ARMY RESEARCH LABORATORY

Optimization of a Three-Dimensional Diagnostic Flow Solver

by Chatt C. Williamson and Yansen Wang

ARL-TR-5406 **November 2010**

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
Adelphi, MD 20783-1197

ARL-TR-5406 **November 2010**

Optimization of a Three-Dimensional Diagnostic Flow Solver

Chatt C. Williamson and Yansen Wang
Computation and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
November 2010	Progress	FY 2010

4. TITLE AND SUBTITLE

Optimization of a Three-Dimensional Diagnostic Flow Solver

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Chatt C. Williamson and Yansen Wang

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: RDRL-CIE-D
2800 Powder Mill Road
Adelphi, MD 20783-1197

8. PERFORMING ORGANIZATION REPORT NUMBER

ARL-TR-5406

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 unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this study, we investigate the performance of a three-dimensional diagnostic flow solver. The numerical solver is profiled to find the "hot spots" of the code. These hot spots are then optimized for performance using techniques such as loop reordering and cache blocking. OpenMP is used to parallelize the code execution, further enhancing its performance. Finally, preliminary efforts to use NVIDIA Inc. Compute Unified Device Architecture (CUDA) programming model are discussed.

15. SUBJECT TERMS

3DWF, optimization, diagnostic wind flow, Poisson equation, BiCGSTAB

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Chatt C. Williamson
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	26	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-1378

Contents

List of Figures iv

List of Tables iv

1. Background 1

1.1 Problem Statement 1

2. Optimization Roadmap 4

2.1 Operating Environment and Tools for Building the 3DWF Model 4

2.2 Code Execution Profiling Techniques....4

2.3 Serial Optimization Techniques 5

2.4 Parallel Optimization Techniques 6

3. Profiling and Optimization Techniques Applied to 3DWF 7

3.1 Profiling Results....7

3.2 Serial Optimization....9

3.3 Code Object setup_3dwf 9

3.4 Code Objects conv_check and compu_uvw....9

3.5 Code Objects init_uvw and linear_interp....10

3.6 Code Object cgstab....10

3.7 Data Structures 11

3.8 Parallel Optimizations 11

3.9 Code Objects setup_3dwf, init_uvw, conv_check, and compu_uvw....12

3.10 Code Object cgstab....13

3.11 General Purpose GPU Computing 14

4. References 15

Appendix A. Hardware and Software Configuration 17

List of Symbols, Abbreviations, and Acronyms 18

Distribution List 19

List of Figures

Figure 1. Flow chart of iterative method. 2

Figure 2. Matrix structure for the Laplacian operator using a seven-point stencil (25x25x13) subject to boundary conditions (nz=number of zeros).....3

Figure 3. Pseudo code for the Preconditioned BiConjugate Gradient Stabilized Method. (reproduced from Barrett et. al, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods"). 3

Figure 4. Output from the Linux gprof utility.....8

List of Tables

Table 1. Performance characteristics of 3DWF for different compiler flags used to control the optimization level.....7

Table 2. Profile of the code execution using cpu_time intrinsic to ascertain the total wall clock time each section of code takes to execute.....8

Table 3. Wall clock timings per iteration for conv_check and compu_uvw before and after optimizations as well as the speedup. 10

Table 4. Total number of loops parallelized for each of the subroutines. 12

Table 5. Observed timings for each of the subroutines parallelized using OpenMP directives. ..12

Table 6. Observed speedups for each of the parallelized routines.....12

Table A 1. Hardware Description. 17

1. Background

Modeling the dispersion of contaminates is an interesting problem in many respects. The numerical models derived are often computationally quite intensive. Contributing to the computational intensity is the size of the problem. It is usually desirable to model at a resolution that is computationally affordable with respect to available resources and time. Implemented on massively parallel systems, the sky is literally the limit. However, when such systems are not readily available or portable, more conservative and/or simplistic approaches are needed.

When modeling dispersion, we often consider two processes—(1) transport and (2) diffusion. Here we can equate transport with the mean flow of the atmosphere and diffusion with turbulent spread. If we consider these two processes separately, we can derive some very simplistic models that are good estimates for how the atmosphere is behaving in a mean sense. One such method to consider, given a set of observations of wind speed and direction, is how the atmospheric flow is behaving throughout the domain in a mean sense. This method is often called diagnostic. We have developed such a model, the 3-Dimensional Wind Flow (3DWF) model, in the U.S. Army Research Laboratory (Wang et al., 2005). While limited, since it does not incorporate the thermal characteristics explicitly, the method is still quite computationally intensive. Such assumptions following the work of Sasaki (1970) and Sherman (1978) give rise to the following functional:

$$E(\vec{u},\lambda) = \int_V \left[\beta_1^2(u_1 - u_1^o)^2 + \beta_1^2(u_2 - u_2^o)^2 + \beta_2^2(u_3 - u_3^o)^2 + \lambda\left(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}\right)\right] dV.$$

In the end, one can solve for the Lagrange multiplier, which is interpreted as pressure in this situation, by using conservation of mass as a constraint subject to boundary conditions. With this estimate, an iterative system is set up to minimize the difference between modeled and observed winds. This variational problem is then recast into a differential equation and solved numerically. The corresponding Euler-Lagrange equation is a Poisson equation for the Lagrange multiplier with flow-through (no obstacle) and no-flow-through (solid obstacle) boundary conditions.

1.1 Problem Statement

A simple model of the iterative system is given in figure 1. This model can be considered in phases. The first phase is initialization, followed by a computation phase, and then an output and/or reduction phase. In this code optimization study, we are mostly interested in the computation phase, though aspects of the initialization phase are also addressed. Here, the computational phase consists of a linear system solver, a velocity correction step, and a convergence test. The linear system that is solved follows directly from the discretization of the spatial problem using a Cartesian grid and the seven-point stencil of the Laplacian operator

(Wang et al., 2003, 2005). The seven-point stencil of the Laplacian operator gives rise to a matrix whose structure is represented in figure 2. This matrix has seven diagonals with a bandwidth that is dependent upon the problem size and the ordering of the spatial discretization points. For our problem, we consider a Cartesian grid that is 401 × 401 × 201 (i,j,k) with a *jik* ordering. This implies that the bandwidth of the matrix is 2*i*j or 320,602. As will be discussed later, this matrix is not very "cache-friendly". While the structure of the matrix is band-diagonal, it is not symmetric once boundary conditions are imposed. This fact limits the types of solvers that can be used. For iterative solvers, the Bi-Conjugate Gradient Stable algorithm is a good choice. Figure 3 is pseudo-code for the BiCGStab algorithm with preconditioning.

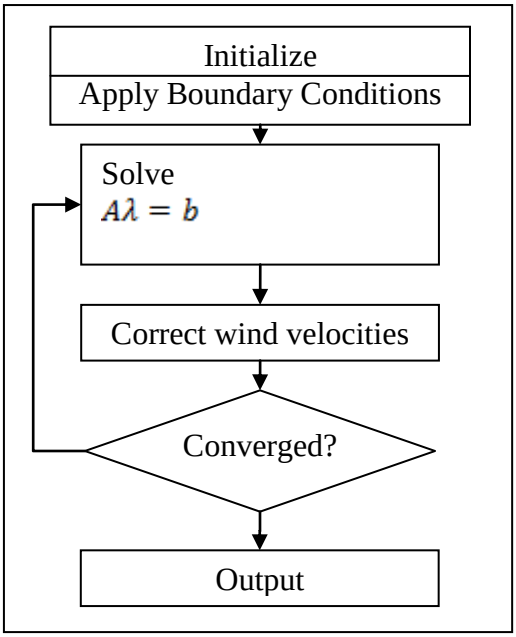

Figure 1. Flow chart of iterative method.

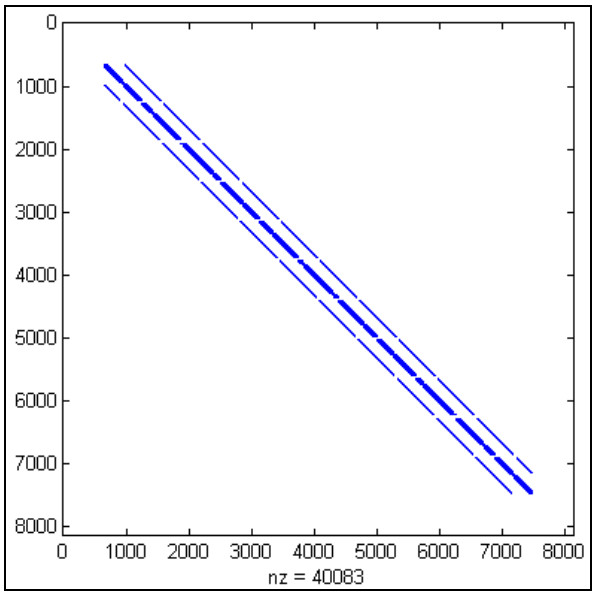

Figure 2. Matrix structure for the Laplacian operator using a seven-point stencil (25x25x13) subject to boundary conditions (nz=number of zeros).

Compute $r^{(0)} = b - A\lambda^{(0)}$ for some initial guess $\lambda^{(0)}$
Choose $\tilde{r}$ for example ($\tilde{r} = r^{(0)}$)
for $i = 1,2, \ldots$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$
    $p^{(i)} = r^{(i-1)}$
  else
    $\beta_{i-1} = \left(\frac{\rho_{i-1}}{\rho_{i-2}}\right)\left(\frac{\alpha_{i-1}}{\omega_{i-1}}\right)$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1}\left(p^{(i-1)} - \omega_{i-1}t^{(i-1)}\right)$
  endif
  solve $M\hat{p} = p^{(i)}$
  $v^{(i)} = A\hat{p}$
  $\alpha_i = \frac{\rho_{i-1}}{\tilde{r}^T v^{(i)}}$
  $s = r^{(i-1)} - \alpha_i v^{(i)}$
  check norm of s; if small enough: set $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p}$ and stop
  solve $M\hat{s} = s$
  $t = A\hat{s}$
  $\omega_i = t^T s / t^T t$
  $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
  $r^{(i)} = s - \omega_i t$
  check convergence; continue if necessary
  for continuation it is necessary that $\omega_i \neq 0$
end

Figure 3. Pseudo code for the Preconditioned BiConjugate Gradient Stabilized Method. (Reproduced from Barrett et Al., "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods").

2. Optimization Roadmap

It is most important in any optimization effort to understand the problem that is before you. What are you trying to optimize for—performance, general applicability for a certain set of problems and/or platforms, etc.? For this study, we are interested in optimizing for performance on a single platform, based on the Intel Core2 micro-architecture. For a description of the hardware, see table A-1 in appendix A. With performance as our particular objective, we will follow a standard method of optimization. We start with serial code as it is and measure the performance using various tools that are available for profiling the execution of the code. With the performance information in hand, we can then pursue serial optimizations to increase the performance of the code. Given that the Intel Core2 micro-architecture is a multi-core chip, we also consider threading and or parallel optimizations once we have achieved good performance from the serial optimization stage. Finally, we can consider acceleration technologies such as CUDA technology to possibly further accelerate the performance.

2.1 Operating Environment and Tools for Building the 3DWF Model

It is important to set the foundation of our study. This study is performed using a Dell Precision T7400 workstation with dual quad-core Intel Xeon CPUs. The operating system used on the system is RedHat Enterprise Linux Release 5, with a 64 bit SMP kernel version 2.6.18. We have chosen to use the GNU Compiler Collection (GCC) version 4.4 (Stallman 2008) to compile the source code of the model. The source code is written in Fortran 90, and so we use gfortran as the main compiler driver. The baseline performance measures are based on a naïve procedure for using the GNU compilers. In other words, we compile with no command line arguments passed to the compile stage except the '-c' flag, which suppresses the executable linking stage producing object code that can be linked at a later step, and language conformance flags. This is naïve because the GNU compilers default to no optimizations. By just including "-O2", one should expect the system to speed up significantly. In various stages of the optimization process, other compiler flags will be used to turn on architectural specific optimizations, such as auto vectorization, and also use the Single Instruction Multiple Data (SIMD) unit and its accompanying instructions.

2.2 Code Execution Profiling Techniques

One is interested in understanding performance when profiling a computer code. When profiling, one looks for "hot spots" or bottle necks that are limiting the performance of the code. There are a number of ways to "profile" the execution of a computer code, and each has its advantages and disadvantages. Three such methods are "poor man's" profiling otherwise known as hand-instrumented, compiler flags to instrument the code, and system level profiling using CPU hardware counters. The first two methods are relatively simple to implement, though they

are typically limited in their granularity and their ability to expose the underlying cause of the performance or lack thereof. The third method is good for understanding the cause of the hot spot, but it is typically limited to users with higher privileges on a given workstation due to the access of the hardware counters. A fourth method of profiling—and we use the term profiling loosely here—is the time utility routine available on most Unix/Linux systems. This, however, is a very course-grained utility, as it only provides the total wall clock time of the execution of the code.

Hand-instrumenting code involves inserting timing routines into the code base. There is an associated overhead with each function or subroutine call, but, hopefully, this is small compared with the execution time of the code being profiled. These times are then reported for analysis. This technique is useful when wanting to understand the time it takes to execute a certain portion of code.

Using compiler flags for profiling is also quite useful. For the GCC, the compiler option to produce instrumented code is -pg for use with the utility program gprof. One can also use the two compiler flags, -fprofile-arcs and -ftest-coverage, and the utility gcov for profiling. It is also recommended that the debug flag -g be included when profiling. This method of profiling stops the executing code on regular intervals and samples which routine is currently executing. The utility gcov provides finer granularity than gprof, as it can be used to assess performance down to the specific line of code.

A utility program such as Oprofile (Levon, 2009) does not require any special instrumentation of the code, but as mentioned before, it does require some elevated privileges to be used properly. This utility can be used to set collection and sampling in the hardware counters for a number of events. Some events that are of interest when profiling code for performance are TLB and L2 cache misses. The accompanying tools opreport, opannotate, and opgprof are used to analyze the sampled data.

2.3 Serial Optimization Techniques

Significant performance increases can often be realized with simple optimization techniques. One needs to consider the characteristics of the central processing unit (CPU) and the memory hierarchy when performing serial optimizations. Knowing the type, number, and operation of the functional units available on a CPU can help in structuring code and data. It also helps in choosing appropriate optimization flags for the compiler being used. Considering the memory hierarchy is called cache-aware programming.

Cache-aware programming is defined as trying to optimize both temporal and spatial locality of data and code in the CPU's cache. Two simple techniques for increasing locality and or reuse are loop reordering and cache blocking. Data structures are also very important when it comes to performance. If at all possible, data should be accessed using unit stride and aligned to the appropriate word boundaries for the CPU under consideration. Loop reordering is one

optimization technique that provides unit stride access. Block caching increases the reuse of data in the cache. Both of these techniques can lead to significant performance increases. Other techniques, such as loop unrolling, can also be beneficial.

As stated previously, knowing the type and number of functional units that a particular micro-architecture employs can lead to coding choices that provide significant performance gains. For example, if the micro-architecture exposes a SIMD unit, one could write loops or choose algorithms that could be easily vectorized. This vectorization also helps in increasing the performance of code execution because the throughput can be significantly enhanced. A properly structured loop can usually be recognized by an optimizing compiler and automatically vectorized.

Finally, using computational libraries, such as the BLAS (Basic Linear Algebra Subprograms) (Lawson et al., 1979), can lead to performance increases. This is especially true for vendor-supplied libraries, such as Intel's Math Kernel Library (MKL) or AMD's Core Math Library (AMCL). These vendors have optimized the BLAS for their architectures for the user. One other package is GotoBLAS (Goto and van de Geijn, 2002), which optimizes the BLAS routines in light of TLB structure.

2.4 Parallel Optimization Techniques

There are basically two different paradigms for parallel execution. The first, shared memory, is best represented typically by the OpenMP (Dagum and Menon 1998) model. The second paradigm is distributed memory, often represented by the Message Passing Interface (MPI) (Gropp et al., 1998). In this study, we investigate shared memory optimizations using OpenMP directives. This choice is driven by two factors: the application space and the ubiquity of multi-core architectures. While MPI could be deployed on multi-core CPUs, there is an associated overhead with the explicit communications in MPI that could potentially limit overall performance.

Parallelizing using OpenMP involves inserting directives in the code to tell the compiler that it is safe to parallelize the code. There are many directives defined in the OpenMP standard. Anecdotally, one can realize the most performance for the least amount of work using the "!$omp Parallel Do" directive with accompanying clauses. One can also define parallel regions using the "!$omp Parallel Region" directive with "!$omp Do" to accomplish the same thing. There is an inherent advantage in doing this—the threads spawned in a parallel region remain active using CPU cycles. This is not the case with the "!$omp Parallel Do," which idles threads that are spawned once they reach an implicit barrier and the code enters serial execution. With OpenMP v3.0 standard, the advantage is negated by the environment variable OMP_WAIT_POLICY. If this is set to ACTIVE (i.e., using the bash shell command "export OMP_WAIT_POLICY=ACTIVE"), then the spawned threads continue to spin and use CPU cycles, and are not idled. Other directives are of interest but are not used in this study. For example, "!$omp Parallel Sections" with the associated directive "!$omp Section" can be used to

run sections of independent code in parallel; a linear solver using red-black ordering, for example, can be parallelized using this method.

Finally, one other technique of parallel optimization is the exploitation of massively parallel architectures, such as graphics processing units (GPU). The burgeoning field of General Purpose GPU (GPGPU) code optimization and performance acceleration is emerging. NVIDIA has exposed their graphic hardware using CUDA, which is a c-like programming language that can be used to accelerate numerical computation using a GPU.

3. Profiling and Optimization Techniques Applied to 3DWF

First we will consider the base line case discussed previously. The problem set is for a 401 × 401 × 201 Cartesian grid with four outer iterations and 10 inner iterations for the BiCGStab algorithm. Using the Linux "time" utility, we observe the total wall clock runtime of the flow solver for various combinations of compiler flags. These data are reported in table 1. We employ a best-effort benchmark and report the best wall clock time for each case. This entails running the flow solver five times for each compiler option configuration, establishing a base case against which later optimization efforts can be compared.

Table 1. Performance characteristics of 3DWF for different compiler flags used to control the optimization level.

gfortran compiler optimization flags	Best Effort Wall clock Time (s)	% Performance Increase relative to base
-c -ffree-form	351.988	-
-c -O2 -ffree-form	195.411	80.1
-c -O2 –ffree-form –ftree-vectorize -march=core2 -msse3	188.157	87.1

As is demonstrated in table 1, the choice of compiler flags can significantly improve performance. Setting the optimization level of the compiler to -O2 resulted, as expected, in a significant increase in performance. Adding auto vectorization and architecture flags increased performance further.

3.1 Profiling Results

In order to profile the code execution, the necessary flags were added to the command line using the best case reported in table 1. The gprof utility output is reported in figure 4.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 75.91    126.96   126.96        4    31.74    31.74  cgstab_
 11.94    146.92    19.96        3     6.65     6.65  linear_interp_
  6.88    158.43    11.51        4     2.88     2.88  compu_uvw_
  2.37    162.39     3.96        4     0.99     0.99  conv_check_
  1.57    165.01     2.62        1     2.62   167.27  MAIN__
  1.05    166.76     1.75        1     1.75     1.75  setup_3dwf_
  0.22    167.12     0.36        1     0.36     0.36  terrain_bc_
  0.09    167.27     0.15        1     0.15    20.11  init_uvw_
  0.00    167.27     0.00        1     0.00     0.00  google_map_
```

Figure 4. Output from the Linux gprof utility.

From the output of the gprof profiling utility, we see that there are six subroutines that consume most of the time. The five subroutines cgstab, linear_interp, compu_uvw, conv_check, and setup_3dwf are of interest for optimization efforts. The most expensive subroutine is expectedly the routine that implements the BiCGStab algorithm. Here we observe that each call to cgstab takes, on average, 31.74 s. The second-most expensive routine is linear_interp. This routine is called three times from init_uvw. In order to compare with the "poor man's" method, we include the time of linear_interp with its parent subroutine init_uvw.

Using "poor man's" profiling, a similar picture emerges as to that painted by gprof. Table 2 reports the wall clock time for the execution of each of the expensive subroutines.

Table 2. Profile of the code execution using cpu_time intrinsic to ascertain the total wall clock time each section of code takes to execute.

Subroutine	**Total Time (s)**	**Time per call (s)**
cgstab	139.0	34.90
init_uvw	22.73	22.73
compu_uvw	12.68	3.17
conv_check	4.36	1.09
setup_3dwf	1.88	1.88

So given these two profiles, we know where to apply our serial optimization efforts. While there appears to be a discrepancy between the total times of the two methods used, this is to be expected. The utility provides a statistical sample of the execution time and not the total timing. Also, there is a slight amount of overhead in the call to the intrinsic subroutine cpu_time. There is also some time consumed in reporting the timing to the console.

3.2 Serial Optimization

In this section we will discuss serial optimization applied in order from the least expensive subroutine, setup_3dwf, to the most expensive subroutine, cgstab. After the serial optimizations were applied, significant performance increases were observed. The total wall clock time was reduced from 188.157 s to 165.311 s. Nearly half of the increase was attributed to optimizations and restructuring of init_uvw and linear_interp.

3.3 Code Object setup_3dwf

There is a minimal chance for significant serial optimization for this section of code. Also taking just slightly less than 2 s of computation time it would hardly seem worth it. We only address it now because restructuring three loops in the code does produce some benefit and makes this section of code more amenable to parallelization later on.

Two loops of code have the following structure:

```
x(1)=0
do i=2,ni
        x(i)=x(i-1)+dx
end do
```

For a fixed dx, these loops are best structured as follows:

```
x(1)=0
do i=2,ni
        x(i)=x(1)+dx*(i-1)
end do
```

The performance increase here is minimal, though it lays the predicate for parallel optimization. The second technique applied to this code object is also instructive and results in a more noticeable performance increase than the previous one. This technique is loop fusion. For this case, we move the initialization of a variable out of complicated triply nested loop and fuse it with a single nested loop that iterates through the entire vector. These optimizations take the wall clock time from 1.88 s to 1.67 s—admittedly not a significant performance increase, but illustrative of some simple optimization techniques.

3.4 Code Objects conv_check and compu_uvw

We have combined the discussion of the optimizations of these two code segments together since the optimizations for each are similar. There are two significant barriers to increased performance in these two code segments. The first is a compare inside a loop, and the second is the mixing of 1-d and 3-d array access.

Complex or multiple compares inside a loop quickly become expensive, as the program may experience a significant number of jumps, causing functional unit stalls. This type of coding should be avoided if at all possible. Fortunately, in this particular code, this is mitigated with splitting the loop on the outer-most index. This can be done if we know a priori the result of the

condition. In this case, we can drop the conditional from the loop and observe a significant speed increase. The increase is dependent upon what the relative size of the remaining loops is after the splitting.

The second impediment to performance is a little more subtle. With mixed indexing—for example, 1-dimensional and 3-dimensional arrays being referenced in loops—choosing the appropriate loop order is important. In this particular case, the 3-dimensional arrays follow an ijk ordering, but the 1-dimensional array is ordered in memory following a jik scheme. As it turns out, the best choice in this situation was to reorder the loops from kij to kji, favoring the 3-dimensional array access.

Table 3 reports the wall clock timings per iteration and speed-ups using these two optimizations.

Table 3. Wall clock timings per iteration for conv_check and compu_uvw before and after optimizations as well as the speedup.

	Wall clock per iteration before Optimization (s)	**Walk clock per iteration after optimization (s)**	**Speedup**
conv_check	1.09	0.54	2.02
compu_uvw	3.17	1.48	2.14

These optimizations evidenced excellent speed-ups for these two subroutines, each with a speed-up factor in excess of 2. Since each subroutine is executed four times in the benchmark study, we observe a savings of almost 9 s in total wall clock time.

3.5 Code Objects init_uvw and linear_interp

Addressing init_uvw, or more specifically linear_interp, presents an interesting case when it comes to optimization. An analysis of the source code is what led to a significant increase in performance. The key insight here was to realize that there was a significant amount of extraneous work that was being done. Limiting this work by adjusting the loop indices to the affected regions yielded significant performance increase. This code had a similar malady as the previous two with if-constructs embedded in do-loops. While these could not be easily removed, one could split the loop as before. However, the portion of loop that could be split using a priori information happened to be extraneous work. Eliminating this extra unproductive work resulted in an increase in performance. Along with loop reordering to be more cache-friendly, the resultant code was roughly 94% faster. The observed wall clock time was reduced from 22.73 s to 11.74 s.

3.6 Code Object cgstab

The serial optimization of this portion of code proved to be a little more problematic. Various methods of optimization were used that provided little, if any, benefits when it came to performance. This is the most expensive portion of the code by far. Without any code restructuring or optimization, this code consumed just shy of 35 s per iteration. We attempted

blocking loops for better cache performance, flattening of loop structures to hopefully expose possible vectorization to the compiler, and special library calls such as to BLAS and GotoBLAS.

There was an interesting interplay as various optimizations were applied to the loops present in the code. Sometimes we would increase the performance associated with the loop or portion of code that we were in the process of optimizing. However, this would then have a detrimental effect on the performance of sections of code that had been previously optimized. We finally settled on a given set of optimizations that appeared to work together. This set of optimizations included flattening some loops, blocking a few, and leaving the remainder unchanged. Overall the performance increased from 34.9 s per iteration to 34.3 s per iteration, an overall time savings of 2.4 s.

It should be noted that the attempt to use GotoBLAS proved to be unproductive. When linking using the optimized library, we actually observed that performance decreased significantly. We suspect that this was due to threading issues in GotoBLAS in the way that it was compiled.

Overall, this particular code segment has very bad cache performance, which has proven to be the limiting factor in any serial optimizations.

3.7 Data Structures

A final concept that should be considered when performing serial optimizations is data structures. While the issue was not specifically addressed in this study, there is still a significant potential impact on performance. Currently, this model code represents and stores the diagonal matrix A associated with the discretization as seven 1-dimnesional arrays. This appears to be suboptimal, perhaps because the seven arrays are not stored contiguously in memory as they would be if the matrix were stored using a common sparse matrix format, such as the DIAG format. If the arrays were not stored contiguously in memory, it would lead to poor performance in the cgstab subroutine where these arrays are used in the matrix vector multiply operation of the BiCGStab algorithm.

One other aspect with respect to data structure, as mentioned previously, is the use of jik data ordering. This choice leads to a large bandwidth for the diagonal matrix. For the current problem, this bandwidth could be halved by using a kij or kji data ordering scheme. Bandwidth comes into play when we are accessing the elements of the vector in the matrix vector multiply operation. The smaller the bandwidth, the better the chances are for cache reuse.

3.8 Parallel Optimizations

The methodology chosen for parallelization is shared memory using OpenMP v3.0 directives. By far the most used directive is the “parallel do” directive with its associated clauses. Across all files, a total of 35 loops were parallelized using OpenMP. Table 4 lists the number of loops that were parallelized for each of the subroutines parallelized.

Table 4. Total number of loops parallelized for each of the subroutines.

	Number of Loops Parallelized	Parallelization Construct
setup_3dwf	6	parallel do
init_uvw	4	parallel do
conv_check	2	parallel do
compu_uvw	11	parallel do
Cgstab	12	parallel do, parallel do (reduction:+)

3.9 Code Objects setup_3dwf, init_uvw, conv_check, and compu_uvw

As listed in table 4, the total number of loops parallelized for these code segments was 23. The parallelization of each of these loops was fairly straightforward using the "parallel do" OpenMP directive. Table 5 reports the timing for each subroutine, given the number of threads used. For reference, the best serial time is also reported.

Table 5. Observed timings for each of the subroutines parallelized using OpenMP directives.

Subroutine	Serial (s)	OpenMP(s)		
		1 Thread	2 Threads	4 Threads
setup_3dwf	1.67	1.59	0.84	0.53
init_uvw	11.74	11.57	6.68	3.93
compu_uvw	1.48	1.51	0.80	0.53
conv_check	0.54	0.58	0.33	0.22
Cgstab	34.30	36.32	27.88	25.45
Sum	158.69	166.80	123.56	109.26
Total Time (wall clock)	165.31	173.58	130.69	116.27

As is to be expected when running the parallelized program, we generally observe a slight overhead compared to the optimized serial program. The two exceptions in this case are for setup_3dwf and init_uvw. In these instances, we speculate that the parallel directives in portions of the code exposed to the compiler further optimizations. We also observe reasonable scalability through four threads for all the subroutines but cgstab (see table 6). Recall that speed-up is limited by Amdahl's law, which basically states that the speed-up is limited by the portion of the code that must execute in serial.

Table 6. Observed speedups for each of the parallelized routines.

Subroutine	Speedup for 2 Threads	Speedup for 4 Threads
setup_3dwf	1.99	3.15
init_uvw	1.76	2.99
compu_uvw	1.64	2.45
conv_check	1.85	2.79
cgstab	1.23	1.35
Overall	1.26	1.42

3.10 Code Object cgstab

By far cgstab is the most expensive subroutine and also deserves the most attention. As indicated in table 4, 12 loops of this subroutine were parallelized. In the process of parallelization, two loop fission optimizations were performed. This resulted in two more loops than there were in the serial version. All of the fissioned loops were parallelized. The total number of loops after the fission optimization was 17. The 12 loops that were parallelized were done in a straightforward manner.

Among the 12 loops that were parallelized were two loops that carried out the matrix-vector multiply operation as part of the BiCGStab algorithm. A third loop, which computed the initial residual, was also parallelized, though it was not part of the BiCGStab algorithm. Of the four scalar products in the BiCGStab routine, two exhibited a performance increase and two did not. The performance increase was most likely due to cache reuse. The two scalar products that were parallelized use the "parallel do reduction" directive on a single loop. These resulted in slightly different residuals as compared to the serial version, but they were acceptable. It should be noted that there is no expectation that the residuals should be the same following a reduction operation using floating point arithmetic. The two vector updates of the BiCGStab routine were parallelized. Two scaling loops associated with the "solve Mp=p0" section of the algorithm were also parallelized. Finally, the loop setting the initial residual and initializing vectors to zero was also parallelized. The zeroing of vectors was an attempt to exploit the first touch memory optimization. The remaining loops were reductions associated with the loop fissions.

Of more interest are the five loops that were not parallelized. These loops are the ones that are limiting the scalability of the cgstab subroutine. The first loop that was not parallelized set up the preconditioning matrix diagonal. This one is computed once every time cgstab is called. Serial timings indicate that this loop takes about 0.5 s of CPU time. The other four loops were the forward and backwards sweeps associated with solving the linear system involving the preconditioning matrix. The backward sweep took roughly twice as long as the forward sweep. Undoubtedly, this asymmetry in computation time is associated with poor cache behavior.

There are strategies to parallelize such loops, such as using level set scheduling or topological ordering. No strategies were employed to parallelize this section of code. It has been reported (Rothberg and Gupta 1990) that the scaling is not that good.

An estimate of the serial time to complete these loops is on the order of 0.65 s per linear system solve. With the BiCGStab performing 10 iterations, these four loops combine for a total CPU time of roughly 13.5 s. Combining this estimate with the other known serial costs for scalar products yields a total scalar computation time of roughly 15.5 s. With Amdahl's law in mind, and assuming perfect scalability of the parallel loops, we can at best see a speedup factor of 2.2. We are pleased with the fact that we achieved a significant fraction—1.35 or 61%—of the total estimated speed-up. Here, too, we suspect that data structures are a limiting factor, as well. Poor cache reuse in general is a killer when it comes to overall performance.

3.11 General Purpose GPU Computing

General purpose GPU computing is an up-and-coming field of research in the computational sciences. Some initial work in porting portions of the cgstab subroutine to use GPU computing has been done. The computing model that we chose to use was NVIDIA's Compute Unified Device Architecture (CUDA) (NVIDIA, 2008). This architecture programming uses a c-like language. Since the flow solver code is implemented in Fortran 90, we needed to use an interface between the two programming languages. Flagon (Gumerov, 2007) is one such interface.

Initial attempts to use Flagon were not fruitful. In our initial efforts, we chose to test the scalar or dot product. This particular routine contained a bug in the interface implementation provided by NVIDIA. We were able to correct this bug and compile the Flagon interface modules using the GCC. Once we did this, we were able to link to the CUBLAS library routines. While successful, we did not observe any appreciable performance increase. One reason for this is that although the scalar product on the CUDA enabled device was nearly 10 times faster, the memory transfers negated the improvements. The CUBLAS SDOT was faster even with memory transfers, but not by much. We also realize that the scalar products are not a significant portion of the total computation time, but this was our first attempt at using Flagon and CUDA.

Future work will include the use of a CUDA kernel for the sparse matrix-vector multiply. Appreciable gains may be realized, but once again, the performance will be limited by the loops that are not parallelized.

4. References

Sasaki, Y. Some Basic Formalisms in Numerical Variational Analysis. *Mon. Wea. Rev* **1970**, *98*, 875–883.

Sherman, C. A. A Mass-consistent Model for Wind Fields Over Complex Terrain. *J. Appl. Meteor.* **1978**, *17*, 312–319.

Wang, Y.; et al. *A High Resolution, Three-Dimensional, Computationally Efficient, Diagnostic Wind Model: initial Development Report*; ARL-TR-3094; U.S. Army Research Laboratory: Adelphi, MD, October 2003.

Wang, Y.; Williamson, C.; Garvey, D.; Chang, S.; Cogan, J. Application of a Multigrid Method to a Mass-Consistent Diagnostic Wind Model. *Journal of Applied Meteorology* **2005**, *44* (7), 1078–1089.

Lawson, C. L.; et al. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. On Math. Soft.* **1979**, *5* (3), 308–323.

Dagum, L.; Menon, R. OpenMP: An Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering* **1998**, *5* (1), 46–55.

Gropp, W.; et al. *MPI: The Complete Reference* (2nd ed.). The MIT Press, September 1998.

NVIDIA. *NVIDIA CUDA Programming Guide 2.0* 2008.

Barrett, Richard; et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. 2nd Edition, SIAM.

Saad, Yousef. Iterative Methods for Sparse Linear Systems. 2nd Edition (available online), January 2003.

Gumerov, N. A.; et al. Middleware for Programming NVIDIA GPUs from Fortran 9X. Poster at Supercomputing 2007.

Goto, K.; van de Geijn, R. A. *On Reducing TLB Misses in Matrix Multiplication*; Tech. Rep. CS-TR-02-55; Department of Computer Sciences: The University of Texas at Austin, 2002.

Levon, J. Oprofile manual. Accessed December 2009. http://oprofile.sourceforge.net/doc/index.html.

Stallman, R. M.; and the GCC Developer Community. Using the GNU Compiler Collection. GNU Press, 2008.

Rothberg, E.; Gupta, A. *Parallel ICCG on a Hierarchical Memory Multiprocessor – Addressing the Triangular Solve Bottleneck*; Report No. STAN-CS-90-1330, Stanford University, 1990.

Appendix A. Hardware and Software Configuration

Table A-1. Hardware description.

Dell Precision Workstation T7400	
CPU	(2x) Intel Xeon X5472 @ 3.0 GHz, 12MB L2 Cache • Quadcore • MMX, SSE, SSE2, SSE3, SSE • Based on Intel Core2 Architecture
HDD	250 GB
Memory	8GB
NVIDIA Tesla C1060	CUDA enabled device
OS	RedHat Enterprise Linux

- Software
  - Linux 64 bit SMP kernel (v2.6.18)
  - NVIDIA CUDA SDK
  - Flagon Fortan 90 Interface to CUDA
  - GNU Compiler Collection v.4.4 (gfortran)
  - gprof and gcov
  - Oprofile Suite

List of Symbols, Abbreviations, and Acronyms

3DWF	Three-Dimensional Wind Flow
AMCL	AMD Core Math Library
BLAS	Basic Linear Algebra Subprograms
BiCGStab	Bi-Conjugate Gradient Stable
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GCC	GNU Compiler Collection
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
L2	Level 2
MKL	Math Kernel Library
MPI	Message Passing Interface
SIMD	Single Instruction Multiple Data
TLB	Translation Lookaside Buffer

NO. OF COPIES	ORGANIZATION
1 ELEC	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1 CD	OFC OF THE SECY OF DEFNS ATTN ODDRE (R&AT) THE PENTAGON WASHINGTON DC 20301-3080
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV & ENGRG \ CTR ARMAMENT ENGRG & TECHNLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
1	PM TIMS, PROFILER (MMS-P) AN/TMQ-52 ATTN B GRIFFIES BUILDING 563 FT MONMOUTH NJ 07703
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD A RIVERA FT HUACHUCA AZ 85613-5300
1	COMMANDER US ARMY RDECOM ATTN AMSRD AMR W C MCCORKLE 5400 FOWLER RD REDSTONE ARSENAL AL 35898-5000
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
1	US ARMY RSRCH LAB ATTN RDRL CIM G T LANDFRIED BLDG 4600 ABERDEEN PROVING GROUND MD 21005-5066

NO. OF COPIES	ORGANIZATION
1	DIRECTOR US ARMY RSRCH LAB ATTN RDRL ROE V W D BACH PO BOX 12211 RESEARCH TRIANGLE PARK NC 27709
4	US ARMY RSRCH LAB ATTN RDRL CIE D D HOOCK BATTLEFIELD ENVIR DIV ATTN RDRL CIE R DUMAIS ATTN RDRL CIE D S LUCES ATTN RDRL CIE M D KNAPP BATTLEFIELD ENVIR DIRCTRT BLDG 1622 WHITE SANDS MISSILE RANGE NM 88002-5501
17	US ARMY RSRCH LAB ATTN IMNE ALC HRR MAIL & RECORDS MGMT ATTN RDRL CIE D C WILLIAMSON (10 COPIES) ATTN RDRL CIE D D GARVEY ATTN RDRL CIE D G D HUYNH ATTN RDRL CIE D Y WANG ATTN RDRL CIE S A WETMORE ATTN RDRL CIM L TECHL LIB ATTN RDRL CIM P TECHL PUB ADELPHI MD 20783-1197

TOTAL: 30 (28 HCS, 1 CD, 1 ELEC)

INTENTIONALLY LEFT BLANK.