EXPERIENCES IN OPTIMIZING A NUMERCIAL WEATHER PREDICTION MODEL: AN EXERCISE IN FUTILITY?

Daniel B. Weber and Henry J. Neeman

100 East Boyd Street, Room 1110
Center for Analysis and Prediction of Storms
University of Oklahoma
Norman, Oklahoma 73019
dweber@ou.edu
hneeman@ou.edu

ABSTRACT

This paper describes the basic optimization methods applied to a Numerical Weather Prediction model, including improved cache memory usage and message hiding. The goal is to reduce the wall clock time for time critical weather forecasting and compute intensive research. The problem is put into perspective via a brief history of the type of computing hardware used for NWP applications during the past 25 years and the trend in computing hardware that has brought us to the current state of code efficiency. A detailed performance analysis of an NWP code identifies the most likely parts of the numerical solver for optimization and the most promising methods for improving single and parallel processor performance. Results from performance-enhancing strategies are presented and help define the envelope of potential performance improvement using standard commodity based cluster technology.

1. INTRODUCTION

Real time weather forecasting can be used to minimize impacts of severe or unusual weather on society, including the transportation industry, energy and food production markets, and manufacturing facilities. Yet the combined weather related losses remains significant with more than $13B in losses annually [1]. A portion of these losses could be mitigated with more timely and accurate weather forecasts, especially weather related to severe storms and tornadoes. At present, severe weather forecasting is limited by computer processing power, as the National Center for Environmental Prediction is not able to generate weather products for public use with sufficient detail to resolve individual thunderstorms and to distinguish between non-rotating and rotating thunderstorms. During the 2005 National Weather Service Storm Prediction Center (SPC) spring forecasting experiment, forecasters, researchers, students and meteorologists assessed several different numerical forecasts of severe weather conditions across the eastern two-thirds of the US [2]. One of the goals was to assess the variability of forecasts with differing horizontal resolutions, such as the number of

mesh points for a given distance. In particular, two forecasts were compared, the 2km forecast submitted by the Center for the Analysis and Prediction of Storms (CAPS), and the 4-km forecast submitted by the National Center for Atmospheric Research (NCAR). On April 28, 2005, the two forecasts differed in the type of convection forecast. The 2-km forecast produced thunderstorms that split and exhibited hook echoes, as evident from the simulated radar field, while the 4-km grid forecast produced a line of storms (Fig. 1). Both forecasts did not verify over Central

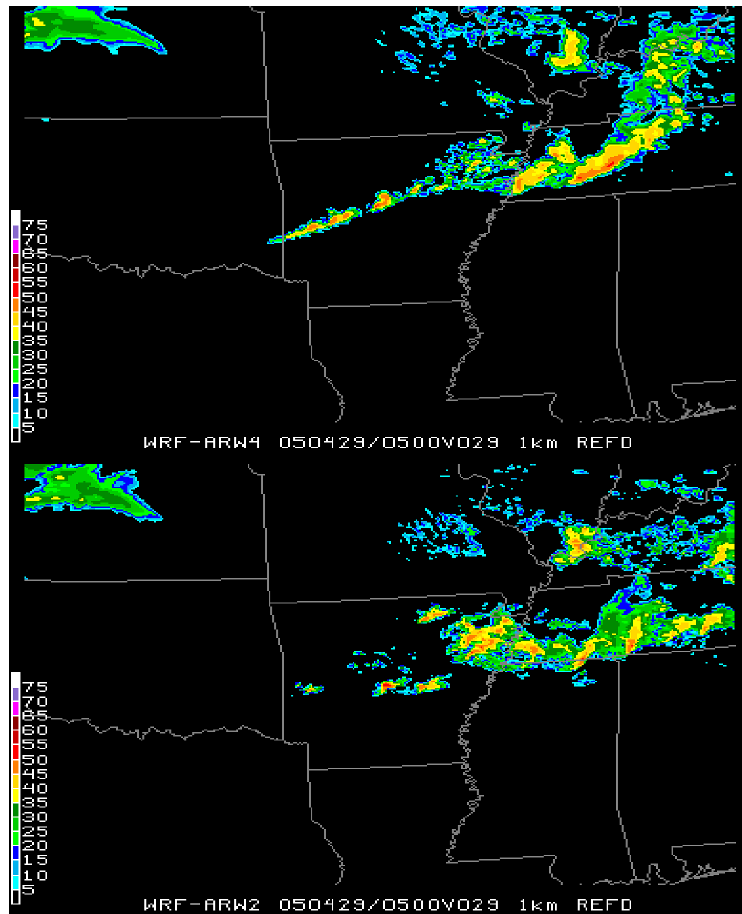

Figure 1. Model predicted radar reflectivity values, similar to that expressed on WSR-88D NWS radar. Top panel is the 4-km Gird spacing WRF forecast and the bottom panel is the 2-km WRF simulated radar forecast. Note red and orange colored isolated regions over central Arkansas depicting individual storms and the line of radar indicated storms in the top panel across central Arkansas. Plots courtesy of the NSSL Spring Experiment web site and data base for April 28, 2005, found at: www.spc.noaa.gov/exper/Spring_2005/archive/.

Arkansas, but the additional information available in the 2-km forecast, related to the potential for the atmosphere to develop hook echo producing/rotating thunderstorms, convinced the forecasters to consider an elevated risk of tornadoes in that region. The above example is one of many possible illustrations used to convey the importance of mesh resolution for predicting severe thunderstorm activity. The 2km forecasts were 8 times more expensive to produce due to the 4-fold increase in horizontal mesh points and a 2-fold increase in the time step resolution.

Two potential solutions exist to reduce forecast computational requirements and increase the warning time of the perishable forecast products: 1) provide larger, more powerful computers and/or 2) improve the efficiency of the numerical forecast models. This paper will focus on the latter, since we have little influence on the type or size computers that are purchased and developed. But, the authors recognize that each year brings more computing capacity to the end user, as illustrated in Figure 2, which shows the change over time of the most capable Top 500 system. In 1993, the peak performance was approximately 200 GFLOPS, following Moore's Law, a doubling of computational capacity every 18 months, we would expect a 50 TFLOP system in 2005, compared to an existing 350+ TFLOP system. Even with this dramatic increase in capability, large-scale thunderstorm resolving forecasts remain unattainable. For example, a nationwide severe thunderstorm 30-hour forecast using 1-km mesh spacing would require 300 Petaflops per ensemble member, in which a few dozen ensemble members would be required to produce significant error statistics and forecast spread. A single forecast/ensemble member of this size would require 70 hours on a 3000 processor supercomputer, clearly unobtainable on today’s and any near future computer system.

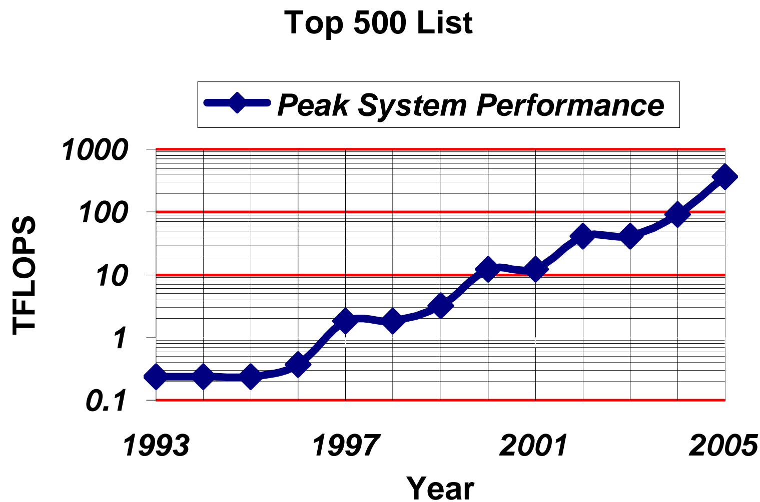

Figure 2. Plot of the peak performing computer, given in Teraflops (TFLOPS), from the past Top 500 lists [3].

From the early 1970's until the late 1990's, the dominant supercomputing platform used by numerical weather prediction centers consisted of proprietary hardware and software solutions using costly vector computing architecture. Figure 3 displays the history of the architecture type and number of cycles provided by Top 500 systems during the past 15 years and shows the vector architecture dominance until the late 1990's. Vector architecture made use of very fast access to memory and very efficient pipelined vector computing units and registers to process large amounts of data and computations efficiently. The access to main memory was generally via several memory pipelines that fed the floating point processing units. As a result of the availability of the special hardware, scientists built software using the Fortran 77 and Fortran 90 standards and prepared the computing loops in a vector friendly manner. A key vector processing requirement was to ensure stride 1 access of memory in the innermost of triply nested loops.

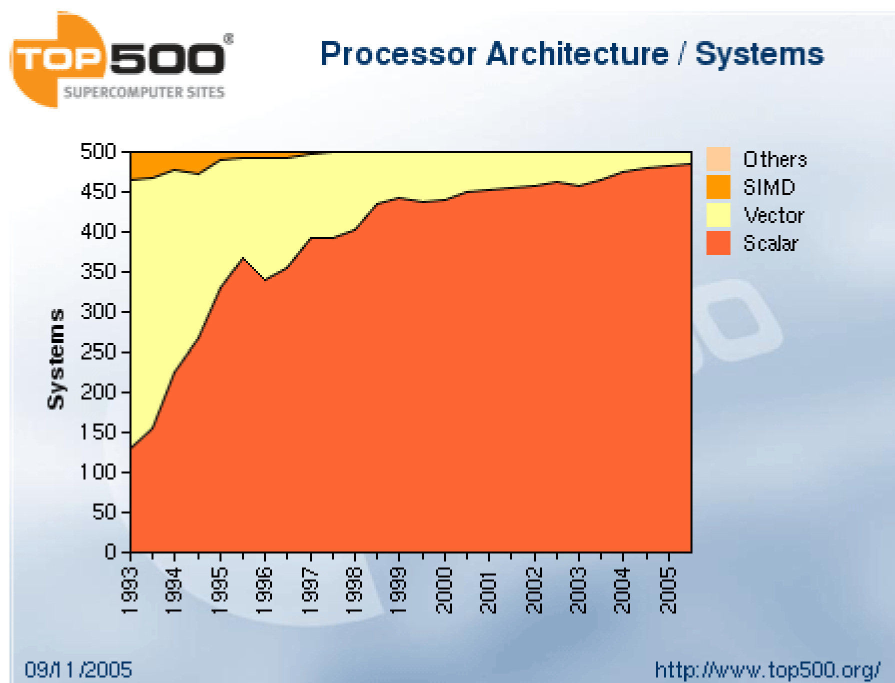

Figure 3. Percentage of cycles according to processor type, compiled from the Top 500 listed systems for the past 13 years [3].

This computing paradigm suited computing centers that were adequately funded, but these centers were severely limited in the number of users that could use the system at one time, due to the relatively high cost per processor. Software that used vector supercomputers could achieve greater than 75% of the peak floating point processing performance, usually measured in floating point operations per second (FLOPS), if the loops were vectorized by the compiler. Numerical weather prediction models, such as the Advanced Regional Prediction System [4, 5], which was developed in the

early to mid-1990's, continued to be developed using vector techniques. Most recently, the Weather and Research Forecast (WRF) model [6] considered the efficiency on new super-scalar based computing architectures. The developers performed several experiments to determine the best loop configuration for use on scalar and super-scalar as well as vector architectures. They found that switching the outer two loops from the traditional i,j,k configuration to the i,k,j format achieved both good vector performance and better scalar performance, on the order of 10-15%. The attention to code performance as a function of loop configuration is a testament to the extent to which scientists must go to recover some of the lost per processor performance associated with commodity technologies. Scalar and Super-Scalar architectures provide very fast processing units at a fraction of the cost of a vector processor. The cost difference is achieved by using memory that is much slower than the vector counterpart. In order to offset the slower memory access associated with scalar technology, a smaller and faster intermediate memory, cache memory, is used to enhance memory performance on scalar systems. For example, the memory access speed of Intel Pentium4 processor running at 3.4 GhZ (Prescott variety of the Pentium4 processor) ranges from 53-284 cycles [7]. Benchmarks, in terms of MFLOP ratings, of the ARPS weather forecast models on a variety of platforms is presented in Figure 4. The results indicate that the ARPS is achieving approximately 5-15% of peak performance on super-scalar architecture, much lower than that of vector architecture.

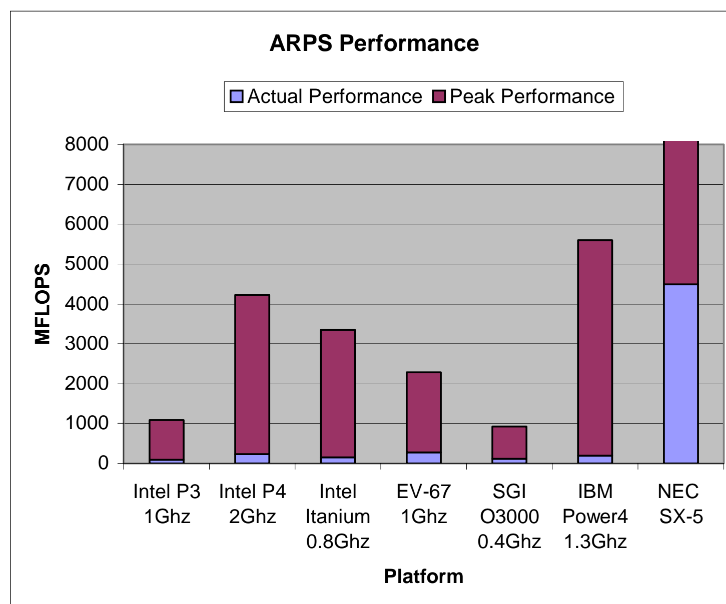

Figure 4. ARPS MFLOP ratings on a variety of computing platforms. Mesh sizes are the same for all but the NEC-SX, which employed a larger number of grid points and with radiation physics turned off.

Memory access and compiler developments have not kept pace with peak processing capacity. Some may comment that if the model runtime is being reduced by applying the economical super-scalar technology in cluster form, then why bother with optimizing the software for efficient use of the new technology? This is a valid argument, but we contend that if performance is enhanced to levels equaling vector efficiencies and sustained with advancements in scalar processing, new science can be performed with the resulting speedup of the weather codes on cluster technology. The potential gains could be viewed from two perspectives without additional computing costs. First, if an eight-fold increase in efficiency is realized, the mesh resolution of an existing forecast could be increased in the horizontal directions by a factor of two, thus improving the quality of the forecast. The other possibility is that if the forecasts are extremely perishable, such as those currently associated with thunderstorm prediction, the results could be obtained eight times sooner than with the unoptimized model. Thunderstorm predictions are perishable because they are sensitive to the initial conditions and the non-linear error growth associated with the strongly nonlinear processes governing the movement, redevelopment and intensity of moist convection. In addition, the initialization of these forecasts use the most recent observations to best estimate the state of the atmosphere. The initialization process can take up to an hour, and the numerical forecast 2 hours, leaving only 3 hours of useful forecast remaining. Also, thunderstorm forecasts require significantly increased computing resources, a factor of 16 for each doubling of the mesh resolution.

We propose that the code enhancements suggested within this paper are applicable to both vector and scalar architectures and are especially effective on computers with high processor-to-memory speed ratios. This paper presents an optimization case study of a research version of the ARPS model developed at the University of Oklahoma. The techniques that were employed preserve the excellent vector performance as well as improved scalar processor throughput. Section 2 briefly describes the numerical model and presents a snapshot of the CPU usage and a review of optimization techniques that have been applied in the past to this software. Section 3 presents a recently developed approach to enhance single processor performance using tiling methods. Section 4 presents results from optimizing the parallel forecast model and Section 5 summarizes the optimization techniques applied to an off-the-shelf computing code, thoughts on how to build an efficient framework at the beginning of the code development, and a plan for future efforts.

2. Performance Profile and Review of Past Optimization Techniques

Optimizations can be characterized in two categories: within a processor (serial) and between processors (parallel) across a set of distributed processors using inter-processor communications. One effort [8] has documented the parallel enhancements of the ARPS but did not investigate single processor performance. Following the primary theme of this paper, this section and the next discuss optimizations for NWP models using a single processor on both scalar and vector architectures.

2.1 Model Description and Performance Profile

Results from two NWP models are given in this paper, the ARPS [4,5] and a research variant, ARPI [9]. Both models have been optimized using several techniques described within. Results from the research version, ARPI, will be the primary focus of this study, but the results can be generally applied to both models.

ARPI employs a finite difference mesh model solving the Navier-Stokes equations. The model invokes a big time step increment for determining the influences due to advection, gravity and turbulent mixing processes as well as the earth's rotational effects, surface friction and moist processes. A much smaller time step, typically one-eight of the big time step, is required to predict the propagation of sound waves in the fully compressible system. The model computes forcing terms for two different time levels for the velocity variables and pressure. Potential temperature, turbulent kinetic energy and moisture quantities (water vapor, cloud particles and rain drops) are updated solely by large time step processes prior to the small time step, whereas the velocities and pressure are updated using both large and smaller time step processes. A sample set of the governing equations for the velocities and pressure in Cartesian coordinates are given in equations (1)-(4) without forcing from the earth's rotation. The equations for temperature, moisture, and turbulent kinetic energy are omitted for brevity.

$$\frac{\partial u}{\partial t} = -(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}) - \frac{1}{\rho}\frac{\partial p}{\partial x} + Turb + Cmix \quad (1)$$

$$\frac{\partial v}{\partial t} = -(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}) - \frac{1}{\rho}\frac{\partial p}{\partial y} + Turb + Cmix \quad (2)$$

$$\frac{\partial w}{\partial t} = -(u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}) - \frac{1}{\rho}\frac{\partial p}{\partial z} + Turb + Cmix \quad (3)$$

$$\frac{\partial p}{\partial t} = -(u\frac{\partial p}{\partial x} + v\frac{\partial p}{\partial y} + w\frac{\partial p}{\partial z}) - \rho(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}) \quad (4)$$

The model variables are defined using a staggered grid, with scalar quantities such as temperature, pressure and moisture offset in location ½ mesh point from the velocity variables in their respective directions (Figure 5). For example, pressure is defined at the center of a mesh cell and the east-west velocity (u) is defined on the mesh cell face ½ mesh cell to the left and right of the pressure. This is true for the north-south velocity (v) and vertical velocity (w) in their respective directions. The mesh staggering is a key aspect of the model and will be important when applying tiling

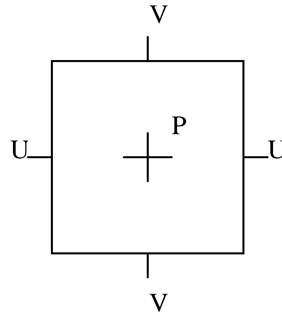

Figure 5. Relationship of horizontal velocities with pressure on the Arakawa staggered C grid [10].

concepts to the loop index limits in Section 3. The order of computation is important in this model, since the updated pressure is dependent on the new velocity field within the same small time step. The order of computation is presented below in terms of major computational tasks. The large time step forcing components, advection, gravity, and mixing and moisture conversion processes, are computed first and represent slowly evolving physical processes. The acoustic wave processes occur at a much faster rate that the large time step processes and thus are computed on a separate time scale than the large time step and use a small time step. The total number of big time steps is used to advance the forecast to the desired time. The number of small time steps within a single large time step is given by the relation:

```
Small_steps_per_big_step = 2*dtbig/dtsmall ! 2*dtbig must be divisible by dtsmall

    DO bigstep = 1,total_number_big_steps
        Update turbulent kinetic energy
        Update potential temperature
        Update moisture conversion
        Compute static small time step forcing for u-v-w-p (advection,
         mixing, gravity)

        DO smallstep = 1, small_steps_per_big_step

            Update horizontal velocities (u-v)
            Update vertical velocity (w) and pressure (p)

        END DO ! Iterate Small Time Step

    END DO ! Iterate Large Time step
```

Note that the model advances the solution forward by two big time steps, applying forcing from t-dtbig step to t+dtbig. The model makes use of function calls to the system clock to obtain estimates of the amount of time consumed by various regions

of code during the forecast simulation. Table 1 contains the output of the timing instrumentation. For a domain of 100x35x35 mesh points simulating a warm bubble within the domain, the total time for a 12 second simulation was 9.44 seconds, of which approximately 2.5 seconds were required by the small time step, or about 25% of the overall CPU requirements. Other regions of notable CPU cycles usage include the turbulence and the moisture solvers; the later requires non-integer power operations, which are extremely slow compared to multiplies and adds. Note: Longer forecast integrations yield similar results in terms of the percentage of the total CPU time, adjusting for the initialization requirements. The large time step to small time step ratio in this case was 6:1, dtbig = 6 seconds and dtsmall = 1 second. These results identify regions of the model that can be investigated further. For instance, the small time step is a likely candidate for optimizing as the percent of the total (from u,v,w, and p) is more than 20%, as is the turbulence calculations, computational mixing, and the moisture conversion processes.

Table 2 presents a summary of important characteristics regarding the memory usage and memory reuse patterns, as well as a count of floating point operations, and MFLOP rates on a Pentium3 700 Mhz processor. These requirements change between forecast models and is a function of the order of computation of the equation set. The forecast model achieved a rate of 75 MFLOPS on a Pentium3 700 Mhz Linux Based laptop, capable of 700 MFLOPS.

Table 1. Timing statistics for ARPI

Process	Seconds	Percent of Total
Initialization	= 1.25	13.2
Turbulence	= 1.97	20.9
Advect u,v,w	= 0.28	3.0
Advect scalars	= 0.42	4.4
UV solver	= 0.81	8.6
WP solver	= 1.61	17.1
PT solver	= 0.03	0.3
Qv,c,r solver	= 1.07	11.3
Buoyancy	= 0.06	0.6
Coriolis	= 0.00	0.0
Comp. mixing	= 1.03	10.9
Maxmins	= 0.37	3.9
Message passing	= 0.00	0.0
Nesting	= 0.04	0.5
Interpolation	= 0.00	0.0
Output	= 0.38	4.0
Miscellaneous	= 0.13	1.3
Total Time	= 9.44	100.0

Floating point instructions and rates of computations were computed using the Performance Application Programming Interface (PAPI, Version 2.14) installed on a LINUX (Redhat Version 7.3) based Pentium3 laptop. Both low and high level PAPI calls were inserted into the code for use in this study. Specifically,

```
fp_ins = -1
call PAPIF_flops(real_time, cpu_time, fp_ins, mflops, ierr)

…perform computations….

call PAPIF_flops(real_time, cpu_time, fp_ins, mflops, ierr)
```

and initializing fp_ins to -1 resets the hardware counters and allows for multiple uses within a subroutine or program. The above provides the user with time between PAPIF_flops calls, the total number of instructions, and the rate of floating point computations. The Intel Pentium3 processor has two hardware counters, limiting the number of low level calls to two individual operations or one derived, such as MFLOPS. Other processor types, such as the Pentium4 and several IBM and AMD varieties contain several hardware counters.

Table 2. ARPI memory requirements and usage patterns. FPI represents the number of floating point instructions required during a single pass through the operation, and MFLOPS, millions of floating point operations, is the compute speed at which the solver computes the solution. Array reuse shows the number of arrays that are used out of the total number of arrays accessed on the rhs of computations. Note: Values represent a special case in which only one pass thru both the big and small times steps was used. (dtsmall = 2 * dtbig). The MFLOP rating was obtained using a mesh size of 100x35x35 mesh zones on a Intel Pentium3 operating at 700 Mhz with a peak flop rating of 700 Mflops. Example 5 is presented in Section 3.

Solver	Array Reuse/total # of arrays used in R.H.S terms	#3-D Arrays/# of different arrays reused in R.H.S terms	#3-D Loops	FPI/ Mesh point	Mflops
Example 5	4/5	2/1	1	6	297
Turb	437/ 486	31/29	67	365	73
Solve Temp Moisture	610/ 707	43/38	116	810	93
Prep UV-WP	343/ 391	29/26	66	313	92
Prep small	23/44	28/16	10	35	40
Solve uv	21/30	9/3	2	36	115
Solve wp	28/42	15/8	6	46	78
Total	-/-	80/-	267	1605	75

2.2 Single Processor Performance Enhancements

This category is related to all techniques that can be applied in a serial manner. We will present the following codes optimization techniques:

-traditional compiler option selections
-removing divides (strength reduction)
-removing unnecessary memory references and calculations
-loop merging
-using hardware specific optimization
-loop collapsing (vector architecture)

2.2.1 Traditional Compiler Option Selections

An enhanced level of optimization can be achieved via compile-line options that make use of vector hardware, pipelining, loop unrolling, superscalar processing and other compiler and processor capabilities (Table 3). But most compilers still fall short of generating code that performs at close to the peak performance, especially for most real world applications that require large amounts of memory and contain complicated loop operations. As evident by the results presented in Section 2.1, the ARPS, and its research sister ARPI, achieve only 10% of peak performance using standard compiler options. Below is a collection of compiler options used by the ARPS and ARPI on a variety of computing platforms, including TeraGrid resources. In general, the typical optimization setting provided with the compilers will provide a good starting point in terms of optimizing real world applications. For the Intel compilers on IA-32 hardware, the –O2 option coupled with the vectorizer flag, -xK or –xW will generate the most efficient code, but perform at approximately 10% of peak. On Itanium based platforms, the –ftz option will improve performance by a factor of 500% over the –no_ftz option due to moisture solvers tendency to perform calculations on very small numbers (underflows). Several tests were performed on the Intel and Portland Group compilers to investigate various compiler options including array padding. Given the complexity of forecast models, - 75 three-dimensional arrays and hundreds of triply nested loops; tests using a variety of compiler options did not produce a combination with significant reductions in model runtime. On the IBM Power4 platform, -qhot enables high-order loop transformations and produced code that contained runtime errors (segmentation faults).

2.2.2 Divides

On scalar architectures, divides require several times more cycles than multiplications or additions require; for example, on the Intel Pentium4 Xeon processor, a multiply, add, and divide require 7, 5, and 28 cycles, respectively [11]. Loops that require divide operations can sometimes use strength reduction [12] to make use of multiplies, adds etc. instead of divides. If the original denominator is time invariant, then a reciprocal array can be pre-calculated and used to replace the divide with a multiply. Example 1, taken from the ARPI w-p explicit small time step solver, depicts this technique. The variable ptbari represents the reciprocal of the base state potential temperature, a quantity that remains constant during the forecast and can be computed once during the initialization step to replace the divide with a multiplication of a reciprocal. The loop that includes a divide computes at a rate of 74 MFLOPS while the reciprocal version computes at a rate of 98 MFLOPS. Several other operations are included in the loop and helped defer some of the costs of the divide operation. Other kinds of solvers, (e.g. tridiagonal) which cannot exploit this technique, are used in

both the ARPS and ARPI to implicitly solve the vertical velocity and pressure equations on the small time step.

Table 3. Common compiler options used in ARPS and ARPI.

Processor	Compiler	Options	Comments
Intel Pentium3	Intel (ifort)	-O3 –xK	-O2 contains: loop unrolling, strength reduction, and others, -O3 adds: align, prefetching, scalar replacement, and loop transformations and others, –xK produces code for Pentium3 processors
Intel Pentium4	Intel (ifort)	-O3 –xW	Same as above, but produces code for Pentium4 processors.
Intel Itanium-2	Intel (ifort)	-O3 –ftz	Same as the above, -ftz (i32 and i64) enables flush denormal results to zero.
AMD Opteron Pentium4 Pentium3	Portland Group (pgf90)	-fastsse -fast*	fastsse = -fast -Mvect=sse –Mscalarsse -Mcache_align –Mflushz no sse
Alpha EV-68	Compaq	-O5	-fast, + others, and -O4 with, -tune host, -assume bigarrays, strength reduction, code scheduling, inlining, prefetching, loop unrolling, code replication to eliminate branches, software pipelining, loop transformations and others
Power4	XLF90	-O3 -qmaxmem=-1 -qarch=auto -qcache=auto -qtune=auto -qstrict -qfixed	instruction scheduling, strength reduction, inlining, loop transformations and others

* used for obtaining cache optimization results in Section 3. Compile option details were assembled from the MAN page compiler command and output.

Example 1. Divide

```
  DO k=kbgn,kend
   DO j=jbgn,jend
    DO i=ibgn,iend
     pprt(i,j,k)=tem3(i,j,k)+dtsml1*(tema*(w(i,j,k)+w(i,j,k+1))/ptbar(i,j,k)
:   -temb*tem4(i,j,k)*j3(i,j,k)*(w(i,j,k+1)-w(i,j,k)))
    END DO
   END DO
  END DO
```

Multiply by the reciprocal

```
DO k=kbgn,kend
 DO j=jbgn,jend
  DO i=ibgn,iend
   pprt(i,j,k)=tem3(i,j,k)+dtsml1*(tema*(w(i,j,k)+w(i,j,k+1))*ptbari(i,j,k)
:   -temb*tem4(i,j,k)*j3(i,j,k)*(w(i,j,k+1)-w(i,j,k)))
  END DO
 END DO
END DO
```

2.2.3 Loop Merging

This technique takes two loops that are serial in nature, in which the results of the first loop are used in the second loop, and combines the loops into one larger loop. Example 2 presents two loops from the ARPI explicit W-P solver to which this technique has been applied. The overall number of computations remain the same. An advantage is obtained since the result from the first loop is not stored and retrieved from the tem1 variable in memory and there is no first loop and associated loop overhead. Merging the loops produces a decrease in the elapsed time from 0.063 to 0.042 seconds, and a FLOP rating increasing from 66 MFLOPS to 98 MFLOPS, an improvement of approximately 50%. A disadvantage to this approach is that the resulting code is less modular and therefore more difficult to understand.

The ARPS model was initially very modular: each subroutine or function contained a single three-dimensional loop, which was readable but inefficient so the ARPS was restructured and now contains very few calls to such operators. (for example, averaging temperature to a vector point (avgx (T))). The ARPS model achieved a speedup of approximately 20-25% from this restructuring, depending on the platform.

Example 2. Initial ARPI Explicit WP Solver loops:

```
DO k=kbgn,kend
 DO j=jbgn,jend
  DO i=ibgn,iend
   tem1(i,j,k)=-temb*tem4(i,j,k)*j3(i,j,k)*(w(i,j,k+1)-w(i,j,k))
  END DO
 END DO
END DO

DO k=kbgn,kend
 DO j=jbgn,jend
  DO i=ibgn,iend
pprt(i,j,k)=tem3(i,j,k)+dtsml1*(tem1(i,j,k)+tema*(w(i,j,k)+w(i,j,k+1))*ptbari(i,j,k))
  END DO
 END DO
END DO
```

Modified ARPI WP Explicit Solver Loop:

```
 DO k=kbgn,kend
  DO j=jbgn,jend
   DO i=ibgn,iend
    pprt(i,j,k)=tem3(i,j,k)+dtsml1*(tema*(w(i,j,k)+w(i,j,k+1))*ptbari(i,j,k)
:   -temb*tem4(i,j,k)*j3(i,j,k)*(w(i,j,k+1)-w(i,j,k)))
   END DO
  END DO
 END DO
```

2.2.4 Redundant Calculations

Computer programmers, for reasons of simplicity and ease of understanding, often write code that recomputes quantities that are static for the lifetime of the application run or are time invariant over parts of the application run time. After writing a computationally intensive program, it is often advantageous to review the final code product, as several intermediate results may be present and could be pre-computed and saved for use during the application run time or recomputed less frequently, as needed. For example, the ARPI model splits the solution process into the large time step which computes the gravity, coriolis, advection, and mixing contributions to the new value of the time dependant variables, and the small time step, which determines the sound wave contributions, to produce the final value of the variable at the new time. For this particular solution technique, the large time step forcing is held constant over the small time step iterations, and thus can be computed once, outside the small time step. This technique usually requires additional temporary arrays to be declared and used for storing intermediate values that are insensitive to the small time step result. An example of this approach can be found in the relationship between the big time step and the small time step. Here, equivalent potential temperature averaged in the i, j, k directions separately, are time variant with respect to the large time step but time invariant with respect to the small time step, so they are pre-calculated during the large time step. Example 3 below presents the final computation of the vertical velocity (W) on the small time step for the explicit solver using a rigid upper boundary. Note that the pttotaz and j3az variables represent the averaging of the potential temperature and the coordinate transformation jacobian in the vertical direction. Both operations are invariant in the small time step and the averaging of the Jacobian is invariant over both time steps. Computing the averaging outside of the small time step for potential temperature and j3 saves on the order of nx*ny*nz*(2*dtsbig/dtsmall)*number_big_steps*4 computations for each forecast. Note that the vertical averaging of j3 could be moved to the initialization phase of the forecast, as is done in the ARPS, but requires an additional three-dimensional array. For a forecast run of 2 hours in duration and of the order of 103x103x103 mesh points, a small domain by current forecast standards, the total number of computations saved is $576 * 10^9$.

Example 3: Final ARPI W calculation in the W-P Explicit Solver:

```
DO k=wbgn,wend ! compute the final vertical velocity for the rigid lid case
 DO j=jbgn,jend
  DO i=ibgn,iend
   w(i,j,k)=w(i,j,k)+dtsml1*(wforce(i,j,k)-(tema*pttotaz(i,j,k)*j3az(i,j,k)
:          *(pprt(i,j,k)-pprt(i,j,k-1))))
  END DO
 END DO
END DO
```

2.2.5 Hardware Specific Operations

Some scalar architectures make use of fused multiply-add units to help improve performance and reduce memory access requirements. A common hardware specialty is the MADD or FMA operation that is specifically designed to improve performance associated with computing dot products. The hardware combines the two operations into one that requires only one cycle per set of operands. The MIPS processor has this capability. This technique is not very helpful when solving hyperbolic equations such as the Navier Stokes equations used in contemporary numerical weather prediction systems. For example, a typical operation associated with the solution of a Partial Differential Equation is to take a spatial derivative in one direction and then add the result, or to average a variable and then take the derivative. Example 3 shows a complicated set of operations. The innermost calculation involves computing the vertical derivative of the perturbation pressure (vertical pressure gradient), which involves a subtraction and then a multiply, the opposite of the MADD operation. But the result of the subtraction can be used in the next operation. The result of the three multiplications coupled with the subtraction of the result from the wforce array comprise a MADD. In addition, the multiplication of this result is added to the W array to create another MADD.

2.2.6 Loop Collapse

Vector processors achieve high rates of throughput by performing loads, stores, and floating point operations at the same time using a number of pipelines in parallel. This process is known as chaining, and on the NEC SX-5 both loads and stores are chained with the floating point units. Thus, for the code to achieve peak performance, the loads, computations, and stores must be consecutive, allowing for chained operations. The strength of the vector approach is to load the pipeline with data and allow the processing unit to remain filled, thus reducing the relative overhead associated with pipelining. Loop collapsing can be used to extend the vector length of a loop to the machine peak value and extend the chaining process. Compilers are not able to fully collapse loops when the loop limits do not extend to the full dimensional length of the arrays in the loop. But with some applications, such as ARPS, the inner loop limits can be altered to expand the full length of the array dimensions. This technique is appropriate even if the array indices during the collapsed loop calculations extend beyond the defined limits for the inner dimensions. This is possible since the memory locations of the nx+1 index, for an array defined by array1(nx,ny,nz) is the next

element within the array. In addition, the computations at the extended mesh points are not needed and thus do not degrade the solution. The collapsed loop contains a larger number of computations than before, but the rate at which the computations are performed far exceeds that of the original loop, reducing the loop time considerably. Example 4 depicts a non-collapsed and collapsed loop; the original loop extends from 2, nx-1, and 2, ny-1, whereas the new loop is performed over the full inner dimensions, 1 to nx and 1 to ny. On vector machines, if a break exists between the end of one vector and the start of the next vector, chaining is terminated. With this modification, the vector length is expanded to cover nx by ny values. This greatly enhances the performance when nx and/or ny are less that the vector machine length. Results from the example 4 are displayed in Figure 6. The drop in performance when vector lengths are smaller than the peak value is evident and the modified loops enhance the performance of the operations. The tests were performed on a NEC SX-5 computer courtesy of Dr. John Romo and NEC Houston [13]. The results indicate that the much improved performance can be achieved with inner loop lengths much less than the machine vector length (256). Fortunately, the ARPS code is largely comprised of three-dimensional nested loops in which the array indices vary according to the required operation. Figure 7 displays results from applying collapsed loops to the ARPS primary solvers. Notice that the overall FLOP rating is much lower than several of the primary solvers values due to non-vectorization of the cloud and radiation transfer process solvers.

Example 4. Non-collapse and collapsed loop, ARPS Model Divergence Loop

```
 DO k=2,nz-2                  ! Non-collapsed loop
   DO j=1,ny-1                ! vectorizer is halted at ny-1 and nx-1
     DO i=1,nx-1
       pdiv(i,j,k)= -csndsq(i,j,k)*(mapfct(i,j,7)*((tem1(i+1,j,k)-tem1(i,j,k))*tema
&         +(tem2(i,j+1,k)-tem2(i,j,k))*temb)+(tem3(i,j,k+1)-tem3(i,j,k))*temc)
     END DO
   END DO
 END DO

DO k=2,nz-2                   ! Collapsed loop
   DO j=1,ny                  ! vector length is now extended to nx*ny
     DO i=1,nx
       pdiv(i,j,k)= -csndsq(i,j,k)*(mapfct(i,j,7)*((tem1(i+1,j,k)-tem1(i,j,k))*tema
&         +(tem2(i,j+1,k)-tem2(i,j,k))*temb)+(tem3(i,j,k+1)-tem3(i,j,k))*temc)
     END DO
   END DO
 END DO
```

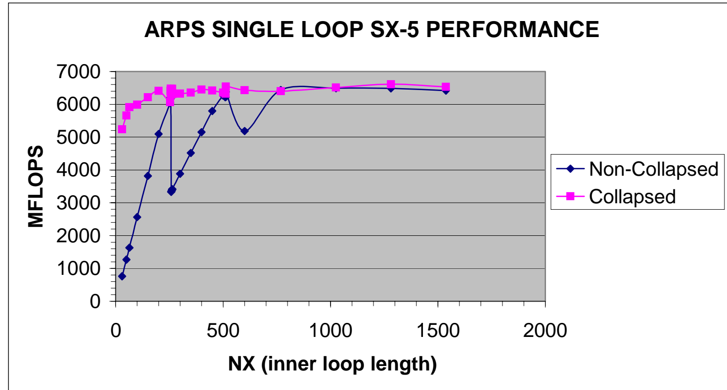

Figure 6. Results from a collapsed and non-collapsed loop performance. The SX-5 vector length is 256, and is comprised of 8 self-contained pipelines of length 32 with a peak floating point operation rating of 8 GFLOPS.

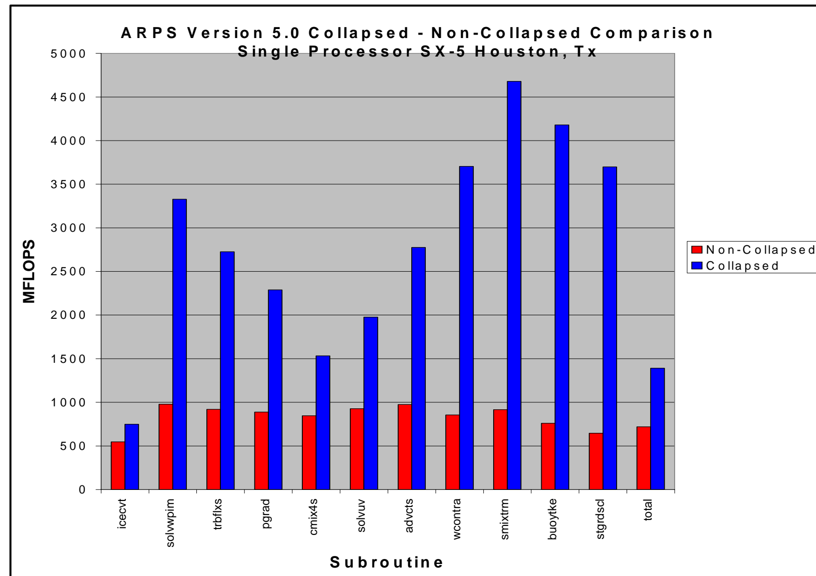

Figure 7. Results from collapsing ARPS loops for the top 11 most time intensive subroutines as determined by FTRACE in descending order from left to right. The red bars indicate the original F90 unoptimized code and the blue bars represent the F90 optimized or collapsed loops. The last set of bars on the right represent the overall ARPS FLOP rating for a grid with 37x37x35 mesh points [13].

3. Cache Optimization

With the rise of commodity-based supercomputers (see Figure 3, above), the cost of the components of a supercomputer is of great concern to vendors, to funding agencies and to purchasers. Memory and CPU prices have dropped significantly during the past 15 years, but the connection between them, whether large amounts of much faster cache memory or specialized high-performance memory bus technology, has remained relatively expensive. As a result of the relationship between speed and price, performance of real applications has suffered when porting from vector architectures, which have fast memory, to scalar architectures, which suffer from slow memory access. Today's scalar processors are capable of executing multiple instructions per cycle and billions of instructions per second. Yet the compilers are not sufficiently developed to enable most applications to achieve near peak performance, largely because of the storage hierarchy. For example, the fastest Pentium4 CPU currently available (3.8 GHz) can achieve a theoretical peak of 7.6 GFLOPs, and thus the registers can in principle consume as much as 182.4 GB of data per second – yet the fastest commonly available RAM (800 MHz Front Side Bus) can only move data at a theoretical peak of 6.4 GB per second, a factor of 28.5 difference in performance. As this hardware mismatch has intensified, the burden of improving performance necessarily has fallen increasingly on application software rather than on hardware, operating systems or compilers.

Several options have been proposed to improve performance of applications, including those techniques described in Section 2. But an arguably even more crucial issue is the set of techniques that address the ever-increasing gap between elements of the storage hierarchy, especially between CPU, cache and RAM. But a technique that is associated with the reuse of data in fast multilevel caches shows promise as a serious alternative and a more complete solution to applications that are memory bound (marching through large amounts of data during computations). Iragoin and Triolet [14] proposed what they originally termed *supernoding*, now commonly known as *tiling*, can reduce the overhead associated with accessing the same memory location several times during a series of computations. This approach allows the reuse of data in the faster L1, L2, and L3 caches commonly available with scalar processors. Tiling involves moving through a series of computations on a subset of the data that is small enough to fit in cache, and then moving on to another such subset, requiring a collection of data that resides in a specified cache on the processor.

An illustration of this principle is provided in Table 2, above. The research weather forecast model contains approximately 80 three dimensional arrays that are used during a forecast. Memory optimization results for this code using a typical commodity scalar processor, the Intel Pentium3 processor, are presented in the rest of this section. The Intel Pentium3 memory specifications are provided in Table 4 for reference.

Table 4. Intel Pentium3 Coppermine memory and processor specifications [15]

Operation	Throughput	Latency
Add	1 cycle	3 cycles
Multiply	1-2 cycles	5 cycles
Divide	18 cycles	Not pipelined
L1 access	32 byte line	3 cycles
L2 access	256 bit bus	7 cycles
Load	1/cycle	3 cycles
Store	1/cycle	1 cycle
Memory Bandwidth (100Mhz Front Side Bus)	680MB/sec	25-60 cycles

3.1 Setting Tile Loop Bounds and MPI Concerns

The solutions resulting from tiled and non-tiled cases of the same model must be identical to within machine roundoff error, although the number of total computations may vary depending on the amount of recalculation of data associated with tiled loops that recompute scalar quantities. Several different loop bound cases exist in the model, so a system to handle each specific case required was developed over several months, and most of the details are not reproduced in this paper.

3.2 Example Tiling Calculations and Results

Three simple programs were created that investigated the L1 and L2 cache behavior of a stencil computation on a Pentium3 laptop. The point of this exercise is to become familiar with the use of PAPI within a Fortran 90 program, compiled with the Intel Fortran 90 Version 7.1 compiler, and to expose the cache behavior of the selected platform with the type of operations that are applied in the forecast model. These results will be compared to tiling results later in this section. A single loop was created in each program that computed a quantity using a stencil for a given loop direction. For example, the following loops and calculations used a 5-point stencil in the I and J-directions respectively. In the below example, data reuse occurs in the i-direction (i+2, i+1, i, i-1, i-2) as the loops are advanced. The loop limits, nx, ny, nz were varied to adjust the memory requirements of the problem. Arrays U and PT were allocated, initialized, and deallocated for each change in nx, ny, and nz. Similar tests were performed for reusing data in j and k directions.

Results from the above tests are shown in Table 5 and Figures 8 and 9. FLOP ratings and L1 and L2 cache miss rates are presented as a function of data size/memory requirements (sum of the allocated array sizes). Results from the K-stencil tests are similar to the J-stencil tests and are not shown. For the J-stencil test, the peak FLOP rating is 297 MFLOPS, or 42% of the processor float point peak, with the best performance associated with the largest inner loop lengths (nx), a common result of

Example 5. I- and J-stencil sample calculations depicting data resuse.

```
DO n = 1, loopnum   ! Loopnum = 80
  DO k = 1,nz
   DO j = 1,ny     ! i-stencil calculation
     DO i = 3,nx-2
      pt(i,j,k) =(u(i+2,j,k) + u(i+1,j,k) – u(i,j,k) +
                  u(i-1,j,k) – u(i-2,j,k)) * 1.3 * n
     END DO
   END DO
  END DO
END DO

DO N = 1,loopnum   ! Loopnum = 80
  DO K = 1,nz
   DO J = 3,ny-2    ! j-stencil calculation
     DO I = 1,nx
      Pt(i,j,k) = (u(i,j+2,k)+u(i,j+1,k)-u(i,j,k)+u(i,j-1,k)-u(i,j-2,k))*1.3*n
     END DO
   END DO
  END DO
END DO
```

all performance tests. For the I-stencil, the results depend on the memory footprint of the loops, with a peak computing rate of 268 MFLOPS (38% of peak) for data sizes less than 4 KB, and 152 MFLOPS (21% of peak) for data sizes greater than 4 KB. All stencil configurations show a significant reduction in the computation rate when data sizes/memory requirements are greater than the L2 cache size (256 KB), an expected result given the latency associated with main memory and the L2 cache memory. This result indicates that a factor of two speedup in the computation occurs when the problem fits completely within the L2 cache. Figure 9 displays the total L1 and L2 cache misses for the j-stencil case and shows that the L2 misses are minimal until the data/memory footprint is similar to the L2 cache size (256 KB). The L1 misses are also closely linked to the L1 cache size, with very few misses reported for data sizes less than 16 KB, the size of the L1 data cache. The observed k-stencil cache miss rates (not shown) are similar to the j-stencil results, but the i-stencil L1 and L2 misses are higher than the j-stencil cache misses. It is not clear why the discrepancy exists. Several tests were performed to uncover the reduced performance of the i-stencil case for data sizes > 4 KB for the Intel compiler, including the analysis of all possible Pentium3 PAPI available counters. The cause of the reduced FLOP performance for the i-stencil for the Intel compiler remains unknown, as the counter data for all available Pentium3 counters yielded little additional information that could be used to obtain the cause for the reduced performance. Additional tests have been performed that remove prefetching and loop unrolling from the Intel compiled code, but the results are inconclusive.

Experiments have also been performed using the Portland Group Fortran compiler and the results contradict the Intel results, with peak FLOPS ratings of 262MFLOPS for the i-stencil for all maximized inner loop lengths and data memory requirements less than 256KB. The j- and k-stencil tests indicate reduced performance for the PORTLAND compiled code as compared to the Intel results. For tests requiring greater than 256KB, the reduction in performance is approximately 50-60% less than the <256KB cases. For all problem sizes, the maximum computational rate is approximately 40% of the peak. The number of computations and data reuse for Example 5 (J-stencil) is also provided in Table 2 for comparison.

Table 5. Sample loop computational FLOPS and Cache results

Stencil	**MFLOPS**	
	Intel	PORTLAND
I-Stencil	268 < 4KB* 152 > 4KB* 115 > 256KB*	262 < 256KB* 135 > 256KB*
J-Stencil	297 < 256KB* 150 > 256KB*	181 < 256KB* 118 > 256KB*
K-Stencil	296 < 256KB* 135 > 256KB*	172 < 256KB* 110 > 256KB*

* Total data/memory size, Peak computational rate is 700MFLOPS. The Intel compiler option was –O3 and the PORTLAND compiler option used was –fast.

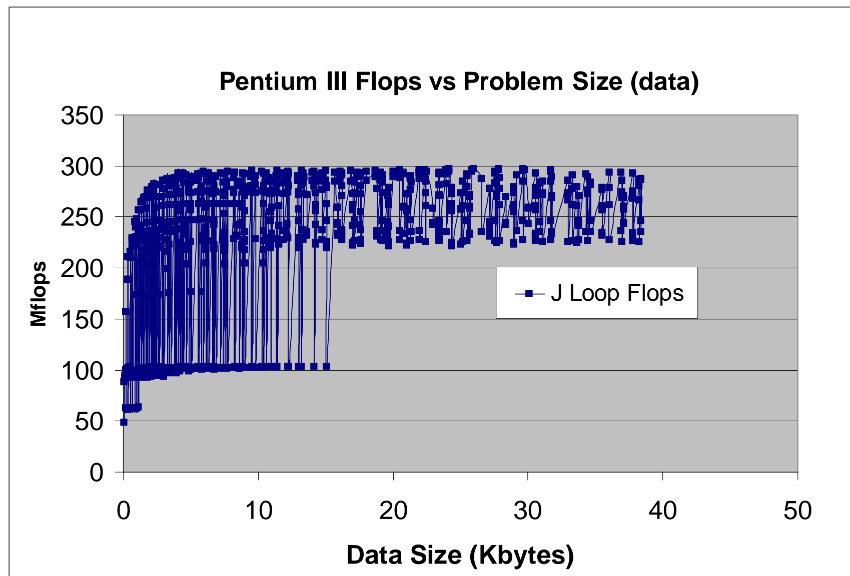

Figure 8. FLOP rating for j-stencil computations obtained using the Intel IFC Version 7.1 Fortran compiler and a Pentium3 DELL laptop running Redhat LINUX 7.3 with a peak throughput of 700MFLOPS [16].

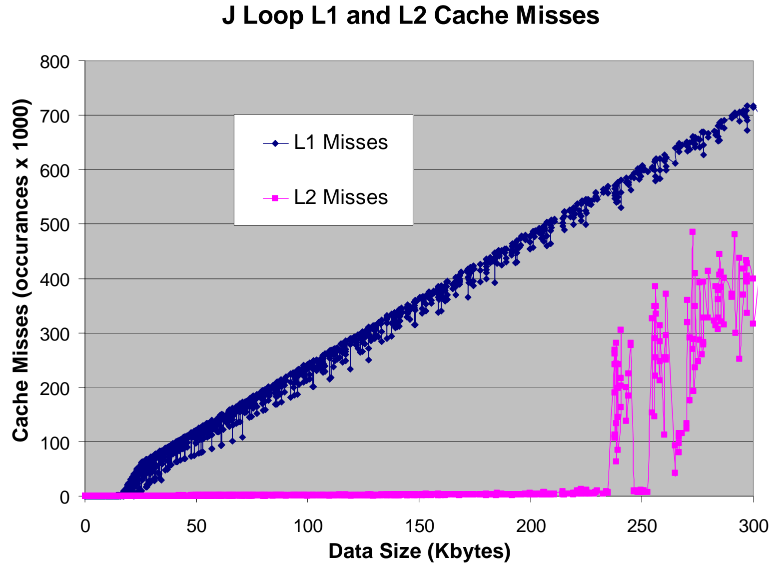

Figure 9. L1 and L2 total cache misses for the J-stencil computation [16].

3.3 Array Referencing Analysis of ARPI

Since the current code efficiency without tiling, as reported in Table 2, is on the order of 10-15% for the Pentium3 Architecture, further analysis of the model computations and memory usage/access patterns is required to obtain better serial performance. On the Pentium3 the L1 cache is 16KB, an amount too small for extended use in a real application. We will focus our efforts on optimizing L2 cache reuse for the primary cache optimization strategy. In ARPI, the primary data memory usage comes from the numerous three dimensional arrays that are accessed and updated during each time step. The number of three dimensional arrays within each solver is listed in Table 2 with the corresponding number of times they are used provided in the second column. A wide range of data reuse is observed (44-93%) across the primary solving routines. In the third column, the array reuse is portrayed in terms of the number of arrays reused on the right hand side of computational expressions as a function of the total number of arrays referenced on the right hand side. This measure of data reuse shows a better reuse rate (52 – 90%). The larger the ratio in Column 3 the higher the potential gains in code speedup when using tiling. The lowest reuse cases involve a small number of loops within the small time step, limiting the amount of potential speedup of the small time step. From Table 2, it appears that there are far more large time step computations/memory references than in the small time step. This is true, but in typical convective scale simulations, the large time step is 5-10 times larger than the small time step, thus the small time step is computed several times for each large time step, resulting in a final computational weighting of the small time step to

approximately 35% of the overall computations. Tiling the small time step, though providing a limited amount of potential improvement given the smaller data reuse ratios, will be easier to implement since the number of loops in the U-V (horizontal velocities) and W-P (vertical velocity and pressure) solvers is very small (8), especially in comparison to the remaining loops (259) in the other primary numerical solvers.

3.4 Tiling the Small time Step

The small time step is the easiest part of the model to initiate tiling. The number of loops involved is small (8) and only a few subroutine are involved. For the U-V solver, there are two loops that compute the new U and V velocities, which contain several three dimensional arrays and reuse pressure several times. The total number of three dimensional arrays used in the small time step to update the velocities and pressure is 19. There are two approaches to tiling the small time step, tile over all computations (U-V and W-P) or contain the U-V and W-P solvers into separate tile loops. Due to the staggered grid [10] used define the variable locations, velocity quantities are offset half of a mesh point from the scalar quantities (pressure, temperature, turbulent kinetic energy, moisture) and the resulting tile loops limits are different for the computation of pressure and the computation of the velocities. This issue will be addressed further later in this section. We present results for both cases; two separate tiles for the computation of U-V and W-P and a single tile for the U-V and W-P update process. The Intel Fortran compiler version did not provide reasonable results, in terms of multiple PAPI MFLOPS and other counter data requests, thus all results presented hereafter were obtained with the PORTLAND Fortran compiler.

3.4.1 U-V Solver

The U-V solver contains only two three-dimensional loops and thus will be reproduced for discussion. The loops are:

To solve for U:

```
     DO k=uwbgn,wend ! compute -cp*avgx(ptrho)*(difx(pprt)
      DO j=jbgn,jend !       + avgz(j1*tem3))
       DO i=ubgn,uend ! note ptforce is cp*avgx(ptrho)
        u(i,j,k,3) = u(i,j,k,3)+dtsml1*(uforce(i,j,k)-
  :          ptforce(i,j,k)*(dxinv*(pprt(i,j,k,3)-pprt(i-1,j,k,3))
  :         + tema*j1(i,j,k+1)*((pprt(i,j,k+1,3)+pprt(i-1,j,k+1,3))
  :           - (pprt(i,j,k,3)+pprt(i-1,j,k,3)))
  :         +tema*j1(i,j,k)*((pprt(i,j,k,3)+pprt(i-1,j,k,3))
  :          - (pprt(i,j,k-1,3)+pprt(i-1,j,k-1,3)))))
       END DO
      END DO
     END DO
```

and to solve for V:

```
      DO k=vwbgn,wend ! compute -cp*avgy(ptrho)*
       DO j=vbgn,vend !   (dify(pprt)+avgz(tem3))
        DO i=ibgn,iend ! note tke(i,j,k,3) is cp*avgy(ptrho)
         v(i,j,k,3)=v(i,j,k,3)+dtsml1*(vforce(i,j,k)-
  :          tke(i,j,k,3)* (dyinv*(pprt(i,j,k,3)-pprt(i,j-1,k,3))
  :          + tema*j2(i,j,k+1)*((pprt(i,j,k+1,3)+pprt(i,j-1,k+1,3))
  :            -(pprt(i,j,k,3)+pprt(i,j-1,k,3)))
  :          +tema*j2(i,j,k)*((pprt(i,j,k,3)+pprt(i,j-1,k,3))
  :            -(pprt(i,j,k-1,3)+pprt(i,j-1,k-1,3)))))
        END DO
       END DO
      END DO
```

Pressure (pprt) is reused several times within each loop, but others arrays such as uforce, vforce, tke, ptforce are only used once. The results from tiling only the U-V solver loops are displayed in Table 6 and reveal a negligible performance improvement for the tiled loop over the non-tiled loop. This is due to the low number of variables reused by the small number of loops (2). The U-V solver requires nine three-dimensional arrays and thus allows a maximum of 7111 mesh points to reside within the L2 cache. The larger the number of mesh points that can fit into cache for a particular solver, the less the loop overhead and fewer tiled loops. As the results indicate, tiling the U-V loop separately provides little improvement over the non-tiled version.

Table 6. MFLOPS and memory requirements for the various ARPI solvers with and without tiling.

Solver	# of Arrays	# of mesh points/ 256KB	Memory Requirements (KB)*		MFLOPS	
			No Tiling	Tiled	No Tiling	Tiled
U-V Only	9	7111	4410	180	115.7	116.3
W-P Only	15	4266	7350	150	79.4	92.7
U-V-W-P	19	3368	9310	190	91.3	105
Prep Small Time Step	28	2285	13720	280	42.2	51.1

*No-tiling memory requirements are determined by the full number of mesh points multiplied by 4 bytes and the number of three dimensional arrays. For the tiled cases, memory requirement calculations were determined by the tiles size that registered the peak performance. The L2 cache is 256KB and code was compiled with the PORTLAND GROUP Fortran compiler.

3.4.2 W-P Solver

The final W-P solution process involves 4 times as many three-dimensional loops than the final U-V solver and as a result the loops are not reproduced here. The W-P solver directly follows the U-V solver and tiling loops via a subroutine call. In this first U-V and W-P solution case, the W-P solver is contained within a separate tiling loop, for a total of two distinct tiling loops within the small time step. The results of the W-P tiling tests are presented in Table 6 and show a moderate improvement (16.7%) for the tiled loop vs the non-tiled loop. The memory footprint for the optimally tiled case is approximately 190KB, approximately one-third smaller than the L2 cache size. The optimal loop tiling size tested and reported in Table 6 for the W-P Only case was 100 x 5 x 5 mesh points. Other tiling tests were performed that revealed that the maximum inner loop length always performed better than smaller inner loop lengths. We concede that a better choice of tile size would further optimize the loop performance for all of the solvers tested and reported in this work. But, we contend that the current examples provide sufficient evidence to validate the effectiveness/ineffectiveness of tiling a weather prediction application.

3.4.3 Combined U-V-W-P Solver

In this configuration, a single tiling loop is used to solve for U-V-W-P. This configuration has more potential for data reuse since the variables U, V, J1, and J2 are available for reuse in the W-P solvers. Variables TKE, PTFORCE, UFORCE and VFORCE are not used in the W-P solver. This solution configuration required more arrays and allows for fewer mesh points to be included in the L2 cache. The results of this test (U-V-W-P) is provided in Table 6 and shows an enhancement of the MFLOPS ratings (15%) for the tiled case over the non-tiled case, similar to that of the W-P Only case.

3.5 Large Time Step

As eluded to earlier, the majority of the number of three-dimensional loops reside in the large time step part of the weather prediction model. This is true for other models such as the ARPS and the WRF. Implementing tile loop bounds into the existing big time step loops is a significant man-power intensive task that could take months to complete. Due to the complexity of tiling the large time step, only a portion of the large time step computations have been tiled within ARPI at this time. Results are shown for the existing tiling components and work continues towards the complete tiling of the large time step solvers. We believe that the largest enhancement of the code efficiency, through the use of tiling, can be achieved on the large time step, since the largest levels of reuse to total number of arrays used resides within the large time step solvers. As discussed in Section 2.1, the small time step solvers are the last part of the integration process to produce the next future state of the atmosphere. Therefore we choose to work backwards from the small time step and introduce/develop tiled loops into the large time step computations.

3.5.1 Pre-Small Time Step Computations

The computations prior to the small time step prepare several intermediate quantities that are constant on the small time step and are therefore pre-computed and used in the small time step (see method described in Section 2.2.4). The loops within this part of the code include relatively few computations (35) per the number of loops (10) and are related to memory swaps, time filtering and spatial averaging. The results for the tiling tests are presented in Table 6 and show a 25% improvement over the non-tiled version, but only experience on the order of 50 MFLOPS.

3.5.2 Other Large Time Step Computations

Work continues in tiling the large time step. At the present time, we are working on the tiling of the advection processes for the U,V,W, and P variables. When this is completed, the advection for the other variables, PT, TKE and the moisture quantities will be simplified.

4.0 Parallel Processing Enhancements

Traditionally, NWP models use the distributed memory paradigm to achieve parallelism, typically using the Message Passing Interface (MPI) with data decomposition, in which all processors obtain the identical code but operate locally on a different physical part of the global physical domain. For example, a typical use of the parallel model for ARPS, WRF or ARPI across 4 processors could be to apply a processor each to the west and east coast regions of the US and two processors to the central portion of the US to complete a full continental scale weather forecast. Messages are created and passed to neighboring processors to update solution boundary conditions for use in the next time step. The above model assumes that the domain decomposition, the method for parallelization, occurs in the i- and j-directions only. This choice is supported by a code analysis of the potential number of required updates in the three model spatial dimensions. It was found that the number of potential vertical message passing events was significantly larger for the k- or vertical direction than for either of the i- or j-direction message passing events, but the authors concede that for experiments using enhanced vertical resolution, vertical message passing may be needed. The result of decomposing the physical domain is that each processor performs calculations on different data, representing different yet overlapping sub-regions of the global domain.

For each application (ARPS, WRF , and ARPI), some of runtime is spent performing local computations and some is spent updating boundary zones via message passing. The ultimate goal of any optimization effort involving message passing is to minimize the time spent passing messages between processors and thus to achieve a performance profile that is nearly 100% computations. Parallel application performance varies widely depending on the need for data along processor boundaries. To achieve this level of parallel performance, the message passing

component must be either eliminated or reduced significantly. This section presents two methods for reducing message passing overhead on parallel applications.

4.1 Reducing Message Passing Requirement via Model Structure

Software structure can simplify code implementation, readability and performance. Parallel programming approaches that require updating of the boundary values can be simplified via several techniques, most of which are related to increasing the number of fake or buffer zones along the boundaries to reduce the frequency of required send/receive pairs. The numerical schemes used to approximate the mathematical equations are key aspects of how to prepare the code for minimal message passing. Ultimately, one could build into the top level program, the ability to set the number of fake zones to achieve fewer messages. Consider the following differential equation provided in Example 6, in which a fourth order centered differencing scheme has been applied to obtain the east-west advection of temperature.

Example 6. Fourth order i-stencil advection operation for U:

$$advu^t = \frac{4}{3}[\overline{\overline{u}^x \delta_x u}^x] - \frac{1}{3}[\overline{\overline{u}^{2x} \delta_{2x} u}^{2x}]$$

where the x overbars denote averaging over *n* points and the delta operator is a differential operation in the denoted direction (in this case, along the *x*-axis). The corresponding Fortran loop is:

```
tema = dxinv/24.0
temb = 2.0*dxinv/3.0
DO k=kbgn,kend  ! compute 4/3*avgx(tem2)+1/3*avg2x(tem3)
 DO j=jbgn,jend  ! signs are reversed for force array.
  DO i=ibgn+1,iend-1
   force(i,j,k)=force(i,j,k)
:  +tema*((u(i+1,j,k)+u(i+2,j,k))*(spres(i+2,j,k)-spres(i,j,k))
:     +(u(i,j,k)+u(i-1,j,k))*(spres(i,j,k)-spres(i-2,j,k)))
:   -temb*(u(i+1,j,k)*(spres(i+1,j,k)-spres(i,j,k))
:      + u(i,j,k)*(spres(i,j,k)-spres(i-1,j,k)))
  END DO
 END DO
END DO
```

Notice that values of the U velocity at i+2 and i-2 are needed to properly compute the term at i. In the ARPI model, the number of fake mesh points was deliberately set to 2 on each processor boundary face to accommodate this requirement (i+2 and i-2). Other models, such as the ARPS, only include one fake mesh point and therefore must communicate results from one processor to another to compute the above advection forcing term. This type of computation, one that requires two fake mesh points, also occurs in the turbulence and in the 4th order forcing terms for computational mixing. With only one fake mesh point specified in the model, the

computational mixing calculations are performed in two stages: the first derivative is computed and then that result is used to compute the second derivative over the same region. But the second derivative does not have the result of the first derivative at i+1 and i-1, especially at an interior boundary, and thus the values must be updated from nearby processors. This would require a message passing event. But, if the initial calculation was computed over a region that is increased by one unit in each direction, the first derivative could be expanded to include sufficient coverage to support the second derivative calculation without a message passing event, at the cost of a modest increase in memory footprint and a modest amount of redundant calculation. By building the additional fake mesh points into model, no MPI overhead is incurred for these operations. This approach is used in the ARPI research model. A generalized approach to this method could be implemented by adding a parameter, defined at runtime by the user, that allows for changes in the number of fakes zones, distinguishing between the number of zones needed by the numerical differencing scheme and the number that establishes the optimal balance between redundant computation versus communication. The result of implementing this message passing optimization technique into the model is that there remains only message passing event for each of the time dependant variables, or 9 events per one pass through the big and small time steps (for the case when big time step = small time step) (Table 7).

Table 7. Summary of MPI message passing optimization techniques in terms of the number of messages within ARPI.

	Number of Message passing events per processor		
Solver	Unoptimized	Method #1 Fake Mesh point expansion	Method #2 Message Grouping
Advection	36	0	0
Computational Mixing	16	0	0
Turbulent Mixing	28	0	
Update variables	9	9	3
Total	89	9	3

4.2 Message Grouping

Message grouping can be described as combining multiple message passing events into a single larger event. The advantage to this approach is that the constant latency charge for initiating a message passing event is the same for small or large messages, so this approach removes the overhead of multiple latencies that would be associated with several single message passing events. Figure 10 displays the scalability diagrams for two cases using the single and multipass approaches with ARPI on the now retired SGI Origin 2000 supercomputer at NCSA (Balder). The results show that the several single message passing events are less efficient than three larger message events, leading to a performance improvement of approximately 50% in the message

passing component of the simulation. This optimization method was applied in combination with the technique described in Section 4.1. As a result of both methods described in Sections 4.1 and 4.2, ARPI now contains only three message passing points within the model, reducing the number of message passing events from 89 in the original implementation. The following message grouping is used: large time step group contains potential temperature, moisture quantities (3) and turbulent kinetic energy, small time step group #1 U and V velocities, and small time stem group #2 vertical velocity and pressure. With message grouping, the same amount of data is passed between processor, but the latency is reduced significantly.

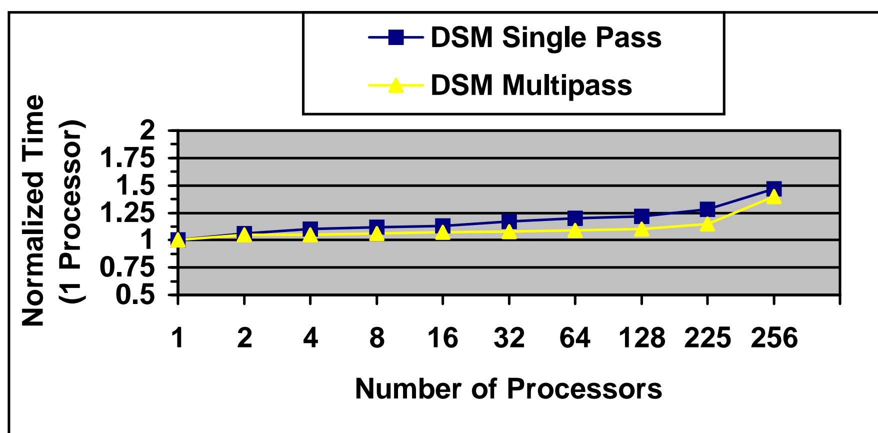

Figure 10. Scalability diagram depicting the total normalized wall clock time for ARPI for the two cases: single pass - each variable is updated with a single message (blue curve) and multipass - grouping variable updates with other variables computed in nearby code/operations (yellow curve). The domain is 23×23×63, dtbig = 20 sec, dtsml = 4 sec.

4.3 Message Hiding

The time dependant variables in the model are computed in a serial fashion – turbulent kinetic energy, potential temperature, water quantities within the large time step, and the velocity components and pressure within the small time step. This serial implementation of the solver creates opportunities for message hiding via computations. Message hiding can reduce the wall clock time of an application by overlaying the MPI operations with computations. MPI allows the user to send messages a variety of ways including blocking and non-blocking. Blocking communication requires the sender and receiver to be ready and waiting for the exchange, performing no other task while waiting for their interlocutor (another processor) to perform a corresponding message operation. Thus, there is the potential for one processor to prepare and send a message with the receiving processor idle until the message has been received, which is a function of network bandwidth and current load.

By contrast, the non-blocking approach allows the code developer the option of posting a receive before the sender has prepared the message, and then continuing on with other computations while waiting for the message to arrive. The receiving processor can periodically check to see whether the message has arrived and then process the data and move on to the next operation. Ultimately, the goal of the use of non-blocking communication is to perform computations while the message is in transit, thus adding additional parallelism to the program and hiding the message passing component with existing computations.

However, there is a limit to the extent to which a message can be masked by computations. Consider the timing statistics presented in Table 1. Since that example represents a single processor run, there is no MPI component. But imagine that Table 1 also showed 3 seconds attributed to message passing; the new total time would be roughly 12 seconds. If we were to mask all of the MPI time, this new optimized version's timing would look similar to the single process version. An example of message hiding within ARPI is provided in Example 7 and is a modification of the description of the ARPI solution process supplied in Section 2.1. This case shows that upon the computation of TKE and other large time step variables, an asynchronous communication can be initiated, since the data is not used immediately by the next group of operations. The advection and computational mixing solvers are called to prepare U, V, W, and P big time step forcing, which requires approximately 15% of the total wall clock time for the simulation. Thus, if the asynchronous messages associated with the large time step were 15% or more of the total run, then the wall time would be reduced by the amount of time required to compute during the asynchronous send/receive, or approximately 15%.

Example 7. Message hiding within ARPI:

```
DO bigstep = 1,total_number_big_steps
        Update turbulent kinetic energy
        Update potential temperature
        Update moisture conversion
*Initiate asynchronous send/receives
        Compute static small time step forcing for u-v-w-p (advection,
         mixing, gravity and others) equates to 15% of wall time
*Complete the asynchronous send/receives
        Compute small time step quantities using some of the data
        associated with the big time step asynchronous send

        Proceed to the small time step solver…
```

As depicted in Table 7, there remain two other message passing sections within ARPI, after the computation of U and V, and after the computation of W and P, all within the small time step. Since the small time step contains a limited amount of work (8 loops) the amount of message hiding typically is not large. Reviewing the timing data of the

model shows that the small time step requires approximately 25% of the wall clock time, while 8% is spent on updating U and V and 17% on updating W and P.

Implementation of the message hiding concept into the model is currently underway, and no results are available yet for this optimization. We are currently debugging the first step in the message hiding process, the installation and testing of the large time step asynchronous send/receive pair. A recent benchmark showing the scalability of ARPI is presented in Figure 11 for a Xeon 64 cluster (3.2 GHz, Infiniband). The model was configured with 150m spacing between mesh points and run for 30 seconds of a much larger two hour research simulation used in an NSF supported project. The local domain size and the number of mesh points were held constant, and the number of processors was varied to measure the scalability of the supercomputer. A perfect scaling code and computer system would be represented on the graph as a line of zero slope. All timings presented in the graph were normalized by the single processor result. The 2 processor results utilized two processors per node and show the performance degradation when both processors are accessing the main memory during the memory intensive research run (323x57x137 mesh points per processor requires 780mb of memory/processes).

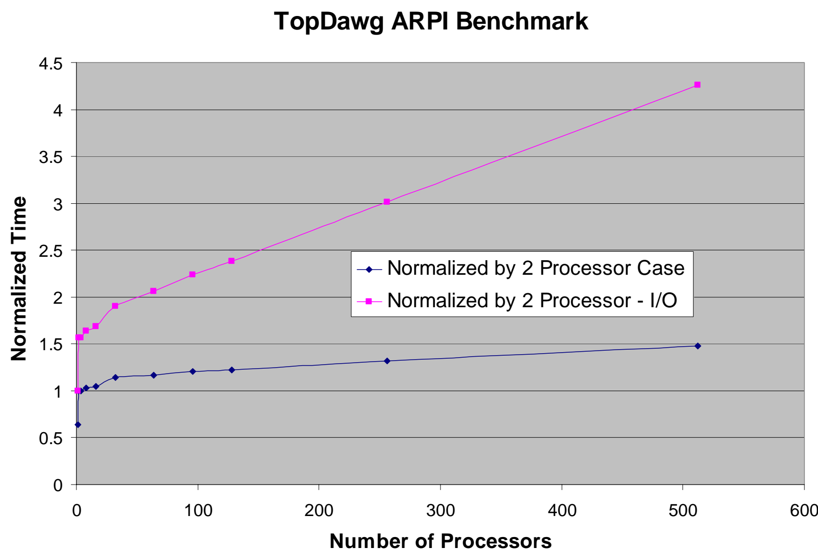

Figure 11. ARPI scalability chart depicting the time required to perform a benchmark simulation on the OSCER TopDawg Supercomputer.

5.0 Discussion and Future Work

The optimization experiences included in this work cover many years of service to the numerical weather prediction community. The authors have spent thousands of hours developing, testing, and implementing optimization plans in order to improve the

response time of a perishable severe weather forecast on available computing resources. These experiences are summarized in this paper in order to document the many different approaches to optimization that have been applied to forecast models on a wide variety of computing platforms during the past 10 years. Noticeable improvements in the model performance have been realized as a direct result of these optimization efforts. Some methods have proved more fruitful than others, such as the loop merging, removal of divides and unnecessary computations, and careful code design and message grouping for use with the message passing components. The use of tiling in the main solver of the model shows promise, but the large time step solvers are as yet mostly untiled and will require substantial labor in the near future. The message hiding technique will be very useful in the large time step part of the model, but may have limited use in the small time step section. Estimates of the manpower required to perform all of the methods included in this paper is presented in Table 8 and gives perspective to the scope of the problem of optimizing a legacy forecast model.

The author's estimate that nearly two man-years are required for optimizing the ARPS and ARPI models, which includes developing plans, techniques, implementation and testing. Clearly, 23 months is a significant amount of time focused on optimizing a forecast model, thus raising the question: could the work have been performed in less time by designing a more flexible model framework? The final section presents retrospective thoughts on how a fluid dynamics code could be designed and implemented to achieve peak performance and to increase the shelf life and quality of a given thunderstorm prediction.

Table 8. Manpower estimates for various optimization methods implemented into a legacy numerical weather prediction model. The estimates assume that the developer has an excellent working knowledge of the source code and solution methods.

Optimization Method	Implementation Time (Worker Months)
Traditional Compiler Options	2 (ongoing)
Divides	1
Loop Merging	2
Removing unnecessary computations	2
Hardware Specific (MADD's)	1
Loop Collapse	2
Tiling	6-8 (not completed)
MPI Code Structure/Restructuring	2
Message Grouping	2
Message Hiding	1 (not completed)
Total	23

5.1 Alternative Approach to Code Development

A challenge faced in the development of transport codes such as numerical weather forecasting models is that they typically are developed without taking cache optimization and parallelism fully into account from the outset. These applications, once completed, are extremely difficult and/or expensive to rewrite, as shown in Table 8. Clearly, substantial performance gains can be achieved when performance is an initial design goal rather than an afterthought.

How could a transport code be designed for performance? A common theme in recent research has been the use of multiple subdomains per processor (e.g., Charm++ [17]), asynchronous communication, and user-tunable array stencil sizes. These strategies address both cache optimization and communication cost minimization and were part of the methods described and tested by the authors on existing code.

Consider the case of an arbitrary collection of subdomains (blocks) on each processor. If each block is sufficiently small, then much computation on that subdomain will occur after the subdomain has been loaded into cache, and if the user has influence over the block size, then typically each subdomain will fit in cache. In addition, the presence of multiple subdomains per processor allows communication to and from a subdomain to be overlapped by computation on one or more other subdomains.

Finally, allowing array ghost halo regions to expand to a user-chosen depth can reduce the frequency of communication, by allowing redundant computation of a modest number of zones near the ghost halo to play the same role as communication, but at a substantially reduced time cost. For example, if the numerical stencil requires only one ghost halo zone in each direction, but the array stencil is three ghost halo zones in each direction, then three timesteps can be computed before a communication is required: after a communication, the first timestep will use the outermost ghost halo zone as its boundary data and will redundantly compute the middle and innermost ghost halo zones (which properly "live" on another processor); the second such timestep will use the middle ghost halo zone for its boundary data and will redundantly compute the innermost; the third such timestep will use the innermost with no redundant computation, and will then require communication immediately after.

Most importantly, if all of these strategies are user-tunable at runtime, then a modest amount of benchmarking – a "bake-off" to find the best subdomain and ghost halo sizes, in much the same manner as ATLAS [18,19] – can determine empirically the optimal balance of computation and communication for any architecture of interest, rather than having to rely on cumbersome or unrealistically simple performance models.

REFERENCES

[1] Pielke, R.A. and R. Carbone, 2002: Weather impacts, forecasts and policy. *Bull. Amer. Meteor. Soc.* 83, 393-403.

[2] http://www.spc.noaa.gov/exper/Spring_2005/

[3] http://www.top500.org/

[4] Xue, M., K. K. Droegemeier, and V. Wong, 2000: The Advanced Regional Prediction System (ARPS) – A multi-scale nonhydrostatic atmospheric simulation and prediction model. Part I: Model dynamics and verification. *Meteor. Atmos. Phys*., **75**, 161 – 193.

[5] Xue, M., K. K. Droegemeier, V. Wong, A. Shapiro, K. Brewster, F. Carr, D. Weber, Y. Liu, and D. Wang, 2001: The Advanced Regional Prediction System (ARPS) – A multi-scale nonhydrostatic atmospheric simulation and prediction model. Part II: Model physics and applications. *Meteor. Atmos. Phys*., **76**, 143 – 165.

[6] Skamarock, W.,C., J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, 2005: A description of the Advanced Research WRF Version 2. NCAR Technical Note #468, 83pp.

[7] http://reviews.zdnet.co.uk/hardware/processorsmemory/0,39024015,39164010,00.htm

[8] Johnson, K.W., J. Bauer, G. Riccardi, K.K. Droegemeier, and M. Xue, 1994: Distributed Processing of a Regional Prediction Model, *MWR*, 122, p.2558-2572.

[9] Weber, D. B., 1997: An investigation of the diurnal variability of the Central Colorado Downslope Windstorm, Ph.D. Dissertation. The University of Oklahoma, 242pp. [Available from School of Meteorology, University of Oklahoma, 100 E. Boyd, Norman, OK 73019 and online at: www.caps.ou.edu/~dweber]

[10] Arakawa, A., 1966: Computational design for long term integration of the equations of motion: Two-dimensional incompressible flow. Part I. J. Comput. Phys. 1, 119-143.

[11] *Intel Pentium4 and Pentium Xeon Processor Optimization: Reference Manual*, Intel Corp, 1999-2002.

[12] Dowd, K, and C. Severance, High Performance Computing, 2nd ed. O'Reilly & Associates Inc., 1993.

[13] Weber, D., K.K., Droegemeier, K. Brewster, H.-D. Yoo, J. Romo, 2001: Development of the Advanced Regional Prediction System for the Korea Meteorological Administration, Project TAKE Final Report, 49pp.

[14] Irogoin F. and R. Triolet, 1989: Supernode Partitioning. Proceeding of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988).

[15] http://bebop.cs.berkeley.edu/pubs/vuduc2003-ata-bounds.pdf

[16] Weber, D.B., 2003: The Current State of Numerical Weather Prediction on Cluster Technology - What is Needed to Break the 25% Efficiency Barrier? ClusterWorld Conference and Expo, San Jose, CA, June 2003

[17] http://charm.cs.uiuc.edu/

[18] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software, 1998. Winner, best paper in the systems category, SC98: High Performance Networking and Computing.

[19] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing,* 1999. CD-ROM Proceedings.