{\partial x} + \frac{\partial Pv}{\partial y} = 0\,,$$

where u and v are the velocity components in the x and y directions, P is pressure, ζ is the vorticity: $\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ and H, related to the height field, is given by: $H = P + (u^2 + v^2)/2$. It is required to solve these equations in a rectangle $a \le x \le b, c \le y \le d$. Periodic boundary conditions are imposed on u, v, and P, each of which satisfies $f(x+b, y) = f(x+a, y)$, $f(x, y+d) = f(x, y+c)$.

A scaling of the equations results in a slightly simpler format. Introduce mass fluxes $U=Pu$ and $V=Pv$ and the potential velocity $Z=\zeta/P$, in terms of which the equations reduce to:

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0\,,$$
$$\frac{\partial v}{\partial t} + ZU + \frac{\partial H}{\partial y} = 0\,,$$
$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0\,.$$

5.1. Discretization

We have discretized the above equations on a rectangular staggered grid with periodic boundary conditions. The variables P and H have integer subscripts, Z has half-integer subscripts, U has integer and half-integer subscripts, and V has half-integer and integer subscripts respectively.

Initial conditions are chosen to satisfy $\vec{\nabla}\cdot\vec{v}=0$ at all times. We time difference using the Leap-frog method. We then apply a time filter to avoid weak instabilities inherent in the leap-frog scheme:

$$F^{(n)}=f^{(n)}+\alpha\,(f^{(n+1)}-2f^{(n)}+f^{(n-1)})\,,$$

where α is a filtering parameter. The filtered values of the variables at the previous time-step are used in computing new values at the next time-step. For a complete description of the discretization we refer to[2].

5.2. CRAY Fortran Code

It is interesting to compare the actual code for the CRAY and SPS-2 implementations. The Fortran code implementing the above explicit algorithm involves a 2D rectangular grid with variables: $u(i,j)$, $v(i,j)$, $p(i,j)$, $z(i,j)$, $psi(i,j)$, $h(i,j)$. There are three main loops, two corresponding to the leap-frog time propagation of various quantities, and one for the filtering step. A typical code sequence, used in the updating of the U, V and P variables, is:

```
  do 200 j=1,n
  do 200 i=1,m
   unew(i+1,j) = uold(i+1,j)+
          tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)+cv(i,j+1)+cv(i,j)
          +cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
   vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1)+z(i,j+1))
          *(cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j))
          -tdtsdy*(h(i,j+1)-h(i,j))
   pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
          -tdtsdy*(cv(i,j+1)-cv(i,j))
200 continue
```

Each such loop is followed by code to implement the periodic boundary conditions. Note that there are loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

5.3. SPS-2 Code

For the SPS-2 implementation, we simply parallelize the code in the way described in section 4, introducing two calls to *decompose* followed by a double PARDO over tasks. Each task performs a double DO loop using indices provided by *decompose*. That is the only code modification required for the main loops. Similarly the boundary condition loops are parallelized by treating them as 1D grids and using a single call to *decompose* for each side of the subgrid. In practice task granularity for the boundary loops can become very small and so we have used a smaller number of tasks to parallelize the boundary than were used in the interior of subgrids.

5.4. Performance Results: SPS-2 and other Systems

We have compared the explicit model on a CRAY-YMP[*], on the Connection Machine CM-2, on the Intel iPSC/860, on the SUPRENUM-1 and on the SPS-2. In each case we have solved the largest grid size that would fit in memory.

The CRAY-YMP performed the explicit computations at 1532 Mflops with 8 processors on a 512×512 grid. Connection Machine CM-2 32-bit performance was 4475 Mflops on a 2048×2048 grid, and was 2,893 Mflops in 64-bit arithmetic.

The SPS-2 performed the explicit model at 2.71 Mflops with 64 processors on a 1024×1024 grid, and performed at .117 Mflops on a single processor with a correspondingly smaller grid. Thus parallel efficiency is about 36%. As with the relaxation examples, we studied many potential ways to decompose the grid among the tasks. The most efficient proved to be to use full column slices.

Table 13: Performance of Shallow Water Equations

Machine	Processors	Grid Size	Mflops
CM-2	65,536	8096×8096	4475.
CRAY-YMP	8	512×512	1532.
CRAY-XMP	4	512×512	560.
iPSC/860	128	1024×2048	543.
SUPRENUM-1	16	512×512	72.
SPS-2	64	1024×1024	2.710
MIPS	1	128×128	1.431
SUN Sparc 1	1	256×256	1.389
SPS-2	1	128×128	.117

ACKNOWLEDGEMENTS

Most of the development work for the computations described here was performed on the SPS-2 computer at the Center for Applied Parallel Processing (CAPP), University of Colorado. We would also to thank Myrias Research Corporation for providing helpful tips on maximizing performance.

[*] All CRAY measurements were performed by R. Sato of NCAR.

References

1. O. McBryan and R. Pozo, ''Performance Evaluation of the Myrias SPS-2 Computer,'' CS Dept Technical Report, University of Colorado, Boulder, 1990.
2. G.-R. Hoffman, P.N. Swarztrauber, and R.A. Sweet, ''Aspects of using multiprocessors for meteorological modeling,'' in *Multiprocessing in Meteorological Models*, ed. D. Snelling, pp. 126-195, Springer-Verlag, Berlin, 1988.
3. R. Sadourny, ''The dynamics of finite difference models of the shallow water equations,'' *JAS*, vol. 32, pp. 680-689, 1975.
4. G.L. Browning and H.-O. Kreiss, ''Reduced systems for the shallow water equations,'' *JAS*, to appear.
5. O. McBryan, ''New Architectures: Performance Highlights and New Algorithms,'' *Parallel Computing*, vol. 7, pp. 477-499, North-Holland, 1988.
6. O. McBryan and R. Pozo, ''Performance Evaluation of the Evans and Sutherland ES-1 Computer,'' CS Dept Technical Report, University of Colorado, Boulder, 1990.
7. O. McBryan, ''A Comparison of the Intel iPSC860 and SUPRENUM-1 Parallel Computers,'' *Supercomputer*, 1991, to appear.
8. O. McBryan, ''Connection Machine Application Performance,'' in *Scientific Applications of the Connection Machine,*, ed. Horst Simon, pp. 94-115, World Scientific Publishing Co., Singapore, 1989.