

# THE SCALABLE MODELING SYSTEM: A HIGH-LEVEL ALTERNATIVE TO MPI

M. GOVETT, J. MIDDLECOFF, L. HART, T. HENDERSON\*, AND D.SCHAFFER\*

*NOAA/OAR/Forecast Systems Laboratory  
325 Broadway, Boulder, Colorado 80305-3328 USA  
Email: [govett@fsl.noaa.gov](mailto:govett@fsl.noaa.gov)*

*\*[In collaboration with the Cooperative Institute for Research in the  
Atmosphere, Colorado State University, Ft. Collins, Colorado 80523 USA]*

A directive-based parallelization tool called the Scalable Modeling System (SMS) is described. 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 efficiently on both shared and distributed memory high-performance computing platforms. SMS provides tools to support partial parallelization and debugging that significantly decreases code parallelization time. The performance of an SMS parallelized version of the Eta model is compared to the operational version running at the National Centers for Environmental Prediction (NCEP).

## 1 Introduction

Both hardware and software of high-end supercomputers have evolved significantly in the last decade. Computers quickly become obsolete; typically a new generation is introduced every two to four years. New systems utilize the latest advancements in computer architecture and hardware technology. Massively Parallel Processing (MPP) computers now comprise a wide range and class of systems including fully distributed systems, fully shared memory systems called Symmetric Multi-Processors (SMPs) containing up to 256 or more CPU's, and a new class of hybrid systems that connect multiple SMPs using some form of high speed network. Commodity-based systems have emerged as an attractive alternative to proprietary systems due to their superior price performance and to the increasing adoption of hardware and software standards by the industry. Programming on these diverse systems offer many performance benefits and programming challenges.

The primary mission of the National Oceanic and Atmospheric Administration's (NOAA's) Forecast Systems Laboratory (FSL) is to transfer atmospheric science technologies to operational agencies within NOAA, such as the National Weather Service, and to others outside the agency. Recognizing the importance of MPP technologies, FSL has been using these systems to run weather and climate models since 1990. In 1992 FSL used a 208 node Intel Paragon to produce weather

forecasts in real-time using a 60km version of the Rapid Update Cycle (RUC) model. This was the first time anyone had produced operational forecasts in real time using a MPP class system. Since then, FSL has parallelized several weather and ocean models including the Global Forecast System (GFS) and the Typhoon Forecast System (TFS) for the Central Weather Bureau in Taiwan [15], the Rutgers University Regional Ocean Modeling System (ROMS) [8], the National Centers for Environmental Prediction (NCEP) 32 km Eta model [17], the high resolution limited area Quasi Non-hydrostatic (QNH) model [16], and FSL's 40 km Rapid Update Cycle (RUC) model currently running operationally at NCEP [2].

Central to FSL's success with MPPs has been the development of the Scalable Modeling System (SMS). SMS is directive-based parallelization tool that translates Fortran code into a parallel version that runs efficiently on both shared and distributed memory systems. SMS was designed to reduce the effort and time required to parallelize models targeted for MPPs, provide good performance, and allow models to be ported between systems without code change. Further, directive-based SMS parallelization requires no changes to the original serial code.

The rest of this paper describes SMS in more detail. Section 2 introduces several approaches to code parallelization, followed by an overview of SMS in Section 3. Section 4 describes the flexibility and simplicity of code parallelization using SMS and explains how this tool has significantly decreased code parallelization time. Section 5 describes several performance optimizations available in SMS and compares the performance of NCEP's operational Eta code with the SMS parallelized Eta. Finally, Section 6 concludes and highlights some additional work that is planned.

## 2 Approaches to Parallelization

In the past decade, several distinct approaches have been used to parallelize serial codes.

*Directive-based Micro-tasking* – This approach was used by companies such as Cray and SGI to support loop level shared memory parallelization. A standard for such a set of directives called OpenMP, has recently become accepted in the community. OpenMP can be used to quickly produce parallel code, with minimal impact on the serial version. However, OpenMP does not work for distributed memory architectures.

*Message Passing Libraries* - Message-passing libraries such as Message Passing Interface (MPI), represents an approach suitable for shared or distributed memory architectures. Although the scalability of parallel codes using these libraries can be quite good, the MPI libraries are relatively low-level and can require the modeler to

expend a significant amount of effort to parallelize their code. Further, the resulting code may differ substantially from the original serial version; code restructuring is often desirable or necessary. One notable example of this strategy is the Weather Research and Forecast (WRF) model which was designed to limit the impact of parallelization and parallel code maintenance by confining MPI-based communications calls into a minimal set of model routines called the mediation layer [20].

*Parallelizing Compilers* – These solutions offer the ability to automatically produce a parallel code that is portable to shared and distributed memory machines. The compiler does the dependence analysis and offers the user directives and/or language extensions that reduce the development time and the impact on the serial code. The most notable example of a parallelizing compiler is High Performance Fortran (HPF). In some cases the resulting parallel code is quite efficient [23], but there are also deficiencies in this approach. Compilers are often forced to make conservative assumptions about data dependence relationships, which impact performance [13]. In addition, weak compiler implementations by some vendors result in widely varying performance across systems [4, 21].

*Interactive Parallelization Tools* - One interactive parallelization tool, called the Parallelization Agent, automates the tedious and time-consuming tasks while requiring the user to provide the high-level algorithmic details [14]. Another tool, called the Computer-Aided Parallelization Tool (CAPTools), attempts a comprehensive dependence analysis [13]. This tool is highly interactive, querying the user for both high level information (decomposition strategy) and lower level details such as loop dependencies and ranges that variables can take. While these tools offers the possibility of a quality parallel solution in a fraction of the time required to analyze dependencies and generate code by hand, limitations exist in their ability to offer efficient code parallelization of NWP codes that contain more advanced features (e.g. nesting, spectral transformations, and Fortran 90 constructs).

*Library-Based Tools* – Library-based tools, such as the Runtime System Library (RSL) [18] and FSL's Nearest Neighbor Tool (NNT) [22], are built on top of the lower level libraries and serve to relieve the programmer of handling many of the details of message passing programming. Performance optimizations can be added to these libraries that target specific machine architectures. Unlike computer-aided parallelization tools such as CAPTools, however, the user is still required to do all dependence analysis by hand.

In simplifying the parallel code, these high level libraries also reduce the impact to the original serial version. Parallelization is still time consuming and invasive, since code must be inserted by hand and multiple versions must be maintained. Source translation tools have been developed to help modify these codes

automatically. One such tool, the Fortran Loop and Index Converter (FLIC), generates calls to the RSL library based on command line arguments that identify decomposed arrays and loops needing transformations [19]. While useful, this tool has limited capabilities. For example, it was not designed to handle multiple data decompositions, interprocessor communications, or nested models.

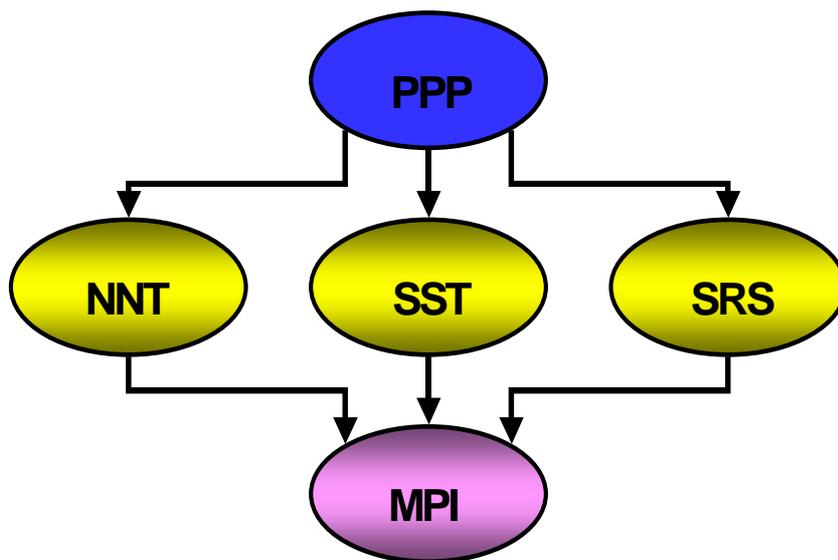
Another tool of this type, and the topic of this paper, is a directive-based source translation tool that is a new addition to SMS called the Parallel Pre-Processor (PPP). The programmer inserts the directives (as comments) directly into the Fortran serial code. PPP then translates the directives and serial code into a parallel version that runs on shared and distributed memory machines. Since the programmer adds only comments to the code, there is no impact to the serial version. Further, SMS hides enough of the details of parallelism to significantly reduce the coding and testing time compared to an MPI-based solution.

### **3 Overview of SMS**

SMS consists of two layers built on top of the Message Passing Interface (MPI) software. The highest layer is a component called the PPP, which is a Fortran code analysis and translation tool built using the Eli compiler construction software [7]. PPP analysis ensures consistency between the serial code and the user-inserted SMS parallelization directives. After analysis, PPP translates the directives and serial code into a parallel version of the code.

In addition to loop translations, array re-declarations, and other code modifications, the parallel version contains PPP generated calls to SMS library-based routines in the Nearest Neighbor Tool (NNT), Scalable Spectral Tool (SST) and Scalable Runtime System (SRS) shown in the Figure 1. NNT is a set of high-level library routines that address parallel coding issues such as data decomposition, halo region updates and loop translations [22]. SRS provides support for input and output of decomposed data [9]. SST is a set of library routines that support parallelization of spectral atmospheric models. These libraries rely on MPI routines to implement the lowest layered functionality required.

# THE SCALABLE MODELING SYSTEM (SMS)



**Figure 1.** Functional diagram of the layers of SMS that are built on top of MPI

Early versions of SMS did not contain the highest level PPP layer. Instead, model parallelization was accomplished by inserting NNT, SST and SRS library calls directly into the parallel code. While a number of models were successfully parallelized using this method, the serial and parallel versions of the code were distinctly different and had to be maintained separately [3,1,10]. Conversely, directive-based parallelization permits the modeler to maintain a single source code capable of running on a serial or parallel system. Modelers are able to test new ideas on their desktop, yet can easily generate parallel code using PPP when faster runs on an MPP are desired. Figure 2 illustrates code parallelization using SMS directives and PPP to generate the parallel code.

## Code Parallelization using SMS

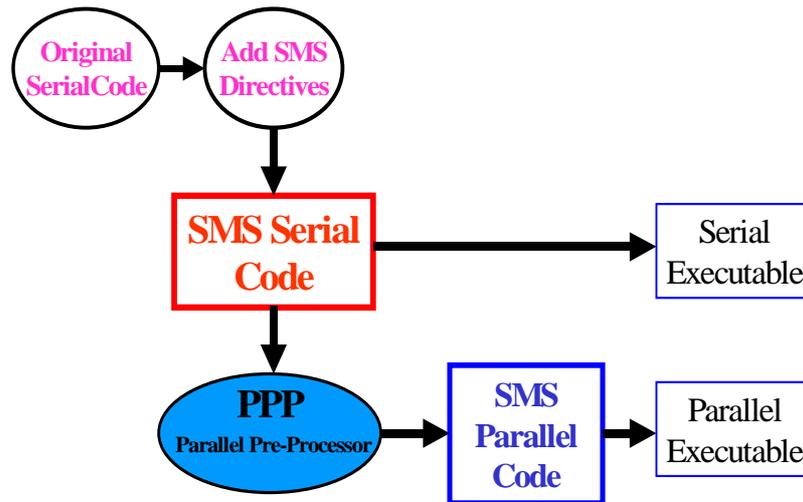


Figure 2. SMS directives are added to the original serial code during code parallelization. The SMS serial code can then be run serially as before, or parallelized using PPP to generate an MPP-ready parallel code.

To simplify the user's interface to parallelization, the number of directives available in the SMS toolkit is minimized. Currently 20 SMS directives are available to handle parallel operations including data decomposition, communication, local and global address translation, I/O, spectral transformations and nesting [5]. Further, when PPP translates the code into its parallel form, it changes only those lines of code that must be modified; the rest of the serial code including comments and white space remain untouched.

Another advantage of this approach is that directives serve to abstract the lower level details of parallelization that are required to accomplish complicated operations including interprocess communication, process synchronization, and parallel I/O. An illustration of an SMS abstraction is the use of a high-level data structure, called a decomposition handle, which defines a template that describes how data will be distributed among the processors. Two SMS directives are

required to declare and initialize the user-specified data decomposition structure (*csms\$declare\_decomp* and *csms\$create\_decomp*). A layout directive (*csms\$distribute*) is then used to associate arrays with this data decomposition.

Once data layout has been defined, the user does not need to be concerned with how data are distributed to the processors or how data will be communicated - SMS handles these low-level details automatically. SMS retains all information necessary to access, communicate, input and output decomposed and non-decomposed arrays through the use of the user-specified decomposition handle.

For example, to update the halo (ghost) region of arrays *x* and *y* between neighboring processors, the user is only required to insert

```
csms$exchange( x, y )
```

into the serial code at the appropriate place. SMS automatically generates code to store information about each variable to be exchanged (global sizes, halo thickness, decomposition type, data type), and then perform the communications necessary to update the halo points of each process. Using the information contained in the decomposition handle, SMS determines how much of the halo region each process must be exchanged, where the information must go, and where it should be stored. Process synchronization is also handled by SMS for these communication operations.

Using this encapsulation strategy other communication operations, including reductions (*csms\$reduce*), transferring data between decompositions (*csms\$transfer*), and the gather and scatter of decomposed data (*csms\$serial*) between global and decomposed arrays are easily handled at the directive level. Further, input and output of data to or from disk require no SMS directives or any special treatment by the user.

Figure 3 shows an example of an SMS program in which the decomposition handle *my\_dh* is declared (line 3) and then referenced by directive (*csms\$distribute*: lines 5, 9) to associate the first array dimension with the first dimension of the decomposition for the arrays *x* and *y*. Once the data layout has been specified via directive, SMS handles all the details required for halo updates (*csms\$exchange*: line 19), reductions (*csms\$reduce*: line 27), and I/O operations (no directives required).

Once SMS understands how arrays are decomposed, parallelization becomes primarily an issue of where in the code the user wishes to perform communications and not how data will be moved to accomplish these operations. The user is still required to determine by dependence analysis where communication is required in

their code, but a single directive is generally all that is required once this information is known. Further information about the use of SMS directives is available in the SMS User's Guide [11].

## Code with SMS Directives

```

1:      program DYNAMIC_MEMORY_EXAMPLE
2:      parameter(IM = 15)
3:      CSMS$DECLARE_DECOMP(my_dh)
4:
5:      CSMS$DISTRIBUTE(my_dh, 1) BEGIN
6:          real, allocatable :: x(:)
7:          real, allocatable :: y(:)
8:          real xsum
9:      CSMS$DISTRIBUTE END

10:     CSMS$CREATE_DECOMP (my_dh, <IM>, <2>)

11:         allocate(x(im))
12:         allocate(y(im))
13:         open (10, file = 'x_in.dat', form='unformatted')
14:         read (10) x

15:     CSMS$PARALLEL(my_dh, <i>) BEGIN
16:         do 100 i = 3, 13
17:             y(i) = x(i) - x(i-1) - x(i+1) - x(i-2) - x(i+2)
18:         100 continue
19:     CSMS$EXCHANGE(y)
20:         do 200 i = 3, 13
21:             x(i) = y(i) + y(i-1) + y(i+1) + y(i-2) + y(i+2)
22:         200 continue
23:         xsum = 0.0
24:         do 300 i = 1, 15
25:             xsum = xsum + x(i)
26:         300 continue
27:     CSMS$REDUCE(xsum, SUM)
28:     CSMS$PARALLEL END
29:     print *, 'xsum = ', xsum
30:     end

```

Figure 3. SMS directives are used to map sub-sections of the arrays *x* and *y* to the decomposition given by “my\_dh”. Each process executes on its portion of these decomposed arrays in the parallel region given by *csms\$parallel*.

Alternatively, when an operation such as a halo update is done with MPI, either each variable is exchanged separately, or in some cases, multiple arrays can be exchanged at the same time using an MPI-derived type or common block. In addition, the programmer must determine its neighbors and decide if communication is required. While not a difficult operation, it can be a tedious and time-consuming endeavor. One example of this complexity can be found in the Eta

code where a key communications routine containing over 100 lines of code was replaced with a single exchange directive during SMS parallelization.

#### **4 Code Parallelization using SMS**

The parallelization of codes targeted for MPPs can be a difficult and time-consuming process. The objective in developing SMS was to design a tool that is easy to learn and use, and to provide support for operations that simplify and speed-up code parallelization. This section highlights some of the features of SMS that have been developed to achieve these goals.

##### *4.1 Code Generation and Run-Time Options*

SMS control over the generation and execution of code can be divided into three areas: parallelization directives, command line options, and run-time environment variables. SMS directives, discussed in Section 3, are the most obvious way to control when, where and how code parallelization should be done.

SMS also provides the user with command line options to modify code translation. User access to parallel code generation using PPP is provided through a script that runs a series of executables to transform the serial code. Several command line options are available in this script that affect parallel code generation including type promotion (eg. `--r8`), retain translated code as comments, and a verbose level to warn of inconsistencies encountered during translation.

Users can also control the run-time behavior of SMS parallel code using environment variables. Environment variables are used to control when sections of PPP- translated user code will be executed. For example, conditional execution of generated code is used to verify the correctness of a parallelization where global sums are required, and for debugging purposes. This allows users to debug and verify parallelization without requiring that code be re-generated after correctness of results is established (discussed below).

Environment variables are also used to control the run-time behavior of SMS to: configure the layout of processors to the problem domain, designate the number of processors used to output decomposed arrays to disk, determine the type of input/output files that will be read/written (MPI-I/O, Native I/O, parallel file output, etc.), and tune model performance.

## 4.2 Advanced Parallelization Support

There are three phases to any code parallelization effort: code analysis, code parallelization, and debugging. Code analysis generally involves finding data dependencies that exist in the code, and based on this information, determining a data decomposition and parallelization strategy. SMS does not currently offer user support for code analysis; however, plans to provide this capability will be discussed in Section 6.

*Code Parallelization* - SMS provides support for simplifying code parallelization. Recognizing that code parallelization becomes simpler to test and debug when it can be done in a step-wise fashion, the user can insert directives to control when sections of code will be executed serially (*csms\$serial*). Serial regions are implemented by gathering all decomposed arrays, executing the code segment on a single node, then scattering the results back to each processors sub-region as illustrated in Figure 4. In this example, the routine *not\_parallel* executes on a single node referencing global arrays that have been gathered by the appropriate SMS routines.

While the extra communications required to implement gather or scatter operations will slow performance, this directive permits users to test the correctness of parallelization during intermediate steps. Once assured of correct results, the user can remove these serial regions and further parallelize their code. This directive has also been useful in handling sections of code where no SMS support for parallelization is currently available such as NetCDF I/O. Further, if adequate performance is attained, some sections of code can be left unparallelized.

*Debugging* - Once SMS directives have been added to the serial source, the parallel code must be run to verify the correctness of parallelization. To ensure correctness, output files should be examined to verify that the results are exactly the same for serial and parallel runs of the code. Since summation is not associative, reductions may not lead to exactly the same results on different numbers of processors. To alleviate this inconsistency, SMS provides a bit-wise exact reduction capability which performs exactly the same arithmetic operations that would be executed in the serial program. This capability is particularly useful when the reduction variables feed back into model fields that are output or compared. Bit-wise exact results also permit the user to verify results exactly against the serial version and ensure the accuracy and correctness of the parallelization effort.

Building on the bit-wise exact reduction capability, two SMS directives have been developed to support debugging that have significantly streamlined model parallelization, reduced debugging time, and simplified code maintenance. The first directive, *csms\$check\_halo*, permits the user to verify that halo region values are up to date. Using this directive, the halo region values of each user-specified array is

compared with their corresponding interior points on the neighboring process. If these values differ, SMS will output the differences and exit. This information helps determine where an exchange or halo update may be required to ensure correctness.

## Incremental Parallelization

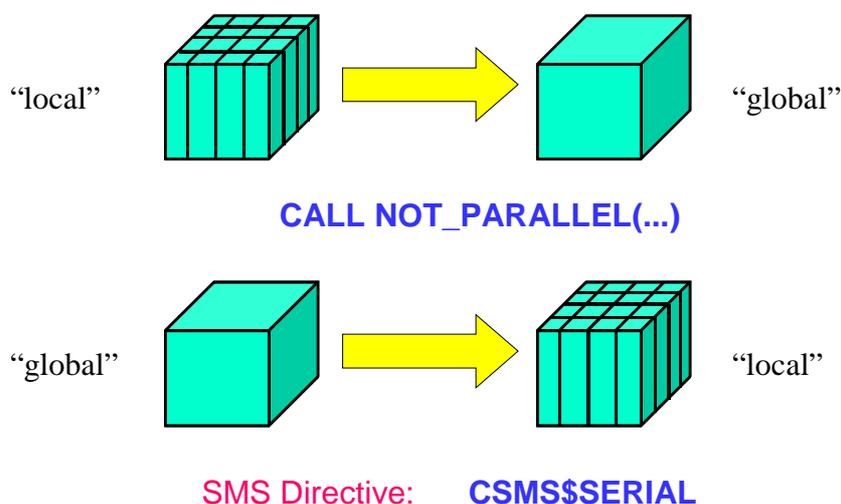


Figure 4. An illustration of SMS support for incremental parallelization. Prior to execution of the serial region of code, decomposed arrays are gathered into global arrays, referenced by the serial section of code, and then results are scattered back out to the processors at the end of the serial region.

The second debug directive, *csms\$compare\_var*, provides the ability to compare array values for model runs using different numbers of processors. For example, the programmer can specify a comparison of the array “x”, for a single processor run and for multiple processors by inserting the directive:

**csms\$compare\_var ( x )**

in the code and then entering appropriate command line arguments to request concurrent execution of the code. The command:

```
smsRun 1 mycode 2 mycode
```

will run concurrent images of the executable *mycode* for 1 and 2 processors. Wherever *csms\$compare\_var* directives appear in the code, user-specified arrays will be compared. If differences are found SMS will display the name of the variable (*x* for example), the array location (e.g. the *i, j, k* index) and the corresponding values from each run, and then terminate execution.

The ability to compare intermediate model values anywhere in the code has proven to be a powerful debugging tool during code parallelization. The effort required to debug and test a recent code parallelization was reduced from an estimated eight weeks down to two simply because the programmer did not have to spend inordinate amounts of time determining where the parallelization mistakes were made.

Additionally, this directive has proven to be a useful way to ensure that model upgrades continue to produce the correct results. For example, after making changes to serial code the modeler executes the debug sections of code (generated by *csms\$compare\_var*), controlled through a command line option, in order to verify that the intermediate results are still correct. By allowing the programmer to test parallelization in this way, code maintenance becomes much simpler for everyone.

## 5 Performance and Portability

As stated in the introduction, SMS has been used to successfully parallelize a number of mesoscale and global forecast models. These models have demonstrated good performance and scaling on a variety of computing platforms including IBM SP, Intel Paragon, Cray T3E, SGI Origin, and Alpha-Linux clusters. This section details some of the portability and performance optimizations available with SMS and then highlights some results of a recent comparison for the operational Eta model.

### 5.1 SMS Optimizations

Model performance can vary significantly depending on the hardware and architecture of the target system and the run-time characteristics of the code. Architectural differences affecting performance include processor speed, the access times and size of each type of memory (register, cache, main memory), bandwidth of the communication pathways, and speed of peripherals such as disks [12]. Issues that affect model performance include the compiler implementation, size and frequency of I/O operations, frequency and type of interprocessor communications, and data locality. SMS has been designed so that models can be ported between

systems without code change, to both run efficiently across shared and distributed memory systems and to provide options that tune the model for the best performance.

Portability has become increasingly important both because high-end computer system hardware changes frequently and because codes are often shared between researchers who run their models on different systems. To ensure portability across shared and distributed memory systems, SMS assumes that memory is distributed; no processor can address memory belonging to another processor. Despite the assumption that memory is distributed, the performance on shared memory architectures is good due to efficient implementations of MPI on these systems. Also, when an SMS parallelized model runs successfully on one system, it can easily be ported and run on another computing platform. For example, it took only two hours to port the ROMS model, parallelized for Alpha-Linux, to an SGI Origin system.

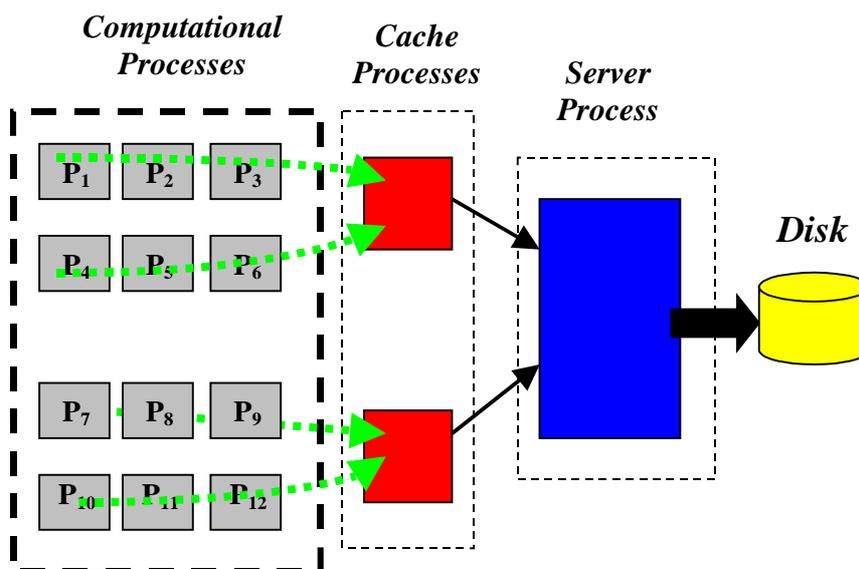
SMS provides several techniques to optimize models for high performance. One is to make architecture-specific optimizations in the lower layer of SMS. During a recent FSL procurement, one vendor replaced the MPI implementation of key SMS routines with the vendor's native communications package to improve performance. Since these changes were made at a lower layer of SMS, no changes to the model codes were necessary.

SMS also supports other performance optimizations of interprocessor communications including array aggregation and halo region computations. Array aggregation permits multiple model variables to be combined into a single communications call to reduce message-passing latency. SMS also allows the user, via directive, to perform computations in the halo region in order to reduce communication. Further details regarding these communication optimizations are discussed in the SMS Users Guide [11] and overview paper [6].

Performance optimizations have also been built into SMS I/O operations. By default, all I/O is handled by a single processor. Input data are read by this node and then scattered to the other processors. Similarly, decomposed output data are gathered by a single process and then written asynchronously. Since atmospheric models typically output forecasts several times during a model run, these operations can significantly affect the overall performance and should be done efficiently.

To improve performance, several options can be specified at run-time via environment variable. One option, illustrated in Figure 5, allows the user to dedicate multiple output processors to gather and output these data asynchronously. This allows compute operations to continue at the same time data are written to

disk. The use of multiple output processors has been shown to improve model performance by up to 25% [10].



**Figure 5.** An illustration of SMS output when cache processes and a server process are used. SMS output operations pass data from the computational domain to the cache processes. Data are re-ordered on the cache processes before being passed through the server process to disk.

Another output option allows the user to specify that no gathering of decomposed arrays be done; instead each processor writes out its section of the arrays to disk in separate files. This option allows users to take advantage of high-performance parallel I/O available on some systems including the IBM SP2. After output cycles are complete, post-processing routines can be run as a separate operation to reassemble the array fragments.

### 5.2 Eta Model Parallelization

As a high-level software tool, SMS requires extra computations to maintain data structures that encapsulate low-level MPI functionality that could lead to potential performance degradation. While a number of performance studies have been done using SMS in recent years, no study has been done to measure the cost of the SMS overhead. To measure this impact, a performance comparison was done between the hand-coded MPI based version of the Eta model running operationally at NCEP,

and the same Eta model parallelized using SMS. The MPI Eta model was considered a good candidate for fair comparison since it is an operational model and has been optimized for high performance on the IBM SP2. Performance optimizations of NCEP's Eta model include the use of IBM's parallel I/O capability which offers fast asynchronous output of intermediate results during the course of a model run.

To accomplish parallelization, the MPI Eta code was reverse engineered to return the code to its original serial form. This code was then parallelized using SMS. Code changes included restoring the original global loop bounds found in the serial code, removing MPI-based communications routines, and restoring array declarations. Fewer than 200 directives were added to the 19,000 line Eta model during SMS parallelization. To ensure correctness of parallelization, generated output files were bit-wise exact compared for both serial and parallel runs.

Table 1: Eta model performance for MPI-Eta and SMS-Eta run on NCEP's IBM SP-2. Times are for a two-hour model run.

Processors	Time	Speedup	Efficiency
24	78	1.00	1.00
32	59	1.32	0.99
48	45	1.73	0.87
88	27	2.88	0.79

After parallelization was complete, performance studies were done to compare SMS Eta to the hand-coded MPI Eta. In these tests, identical run-times were measured on 88 processors of NCEP's IBM SP2 for a two hour model run. Further tests on FSL's Alpha Linux cluster, shown in Table 1, illustrate good performance and scaling. Further analysis of these performance results is planned. However, these results demonstrate that SMS can be used to speed and simplify parallelization, improve code readability, and allow the user to maintain a single source, without incurring significant performance overhead.

## 6 Conclusion and Future Work

A directive-based approach to parallelization (SMS) has been developed that can be used for both shared and distributed memory platforms. This method provides general, high level, comment-based directives that allow complete retention of the serial code. The code is portable to a variety of hardware platforms. This

parallelization approach can be used to develop portable parallel code on multiple platforms and achieve good performance.

As we continue to parallelize more atmospheric and ocean models additional features are being added to SMS to enhance its usefulness. Parallelization of these models for MPPs has driven the development of SMS for the last ten years. Based on this experience, we have developed a tool that significantly decreases the time required to parallelize models. Further, we offer a simple, flexible user interface, provide tools that permit partial parallelization, simplify debugging and can verify the correctness of the model results exactly. In addition, our experience in working with a variety of computing platforms has allowed us to develop a tool that provides flexible high-performance portable solutions that are competitive with hand-coded vendor specific solutions. We have also demonstrated in the parallelization of NCEP's Eta model that the SMS solution performs as well as the MPI based operational version of the code.

### 6.1 Future Work

SMS currently supports the analysis and translation of Fortran 77 with added support for some commonly used Fortran 90 constructs such as allocatable arrays, limited module support, and array syntax. However, full support is planned for all of the Fortran 90 language including array sections, derived types, and modules. Another upgrade will enable the PPP translator to generate OpenMP code. Further, for state-of-the-art machines that consist of clusters of SMPs, a parallel code that implements tasking "within the box" using OpenMP and message passing "between the boxes" using MPI may be optimal. The PPP translator could be designed to generate both message passing and micro tasking parallel code.

We would also like to reduce the dependence analysis and code modification time (insertion of directives) required to parallelize a model. Development has begun on a tool, called *autogen*, to analyze the user code and automatically insert SMS directives into the serial code. A typical model (20-30K source lines) parallelized using SMS requires the insertion of about 200 directives into the code. *Autogen* could automatically generate the two most common SMS directives (*csms\$parallel* and *csms\$distribute*) that account for roughly half of the directives users must add to the serial code.

As the analysis capabilities of this tool grow, we expect to further reduce the number of directives that must be inserted by the user. However, one limitation of *autogen* is that it does not provide interprocedural analysis of the code. Therefore, we would like to combine SMS code translation capabilities with a semi-automatic dependence analysis tool. This tool would automatically insert SMS directives into

the serial code, from which a parallel version could be generated using PPP in order to further simplify parallelization.

## References

1. Baillie, C., MacDonald A.E. and Lee J.L., QNH: A numerical Weather Prediction Model developed for MPPs. International Conference HPCN Challenges in Telecomp and Telecom: Parallel Simulation of Complex Systems and Large Scale Applications. Delft, The Netherlands (1996).
2. Benjamin, S., Brown J., Brundage K., Kim D., Schwartz B., Smirnova T., and Smith T., The Operational RUC-2, 16<sup>th</sup> Conference on Weather Analysis and Forecasting, AMS, Phoenix, (1998) pp.249-252.
3. Edwards, J., Snook J., and Christidis Z., Forecasting for the 1996 Summer Olympic Games with the NNT-RAMS Parallel Model, 13<sup>th</sup> International Information and Interactive Systems for Meteorology, Oceanography and Hydrology, Long Beach, CA., American Meteorological Society, (1997) pp.19-21.
4. Frumkin, M., Jin H., and Yan J., Implementation of NAS Parallel Benchmarks in High Performance FORTRAN, NAS Technical Report NAS-98-009, NASA Ames Research Center, Moffett Field, CA (1998).
5. Govett, M., Edwards J., Hart L., Henderson T., and Schaffer T., SMS Reference Manual, [http://www-ad.fsl.noaa.gov/ac/SMS\\_ReferenceGuide.pdf](http://www-ad.fsl.noaa.gov/ac/SMS_ReferenceGuide.pdf) (1999).
6. Govett, M., Edwards J., Hart L., Henderson T., and Schaffer D., SMS: A Directive-Based Parallelization Approach for Shared and Distributed Memory High Performance Computers, [http://www-ad.fsl.noaa.gov/ac/SMS\\_Overview.pdf](http://www-ad.fsl.noaa.gov/ac/SMS_Overview.pdf) (2001).
7. Gray, R., Huring, V., Levi, S., Sloane, A., and Waite W., Eli, A Flexible Compiler Construction System., *Communications of the ACM* **35** (1992) pp.121-131.
8. Haidvogel, D.B., Arango H.G., Hedstrom K., Beckman A., Malanotte-Rizzoli P., and Shchepetkin A.F., Model Evaluation Experiments in the North Atlantic Basin: Simulations in Nonlinear Terrain-Following Coordinates, *Dyn. Atmos. Oceans* **32** (2000) pp.239-281.
9. Hart, L., Henderson T., and Rodriguez B., An MPI Based Scalable Runtime System: I/O Support for a Grid Library, <http://www-ad.fsl.noaa.gov/ac/hartLocal/io.html> (1995).
10. Henderson, T., Baillie C., Benjamin S., Black T., Bleck R., Carr G., Hart L., Govett M., Marroquin A., Middlecoff J., and Rodriguez B., Progress Toward Demonstrating Operational Capability of Massively Parallel Processors at Forecast Systems Laboratory, Proceedings of the Sixth *ECMWF Workshop on the Use of Parallel Processors in Meteorology*, European Centre for Medium Range Weather Forecasts, Reading, England (1994).

11. Henderson, T, Schaffer D., Govett M., and Hart L., SMS User's Guide, [http://www-ad.fsl.noaa.gov/ac/SMS\\_UsersGuide.pdf](http://www-ad.fsl.noaa.gov/ac/SMS_UsersGuide.pdf) (2001).
12. Hwang, K., Advanced Computer Architecture: Parallelism, Scalability, and Programmability, *McGraw Hill, Inc.*, (1993) pp.157-256.
13. Ierotheou, C.S., Johnson S.P., Cross M., and Leggett P.F., Computer aided parallelization tools (CAPTools) - Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes, *Parallel Computing* **22** (1996) pp.163-195.
14. Kothari, S., and Kim Y., Parallel Agent for Atmospheric Models, Proceedings of the *Symposium on Regional Weather Prediction on Parallel Computing Environments*, (1997) pp.287-294.
15. Liou, C.S., Chen J., Terng C., Wang F., Fong C., Rosmond T., Kuo H., Shiao C., and Cheng M., The Second-Generation Global Forecast System at the Central Weather Bureau in Taiwan, *Weather and Forecasting* **12**, pp.653-663 (1997).
16. MacDonald, A.E., Lee J.L., and Xie Y., QNH: Design and Test of a Quasi Non-hydrostatic Model for Mesoscale Weather Prediction. *Monthly Weather Review* **128** (2000) pp.1016-1036.
17. 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, 23-28 October 1995; WMO, Geneva, PWPR Rep. Ser. No. 7 **WMO/TD 699** (1995) pp.42-51.
18. Michalakes, J., RSL: A Parallel Runtime System Library for Regular Grid Finite Difference Models using Multiple Nests, *Tech. Rep. ANL/MCS-TM-197*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois (1994).
19. Michalakes, J., FLIC: A Translator for Same-Source Parallel Implementation of Regular Grid Applications, *Tech. Rep. ANL/MCS-TM-223*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois (1997).
20. Michalakes, J., Dudhia J., Gill D., Klemp J. and Shamarock W., Design of a Next Generation Regional Weather Research and Forecast Model, *Proceedings of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, European Centre for Medium Range Weather Forecasts, Reading, England (1998).
21. Ngo, T., Snyder L., and Chamberlain B., Portable Performance of Data Parallel Languages, Supercomputing 97 Conference, San Jose, CA (1997).
22. Rodriguez, B., Hart L., and Henderson T., Parallelizing Operation Weather Forecast Models for Portable and Fast Execution, *Journal of Parallel and Distributed Computing* **37** (1996) pp.159-170.
23. The Portland Group, Parallel Fortran for HP Systems, <http://www.npac.syr.edu/hpfa/bibl.html>(1999).