THE SAME-SOURCE PARALLEL MM5[1]

John Michalakes

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439
michalak@ucar.edu
+1 303 497-8199
Fax: +1 303 497-8181

ABSTRACT

Beginning with the March 1998 release of the Penn State University/NCAR Mesoscale Model (MM5), and continuing through six subsequent releases up to the present, the official version has run on distributed -memory (DM) parallel computers. Source translation and runtime library support minimize the impact of parallelization on the original model source code, with the result that the majority of code is line-for-line identical with the original version. Parallel performance and scaling are equivalent to earlier, hand-parallelized versions; the modifications have no effect when the code is compiled and run without the DM option. Supported computers include the IBM SP, Cray T3E, Fujitsu VPP, Compaq Alpha clusters, and clusters of PCs (so-called Beowulf clusters). The approach also is compatible with shared-memory parallel directives, allowing distributed-memory/shared-memory hybrid parallelization on distributed-memory clusters of symmetric multiprocessors.

1 INTRODUCTION

The Pennsylvania State/National Center for Atmospheric Research Mesoscale Model is a limited-area model of atmospheric systems, now in its fifth generation, MM5 (Grell et al., 1994). It was designed for vector and shared-memory parallel architectures. Two earlier distributed-memory (DM) parallel versions of the model code were developed at Argonne National Laboratory—the Massively Parallel Mesoscale Model (MPMM) and a subsequent Fortran90 implementation, MM90. These were efficient, scalable, and more modular and dynamically configurable (Foster and Michalakes, 1993; Michalakes, 1997b) than the source model. Nevertheless, extensive modification for parallelization prevented integration with the official

[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

version of MM5. The challenge was to produce a DM-parallel version of the model sufficiently close to the original source code that it could be officially adopted, supported, and maintained. This was accomplished in March 1998 with the release of MM5 Version 2 Release 8, the first official version of the model to support distributed-memory parallelism.

Single-source implementation of parallelism has obvious benefits for maintainability, avoiding the effort needed to keep multiple, architecture-specific versions up to date with respect to each other. The "same-source" approach to parallelization, an additional constraint on single-source implementation, also emphasizes avoiding changes to the original source code, a critical factor in NCAR's acceptance of this option. The approach employs an application-specific parallel library and a compile-time source translator to automate and hide parallel mechanisms in the code. The Runtime System Library, RSL (Michalakes, 1997c), provides domain decomposition, local address space computation, distributed I/O, and interprocessor communication supporting parallelization of both the solver and the mesh refinement code. The Fortran Loop and Index Converter, FLIC (Michalakes, 1997a), translates at compile-time to generate a parallelized code (that only the compiler sees) from a single version of the source model. The approach is essentially directiveless, requiring only a small amount of information—sufficiently general and concise to fit on the tool's command line—to direct the translation. Because MM5 already contained multithreading directives (OpenMP) for shared-memory parallelism, the model runs in combined distributed- memory/shared-memory parallel modes on distributed-memory clusters of symmetric multiprocessor (SMP) nodes.

The DM-parallel option to MM5 was released as part of the official model in March 1998. Since that time, MM5 has progressed through seven releases, including a new version, MM5 Version 3, with the distributed memory option intact and functioning as part of the main release of the model. The code is running operationally in real-time forecast mode on an IBM SP at the United States Air Force Weather Agency, Offutt Air Force Base, Nebraska. The model is also in use by the U.S. EPA, the California Air Resources Board, and a number of other research, university, and government users in the United States, Europe, and Asia. Supported platforms in active community use include the IBM SP, Cray T3E, Fujitsu VPP300 and VPP5000, Compaq Alpha SMP-clusters, Beowulf-type PC and Alpha clusters, workstation networks, and the SGI Origin2000. This paper summarizes issues that arise in parallelization of a weather model and describes the tools-based approach used to parallelize MM5. Results are evaluated in terms of impact on model source code as well as model performance and scaling.

2 SAME SOURCE

Architecture-specific coding affects understandability, maintainability, extensibility, reusability, and portability to other, dissimilar architectures. Such coding may manifest itself in how arrays are dimensioned, aligned, and allocated in memory; how loops are nested or otherwise structured (blocked, unrolled, fused); at what level loops are positioned in the subroutine call hierarchy; how iteration is expressed (loops or array syntax); how information is exchanged between subroutines; and, with distributed memory, how communication is implemented. Maintaining separate codes is difficult and time consuming; and because changes and enhancements must be made by hand and tested over all versions, some versions inevitably fall behind. The ability to

exploit a range of computer architectures with a single source code provides obvious benefits. If in addition to a "single source" one wishes to preserve the pre-existing sequential source, the constraint of "same source" is imposed. In other words, the parallelized model is as close as possible to being line-for-line identical with the original, preparallelized version.

Distributed-memory programming provides the most general programming model for both portability and scalability, since distributed-memory programs adapt trivially to shared memory (while the reverse is not true). Portability through distributed memory programming will best position programs to exploit successive advances in high-performance computer architecture, including low-cost high-speed networked "Beowulf" configurations of personal computers, a computational option unavailable to shared-memory programs. Programming for distributed memory provides both portability and scalability. Another emerging architecture is distributed-memory configurations of SMP nodes; distributed-memory programming is an essential component of an overall strategy to exploit these machines. Finally, on distributed/shared memory architectures—distributed-memory machines with additional hardware and software to support shared-memory programming (e.g., the SGI Origin2000)—distributed-memory programming may still provide better scaling because locality is explicitly enforced.

Much of the painful low-level detail originally associated with message-passing programming—domain decomposition, message passing, distributed I/O, and load balancing—has been efficiently encapsulated in application-specific libraries (Hempel and Ritzdorf, 1991; Kohn and Baden, 1996; Michalakes, 1997c; Parashar and Browne, 1995; Rodriguez et al., 1995). These approaches still require modification to the code for iteration over local data, global and local index translation, and distributed I/O. If one is able to design a new model or undertake a major redesign, these issues may be addressed directly in the code, as a number of groups have demonstrated (e.g., ECMWF's IFS and Environment Canada's MC2 models). However, if a *same-source* and not only a single-source implementation is required, additional help is needed.

Source translation removes the remaining difficulties associated with implementing the model efficiently for distributed memory. Further, source translation is applicable to a broader range of performance portability concerns. Loop restructuring, data-in-memory restructuring and realignment, and other manipulations are all effective code transformations for addressing single-processor cache performance, data locality, and communication cost. Source translation and analysis tools also uncover data dependencies in parallel routines (Friedman et al., 1995; Kothari, 1996; Evans et al., 1997; Sawdey and O'Keefe, 1997). Finally, source translators may be used for nonperformance-related code transformations, such as adjoint generation for sensitivities and four-dimensional variational assimilation (Goldman and Cats, 1996). Thus, source translation is a key enabling technology for the single-source development of fully integrated, fully portable models.

3 APPROACH

Parallelizing a weather model for distributed memory parallel computers involves dividing the horizontal dimensions of the domain and assigning the resulting patches to processors. The code is then restructured to compute only the cells stored locally on each processor (by modifying DO

loops and index expressions). Communication is added to exchange data between processors. In an explicit model such as MM5, the communication between processors is nearest neighbor and is used to update extra memory regions around the local partition. Scatter-gather communication is also required to support the exchange of forcing and feedback data between nested domains.

Adapting the model to compute over multiple address spaces requires modifying the code to execute only over the local partition on each processor. This involves modification of loops and indices. There are two approaches: an index expression represents the index of a cell either in the global, undecomposed domain (global view) or in local memory (local view). In either case, the actual indexing of model arrays within the bodies of parallel loops is unaffected; what differs is the expression of the loop ranges themselves, the declaration and storage classes of the decomposed arrays, and the subroutine interfaces. The global view has advantages for new codes, while the local view has advantages for a same-source parallelization of a pre-existing code.

In the global view, ranges of parallel loops in a subroutine are modified to begin and end at the global indices of the first and last cells on the processor. Fortran subrange expressions are used to declare locally sized model arrays whose elements are, nevertheless, globally indexable. The global view allows all index expressions within the subroutine—array indices, tests for boundary conditions, and instances where the value of an index feeds into the computation—to remain as-is. However, since each processor's arrays must be declared using a different subrange (that is, each processor's set of cells starts and ends at different global indices), the mapping of arrays to storage must be dynamic: model arrays must be passed through argument lists or dynamically allocated. Furthermore, local decomposed arrays in the subroutine must also be automatically allocated using subranges, either explicitly or as stack variables. Automatic storage is allowed in Fortran-90 but not in Fortran-77.

In the local view, as in the global view, loop ranges over decomposed dimensions must be modified, but here they begin and end at local indices of the first and last cell stored on the processor regardless of their position within the global domain. This allows array dimensions to be uniform over processors and avoids the need to overhaul existing static data structures. It becomes necessary, however, to translate between local and global meaning under certain circumstances: loop-invariant index expressions (a constant appears as an index into a decomposed array dimension, for example) must be converted from global to local. Index expressions that appear in tests for position in the domain (boundary tests, for example) must be converted from local to global. Index expressions whose values feed into the model computation in some way (computing distances between two points based on their grid indices, for example) must be converted from local to global.

The global view avoids the need to convert between global and local indexing, but it requires greater flexibility in declaring and allocating model storage, and data must be passed between subroutines through argument lists. The global view should be considered for new codes or codes undergoing major redesign. The local view, on the other hand, requires that indices be treated carefully depending on whether they refer to a global or local index, but the local view

can be used without overhauling existing static data structures. Because of this latter feature, the local view was adopted for the MM5 parallelization.

3.1 PARALLEL LIBRARY: RSL

RSL is a Fortran-callable parallel runtime system library for implementing regular-grid models with nesting on distributed-memory parallel computers. It is used to encapsulate many of the lower-level parallel mechanisms that, otherwise, would require extensive addition and modification to the model source code:

- domain specification, decomposition over processors, and remapping,
- intradomain communication (stencil exchanges),
- interdomain communication (nest forcing and feedback),
- local computation on each processor subdomain, and
- distributed I/O.

RSL is implemented atop lower-level MPI message passing and is therefore portable to any platform that supports MPI. RSL and its use in parallelizing MM5 have been described previously (Michalakes, 1997b,c). Although the library eliminates a large amount of explicit parallel mechanism in the code, its use still requires that the code be modified to compute over local processor subdomains (using either the local or the global view described above). Therefore, additional encapsulation and automation are required for a fully same-source approach.

3.2 SOURCE TRANSLATOR: FLIC

The Fortran Loop and Index Converter, FLIC (Michalakes, 1997a), is a Fortran compiler with a special-purpose back end for generating the modified code. It is called as a precompiler prior to invocation of the Fortran compiler when the model is built for distributed memory. Because it employs full lexical, syntactic, and semantic analysis of the input Fortran, FLIC is able to transform the code with minimal direction specified from the command line or from a small file of FLIC directives. Directives are not placed in the code itself.

FLIC examines array references within loops and infers which loops are over decomposed dimensions. It uncovers instances where decomposed dimensions are indexed by loop-invariant expressions and generates global to local index translations. FLIC uncovers instances where expressions of parallel loop variables are used in conditional expressions and generates local-to-global index translations. FLIC does not do automatic dependency or interprocedural analysis, which would be an aid to designing interprocessor communication within a code to be parallelized. This is not a serious shortcoming, however, since the data dependencies are already known and since the primary aim is to minimize the impact of other changes for parallelism on the model source code.

Parallelizing a large code such as MM5 involves (1) uncovering and handling data dependencies with communication and (2) modifying the source code to compute over a local processor subdomain rather than the entire domain. Uncovering data dependencies and designing

communication are conceptually difficult but need doing, essentially, only once. Further, the actual impact on the source code is negligible. The result of several person-months of effort to analyze dependencies and design communication is six calls to RSL message-passing subroutines in 2800 lines of the MM5 nonhydrostatic solver. On the other hand, modifying the code to execute on a local subdomain involves examining 242 files containing more than 50,000 lines of Fortran and, within this, identifying and modifying 560 loops over decomposed dimensions, 158 global-to-local index conversions, and 62 local-to-global index conversions. Yet the rules for identifying and implementing these modifications are straightforward, mechanical, and automatable at model compile time. Thus, there is a significant role for automation, and even a relatively simple mechanism such as FLIC provides enormous benefit.

When the user specifies the distributed-memory parallel option to MM5, the build mechanism (UNIX make utility) automatically invokes FLIC on each source file before passing it to the Fortran compiler. Input to FLIC is the subroutine itself and a short specification that includes the names of key identifiers that are used in the code to declare north/south and east/west decomposed dimensions of multidimensional arrays and the names of subroutines that are called within a loop over a decomposed dimension. In MM5, as with most large simulation codes, the identifiers used to dimension physical dimensions of state arrays are used consistently over the entire code so that a very small amount of information may direct the translation across the whole program.

FLIC examines the array references within the body of each loop in the subroutine. If it finds that an expression containing a loop variable is used to index a decomposed array dimension, the loop statement is considered to be over that decomposed dimension and is transformed. FLIC is able to determine which array dimensions are decomposed by whether they are declared using an identifier that was specified as a key identifier. In case the range of the loop statement also includes an expression with the key identifier, FLIC assumes that loop is already intended to iterate over the local memory rather than the logical domain and does not transform it. One finds instances of this in MM5 where the program is setting arrays to zero or some other initial value. As an example, FLIC is told that MIX is a key identifier that declares the north/south dimension of the domain and MJX is a key identifier that declares the east/west dimension and then invoked on the following (fictitious) code fragment:

```
SUBROUTINE HADV( FTEN, UA, … )
PARAMETER ( MIX = …, MJX = …, MKX = … )
REAL  FTEN( MIX, MKX ), UA( MIX, MJX, MKX )
DO K = 2, KL-1
  DO I = 1, MIX
    FTEN(I,K)=0. ! (1)
  END DO
  DO J = 2, JL-1
    DO I = 2, IL-1
      FTEN(I,K)=FTEN(I,K)-UA(I+1,J+1,K)+UA(I,J+1,K)) ! (2)
    END DO
  END DO
END DO
```

On inspection of the triply nested loop body (2), FLIC finds that loop variable I indexes the first dimension of the array FTEN and infers that the I-loop is over the decomposed north/south dimension because the first dimension of FTEN was declared using MIX. The second dimension of FTEN is dimensioned with MKX, which FLIC knows nothing about, so the K-loop is left alone. The J loop is recognized as an east/west decomposed dimension when FLIC encounters the second index of the array UA. The index expression involves a loop variable, J, and FLIC knows that this dimension is east/west decomposed because it is dimensioned using MJX. FLIC also recognizes that the first I-loop (1) is over a decomposed dimension, but this loop is not translated because it runs from 1 to MIX, one of the key identifiers.

FLIC replaces DO statements for decomposed dimensions with macros that are expanded to what the Fortran compiler sees (transformations underlined):

```
SUBROUTINE HADV( FTEN, UA, … )
PARAMETER ( MIX = …, MJX = …, MKX = … )
REAL  FTEN( MIX, MKX ), UA( MIX, MJX, MKX )
DO K = 1,KL-1
  DO I = 1, MIX
    FTEN(I,K) = 0.
  END DO
  do j = js(2), je(JL-1)
    do i = is(2), ie(IL-1)
      FTEN(I,K)=FTEN(I,K)-UA(I+1,J+1,K)+UA(I,J+1,K))
    enddo
  enddo
END DO
```

The starting and ending indices for the local processor subdomain are computed at run time by the RSL library and stored in the integer arrays js, je, is, and ie. FLIC generates macros rather than the transformation itself to allow targeting other libraries than RSL.

FLIC also recognizes instances where loop invariate expressions are used to index a decomposed dimension of an array. This results in a global-to-local index conversion. Finally, FLIC recognizes instances where a loop variable from a decomposed loop is used in a conditional, for example, a boundary test or other check for position within the global domain. This results in a local-to-global index conversion. Additional information and examples may be found in (Michalakes, 1997a).

4 RESULTS

Two criteria are used to assess the effectiveness of a same-source parallelization: impact on source code and model performance.

One measure of impact is a count of the number of source lines that are added or changed specifically for distributed-memory parallelism. As noted earlier, the number of source lines affected without FLIC or any other mechanism would be enormous and would prevent a same-

Table 1 : Impact on MM5 source

	1. Number of Lines	2. Lines Differing with FLIC	3. Percent	4. Lines Differing with FLIC and CPP	5. Percent
Dynamics	3367	270	8.02	17	0.50
Physics	31097	135	0.43	135	0.43
FDDA	3982	179	4.50	91	2.29
Infrastructure	15066	1359	9.02	329	2.18
Total	53512	1943	3.63	572	1.07

source implementation of DM-parallelism in a pre-existing code. As shown in Table 1, FLIC is effective at eliminating all but a small fraction of source lines specific to distributed memory parallelism: 3.6 percent of the more than 50,000 source lines. The differences appear mostly in sections of the model where the changes for parallelism fall outside the loop and index translation FLIC is designed to handle, such as I/O, nesting, model initialization, and four-dimensional data assimilation. Other differences stem from calls to communication routines in the RSL library. The remainder of DM-parallel specific lines of code is hidden using conditional compilation and file-include directives of the UNIX C-preprocessor, CPP. The last two columns of Table 1 show the effect of using both FLIC and CPP to automate and hide lines of code relating to distributed memory. The second and third columns of Table 1 are more relevant to developers and maintainers of the code. The additional use of CPP to hide DM-parallel code is to avoid bothering non-DM parallel (workstation, shared-memory, and vector supercomputer) users with such details.

Figure 1 shows performance results in both Mflop per second and in hours of simulation per wall-clock compute-hour on a variety of platforms. The EV56 Beowulf timings were conducted on the 64-node Centurion II system at the University of Virginia.[2] Each node has one 533 Mhz Alpha EV56 processor. The model was run using MPI-over-Myrinet message-passing between single-threaded processes. The Compaq EV6 Cluster is an eight-node ES40 system at NCAR with four 500 Mhz Alpha EV6 processors per node. MM5 was run using message-passing between MPI processes, each running one, two, or four OpenMP threads. For each number of processors, only the fastest combination of shared and distributed memory is shown.[3] The SGI Origin2000 timings were obtained in dedicated (exclusive access) mode on 64- and 128-processor configurations of MIPS R12000 300 Mhz processors at SGI. The model was run as multiple single-threaded MPI processes.[4] The IBM SP timings were obtained on the IBM Winterhawk-I system at NCAR which has two Power3 200 Mhz CPUs per node. The model was run with one MPI process per node, each running two OpenMP threads. The Fujitsu timings

[2] MM5 performance data on the Centurion cluster was provided by Greg Lindahl, HPTi. Information on Centurion is available at http://www.cs.virginia.edu/~legion/centurion.

[3] Additional information is available at http://www.mmm.ucar.edu/mm5/mpp/helpdesk/20000106.html.

[4] Model performance data on the Origin was provided by Wesley Jones, SGI Corp.

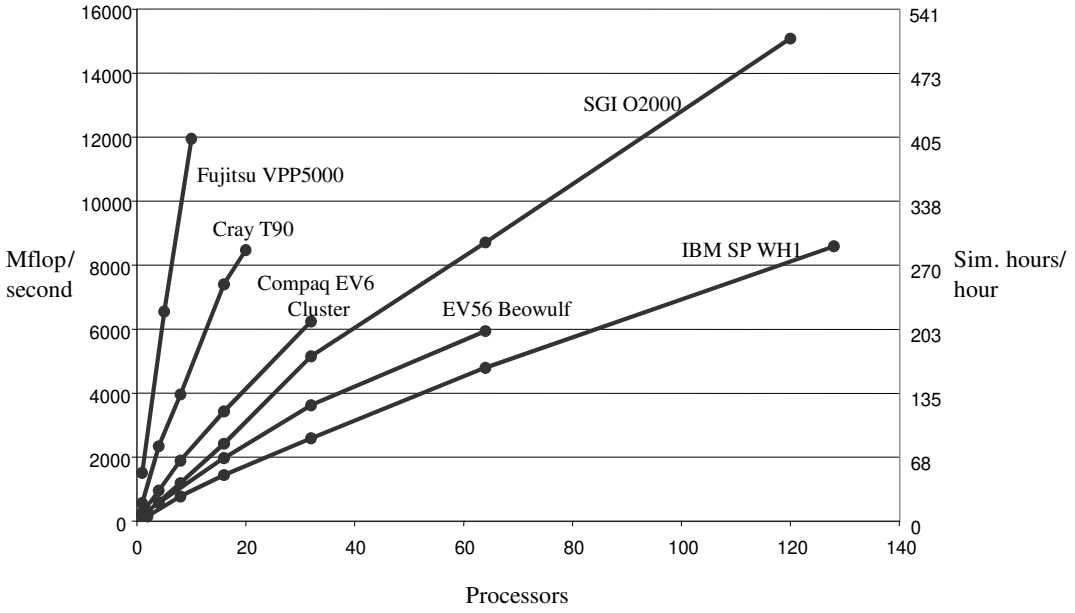

Figure 1. MM5 floating-pint performance on various platforms

were obtained on a VPP5000 installed at the Central Weather Bureau in Taiwan. The VPP5000 is a distributed-memory machine with vector processors, linked by a high-speed crossbar. The model was run using Fujitsu's implementation of MPI and with a one-dimensional data decomposition to preserve vector length in the I-dimension. The Cray T90 timings were obtained on the Cray T932 at the NOAA Geophysical Fluid Dynamics Laboratory. All timings were obtained in dedicated usage mode and show elapsed time except for the one-processor time, which was collected in non-dedicated mode and shows CPU time. All multi-CPU runs were shared-memory Cray microtasked runs (using CMIC$ directives). The T90 was also used to determine the floating-point operation count upon which the Mflop per second estimates in Figure 1 are based.

All runs were of a 36-kilometer resolution domain over Europe; the grid consisted of 136 cells in the east/west dimension, 112 north/south, and 33 vertical layers (503,000 cells). The operation count for this scenario is 2,398 million floating point operations per model time step (81 seconds). I/O and model initialization were not included in the timings. All timings except the T90 runs were performed at single (32-bit) floating-point precision. Scaling was measured as the speedup divided by the factor of increase in the number of processors. The results were as follows:

- Futisu VPP5000, 1 to 10 CPU (1,512 to 11,951 Mflop/second) 79%
- EV56 Alpha Beowulf machine, 4 to 64 CPU (575 to 5,938 Mflop/sec), 65%
- Compaq EV6 Cluster (ES40), 1 to 32 CPU (266 to 6,244 Mflop/sec), 73%

- SGI Origin, 1 to 120 processors (158 to 15,080 Mflop/sec), 80%
- IBM SP WH1 machine, 2 to 128 CPU (155 to 8,594 Mflop/sec), 87%
- Cray T90, 1 to 20 CPU (569 to 8,472 Mflop/sec), 74%

A source of parallel inefficiency is communication overhead, primarily within the time-split portion of the solver, which involves communication and relatively little computation at each of a number of small time steps, compared with the main physics/advection/diffusion time step. Another source of inefficiency is load imbalance, stemming primarily from the fact that the number of processors usually does not evenly divide the number of grid cells in a horizontal dimension, but also from reduced work at domain boundaries and uneven computational load in model physics. Performance and scaling of the same-source parallel MM5 are consistent with earlier, hand-parallelized implementations of the model.

5 CONCLUSION

The set of architectures available to users of the Penn State/NCAR MM5 has been expanded to included distributed-memory parallel computers, providing cost-effective scalable performance and memory capacity for large problem sizes. The same-source approach uses high-level parallel library and source-translation technology for adapting MM5, simplifying maintenance and allowing new physics modules to be incorporated without modification. The approach facilitates maintenance of the DM-parallel option to MM5 as an option within the official version, rather than as a separate stand-alone version. As a result, the DM-parallel option to MM5 (now at Version 3.1) has been a part of seven subsequent model releases since MM5 Version 2.8 in March 1998. The same-source approach is applicable to other, similarly constructed codes when there is a need or desire to develop the code for distributed memory parallel machines without impacting the pre-existing source code. The approach is also compatible with pre-existing loop-level multithreading directives so that the code will run in distributed-memory/shared-memory mode on SMP clusters.

The fact that MM5 is a fully explicit model is a convenient simplification that may not be available in other models, many of which employ implicit methods in their horizontal dynamics (Baillie et al., 1997). Future work involves adapting and expanding this approach to incorporate other computational techniques, including spectral, semi-implicit, and other methods with nonlocal data dependencies. Another focus will be on augmenting source code analysis and translation to address cache and other performance portability issues. Same-source tools and techniques provide a reasonable approach to obtaining good performance over the range of high-performance computers from a single version of a pre-exising model source code.

REFERENCES

Baillie, C., J. Michalakes, and R. Skålin, 1997: *Regional Weather Modeling on Parallel Computers*, Parallel Computing 23 (1997) 2135—2142.

Evans, E.W., S.P. Johnson, P.F. Leggett, and M. Cross, 1997: *Automatic Code Generation of Overlapped Communications in a Parallelisation Tool,* Parallel Computing 23 (1997) 1493—1523.

Foster, I. and J. Michalakes, 1993: *MPMM: A Massively Parallel Mesoscale Model*, in Parallel Supercomputing in Atmospheric Science, G. R. Hoffmann and T. Kauranne, eds., World Scientific, River Edge, New Jersey, pp. 354–363.

Friedman, R., J. Levesque, and G. Wagenbreth, 1995: *Fortran Parallelization Handbook*, Applied Parallel Research Inc., Sacramento.

Goldman, V. and G. Cats, 1996: *Automatic Adjoint Modeling within a Program Generation Framework: A Case Study for a Weather Forecasting Grid-Point Model,* in Computational Differentiation, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds. Society for Industrial and Applied Mathematics, Philadelphia, pp. 184–194.

Grell, G. A., J. Dudhia, and D. R. Stauffer, 1994: *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5),* Tech. Rep. NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado.

Hempel, R., and H. Ritzdorf, 1991: *The GMD Communications Library for Grid-oriented Problems,* Tech. Rep. GMD-0589, German National Research Center for Information Technology.

Kohn, S. R., and S. B. Baden, 1996: *A Parallel Software Infrastructure for Structured Adaptive Mesh Methods*, in Proceedings of Supercomputing '95, IEEE Computer Society Press.

Kothari, S., 1996: *Parallelization Agent for Legacy Codes*, draft technical report, Iowa State University, Ames. See also http://www.cs.iastate.edu/kothari.

Michalakes, J., 1997a: *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.

Michalakes, J., 1997b: *MM90: A Scalable Parallel Implementation of the Penn State/NCAR Mesoscale Model (MM5)*, Parallel Computing (to appear); also preprint ANL/MCS- P659-0597.

Michalakes, J., 1997c: *RSL: A Parallel Runtime System Library for Regional Atmospheric Models with Nesting*, Proceedings of the IMA workshop "Structured Adaptive Mesh Refinement Grid Methods," (to appear); also preprint ANL/MCS-P663-0597.

Parashar, M., and J. C. Browne, 1995: *Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement*, Proceedings of the International Conference for High Performance Computing, pp. 22–27.

Rodriguez, B., L. Hart, and T. Henderson, 1995: *A Library for the Portable Parallelization of Operational Weather Forecast Models,* in Coming of Age: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, World Scientific, River Edge, New Jersey, pp. 148–161.

Sawdey, A. and M. O'Keefe, 1997: *Program Analysis and Overlap Area Usage in Self-Similar Parallel Programs,* in Languages and Compilers for Parallel Computing, number 1035 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997.