THE SAME-SOURCE PARALLEL MM5[1]

John Michalakes

Mathematics and Computer Science Division
Argonne National Laboratory
Chicago, Illinois 60439
michalak@mcs.anl.gov
+1 630 252-6646
Fax: +1 630 252-5986

ABSTRACT

With the March 1998 release of the Penn State University/NCAR Mesoscale Model (MM5), the official version of the model (MM5v2 Release 8) now runs on distributed memory (DM) message-passing platforms. Under an IBM-funded effort, 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 SP2, Cray T3E, and Fujitsu VPP. The approach is compatible with shared-memory parallelism, allowing DM/SM hybrid parallelization on distributed memory clusters of SMP. Preliminary results show that scalability on distributed shared memory computers such as the SGI Origin 2000 also benefits from a distributed memory programming model.

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 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, 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 version of MM5. The challenge was to produce a distributed memory parallel version of the model sufficiently close to the original source code that it could be officially adopted, supported, and maintained. This was accomplished March, 1998, with the release of MM5

[1] This work was supported in part by the Mathematical, Informational, and Computational Sciences Division subprogram of the office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Version 2 Release 8, the first official version of the model to support distributed memory parallelism.

The "same-source" approach to parallelization places an emphasis on preserving the original source code, a critical factor in NCAR's willingness to accept the changes to the official model. 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 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.

The DM-parallel option to MM5 was released as part of the official model in March, 1998, as MM5 Version 2 Release 8. The code is running operationally in real-time forecast mode on an IBM SP at the United States Air Force Weather Agency facility, 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. This paper summarizes issues that arise in parallelization of a weather model and describes the tools-based approach used to parallelizing 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" the user also wishes to parallelize the code while preserving the pre-existing non-parallel source, the additional constraint of "same-source" is imposed.

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, the latest of which is low-cost high-speed networked configurations of personal computers (Cipra, 1997), 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 (and non-mutually exclusive) component of an overall strategy to exploit these machines. Finally, on distributed/shared memory architectures – distributed memory machines with additional hardware and software for to support shared-memory programming (e.g. the SGI Origin 2000) -- distributed memory programming may still provides better scaling because locality is explicitely 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 Environnement Canada's MC2 models). However, if a *same*-source and not only 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). 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). 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 tiles to processors. The code is then restructured to compute only the cells stored locally on each processor (by modifying DO loops and index expressions) and communication is added to exchange data between processors. In an explicit model such as MM5, the communication between processors is essentially nearest neighbor and is used to update extra memory regions around the local partition. Communication is also required to support mesh-refinement in models that support nesting.

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: either establish that an index expression always represents a global index (Global View), or establish that the index expression always represents the index of a cell in local memory (Local View). In either case, the actual indexing of the 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 legacy 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 dynamic memory allocation features such as those found in Fortran-90 must be used. Furthermore, local decomposed arrays in the subroutine must also be dynamically allocated using subranges, either explicitly or as stack variables, which is supported 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 data structures. It becomes necessary, however, to translate between local and global meaning of 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 – most be converted from local to global.

In both the global and local view, modification of looping structures and data declarations are required to adapt the code to distributed memory execution. The global view avoids the need to convert between global and local indexing but requires greater flexibility in declaring and allocating model storage and it requires that data 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 mean a global or local index, but the local view more easily coexists with static data structures and systematic use of COMMON in existing codes. Because of this, the local view was adopted for the MM5 parallelization.

4. PARALLEL LIBRARY: RSL

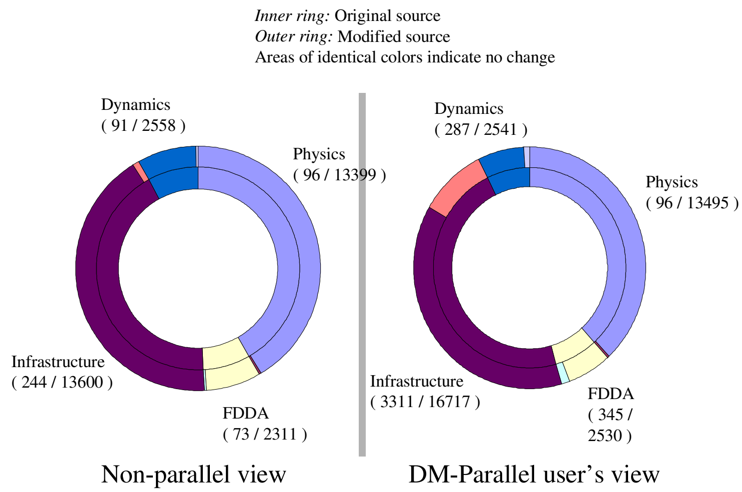

Figure 1 Impact on source

RSL is a 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
- Intra-domain communication (stencil exchanges)
- Inter-domain communication (nest forcing and feedback)
- Local computation on each processor subdomain
- Distributed I/O

RSL and its use in parallelizing MM5 has 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 local or global views described above). Therefore, additional encapsulation and automation is required for a fully same-source approach.

5. SOURCE TRANSLATOR: FLIC

Even with parallel libraries, modifications for local address space computation must be made for distributed memory. Hitherto, modifications had been made manually and appeared explicitly as changes to the source code. The same-source approach transfers the responsibility for making these changes to an automatic tool, the source translator, and in the process removes these changes from view of code developers, maintainers, and users. Translations for distributed memory, cache performance, and computational restructuring include:[i]

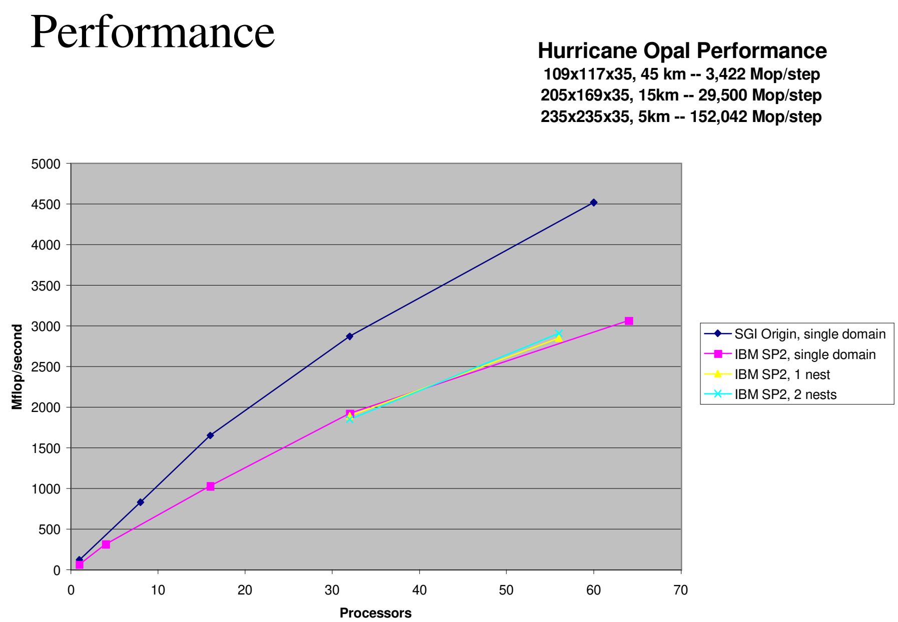

Figure 2. Single and multi-domain performance.

- ✓ Identification and translation of loops over decomposed dimensions,
- ✓ Automatic conversion of loop-invariant expressions indexing decomposed dimensions of model arrays (global to local),
- ✓ Automatic conversion of loop-variable expressions used to test position in the global domain, e.g. for boundary conditions (local to global),
- ❑ Automatic conversion of loop-variable expressions whose values are used within a computation, e.g. computing distance between two points in the logical domain based on their indices (local to global),
- ❑ Automatic insertion of global and local array dimensions into argument lists of subroutines and associated call statements (especially with global view),
- ❑ Automatic disambiguation of expressions used to dimension arrays (local) from loop range expressions,
- ❑ Automatic interprocedural data dependency analysis,
- ✓ Automatic array index reordering,
- ❑ Automatic loop restructuring, and
- ❑ Automatic index algebra to move non-local references to the righthand sides of assignement statements.

We exploit a useful dichotomy in applying source translation to the parallelization task to provide an incremental development path. This is as follows: communication is hard to design but is easy to implement with almost no impact on the source code. Computational restructuring, on the other hand, is straightforward and mechanical, but requires extensive error-prone modification to the source code. Therefore, for parallelizing a code with a minimum of effort and source code impact, there is an advantage to automating computational restructuring tasks, even if it is necessary to defer automatic dependency analysis for a subsequent phase because some tools are still under development.

The Fortran Loop and Index Converter (FLIC) (Michalakes, 1997a) is a Fortran compiler with a special purpose back-end for generating the modified code. Because it employs full lexical, syntactic, and semantic analysis of the input Fortran, it is able to transform the code with minimal direction. 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, and it uncovers instances where expressions of parallel loop variables are used in conditional expressions and generates local to global index translations.

6. RESULTS

The impact on software is extremely small, especially from the point of view of the nonparallel user. Of the 32,000 lines in the model that have been addressed so far, the UNIX diff utility reports 504 lines are different (left half of Figure 1). This view of the code is significant because changes are out of the way of non-parallel users and code developers. One need not even install the DM parallel components, in which case the model is effectively the MM5 code as it exists today.

The right half of Figure 1 shows the parallel user and developer's point of view: the actual number of changes for distributed memory. Physics is virtually unaffected: only 96 of the total 13,495 lines in the parallelized subset are different. Dynamics, which includes communication, is affected slightly more: 287 lines of a total 2,541. Infrastructure, which includes I/O and initialization, effects only 3,300 of a total 16,700. This is due largely to changes relating to distributed I/O, something FLIC does not address. Similarly, the FDDA nudging code is affected because it also includes I/O and several large data reduction operations that FLIC does not, at present, handle.

Figure 2 and Figure 3 show recent preliminary performance results using the MM5 Version 2 Release 8 mode. The results were gathered using the IBM SP at Argonne and the Silicon Graphics Origin 2000 at the University of Illinois (NCSA). Timings on the Origin were performed in dedicated user mode (exclusive access); exclusive access to the processors on the SP was provided by the gang-scheduler. Performance data for other supported platforms – the Cray T3E and Fujitsu VPP300 – were unavailable at the time this paper was prepared. Hurricane Opal data was used to initialize the runs. The box in Figure 2 shows grid sizes, resolutions, and cost in floating point operations (times 10^6) per coarse domain time step for non-nested, singly nested, and doubly nested cases.

Runs using a single domain (non-nested) were conducted to assess performance and scaling of the code in distributed-memory parallel mode on from 1 through 64 processors of the IBM SP (130 Mhz Power2 Superscalar thin nodes with TB3 switch interconnection) and from 1 through 60 processors of the NCSA Origin 2000. The model runs at a rate of 63 Mflop/sec

SGI Origin 2000
Performance

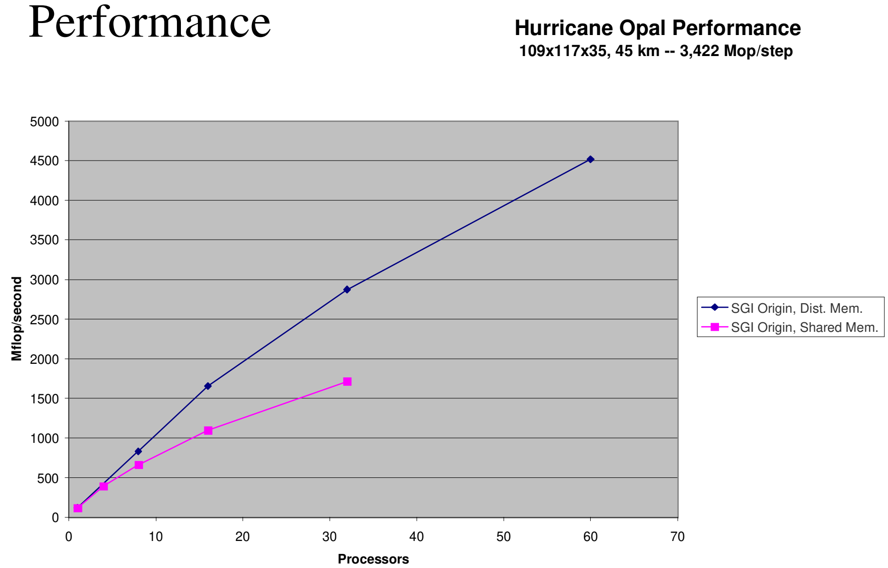

Figure 3. Shared versus distributed memory performance.

on one SP processor versus 118 Mflop/sec on one processor of the origin. On 64 SP processors, the model ran at 3063 Mflop/sec; on the Origin, 4515 Mflop/sec. Parallel efficiency (speedup/P) from 1 to 60 processors on the Origin was 63 percent. IBM SP scaling is super-linear from 1 to 4 processors because of memory effects; therefore a parallel efficiency of 61 percent was calculated from 4 to 64 processors (speedup/P/4). This translates to 14 hours for a 36 hour forecast on one SP processor; 7.7 hours on one processor of the Origin. Running in parallel, the estimated forecast times (exclusive of I/O) are 30 minutes on 64 SP processors and 12 minutes on 60 Origin processors. Singly and doubly nested timings on the SP are shown in Figure 2; performance is actually slightly better because the nested grids are much larger than the coarse domain and therefore run more efficiently.

Figure 3 shows a comparison between shared-memory and distributed memory MM5 runs on the SGI Origin. The distributed memory code is 63 percent efficient from 1 to 60 processors, whereas the shared memory version is only 46 percent efficiency from 1 to 32 processors. Pure distributed memory programs appear able to exploit the low-latency high-bandwidth interconnect to provide scalable performance.

7. CONCLUSION

We have described an effort that will expand the set of architectures that will run the official NCAR version of the MM5, providing the benefit of scalable performance and memory capacity for large problem sizes to users with access to distributed memory parallel computers. The same-source approach uses source-translation technology for adapting MM5, simplifying maintenance and allowing new physics modules to be incorporated without modification. 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 non-local 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 computing options from a single version of the model source code.

REFERENCES

Baillie, C., J. Michalakes, and R. Skålin, 1997: *Regional Weather Modeling on Parallel Computers*, Parallel Computing, (to appear, December 1997).

Cipra, B. A., 1997: "In scientific computing, many hands make light work," SIAM News, 30.

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, PA, 1996, 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, Iowa, 1996. 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.

---

[i] Checked items are those implemented in FLIC or related translation tools in use within the parallel MM5 effort.