

## PERFORMANCE ANALYSIS OF THE SCALABLE MODELING SYSTEM

D. SCHAFFER\* AND J. MIDDLECOFF\*

*NOAA Research-Forecast Systems Laboratory  
Boulder, Colorado*

*E-mail: sms-info@fsl.noaa.gov*

*\*[In collaboration with the Cooperative Institute for Research in the Atmosphere (CIRA),  
Colorado State University, Fort Collins]*

M. GOVETT

*NOAA Research-Forecast Systems Laboratory  
Boulder, Colorado*

T. HENDERSON

*National Center for Atmospheric Research, Climate and Global Dynamics Division  
Boulder Colorado*

The Scalable Modeling System (SMS) is a directive-based parallelization tool. The user inserts directives in the form of comments into existing Fortran code. SMS translates the code and directives into a parallel version that runs on shared and distributed memory high-performance computing platforms. Directives are available to support array re-sizing, inter-process communications, loop translations, and parallel output. SMS also provides debugging tools that significantly reduce code parallelization time. SMS is intended for applications using regular structured grids that are solved using explicit finite difference approximation (FDA) or spectral methods. It has been used to parallelize ten atmospheric and oceanic models but the tool is sufficiently general that it can be applied to other structured grids codes.

The performance of SMS parallel versions of the Eta atmospheric and Regional Ocean Modeling System (ROMS) oceanic models is analyzed. The analysis demonstrates that SMS adds insignificant overhead compared to hand-coded Message Passing Interface (MPI) solutions in these cases. This research shows that, for the ROMS model, using a distributed memory parallel approach on a cache-based shared memory machine yields better performance than an equivalent shared-memory solution due to false sharing. We also find that the ability of compilers/machines to efficiently handle dynamically allocated arrays is highly variable. Finally, we show that SMS demonstrates the performance benefit gained by allowing the user to explicitly place communications. We call for extensions of the High Performance Fortran (HPF) standard to support this capability.

## 1 INTRODUCTION

In the early days of parallel supercomputers, shared memory vector processors comprised the dominant architecture. Most large-scale atmospheric and oceanic models ran on these machines. Although the hardware was expensive, it was relatively easy to do the programming required to obtain good performance. Usually, this was simply a matter of ensuring that the array lengths passed to the vector pipes were sufficiently large (facilitated by smart compilers) and that the parallel code scaled to a few processors; this task is easily achievable with simple multi-tasking directives.

These vector machines are still alive today. The Japanese Earth Simulator, at \$400 million, comprises NEC SX-6 vector platforms, and as of November 2002 was the top-ranked high performance computer in the world (36 teraflop/s) as measured by the LINPACK benchmark suite<sup>1</sup>. However, since the early 1990s, there has been a shift toward distributed memory commodity microprocessor-based solutions. This approach enables high performance computing to be provided at a relatively lower cost.

A recent example demonstrates why this approach is attractive. The National Oceanic and Atmospheric Administration Forecast Systems Laboratory's supercomputer is a Linux cluster constructed from 1500 commodity Intel processors. Its LINPACK performance is 3.6 teraflop/s (TF): sufficient enough for eighth in the November 2002 ranking of the world's fastest supercomputers. While only one-tenth as powerful as the Earth Simulator, its price/performance is \$1.5 million/TF compared to \$11 million/TF for the Simulator.

This method of calculation neglects the hidden cost of developing and maintaining efficient parallel versions of models that are suitable for distributed memory architectures. The required effort includes decomposing the data and computations, identifying where communication is required, and making the appropriate calls to Message Passing Interface (MPI)<sup>2</sup> subroutines to implement this parallelization. The final product is a code crowded with parallel programming constructs that often obscure the scientific calculations

expressed by the model. This results in higher maintenance costs as scientists (often unskilled in writing MPI programs) struggle to keep the distributed memory parallel coding constructs and MPI subroutine calls updated with the evolving serial code. Fundamentally, this is a difficult problem because scientists are forced to do parallel programming at a low level, akin to writing the serial code in assembly language.

A variety of approaches have been developed to attempt to alleviate some of this burden. For example, High Performance Fortran (HPF) is an attempt to automatically parallelize Fortran codes. Despite support by parallel computer vendors and scientific institutions, it has largely failed to live up to the promise of high-performance and ease of use. One shortcoming of HPF is the compiler's inability to determine the optimal locations of inter-processor communication. Although code restructuring and insertion of additional HPF directives can help the compiler do a better job, additional effort is required by the programmer and performance is often significantly less than hand-coded solutions<sup>3</sup>. Renewed efforts to optimize HPF have achieved some success<sup>4</sup>; however performance still lags MPI-based hand-coded solutions by at least 15 percent (often much more), and code restructuring is often required.

Another approach, OpenMP, is a standard for directive-based parallelization targeted for shared and distributed-shared memory applications<sup>5</sup>. A fine grain parallelization can be very quickly constructed using directives at the loop level. However, scalability beyond a few processors requires implementation of a coarse grain parallelization as in the case of ROMS<sup>6</sup>. OpenMP has become more widely used recently as part of hybrid MPI/OpenMP parallel implementations on clusters of Symmetric Multiple Processors (SMPs) as demonstrated by the presentations at this ECMWF conference. Cocke and Christidis<sup>7</sup> show the utility of this approach over a straight MPI implementation. OpenMP has been successful because the use of directives hides detail, simplifying the parallelization process. Even more significantly, it has now become a standard supported by most major vendors.

SMS, similar to HPF, is a directive-based approach that employs a distributed-memory programming model. The user adds directives (comments) to the code that indicate, for example, how data and computations are distributed among the processors. SMS translates the directives and serial code into a parallel version that runs correctly on both shared and distributed memory platforms. Once the parallel code is working, various performance optimizations can be added as described in Section 2. SMS provides support for much but not all of Fortran 90. In contrast to HPF, the user is required to explicitly place communication directives into the code. In this paper, SMS performance is analyzed in detail. Section 2 reviews the key features of SMS. Section 3 examines performance of SMS versions of an atmospheric and an oceanic model. Section 4 concludes by proposing that the SMS prototype demonstrates the need to extend HPF to allow explicitly placed communication.

## 2 KEY FEATURES OF SMS

Since Govett, et al.<sup>8</sup> and Henderson, et al.<sup>9</sup> describe SMS in detail, this section will focus on key features of the tool. In SMS, data are distributed using the **DECLARE\_DECOMP**, **CREATE\_DECOMP**, and **DISTRIBUTE** directives. The first two simply enable the user to identify the existence of a decomposition, specifying the number of dimensions and the halo sizes. The third directive, **DISTRIBUTE**, is key because it describes how arrays are decomposed among the processors. For example,

```
csms$distribute(my_decomp, 1, 2) begin  
    real x(im, jm, km)  
csms$distribute end
```

indicates that the first and second dimensions of  $\mathbf{x}$  are decomposed in blocks.

In HPF, a distribute directive provides the same functionality. Based on its distribute directive and adjacent dependencies prescribed

by the code, HPF automatically places the communications needed to satisfy these dependencies. In contrast, SMS requires the user to determine where communication is needed and then manually place it using the **HALO\_UPDATE** directive. Although this imposes additional burden on the scientist, it underscores the key weakness of HPF in terms of performance; it is simply too difficult for an HPF compiler to optimally place communication. To illustrate these points, suppose a code has three subroutines and a main program:

```
subroutine A
w(1:im) = w(1:im) + (x(0:im-1) + x(2:im+1))/2.0
```

```
subroutine B
w(1:im) = w(1:im) + (y(0:im-1) + y(2:im+1))/2.0
```

```
subroutine UPDATE
! Updates x and y
```

```
program MAIN
do i=1,iterations
  call UPDATE
  call A
  call B
end do
```

Assume that the arrays  $w$ ,  $x$ , and  $y$  are decomposed using the distribute directive in both the HPF and SMS cases. In both subroutines there are adjacent dependencies: the computation of  $w$  at any point  $i$  depends on values of  $x$  (or  $y$ ) at  $i-1$  and  $i+1$ . HPF will do the halo updates of  $x$  and  $y$  separately, just prior to the calculation of  $w$  in each case. On the other hand, in the case of SMS, the user can aggregate the communication of  $x$  and  $y$ , reducing latency:

```
do i = 1, iterations
  call UPDATE
```

```
csms$halo_update(x,y)  
  call A  
  call B  
end do
```

The downside of the SMS approach is the added burden of correctly placing communication directives and maintaining them as the code evolves. SMS provides two debugging directives to mitigate this additional effort. The **CHECK\_HALO** directive enables the user to assert that the halo region of a variable is up to date. If the assertion proves false then an error message is printed and the parallel run stops. For example, the **CHECK\_HALO** directive could be inserted prior to the calculation of  $w$  in subroutine *A* above:

```
csms$check_halo(x, "Checking x before the calculation of w")
```

If the assertion fails, the user knows that a **HALO\_UPDATE** directive is missing or misplaced.

The second debugging directive, **COMPARE\_VAR**, identifies cases where any variable, decomposed or non-decomposed, has different values for two separate runs of the parallel code. In the example above, suppose the **COMPARE\_VAR** directive is inserted after the calculation of  $w$  in subroutine *A*:

```
csms$compare_var(w, "Checking w after it is computed")
```

At runtime, the user can, as an option, simultaneously launch two separate runs of the code (i.e., 1 and 4 processor runs). When the two runs reach the **COMPARE\_VAR** directive, the values of  $w$  are compared and, if there is a difference, an error message is printed and the two runs stop. The **COMPARE\_VAR** is more general than **CHECK\_HALO** in that it also uncovers parallel bugs due to other causes such as failure to decompose an array or execute a global summation.

In addition to allowing the user to explicitly place communication, SMS provides other optimizations. One, the ability for output to be done asynchronously with computations, is implemented by having the compute processors send their data to a designated server processor. The compute processors can then continue with the next set of computations while the server writes to disk. Also, SMS will automatically use the platform-dependent optimal underlying communication library (currently MPI or SHMEM). Finally, at runtime, SMS allows the user to choose a processor layout (a mapping of grid points to processors). The performance benefits of these optimizations are discussed in Section 3.

### 3 PERFORMANCE ANALYSIS

Govett, et al.<sup>8</sup> examined some performance aspects of SMS. This section extends their analysis for two models: the NOAA National Centers for Environmental Prediction Center (NCEP) Eta model and the Rutgers University ROMS model. Dedicated machine time was not available for any of the results presented; instead, the minima of multiple repetitions are recorded. The results cover a variety of platforms whose specifications are shown in **Table 1**.

Eta<sup>10</sup> is a mesoscale weather prediction model used to produce daily weather forecasts for the United States National Weather Service. Govett, et al.<sup>8</sup> compared performance of a hand-coded MPI version of the model run operationally at NCEP with an equivalent SMS version. The MPI version was optimized by IBM for the SP3 architecture. The resolution tested was 223x365x45. **Table 2** is a reprint of the results of the comparison.

The SMS version beats the MPI version at all processor counts. As Govett, et al.<sup>8</sup> point out, this is largely due to the smaller amount of time the SMS code spends doing halo updates. **Table 3** shows these previously unpublished times. The differences arise from the fact that the MPI version communicates the data using nearly twice as many MPI calls as the SMS version, resulting in additional latency.

**Table 1** Hardware specifications for platforms used in SMS performance analysis. The interconnect for the IA and Alpha machines is Myrinet 2000.

Name	Chip	Clock Speed (GHz)	CPUs per node	Memory Bandwidth per PE (GB/s)	FLOP per Cycle	L2 Cache (MB)
T3E	EV5	0.3	1	1.2	2	0.1
IA-32	i686	2.2	2	1.6	2	0.5
Alpha	EV67	0.833	2	1.3	2	4.0
SP3	Power3	0.375	4	0.3	4	8.0
O3K	R14000	0.6	64	1.4	2	8.0
IA-64	Itanium	0.8	2	3.2	2	4.0

**Table 2** Performance of the MPI and SMS versions of the Eta model run on the NCEP IBM SP3. Run times are given in seconds for a full 48-hour model run including initialization and the generation of hourly output files.

Number of Processors	MPI-Eta Time	SMS-Eta Time	SMS faster	SMS-Eta Efficiency
4	11197	10781	4 %	1.00
8	5317	5258	1 %	1.03
16	2878	2774	4 %	0.97
32	1471	1446	2 %	0.93
64	872	820	6 %	0.82
88	694	643	7 %	0.76

An additional experiment was performed in which the SMS version directives were modified so that the number of calls to the communications layer was the same as for the hand-coded MPI version. In this case, at 88 processors, the halo-update time was 238 seconds, roughly equivalent to the MPI version (243 seconds).

We extend the Eta analysis further by examining the benefit of using asynchronous output in SMS. A separate run using the same resolution was executed on the IA-32 machine using 88 processors. This time, output was written at model hours 0, 10, 20, 30, and 40. With asynchronous output turned off, the runtime was 684 seconds, of which 164 was spent doing output. With asynchronous output enabled, the output time was reduced to 54 seconds. This remaining

time is that required for the compute processors to send their pieces of the output to the server processor. The savings of 110 seconds (a 15% improvement) was due to the fact that the server was writing the data to disk at the same time the compute processors proceeded with the next set of computations, hiding the disk write cost.

**Table 3** Halo update times for the MPI and SMS versions of the 48-hour Eta model run on the IBM SP3.

Number of Processors	MPI-Eta Halo Update Time	SMS-Eta Halo Update Time
4	1013	893
8	771	631
16	614	526
32	388	348
64	294	249
88	243	214

The ROMS version benchmarked here is identical to that studied by Govett, et al.<sup>8</sup>. The resolution is 130x130x30. As discussed by the authors, the SMS parallel version was constructed using the existing shared memory parallel code. To simplify the SMS parallelization, the static memory (common block) serial code was converted to dynamic memory (Fortran 90 allocatable arrays) during the compilation process. **Table 4** shows the runtimes for various configurations on the different platforms.

To begin, we examine the single processor performance. The table row labeled “%Peak” shows a large variance in the ratio of sustained to peak performance for the serial (static memory) code over the different machines. Overall, this is not surprising since no attempt was made to optimize the scalar code for each machine. The IA-32 processor results are particularly poor, partly due to the very small L2 cache on these processors, as given in **Table 1**. It may also be that the compiler was not able to take advantage of the “vector” 128-bit registers on the chip in some cases.

**Table 4** Runtimes of the ROMS main model loop for time steps 2-21 on various platforms. “Static” refers to the static memory (common block) serial code. “Alloc” refers to the same code except common blocks are replaced with Fortran 90 allocatable arrays. The other rows refer to SMS parallel times for the given processor counts. “PG” indicates results when SMS and the ROMS model are compiled with the Portland Group Fortran 90 compiler. “IFC” indicates results when the code was compiled using the Intel compiler. The “% Peak” field refers to the percentage of the theoretical peak performance attained by the serial (static) code. The theoretical peak is clock speed \* maximum floating point operations per cycle. The “Eff” field is the parallel efficiency relative to the serial code for that machine. The attained performance is the runtime divided by the measured number of floating point operations (23 GFLOP).

Mach/ Config	T3E	IA-32 PG	IA-32 IFC	Alpha	O3K	SP3	IA-64
Static/ %Peak	n/a	83.4 6.6%	72.5 7.6%	81.7 18.5%	103.5 17.6%	106.8 14.4%	159.6 8.6%
Alloc/ Eff	n/a	91.3 <b>0.91</b>	76.7 <b>0.95</b>	84.4 <b>0.97</b>	102.2 <b>1.01</b>	114.7 <b>0.93</b>	160.0 <b>0.99</b>
1x1 PE/ Eff	n/a	91.8 <b>0.91</b>	76.8 <b>0.94</b>	89.7 <b>0.91</b>	106.6 <b>0.97</b>	116.1 <b>0.92</b>	161.8 <b>0.99</b>
1x2 PE/ Eff	n/a	63.0 <b>0.66</b>	59.3 <b>0.61</b>	47.9 <b>0.85</b>	51.4 <b>1.01</b>	59.6 <b>0.90</b>	80.7 <b>0.99</b>
2x2 PE/ Eff	n/a	29.2 <b>0.71</b>	26.7 <b>0.68</b>	27.0 <b>0.76</b>	29.0 <b>0.89</b>	37.5 <b>0.71</b>	40.9 <b>0.98</b>
2x4 PE/ Eff	105.5 n/a	14.0 <b>0.75</b>	12.6 <b>0.72</b>	13.8 <b>0.74</b>	14.7 <b>0.88</b>	18.8 <b>0.71</b>	20.7 <b>0.96</b>
4x4 PE/ Eff	55.0 n/a	7.7 <b>0.68</b>	6.9 <b>0.66</b>	8.2 <b>0.62</b>	8.1 <b>0.80</b>	11.2 <b>0.60</b>	12.5 <b>0.80</b>
4x8 PE/ Eff	29.2 n/a	4.5 <b>0.58</b>	4.1 <b>0.55</b>	5.1 <b>0.50</b>	4.5 <b>0.72</b>	7.0 <b>0.48</b>	7.7 <b>0.65</b>

The T3E results are not available for fewer than 8 processors because the code does not fit in memory. However, since the code scales nearly perfectly between 8 and 16 processors, we can estimate the single processor run-time as eight times the 8 processor run-time. The result is 844 seconds or 27.3 megaflop/s (MF) per processor. The T3E peak performance is 600 MF so the estimated single processor performance is 4.4% of peak. This poor value is likely due in part to the very small cache on the T3E. Also, since the resolution is small and the I dimension is decomposed in each of the reported cases in **Table 4**, vector lengths are short and so the effectiveness of the T3E memory streams hardware is reduced. Overall, comparison of performance of the static and allocatable array versions shows that some machines/compiler have difficulty attaining the same

performance as the equivalent static memory version. This observation is explored in depth later in this section.

Turning to the SMS parallel version, the results for 1 PE indicate a drop-off for some of the platforms compared to the dynamic memory version of the serial code. When the SMS version is created, halo points are added to all decomposed arrays (as would be the case for a hand-coded MPI version). The additional memory changes the cache behavior of the model. Further investigation would be needed to explain why this causes measurably poorer cache re-use on some platforms but not others.

The remaining rows in **Table 4** show how the SMS parallel version scales. In several cases, there are steep drop-offs when all processors on a node are used, notably the IA-32 (2 cpus/node) and the SP3 (4 cpus/node). This likely occurs because the demand for memory access created by using all CPUs on a node exceeds the available bandwidth. This memory contention does not occur on the IA-64 because it has, for example, twice the memory bandwidth and 8 times the amount of L2 cache as compared to the IA-32. The scalability drops off at 16 and 32 processors for all machines as communication, loop overhead, etc. begin to dominate. The SP3 is the worst case since it has the highest ratio of communication time (**Table 5**) to computation time.

The impact of SMS halo updates on scalability is now considered. The first time each halo directive is encountered at runtime, the communication patterns are stored away to avoid re-computing them. For the remainder of the model run, halo updates are implemented using a four-step process:

1. Search through the list of cached communication patterns to find the correct one.
2. Pack the data to be communicated into buffers.
3. Communicate the contents of the buffers to the appropriate processors.
4. Unpack the received data into the halo regions of the arrays.

Each of these steps was timed within the main model loop. Barriers were placed prior to turning on the timers in order to eliminate the effects of process skew. The sums of these steps are the communication times shown in **Table 5**. The first step is overhead that should not occur in a hand-tuned, application-specific MPI version. This overhead as a fraction of the total communication time is also listed in the table. Although not shown, further measurements indicate that between 30% and 60% of the searching process could be eliminated by directly associating each directive with a cached communication pattern.

**Table 5** Communication times and SMS overhead for various platforms and processor counts. “sms+” refers to the overhead added by SMS as a percentage of the total communication time.

Mach Pes sms+	T3E	IA-32 PG	IA-32 IFC	Alpha	O3K	SP3	IA-64
2 sms+	n/a	1.13 <b>8.8%</b>	1.03 <b>7.1%</b>	1.32 <b>6.5%</b>	0.73 <b>4.0%</b>	0.66 <b>10.8%</b>	0.82 <b>10.8%</b>
4 sms+	n/a	1.64 <b>6.3%</b>	1.54 <b>5.0%</b>	1.77 <b>3.8%</b>	1.29 <b>2.5%</b>	1.05 <b>8.3%</b>	1.54 <b>6.1%</b>
8 sms+	3.92 <b>5.0%</b>	1.71 <b>5.6%</b>	1.61 <b>4.6%</b>	1.85 <b>3.5%</b>	1.34 <b>2.3%</b>	1.93 <b>4.6%</b>	1.82 <b>5.0%</b>
16 sms+	3.99 <b>4.8%</b>	1.59 <b>5.2%</b>	1.52 <b>4.4%</b>	1.62 <b>3.4%</b>	1.36 <b>2.3%</b>	2.31 <b>3.8%</b>	1.64 <b>5.6%</b>
32 sms+	3.39 <b>5.7%</b>	1.40 <b>5.1%</b>	1.33 <b>4.6%</b>	1.49 <b>3.5%</b>	1.27 <b>2.2%</b>	2.30 <b>3.9%</b>	1.51 <b>6.0%</b>

As mentioned in Section 2, SMS is designed to automatically choose an underlying communications package depending on the target platform. On the T3E, this is the SHMEM library. An experiment was conducted in which SHMEM calls were replaced with MPI calls. The measured communication times were 20% higher than those reported in **Table 5**.

We further examine the scalability of the SMS version of ROMS by looking at the performance for different processor layouts and by comparing it to the shared memory parallel results. **Table 6** shows these performance numbers for the Origin 3000. Surprisingly, the SMS 2x8 layout outperforms the 4x4 layout. Consider first the

computations. For a 2x8 layout, the length in the I dimension is longer than a 4x4 layout. This provides more opportunity for software pipelining and pre-fetching of cache lines.

**Table 6** Main model loop runtimes and efficiencies of the SMS and shared memory versions of ROMS on the Origin 3000 for various processor layouts. The efficiencies are relative to the shared memory time for a 1x1 processor layout.

PE Layout	Shared Memory Time	Shared Memory Efficiency	SMS Time	SMS Efficiency
1x1	103.5	<b>1.00</b>	106.6	<b>0.97</b>
1x2	51.80	<b>0.99</b>	51.46	<b>1.01</b>
1x4	26.40	<b>0.98</b>	27.43	<b>0.94</b>
2x2	35.29	0.73	29.01	0.89
1x8	13.98	<b>0.93</b>	14.72	0.88
2x4	18.75	0.69	14.65	<b>0.88</b>
1x16	7.72	<b>0.84</b>	8.41	0.77
2x8	9.85	0.66	7.86	<b>0.82</b>
4x4	n/a	n/a	8.08	0.80
1x28	5.79	<b>0.64</b>	5.51	0.67
4x7	8.87	0.42	4.99	<b>0.74</b>
1x32	5.36	<b>0.60</b>	5.13	0.63
4x8	7.62	0.42	4.53	<b>0.71</b>

In terms of communication, the standard presumption is that the communications time would be shorter for the 4x4 layout since the perimeter is smaller. Although not shown in the table, in this case the 4x4 communication time (1.32 seconds) is longer than the 2x8 communication time (1.19 seconds). Further measurements showed that all of this difference was due to increased packing and unpacking times for the 4x4 case. It is possible that software pipelining and cache-line pre-fetching play a role here as well. Despite the smaller

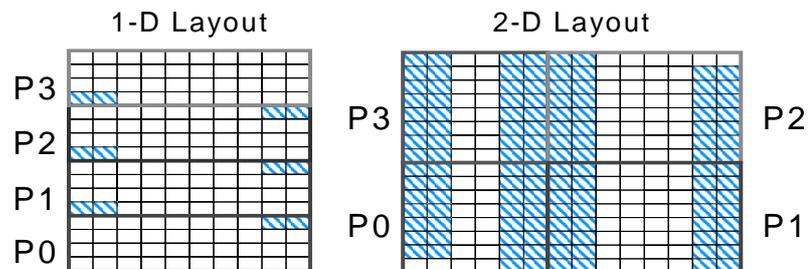
perimeter for the 4x4 case, the actual MPI communication time (step 3 above) was identical for both two cases. Although this requires further investigation, we speculate that latency rather than bandwidth was the dominant factor. The improvement gained by using SMS configuration files to specify a 2x8 layout rather than the default 4x4 layout demonstrate the added value of this SMS feature.

When considering the shared memory parallel version of the ROMS model, we see from **Table 6** that the 1-D layouts outperform 2-D layouts in every case. This is due to the overhead associated with false sharing in the shared memory code. Measurement of secondary cache misses and cache line invalidations by other processors (**Table 7**) supports this argument. The Barotropic Step function profiled in the table consumes the largest portion of runtime in the model. The 2-D shared-memory layout has 4.5 times as many secondary cache misses as the 1-D layout.

**Table 7** Secondary cache misses and invalidations by external processors for the Barotropic Step function as measured by the SGI Speedshop tool during the ROMS main model time step.

Parallel Version	PE Layout	Secondary Cache Misses	External Invalidations
Shared Memory	1x32	4700	7800
Shared Memory	4x8	20000	21000
Distributed Memory	1x32	600	400
Distributed Memory	4x8	400	500

The external cache invalidations reflect a similar story. **Figure 1** illustrates why this is happening. For a 2-D layout, there are far more opportunities for cache misses than a 1-D layout.



**Figure 1** Schematic in which false sharing occurs for 1-D and 2-D processor layouts for shared memory parallelism. False sharing occurs when two processors shared a cache line but not the actual data. So, for example, in the 1-D layout, the last two grid points in the last row of the piece of the array “owned” by P0 map to the same cache line as the first two grid points in the first row of the array piece “owned” by P1. Thus, when P0 writes to that cache line, it invalidates it for P1 and the converse happens when P1 writes to the cache line. If this writing occurs simultaneously then a thrashing effect occurs which degrades performance.

Since the shared-memory parallel version is implemented in a coarse grain fashion, where the parallel loops cover a significant number of computations, it should be efficient enough to reasonably compare it with the SMS (distributed memory parallel) performance. **Table 6** shows that while the codes are fairly equivalent up through 16 processors, subsequently the SMS version beats the shared memory version. Again, this is due to false sharing in the shared memory code. **Table 7** shows that the secondary cache invalidations and external invalidations are negligible in the distributed memory code as compared to shared memory. For a distributed memory parallel code, the data for each processor are stored in completely separate memory locations, eliminating the possibility of false sharing.

We conclude our analysis of the SMS ROMS performance by looking at how the use of dynamic memory impacts the scalability of the parallel code. Since a static memory SMS parallel version was not available, a stand-alone kernel was constructed from the ROMS

Barotropic step routine, at the same resolution. **Figure 2** shows four variations that were created.

Performance of the “allocatable,” “derived-type/pointer,” and “de-referenced pointer” cases were compared to the original “static” case for various platforms and processor counts. As mentioned, to simplify the SMS parallelization, the ROMS code was converted to use allocatable arrays during the compilation process. **Table 8** shows the effect of this conversion. At 32 processors, the “allocatable array” version of the kernel executes at only 64% the efficiency of the original static memory scheme on the IA-32 machine when the Portland Group Compiler is used. In most other cases, the allocatable array scheme does not negatively impact performance as significantly. It is also remarkable that, for 32 processors on the SGI Origin 3000, despite taking a hit for using dynamic memory, the SMS version outperforms the shared memory parallel version (**Table 6**).

In **Table 9**, we see that the performance drop-off for using Fortran 90 pointers with derived types is quite dramatic for the Portland Group compiler case. The SGI Origin 3000 results worsen as well. This could be due to extra precaution the compiler takes when dealing with pointers since they can point to overlapping locations in memory.

To work around these compiler weaknesses, a common trick is to de-reference the pointers by passing them as arguments to the computationally intensive subroutines. **Table 10** shows the results for this scheme. The T3E and IA-32 performances are noticeably better. For the T3E, the “de-referenced” case is actually better than the “allocatable” case.

## ROMS Barotropic Step Kernel

Static:

```
common /c1/ U(IM,JM,KM)
call KERNEL          ! references U
```

Allocatable:

```
real, allocatable :: U(:,:,:)
call KERNEL          ! references U
```

Derived Type/Pointer:

```
type t1
  real, pointer :: U(:,:,:)
end type t1
type (t1) :: o1
call KERNEL          ! references o1%U
```

Dereferenced Pointer:

```
call KERNEL(o1%U)   ! References dummy arg
```

**Figure 2** Illustration of how the ROMS Barotropic Step kernel was constructed with various Fortran coding constructs. In the original code, main model variables were stored in common blocks. In the second variation, the common blocks were replaced with Fortran 90 allocatable arrays. The third variation stores the model variables as Fortran 90 pointers within objects of derived types. The final variation is the same as the third except the pointers are de-referenced by passing them to the kernel subroutine. In this case, within the subroutine, the variables are declared as simple array dummy arguments.

**Table 8** Efficiency of the allocatable array scheme for managing memory in the ROMS Barotropic step kernel relative to the common block variation for various platforms and processor counts. The resolution is 130x130x30.

Mach/ Pes	T3E	IA-32 PG	IA-32 IFC	Alpha	O3K	SP3	IA-64
1	n/a	0.89	1.00	1.02	0.85	0.95	0.98
2	0.96	1.00	1.02	0.99	0.89	0.98	1.03
4	0.88	0.98	0.94	0.97	0.95	0.95	0.99
8	0.88	0.92	0.95	1.01	0.92	0.97	0.99
16	0.92	0.85	0.89	0.92	0.84	0.97	1.00
32	0.89	0.64	0.84	0.97	0.85	0.96	n/a

**Table 9** The same as Table 8 except for the derived-type/pointer scheme.

Mach/ Pes	T3E	IA-32 PG	IA-32 IFC	Alpha	O3K	SP3	IA-64
1	n/a	0.55	0.89	0.68	0.80	0.94	0.98
2	0.94	0.81	1.02	0.74	0.84	1.00	1.03
4	0.87	0.69	0.92	0.68	0.92	0.79	0.99
8	0.87	0.56	0.84	0.70	0.90	1.00	0.99
16	0.90	0.45	0.77	0.72	0.82	0.99	1.00
32	0.88	0.27	0.84	0.80	0.82	0.98	n/a

**Table 10** The same as tables 8 and 9 except for the de-referenced pointer scheme.

Mach/ Pes	T3E	IA-32 PG	IA-32 IFC	Alpha	O3K	SP3	IA-64
1	n/a	0.93	0.97	0.98	0.86	0.95	0.98
2	1.02	1.02	1.03	0.96	0.89	1.00	1.03
4	0.95	0.98	0.98	0.93	0.96	0.83	0.99
8	0.96	0.92	0.90	0.99	0.92	1.00	0.99
16	1.00	0.86	0.83	0.91	0.85	1.04	1.00
32	0.98	0.67	0.77	0.94	0.86	0.98	n/a

#### 4 SUMMARY AND CONCLUSIONS

Here we have examined the performance of SMS parallel versions of the Eta atmospheric and ROMS oceanic models. Previously, Govett, et al.<sup>8</sup> showed that SMS Eta beats the performance of a hand-coded MPI version of the model running operationally at NCEP, largely because it does a better job of aggregating halo updates. The results here show that when the difference in aggregation is removed, the SMS halo update times match those for the MPI version. On the Cray T3E, using SHMEM to implement halo updates resulted in a 20% decrease in communication time over MPI.

By definition, a perfectly tuned hand-coded MPI version of Eta or any other code will beat the performance of an SMS version since the latter is built on top of MPI. However, the hand-coded MPI version of the operational NCEP Eta model demonstrates that this can be difficult in practice. Thus, an advantage of using an approach like SMS is that the effort to implement optimizations does not have to be repeated for every model. The disadvantage of the SMS approach is that the additional software layer adds overhead. However, the SMS ROMS results show that the overhead for the critical halo update communications ranges from 2-11%; one-third to two-thirds of which could be eliminated with an improved SMS design.

To simplify the SMS parallelization, the ROMS code was converted to use Fortran 90 allocatable arrays during compilation. This incurred a performance penalty on some machines. When pointers within derived types are used, the performance is worse than the allocatable array version for some machines. This result is significant since oceanic and atmospheric scientists are beginning to implement object-oriented designs using these constructs. The results here also show that these penalties can be mitigated by “de-referencing” the pointers prior to the calls to core computational routines. However, on some machines, performance penalties remain even with this optimization. Clearly, some vendors need to put additional effort into avoiding performance pitfalls for dynamically allocated memory in Fortran 90.

We also find that, despite the performance hit due to dynamic memory usage on the SGI O3K, the SMS (distributed-memory parallel) version of ROMS still beats the shared-memory parallel version for higher numbers of processors. This is due to the effects of false sharing. The results also show the value added by the ability of the user to choose processor layouts at runtime using SMS configuration files. And, finally, SMS asynchronous output is demonstrated to yield a 15% improvement in runtime for the Eta model for 88 processors on an IA-32 machine.

In this paper, we have seen that SMS provides the user with a clean way to optimally locate communication in a code. Although the requirement to identify where communication is needed adds extra burden on the scientist, SMS debugging directives provide substantial mitigation. The end result is a directive-based tool that simplifies the parallelization process and generates code that has portable high performance.

There are, however, two significant obstacles to the long-term success of SMS. One is the lack of funding available to expand and support its capabilities. To name a few examples of the additional work required, SMS needs to be able to fully support Fortran 90, provide more backend optimizations (such as SHMEM on the Alpha machines), support dynamic load balancing, and provide a means to handle non-rectangular grids. The more fundamental obstacle is the lack of vendor support for the tool. Many meteorological institutions are unwilling to utilize a tool that is not supported by the significant hardware vendors because of the risks involved.

The OpenMP standard was born out of recognition of the value of providing a portable means for hiding the details of parallel programming on shared memory architectures. The presentations at the 2002 ECMWF workshop demonstrated the continued attractiveness of this standard in the meteorological community. The SMS prototype demonstrates that it is feasible to develop a standard for distributed memory architectures that exhibits portable high performance. One reasonable solution might be to improve the existing HPF standard by providing a means for turning off

automatically generated communication and replacing it with user-specified directives. Further, by adding directive-based debugging support, HPF could finally turn the corner toward full success. As a bonus, since using a distributed memory parallelization method avoids false sharing on cache-based shared memory machines, this truly high performance Fortran code may, in some cases, turn out to be a better solution than OpenMP for these architectures.

### Acknowledgements

Access to the Cray T3E was provided by Dr. Sirpa Hakkinen of the National Aeronautics and Space Administration Goddard Space Flight Center. Access to the IA-64 platform was provided by Dr. Bruce Loftis of the National Center for Supercomputing Applications. We wish to thank Dr. Craig Tierney and Leslie Hart for their helpful insights.

### References

1. *The Top 500 Supercomputer Sites as Measured by the LINPACK Benchmark*, November 2002, <http://www.top500.org/lists/linpack.php>.
2. Gropp W, Lusk E, Skjellum A. *Using MPI, Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
3. Frumkin M, Jin H, Waheed A, Yan J. A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000. *Proceedings of Super Computing '98*, Orlando, Florida, [http://www.supercomp.org/sc98/TechPapers/sc98\\_FullAbstracts/Hribar1140/](http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Hribar1140/).
4. The dHPF Compiler Project. <http://www.cs.rice.edu/~dsystem/dhpf/overview.html>
5. OpenMP Specifications <http://www.openmp.org/specs>
6. Haidvogel D, Arango H, Hedstrom K, Beckman A, Malanotte-Rizzoli P, Shchepetkin A. Model Evaluation Experiments in

the North Atlantic Basin: Simulations in Nonlinear Terrain-Following Coordinates. *Dyn. Atmos. Oceans* 2000; **32**: 239-281.

7. Cocke S, Christidis Z. Parallelization of a GCM using a hybrid approach on the IBM SP2. *Proceedings of the Ninth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, Reading, UK, November 13-18, 2000.
8. Govett M, Hart L, Henderson T, Middlecoff J, Schaffer D. The Scalable Modeling System: Directive-based code parallelization for distributed and shared memory computers. Accepted for publication in *J. Parallel Computing*, 2003.
9. Henderson T, Schaffer D, Govett M, Hart L. SMS User's Guide, [http://www-ad.fsl.noaa.gov/SMS\\_UsersGuide.pdf](http://www-ad.fsl.noaa.gov/SMS_UsersGuide.pdf) (2003)
10. Mesinger F. The Eta Regional Model and its Performance at the U.S. National Centers for Environmental Prediction. *International Workshop on Limited-area and Variable Resolution Models*. Beijing, China, WMO/TD 1995; **699**: 42-51.