



Optimization of the NMS6b Weather Model Code

**by Chatt Williamson, Steven R. Thompson, Daniel M. Pressel,
Jeffrey N. Robinson, Dixie Hisley, and George Petit**

ARL-TR-3496

May 2005

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005-5067

ARL-TR-3496

May 2005

Optimization of the NMS6b Weather Model Code

Chatt Williamson, Daniel Pressel, and Dixie Hisley
Computational and Information Sciences Directorate, ARL

Steven R. Thompson, Jeffrey N. Robinson, and George Petit
Raytheon

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.				
1. REPORT DATE (DD-MM-YYYY) May 2005		2. REPORT TYPE Final		3. DATES COVERED (From - To) 2002–2003
4. TITLE AND SUBTITLE Optimization of the NMS6b Weather Model Code			5a. CONTRACT NUMBER	
			5b. GRANT NUMBER	
			5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Chatt Williamson, Steven R. Thompson,* Daniel M. Pressel, Jeffrey N. Robinson,* Dixie Hisley, and George Petit*			5d. PROJECT NUMBER 5U03CL	
			5e. TASK NUMBER	
			5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRD-ARL-CI-HC Aberdeen Proving Ground, MD 21005-5067			8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-3496	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)	
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				
13. SUPPLEMENTARY NOTES *Raytheon, AMSRL-ARL-CI-HC, Aberdeen Proving Ground, MD 21005-5067				
14. ABSTRACT The U.S. Army needs timely and accurate weather forecasting to support the prediction of battlefield conditions. The U.S. Army Research Laboratory Major Shared Resource Center was tasked with optimizing the Nonhydrostatic Model Simulation (NMS) weather forecasting code for potential U.S. Army use. This code was written for parallel execution on shared memory architectures using OpenMP directives. As written, the code does not run on distributed memory nodes. The NMS code consists of ~190,000 lines of Fortran code and 4000 lines of C code and was developed by Dr. Greg Tripoli of the University of Wisconsin. The code features a unique variable-stepped topography representation designed to handle steep slopes. It is designed to faithfully represent flows in the presence of arbitrarily rough topography while maintaining sensitivity to subtle impacts of weak topography. In this report, we give a brief description of the NMS code, followed by the initial performance rate and our optimization goal, a short discussion of our approach, an explanation of the optimization work, our final benchmark results, and finally a brief mention of what future work could be done.				
15. SUBJECT TERMS parallel computing, weather modeling, performance tuning				
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UL	18. NUMBER OF PAGES 26
a. REPORT UNCLASSIFIED	b. ABSTRACT UNCLASSIFIED	c. THIS PAGE UNCLASSIFIED		
			19b. TELEPHONE NUMBER (Include area code) 410-278-9151	

Contents

Acknowledgments	iv
1. The Nonhydrostatic Model Simulation (NMS) Code	1
2. Our Starting Point and Goal	2
3. Target Platforms	2
4. Optimization Work on the IBM SP4	3
4.1 Initial Optimizations.....	3
4.2 Further Optimizations.....	5
4.3 Additional.....	5
5. Cray X1	7
6. SGI	7
7. Conclusions	7
Appendix A. Methods Used for Serial Tuning	9
Appendix B. Additional Methods Used for Parallel Tuning	13
Distribution List	17

Acknowledgments

The authors wish to thank Tom Kendall for his leading the systems' engineering effort to reconfigure the IBM SP4 and the entire systems staff of the U.S. Army Research Laboratory Major Shared Resource Center for their support of this project. This project was supported by a grant of computer time from the Department of Defense High Performance Computing Modernization Program.

1. The Nonhydrostatic Model Simulation (NMS) Code

The U.S. Army needs timely and accurate weather forecasting to support the prediction of battlefield conditions. The U.S. Army Research Laboratory Major Shared Resource Center was tasked with optimizing the Nonhydrostatic Model Simulation (NMS) weather forecasting code for potential Army use. This code is written for parallel execution on shared memory architectures using OpenMP directives. As written, the code does not run on distributed memory nodes. The NMS code consists of ~190,000 lines of Fortran code and 4000 lines of C code and was developed by Dr. Greg Tripoli of the University of Wisconsin.¹ The code features a unique variable-stepped topography representation designed to handle steep slopes. It is designed to faithfully represent flows in the presence of arbitrarily rough topography while maintaining sensitivity to subtle impacts of weak topography. In this report, we give a brief description of the NMS code, followed by the initial performance rate and our optimization goal, a short discussion of our approach, an explanation of the optimization work, our final benchmark results, and finally a brief mention of what future work could be done.

The NMS code performs some initialization work in which the terrain data are read and mapped to the coordinate system to be used for simulation. After initialization is complete, NMS integrates over time steps, periodically dumping output files, and, less frequently, reading updated boundary conditions (see figure 1). The code processes data on multiple grids. One grid (in our case, a 3-km grid) covers the entire domain, and there can be grids of finer resolution inside the largest grid for greater computational accuracy. This occurs in a nested fashion of ever finer and finer grids to an arbitrary degree. The location of finer resolution grids can change from one time step to another.

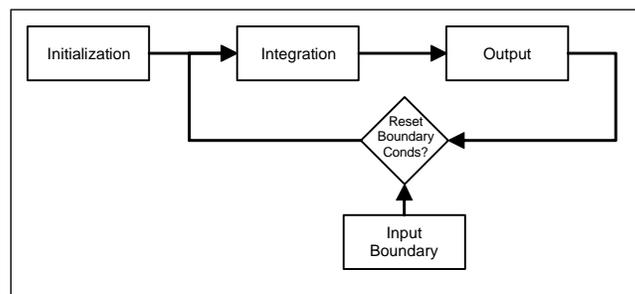


Figure 1. Process flow of the NMS code.

¹ Tripoli, G. J. A Nonhydrostatic Mesoscale Model Designed to Simulated Scale Interaction. *Monthly Weather Review* **1992**, *120* (7), 1342–1359.

A second characteristic of NMS is that the amount of computation work greatly increases (by two to three orders of magnitude) when the code's "micro-physics" is invoked, as for example, when precipitation occurs in the simulation. The nested grids and the microphysics would each contribute greatly to a load-balancing problem if parallel processing were introduced into the code via a fixed-domain decomposition scheme.

2. Our Starting Point and Goal

In November 2002, a multidisciplinary team of engineers and computer scientists was formed to investigate whether a 36-hr forecast could be executed in 1 hr. At that time, a 36-hr forecast on 192 processors of a Silicon Graphics Incorporated (SGI) Origin 3800 required ~26 hr. Thus, improvement by a factor of 26 was desired. The team was asked to perform their best level of optimization within a short time frame, and to develop a long term plan for achieving the 36:1 goal. Besides the SGI, two other platforms were immediately targeted: the IBM SP4 and the Cray X1.

Because seven engineers and computer scientists were working collaboratively on this optimization project, a mechanism for managing software configuration was essential. Though other tools provided more functionality, SCCS (source code control system) was deemed sufficient and was selected because of its familiarity. A script was written to allow members of the team to run jobs independently, yet share the common static input files (such as the terrain data files), so that disk space would be conserved, and time would be saved by not having to copy these large files to the scratch space for each run.

Initial analysis on the SGI Origin 3800 revealed that several hours of compute time were spent in serial processing, and only the integration portion of the code was being executed in parallel. We decided to separate the serial analysis portion of the code from the parallel integration processing. Files were output periodically so that the serial analysis processing could be done offline. This saved ~8 hr of total wall clock time. The initialization work and integration processing now took ~18 hr on 112 processors of an SGI Origin 3800, giving us a 2:1 ratio in simulated time compared to CPU time, but still leaving us short of the goal by a factor of 18.

3. Target Platforms

Because the code is not written in MPI, it can currently only be run on shared memory nodes. This limits its execution to 512 processors on ARL's SGI Origin 3800, to 32 processors on ARL's IBM p690 Power4 (a.k.a., SP4) machine, and to 16 single streaming processors (SSPs) on the Cray X1 at the U.S. Army High Performance Computing Resource Center (AHPCRC).

The clock speed of the IBM SP4 1.3-GHz processor is over three times faster than the 400-MHz clock speed of the SGI Origin 3800 processor, while the ratio of peak speeds of the floating point units is over 6:1. There is also an impressive increase in many of the other capabilities for the IBM SP4. Early benchmarks on a 30-min test case indicated that the execution would speed up by approximately the clock ratio.

ARL's IBM SP4 consists of two 32-processor nodes. Initially, each node had 4 logical partitions (LPARS), each with 8 processors. The hardware configuration of 8 processors per LPAR limited execution to 8 processors, but because all 32 processors on the node would be needed, the machine was reconfigured into a single LPAR per node.

4. Optimization Work on the IBM SP4

On the SP4, the initial 32-processor benchmark on the 30-min test case was conducted in 900 s, and the 36-hr execution was done in 9.5 hr. The processing ratios between the two cases are not the same because the 30-min test case performs the same amount of initialization processing as the 36-hr run, making the relative overhead of initialization greater, and the 36-hr simulation writes some output that is not done in the short test case.

4.1 Initial Optimizations

The code was profiled on the IBM SP4. The results were that the top seven routines consumed 91% of the CPU time. That was fortunate, as it allowed us to concentrate our optimization efforts on roughly 3000 lines of code instead of all 190,000 lines of code. As we optimized these seven subroutines, some of the others just beyond these seven would then become more significant and would enter our realm of interest.

IBM suggested several modifications for improvement, including use of the IBM "mass" library (specially tuned for typical mathematical functions), and eliminating calls to the "nan" function, which performed an apparently unnecessary check for invalid floating point numbers.

We discovered that the code makes numerous calls to a subroutine named `azero` that zeroes the given array. Initially we attempted to optimize this routine by using IBM's `cache_zero` function. Unfortunately, that did not help, though significant time was spent getting it to work. Eventually, we discovered that the vast majority of the time, the subroutine `azero` is called to zero a mere 84 words of memory.

With this discovery, we abandoned previous optimization attempts, and wrote three sections of code: the first section zeroes up to 100 values, the second section handles 101–10,000 values, and the third section zeroes over 10,001 values. The first section is fully unrolled starting at 100

and counting down. This allows the use of a computed “goto,” so that the code branches into the middle of this section and then executes all of the remaining iterations without another branch. The section for moderate-sized arrays is executed serially because the overhead of parallelization would exceed the cost of executing this section serially.² Vendor specific prefetch directives for the SGI and IBM were inserted to specify that the cache lines should be prefetched for store.

Approximately 1 month after the initial 32-processor IBM SP4 benchmark of 900 s was obtained, we performed the benchmark in 600 s. Improvement came by using large pages, optimizing the input/output (I/O) (writing entire arrays instead of looping through values in the array), parallelizing the interpolation processing in the initialization portion of the code, and using better compiler flags.

Several interesting observations were made at this point concerning the performance of the code based on data from IBM’s Hardware Performance Monitor (HPM). Three of those with the greatest degree of relevancy are as follows:

1. There is an extremely low rate of cache misses that miss all the way back to main memory.
2. There is an abnormally high Translation Lookaside Buffer (TLB) miss rate. In fact, on both the IBM SP4 and on the SGI O3K, there are significantly more TLB misses than cache misses (3.1-billion TLB misses vs. 81-million L3 cache misses on the IBM SP4).
3. The code is achieving only ~2%–3% of peak on the IBM SP4. The exact cause of this has not been fully determined, although excessive numbers of TLB misses and unnecessary time initializing array elements to zero certainly hurts.

When parallelizing a code using fine grained parallelism (e.g., OpenMP or High-Performance Fortran [HPF]), one needs to be careful that the amount of work in the parallelized section of code is sufficient to justify the overhead of parallelization. In some cases, it will not be. In other cases, parallelization will improve performance; however, the scalability of the section in question may be extremely limited. This becomes a serious issue when trying to use compiler-generated automatic parallelization on sections of code that have not been manually parallelized. In most cases on the IBM, attempts to use automatic parallelization resulted in a slowdown and had to be abandoned.

Appendix A goes into further detail on the serial optimizations. Appendix B covers the parallel optimizations in greater detail.

² Moore, S. University of Tennessee at Knoxville. Data on the Overhead of Parallelization. Private communication. April 2003.

4.2 Further Optimizations

One of the subroutines determined to be computationally intensive is called by a subroutine in which OpenMP constructs already existed. Nested parallelism was tested by inserting parallel do constructs within this called subroutine. On a 16-processor run on the IBM SP4, the original code was compared against the new version with nested parallelism enabled. The `AIXTHREAD_SCOPE` environment variable was set so that each logical processor was assigned to one physical processor (no thread-sharing by the original team of processors). The addition of nested parallelism reduced the execution time by ~20%.

After modifying code for better TLB use, restructuring code branches to minimize branching, using IBM's "mass" library, and altering the aforementioned "azero" subroutine, the run time for the 30-min case was reduced to 500 s.

Additional effort was expended in the following areas:

1. Blocking expensive routines/loops with a high TLB miss rate (i.e., restructuring the code so as to perform as many operations on data while it is in cache).
2. Identifying that a significant amount of time was spent on division and calculating X to the Y power (e.g., cube root). Fortunately, in many of the more expensive subroutines, ways were found to significantly decrease the number of times these operations were executed. About two thirds of "x to the y power" computations were eliminated by modifying ~20 subroutines and functions. At the same time, some instances of the "min" and "max" functions were reduced.
3. Fine tuning the use of prefetch directives.
4. Improving the performance of the code on the IBM SP4 by using 30 rather than 32 processors. This eliminated contention with the operation system.
5. After reducing the TLB miss rate, it was found that the performance could actually be increased by going back to the default page size. It is difficult to understand why this would be the case.
6. Additional optimizations to reduce the number of branches and integer calculations.

These efforts have reduced the time for the 30-min test case to under 300 s and the time spent running the full 36-hr test case to 4 hr 37 min (see figure 2).

4.3 Additional

It would appear as though the NMS code is far from being bound by the memory bandwidth because it appears to be using <6 GB/s of the 204.8 GB/s available memory bandwidth. However, this has proven not to be the case for reasons discussed at length in appendix B. Tuning to reduce the number of TLB misses has been very beneficial (the hardware prefetch

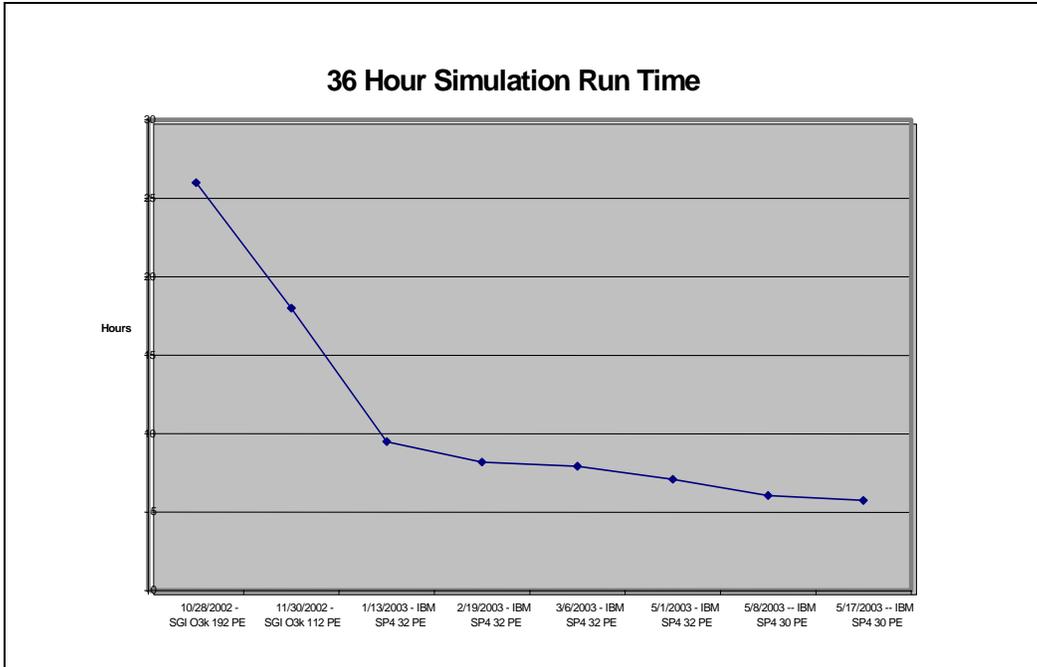


Figure 2. This performance graph shows the progress made over a period of 6 months in optimizing NMS for a 36-hr simulation. The first two data points are benchmarks on the SGI Origin 3800, the others are on the IBM SP4.

engine cannot prefetch across page boundaries). Our best estimates are that by eliminating 90% of the TLB misses, the run time was reduced by nearly 3 hr for the 36-hr simulation. Another promising observation was that the most expensive routine in the program was responsible for over 1500 s of run time. On closer examination, it has been found that this subroutine is processing 20 data streams, which means that the hardware prefetch engine lacks enough stream buffers (by a factor of 2) to properly prefetch the data used by this routine. Making matters worse, one might have hoped that because this routine is called several times consecutively per time step, that once the data resided in cache, it would remain there. Unfortunately, even though there is 512 MB of L3 cache on a single node of the IBM SP4, the 20 data streams require a minimum of 640 MB of L3 cache to fully reside in cache. Efforts to tune this routine reduced its cost to under 900 s (see appendix A for more details).

Further performance increases might be obtained through parallelizing the remaining serial loops in the code. Unfortunately, there are two main obstacles to this task. In some cases, the loops are resistant to parallelization. The second case involves the relative cost of the parallelization event vs. the amount of work in the loop being parallelized (see appendix B for additional information). In many cases, it would be more expensive to parallelize a loop than to leave it alone. Fortunately, most of those loops are not executed often enough to matter. However, a few such loops were identified as being responsible for roughly 1000 s of run time when executing the 36-hr simulation on 30 processors of the IBM SP4. Parallelizing them reduced this time to under 100 s.

5. Cray X1

In parallel with optimization efforts on the IBM, we worked on getting the code to run on the Cray X1. As with the IBM, there were several obstacles to overcome in porting NMS to the Cray. We modified the I/O routines to handle X1 INTEGER*2 data storage format which stores all integer data as one array element per word rather than the packed format used by the NMS code. We uncovered an operating system bug that Cray was able to patch to enable calling Unix systems commands from within Fortran executables. Additionally, we made several code revisions to ensure variables were initialized prior to their use. In all, it took ~1 month of effort to port the code to the X1. After the code was verified to be running correctly, a baseline benchmark indicated performance to be about half that of the IBM SP4 processor. Performance on the Cray X1 is significantly hampered by the fact that most inner loops iterate over the vertical dimension, resulting in a vector length of only 35. We have turned the code over to Cray, and they are working to see if a newer version of their compiler can improve the performance.

6. SGI

We periodically rebenchmarked the SGI Origin 3800 with the source code modifications that improved performance on the IBM SP4. Generally, these changes made no difference in the SGI performance. We attribute this to the superiority of the SGI compiler, in that it presumably already overcame the optimization issues that we had to manually overcome for the IBM compiler. The performance on the SGI appears to be limited by the slower clock speed of the processor and the fact that the code is not completely scalable.

7. Conclusions

Currently, we can run a 36-hr forecast on 32 processors of an IBM SP4 in 4 hr 37 min, thus giving us a speedup factor of ~5.5. To gain another factor of 4.5, improvement would require porting the code to a distributed memory platform. Achieving a good load balance on a distributed memory system when dealing with nested grids would be challenging and is beyond the scope of this project. Alternatively, one might try some of the following products:

1. SGI Altix with up to 256 6-GFLOP processors.
2. Fujitsu makes a SPARC based with up to 128 6-GFLOP processors.
3. HP makes the Super Dome and GS product lines.

Possibly one of these would allow us to reach the original goal of a 1-hr run with the least amount of additional effort.

Appendix A. Methods Used for Serial Tuning

A.1 Introduction

This appendix deals with optimizations at the processor level (serial tuning). The remainder of this appendix will cover the following topics:

1. The way in which the code interacted with the memory system, in particular with the stream buffers.
2. Cache effects.
3. Translation lookaside buffers (TLBs).
4. Methods used to reduce the time spent on integer arithmetic.
5. Large page sizes.

In addition to the approaches to tuning and parallelization discussed in the remainder of this appendix, a number of fairly standard techniques were used to improve the performance of this code. The majority of these techniques were used to eliminate redundant calculations (especially those involving the exponentiation operator). In general, the savings in any one routine was small, but the combined savings amounted to 15–20 min for a 6-hr run.

A.2 The Way in Which the Code Interacted With the Memory System, in Particular With the Stream Buffers

One key aspect of the IBM SP with Power 4 processors is the ability of the processor to stream data into the caches from main memory and on up to the processor. If each L3 cache miss was handled individually as it happened, most of the memory bandwidth would be unusable. Instead, the processor attempts to stream the data into the L3 cache and from there into the processor whenever possible. While prefetch instructions can help to improve the efficiency of this process, the key component is the group of eight stream buffers. This means that up to eight separate data streams can be efficiently moved from main memory at one time. This raises the question, what happens if there are more than eight data streams? The answer is not pretty. The performance of the section of code in question can drop off rapidly. The extent to which this drop off is seen will in part depend on what the overall cache miss rate is. The lower the miss rate, the less of a problem one is likely to see.

In the case of Nonhydrostatic Model Simulation (NMS), the second most expensive subroutine had 20 arrays that it was accessing. Given the structure of this subroutine, this translated into at least 20 data streams. By carefully breaking the middle loop into multiple loops and reordering

some of the inner loops, it was possible to reduce the maximum number of data streams in a middle loop to eight. The outer loop still had 20 data streams, but the stream buffers were primarily concerned with the inner and middle loops. As a result of this and other optimizations, the speed of this routine was doubled.

A.3 Memory Access Patterns

There are primarily five ways in which the access pattern for data can be ordered relative to the order in which it is stored in main memory:

1. While the simplest pattern might be called static, it normally does not have a name associated with it. This pattern accesses a very small number of locations continuously.
2. Random access is used to describe any access pattern that has the potential of being impossible to predict.
3. STRIDE-1 refers to a program accessing a large number of adjacent memory locations in the same order as they are stored in main memory.
4. STRIDE-N is similar to STRIDE-1, but accesses every Nth memory location. Normally, N will be large enough to cause an excessive number of cache and TLB misses.
5. Blocked is a complicated access pattern designed to improve efficiency.

A.4 Cache Effects

An important aspect of the Power 4 processor is the L3 cache. Its latency is 90 ns. The system that was used in this exercise had a 1.3-GHz clock rate (0.77-ns cycle time). This means that if an L3 cache hit stalls the processor without being overlapped with other useful work, the processor will lose the ability to execute 467 floating-point operations. This is significantly more than the amount of lost work associated with L2 cache misses on many competing products. One might object to comparing an L3 cache to an L2 cache; however, based on the size of the caches (16 MB for the L3 cache and 4–8 MB for many L2 caches), this seems to be the most appropriate comparison. This means that with the Power 4 processor, it is important to tune for both the L2 and L3 caches. Unfortunately, on a per-processor basis, the L2 cache is <1 MB in size.

As was previously mentioned, many of the scratch arrays are 84 elements in size. What was not mentioned is that only the first 35 elements of these arrays are being used. Attempts were made to decrease the size of these arrays, however for reasons that were not immediately clear, they broke the program. As a general rule, this would have been little more than a minor annoyance. However, when tuning loops with a STRIDE-N access pattern, one frequently uses blocking. A side effect is to change one-dimensional (1-D) arrays into two-dimensional (2-D) arrays. Given a 4-byte data size and a 128-byte cache line size for the L1 and L2 caches, on average one would expect 76% of the 2-D array to be brought into the L1 cache, even though only 42% of the

elements were being used. A more serious problem was that in many cases the scratch array was no longer fitting in the L2 cache, or at best had a low hit rate due to a high rate of eviction. By moving the declaration of the scratch array to the parent subroutine, we were able to leave the amount of allocated space unchanged. However, inside of the routine where the array was actually being used, we declared the scratch array with the smaller dimensions. This approach was applied to many of the more expensive subroutines and resulted in an overall savings of ~10% of the run time.

A.5 TLBs

All RISC-based and CISC-based systems divide main memory into pages. Similarly, the logical address space of a program is divided into pages. Page table entries (PTEs) are used to map the logical pages to the physical pages in main memory. To improve the performance of this process, a TLB is used to cache recently used PTEs either inside of, or at least close to the CPU. This puts an upper bound on the size of a TLB. Most systems use TLBs with 64–256 entries. The Power 4 processor is equipped with a 1024 entry TLB. Even so, when accessing multiple arrays in a STRIDE-N access pattern, one would not be surprised to find a large number of TLB misses. Using Perfex on Silicon Graphics Incorporated (SGI) Origin 3000 and HPM count on an IBM SP with Power 4 processors, it was found that on both systems there were an extraordinarily large number of TLB misses. Several of the more expensive subroutines were spending 50%–90% of their time on TLB misses on the IBM system.

The most desirable solution to this problem is to either reorder the indices of the arrays, or reorder the loops in a loop nest so that there is a STRIDE-1 access pattern. Many codes perform sweeps through the data first in one direction, and then in another direction. This is a critical part of the algorithm. In such a case, it will be impossible to eliminate all of the STRIDE-N access patterns. The best that can be done in such a situation is to block the code. This approach was used with many of the more expensive loops in NMS. However, in a few cases, the use of GOTOs and difficult-to-follow logic made it difficult to apply this technique. In one such case, it required multiple attempts before the routine was successfully tuned without damaging the logic. Given the complexity of this effort, several related routines that were called less frequently were left alone. Upon conclusion of this project, over 90% of all TLB misses were eliminated. This resulted in an overall savings of at least 20% of the run time.

An interesting side note is that one expensive loop had to be left alone. Attempts to apply blocking to this loop required converting 1-D scratch arrays into 2-D scratch arrays. The 1-D arrays had a high L1 cache hit rate, and a nearly 0% L2 cache miss rate. The 2-D scratch array had a poor L1 cache hit rate and a good, but not great, L2 cache hit rate. As a result of the increased number of L1 and L2 cache misses, the cost of the additional cache misses significantly exceeded the savings from the reduction in TLB misses on the IBM SP with Power 4 processors.

A.6 Methods Used to Reduce the Time Spent on Integer Arithmetic

A surprising finding when running HPMcount was that NMS was spending a significant percentage of its time (probably in excess of 50% of the time, and more like 90% of the instructions) executing integer instructions. It was found that several of the most expensive loops in the most expensive subroutine were constantly testing in inner loops to see if the current location was above or below ground. This information was stored in a 2-D integer array. In some, but not all, cases, it was possible to move this test either to an outer loop, or to move the test out of the body of the loop and into the DO statement itself. In the remaining cases, it was determined that no matter how mountainous the terrain is, the top of the atmosphere is significantly above ground level. Therefore, if one checks ahead of time for what is the highest elevation for land (a 2-D calculation), one can then use that value to move most of the tests outside of the inner most loop. The resulting savings for the 3-D calculation more than offset the cost of the 2-D calculation.

A careful analysis of usage patterns allowed us to avoid creating arrays that were never being used for the specified input data (for other input data it might still be necessary to create the arrays). A selective application of techniques for software pipelining allowed the elimination of many scratch variables and redundant calculations. Overall, the cost of the most expensive subroutine was reduced by more than a factor of 2.

A.7 Large Page Sizes

Documentation from IBM recommends using large memory pages as a way of improving performance. In theory, they should reduce the number of TLB misses. Furthermore, in the case of the Power 4 processor, they should improve the efficiency of the stream buffers. Early efforts to use large memory pages demonstrated their value. However, we were surprised to find that once the program had been tuned along the lines discussed in this report, the use of large pages actually was counter productive. For reasons that are not clear, turning on that feature actually slowed the program down by a few percent.

Appendix B. Additional Methods Used for Parallel Tuning

B.1 Introduction

This appendix discusses the remaining aspects associated with the optimization of the parallel (OpenMP) performance of the code. Issues involving the use of a single 32-processor node of an IBM SP with Power4 processors will be discussed. The remainder of this appendix will cover the following topics:

1. The effect that granularity of parallelization has on performance.
2. Other impediments to achieving linear speed-up.

B.2 The Effect That Granularity of Parallelization Has on Performance

Based on a variety of talks that the author has attended as well as some one-on-one discussions, it appears as though a commonly held misconception is that it is almost always a win to take advantage of any opportunities for parallelization one finds. Even when it is not a win, the general assumption appears to be that it is not much of a loss. As was discussed in Moore,¹ this is most definitely not the case. A rough estimate of the cost of parallelizing a section of code using loop-level parallelism is to estimate the cost of synchronization. For small numbers of processors and bus-based systems of any size, this will be approximately equal to Memory Latency * C * N. C is a constant, and is most likely to be either 2 or 4 (we will assume 4). N is the number of processors. The memory latency is likely to be at least 200 ns, and on some systems may exceed 1000 ns. For purposes of discussion, we will assume 400 ns. Larger systems may use a more sophisticated algorithm for synchronization, in which case the cost would approximately equal Memory Latency * C * C * Log base 2(N), where C and N are defined as previously mentioned. For 32 processors, this gives a synchronization cost of 51 ms. For 256 processors, the synchronization cost, assuming the more sophisticated algorithm, would be at least 28 ms, and might be larger because large systems tend to have larger memory latencies.

From this discussion, one can conclude that to avoid parallel slowdown, a parallelized section of code should require not less than 28 ms to execute on a single processor. In order to show good speed-up, the amount of work should be at least a factor of $100 * N$ greater than this. For 32 processors, that comes to 0.16 s. For 256 processors, that comes to 0.72 s. The importance of this discussion is that the subroutine AZERO was frequently used to initialize arrays to zero. Over 99% of the time, these arrays were 84 elements long. Assuming that the arrays are already in the L1 cache, this could be done in roughly 84 cycles. On an SGI Origin 3000 with 400-MHz

¹ Moore, S. University of Tennessee at Knoxville. Data on the Overhead of Parallelization. Private communication. April 2003.

processors, this comes to 210 ns. On systems with newer processors, this value would be even less. If the array needs to be fetched from memory and later written back to memory, and assuming a usable memory bandwidth of 500 MB/s and 4-byte data, the initialization would take 670 ns. Clearly, these calls should never have been parallelized. On the other hand, a small percentage of the calls involved considerably larger arrays and therefore those calls needed to be parallelized. Based on this analysis, AZERO was split into multiple loops, and depending on the size of the array, different code is selected for execution.

AZERO was not the only short count loop that had been parallelized. We believe that all such loops have now been identified, and the parallelization constructs were either removed entirely or made conditional as in the case of AZERO. This had the unfortunate side effect of increasing the percentage of serial code. In other words, it effectively helps to establish an upper bound on the parallel speed-up when using large numbers of processors. In an attempt to limit this side effect, ways were found to eliminate most of the calls to AZERO. Additionally, efforts were made to maximize the efficiency of AZERO and related routines. One of the most promising of these approaches appeared to be to use one or another of the compiler directives for data prefetching in an attempt to minimize the cost of initializing the large arrays (this was now over 50% of the time spent in AZERO). Unfortunately, it was not until the project was nearly completed that it was discovered that on the IBM SP these directives are normally ignored when they occur in loops parallelized using OpenMP. There appears to be a complicated set of compiler options that one can use to get around this limitation, but we ran out of time before we could make them work.

The initial parallelization effort left many loops to the automatic parallelization option found on most of today's compilers. So long as the loops were simple enough, this was an efficient division of labor. However, a careful analysis of the output from a profiled run indicated that some of these loops were not being parallelized. Individually, the cost of these loops was miniscule. Even when taken in combination, they represented a small percentage of the total CPU time. The problem was that when run on 30 processors, the combined time spent in these loops was 15–20 min for a 6-hr run. Some of the loops proved to be difficult to parallelize, but enough of the work was either parallelized or eliminated through serial optimizations to reduce the run time by at least 15 min.

B.3 Other Impediments to Achieving Linear Speed-up

As is discussed in Moore,¹ when dividing finite units of parallelism between processors, the ideal speed-up will more closely resemble a staircase than a straight line (e.g., with 15 units of parallelism, it can be divided evenly between 5 processors, but not 4 or 6 processors). This effect becomes most pronounced as the number of processors approaches the degree of available parallelism. In the case of Nonhydrostatic Model Simulation (NMS), this effect is further complicated by the internal workings of the code. Based on certain hard-coded parameters, the program clusters multiple units of parallelism into a single group. Effectively, this reduces the

available level of parallelism. When absolutely necessary, one can change the hard coded parameters, thereby increasing the usable level of parallelism. Tests were run on the IBM SP with Power 4 processors to see what effect this would have on the run time. The hope was that so long as the parallelism was roughly an integer multiple of the number of processors being used, the run time would be unaffected. This was not what we found. Instead, the smaller the group size, the greater the usable level of parallelism, but also the greater the run time, by up to 25%. The obvious solution for the IBM SP, where we were limited to using no more than 32 processors, was to use the largest group size possible. However, for systems with larger numbers of processors, this effect would be one more impediment to achieving anything close to linear speed-up.

Another problem has to do with competition between the operating system and the application for processor time. When dealing with a large system such as the SGI Origin 3000 with up to 2048 processors, it is reasonable to expect that a few processors will be reserved for use by the operating system, interactive jobs, and the like. On systems with smaller node counts, the normal practice is to schedule work on all of the processors. Therefore, with a 32-processor node size, one would normally expect the load factor to be 32. Adding in the operating system overhead, one will generally see a load factor of roughly 32.5. When dealing with a single 32-processor coarse grained application, the additional 0.5 units of work is easily spread across the 32 processors, resulting in a predicted parallel speed-up of 31.5. However, NMS is parallelized using loop-level parallelism, which is an example of fine-grained parallelism. Due to the increased frequency of synchronization events, even if the operating system overhead is spread across all 32 processors, it behaves as though it is running on a single processor. Therefore, the predicted speedup when using 32 processors would only be 16, while the predicted speed-up when using 31 processors would be 31. Based on this analysis, measurements were made, and it was found that for the test case there was little difference in performance between 30 and 31 processors (due to the stair-step effect), but that 32-processor runs were noticeably slower. From that point on, all runs were made using 30 processors on a dedicated 32-processor node.

Another impediment to achieving parallel speed-up on an SMP node is the available memory bandwidth on a per-processor basis. On paper, the IBM SP with Power 4 processors has a very impressive memory bandwidth. However, we were not seeing anything close to the peak bandwidth. After checking with IBM, it was determined that the peak bandwidth for a node equipped with <128 GB of main memory is significantly less than the published value. The system that we were using was equipped with 32 GB of main memory per node. In and of itself, this would not appear to be an impediment to achieving linear speed-up; however with prefetching, a single processor should be able to take advantage of at least M/N MB/s of memory bandwidth, where M is the peak memory bandwidth and N is the maximum number of processors in a node. Because the system being used only had a peak memory bandwidth of m , where $m < M$, it is not surprising that one would run out of memory bandwidth before

running out of processors. When dealing with a collection of serial jobs, this is not likely to be a serious problem because different parts of a program will put different levels of stress on the memory system. Therefore, assuming that each job randomly enters and exits the high bandwidth and low bandwidth subroutines, then one might not see much of a problem. Similarly, most jobs parallelized using MPI are coarse grained jobs. While the progress of each process associated with the MPI job is far from random, there is enough of a disconnect that one might see this problem to a much smaller extent. Unfortunately, since NMS is parallelized using loop-level parallelism, which is inherently fine grained, all of the processors progress in lock step between high bandwidth and low bandwidth subroutines. Therefore, even if on average there is enough bandwidth to show linear speed-up, in practice the high bandwidth subroutines are likely to be bandwidth-starved when all/most of the processors are being used by a single job. This appears to be the best explanation as to why some of the subroutines showed nearly linear speed-up, while others showed at best factors of 20–25 speed-ups.

NO. OF
COPIES ORGANIZATION

- 1 DEFENSE TECHNICAL
(PDF INFORMATION CTR
ONLY) DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218
- 1 US ARMY RSRCH DEV &
ENGRG CMD
SYSTEMS OF SYSTEMS
INTEGRATION
AMSRD SS T
6000 6TH ST STE 100
FORT BELVOIR VA 22060-5608
- 1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS
AT AUSTIN
3925 W BRAKER LN STE 400
AUSTIN TX 78759-5316
- 1 US MILITARY ACADEMY
MATH SCI CTR EXCELLENCE
MADN MATH
THAYER HALL
WEST POINT NY 10996-1786
- 1 DIRECTOR
US ARMY RESEARCH LAB
IMNE ALC IMS
2800 POWDER MILL RD
ADELPHI MD 20783-1197
- 3 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CI OK TL
2800 POWDER MILL RD
ADELPHI MD 20783-1197
- 3 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CS IS T
2800 POWDER MILL RD
ADELPHI MD 20783-1197

NO. OF
COPIES ORGANIZATION

- ABERDEEN PROVING GROUND
- 1 DIR USARL
AMSRD ARL CI OK TP (BLDG 4600)

<u>NO. OF</u>	<u>ORGANIZATION</u>
<u>COPIES</u>	
1	PROG DIR C HENRY 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	DEP PROG DIR L DAVIS 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	DISTR CTRS PROJ OFC V THOMAS 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	HPC CTRS PROJ MGR J BAIRD 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	CHSSI PROJ MGR L PERKINS 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	NAVAL RSRCH LAB J OSBURN CODE 5594 BLDG A49 RM 15 WASHINGTON DC 20375-5340
1	NAVAL RSRCH LAB J BORIS CODE 6400 4555 OVERLOOK AVE SW WASHINGTON DC 20375-5344
1	USAF WRIGHT-LAB B STRANG WL/FIMC BLDG 450 2645 FIFTH ST STE 7 WRIGHT PATTERSON AFB OH 45433-7913

<u>NO. OF</u>	<u>ORGANIZATION</u>
<u>COPIES</u>	
1	NAVAL RSRCH LAB R RAMAMURTI CODE 6410 4555 OVERLOOK AVE SW WASHINGTON DC 20375-5344
1	ARMY AEROFLIGHT DYNAMICS DIR R MEAKIN MS 258-1 MOFFETT FIELD CA 94035-1000
1	NAVAL RSRCH LAB J MCCAFFREY HEAD OCEAN DYNAMICS PREDICTION BR CODE 7320 STENNIS SPACE CTR MS 39529
2	USAF WRIGHT-LAB J SHANG WL/FIM 2645 FIFTH ST STE 6 WRIGHT PATTERSON AFB OH 45433-7913
1	USAF PHILIPS LAB S WIERSCHKE OLAC PL/RKFE 10 EAST SATURN BLVD EDWARDS AFB CA 93524-7680
1	NAVAL RSRCH LAB D PAPCONSTANTOPOULOS CODE 6390 WASHINGTON DC 20375-5000
1	AIR FORCE RSRCH LAB/DEHE R PETERKIN 3550 ABERDEEN AVE SE KIRTLAND AFB NM 87117-5776
1	NAVAL RSRCH LAB G HEBURN RSCH OCEANOGRAPHER CNMOC BLDG 1020 RM 178 STENNIS SPACE CTR MS 39529

NO. OF
COPIES ORGANIZATION

1 AIR FORCE RSRCH LAB
INFORMATION DIR
R LINDERMAN
26 ELECTRONIC PKWY
ROME NY 13441-4514

1 SPAWARSSYSCEN (D4402)
R WASILAUSKY
BLDG 33 RM 0071A
53560 HULL ST
SAN DIEGO CA 92152-5001

1 USAE WATERWAYS
EXPERIMENT ST
J HOLLAND
CEWES HV C
3909 HALLS FERRY RD
VICKSBURG MS 39180-6199

1 US ARMY CRD&EC
B PERLMAN
AMSEL RD C2
FT MONMOUTH NJ 07703

1 SPACE & NAVAL WARFARE SYS CTR
K BROMLEY
CODE D7305 BLDG 606 RM 325
53140 SYSTEMS ST
SAN DIEGO CA 92152-5001

1 DEPT CHAIR
T TEZDUYAR
MECH ENGR & MTRL SCI
RICE UNIV MS 321
6100 MAIN ST
HOUSTON TX 77005

1 ARMY HIGH PERFORMANCE
COMPUTING RSRCH CTR
B BRYAN
1200 WASHINGTON AVE
S MINNEAPOLIS MN 55415

1 NAVAL CMD CNTRL &
OCEAN SURVEILANCE CTR
L PARNELL
HPC COORDINATOR 7 DIR
DOD DISTRIBUTED CTR
NCCOSC RDTE DIV D3603
49590 LASSING RD
SAN DIEGO CA 92152-6148

NO. OF
COPIES ORGANIZATION

1 ASSOCIATE DIR
S MOORE
INNOVATIVE CMPTG LAB
CMPTR SCI DEPT
1122 VOLUNTEER BLVD STE 203
KNOXVILLE TN 37996-3450

ABERDEEN PROVING GROUND

16 DIR USARL
AMSRD ARL CI HC
D BROWN
J CLARKE
P CHUNG
J GOWENS
B HENZ
D HISLEY
T KENDALL
P MATTHEWS
R NAMBURU
C NIETUBICZ
R PRABHAKARAN
D PRESSEL
D SHIRES
K SMITH
R VALISETTY
C ZOLTANI

INTENTIONALLY LEFT BLANK.