SINTEF REPORT

SINTEF ICT

Address: NO-7465 Trondheim, NORWAY
Location: Alfred Getz vei 1, NTNU, NO-7491 Trondheim
Telephone: +47 73 59 30 48
Fax: +47 73 59 29 71

Enterprise No. NO 948 007 029 MVA

TITLE

Parallelization of some scientific codes using profiling tools and OpenMP

AUTHOR(S)

Bernt G. Galtrud Karstein Sørli Torbjørn Utnes

CLIENT(S)

SINTEF ICT Applied Mathematics

REPORT NO.	CLASSIFICATION	CLIENTS REF.	
SINTEF A11362	Open	Roger Bjørgan	
CLASS. THIS PAGE	**ISBN**	**PROJECT NO.**	**NO. OF PAGES/APPENDICES**
Open	9788214044379	901110.21	53/3
ELECTRONIC FILE CODE		**PROJECT MANAGER (NAME, SIGN.)**	**CHECKED BY (NAME, SIGN.)**
		Karstein Sørli	Torbjørn Utnes
FILE CODE	**DATE**	**APPROVED BY (NAME, POSITION, SIGN.)**	
	2009-03-19	for Roger Bjørgan, Research Director	

ABSTRACT

The purpose of this report is to exchange our experience with parallelizing existing scientific codes by utilizing profiling tools and the OpenMP application programming interface (API) for multi-platform shared-memory parallel programming in C/C++ and Fortran.

Profiling is an excellent tool to get an indication on which parts of the program to concentrate in order to parallelize scientific codes. In general it is necessary to get more detailed information regarding the relative importance of different sub-blocks inside an interesting subroutine. This information might be obtained by the use of manually inserted timers inside the subroutines. In order to get more reliable results *cpu-timers* should be used instead of *wall-clock-timers*.

The main goal was to parallelize the *Simra* CFD-code as much as possible. Some initial work on less complex and smaller programs did undoubtedly lead to better results for the Simra code. At the moment about 65% of the program has been parallelized. When using 16 cores on the Njord supercomputer, the global speedup resulting from this work is between 2.2 and 2.3 depending on the problem size.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	Parallel computing	Parallelle beregninger
GROUP 2	Partial differential equations	Partielle differensialligninger
SELECTED BY AUTHOR	OpenMP	OpenMP

Parallelization of some scientific codes using profiling tools and OpenMP

Bernt G. Galtrud Karstein Sørli Torbjørn Utnes

March 31, 2009

1 Introduction

We are witnessing the arrival of significant multiprocessing capabilities on PCs and small-group compute servers. Single-user machines with multiple CPUs have been available for several years, but they remain uncommon and have typically only doubled or quadrupled, the number of processors. That is now changing, as *multi-core* chip designs begin to make it economical for typical users to access 8, 16, or even more processor cores. However, it will not be easy to utilize all of this power for applications on the desktop. Most desktop software so far has been developed for single-threaded execution on one CPU, and writing multi-threaded software has traditionally been difficult.

The purpose of this report is to exchange our experience with parallelizing existing scientific codes by utilizing profiling tools and the OpenMP application programming interface (API) for multi-platform shared-memory parallel programming in C/C++ and Fortran.

2 Computing environment

The work described in this report was done on two different computer systems. Most of the development and some initial testing were done on a quite old desktop PC. Some testing was done on a supercomputer. Both systems got advantages and disadvantages.

The computing power of a supercomputer is of course much higher than a desktop PC. It would seem nice to do all the work on the big machine. However, the number of CPU hours available for the project was limited. The supercomputer got another disadvantage. Access to the machine is through a batch system. A *job* is committed to a queue. And the job is executed when enough resources are available. On a desktop PC on the other hand, full and unlimited access can be assured at no extra cost.

2.1 Desktop PC

The PC used was a Dell Precision Workstation 530 MT. The complete manual was lost long time ago, and the following description of the system was the best we managed to get.

It is a dual processor system consisting of two Intel XEON Processors at 1.5GHz. The machine got two 256MB RAM units. The amount of Level 1 cache is unknown, but each processor got an individual Level 2 cache of 256kB.

The operating system installed was Ubuntu Linux version 8.04. (Released in April 2008.) The Fortran compiler used was GNU's gfortran version 4.2.3. The OpenMP version was 2.0

2.2 Supercomputer - Njord

Njord is the current (primo 2009) supercomputer of the Norwegian University of Science and Technology (NTNU) in Trondheim. Njord is a distributed shared-memory system, consisting of IBM p575+ nodes interconnected with a high-bandwidth low-latency switch network [10].

Njord consist of 65 nodes. There are 59 compute nodes, 4 I/O nodes and 2 login nodes. Each compute node consist of 8 dual-core power 5+ processors with 1.9 GHz clock speed. 55 of the compute nodes got 32GB shared memory, and 4 nodes got 128GB RAM.

Each dual-core processor got a common Level 3 cache of 36 MB. Each core got an individual off-chip Level 2 cache of 1.92 MB, and an individual on-chip Level 1 cache of 64 kB. Each node got a total memory band width of 24 GB/s, respectively 14.4-16GB/s read and 7.2-8GB/s write. The address mode is by default 64-bit, but 32-bit is also possible.

The operating system running on Njord is IBM's Unix flavor, AIX v 5.3. And the Fortran compiler used was IBM's xlf90, together with OpenMP v 2.0.

3 Profiling tools

Profiling is a performance analysis of a program as it runs [11]. The main behavior recorded is the frequency and duration of function calls. The output from a profiler is usually either a *trace* or a *profile*. A *trace* is a sequential list of events (calls) as they occur when the program is run. A *profile* is a statistical summary of the events occurring in the program. The profile is most often given either as a *flat profile* or a *call graph*.

The *flat profile* consists of a ranking of the subroutines, constituting the program, based on each subroutine's *self-time*, i.e the time spent in that subroutine alone excluding the time spent in eventual child routines called by that specific subroutine. Among other things the percentage of the total execution time used by each subroutine is also displayed.

A *call graph* gives in addition for each function information concerning which other functions called that specific function and how many times they did it, and which functions that are called from the considered function. In sequential programs a profile is most often enough to get a good overview of the program. In parallel programs on the other hand a trace might be interesting to identify idle times for instance.

There are a wide variety of profiler tools available. A profile might be established using different techniques. Some are very accurate, but induces a very important slow down. Others are a bit less accurate but allows the program to run at nearly full speed.

In this project the GNU tool *gprof* was chosen for the desktop PC. While IBM's very similar tool called *Xprofiler* was used on the Njord supercomputer. Both depend on sampling at regular time intervals, observing which program line is being executed at that time. Some error must be expected but it is very fast. Xprofiler can be used to profile programs containing both sequential and parallel blocks. gprof on the other hand only handles sequential code.

Other tools were tested as well, in particular *Valgrind*. Valgrind is very nice looking and easy to use with an intuitive GUI. But on the kind of problems investigated in this project, the slow down is terrible. (Execution time is at least multiplied by a factor ten.)

Several good tutorials for using gprof can be found on the Internet. (See for instance the one published by Jay Fenlason and Richard Stallman [12].) In order to use gprof three steps are necessary. First compile and link the source code with profiling enabled. This is done by giving the *-pg* flag to the compiler. Second, run the program as usual with eventual arguments and I/0-files. The expected output from the program will be produced, but also a file called *gmon.out*. This file contains the raw profiling data. The last step of the process is to run the gprof program with the gmon.out file as an argument. This interprets the raw data and prints both a flat profile and a call graph to a textfile. There are also printed some information about how to interpret the profiling information. These explanations are quite good and permit to easily comprehend the profile.

Several examples of profiles can be found in the appendix. The profiles given are of the programs investigated in this project. These examples will be treated more carefully at a later stage in this report.

4 Initial work on basic problems

In order to get a hands-on experience with OpenMP as fast as possible we started with two simpler problems, i.e. *linear systems of equations* and *a Dirichlet type Poisson equation problem*. These problems are important parts of all CFD solvers.

4.1 System of linear equations

We parallelize a Fortran90 program which re-implements the SGEFA/SGESL linear algebra routines from LINPACK for use with OpenMP ([1],[2],[3],[4], [5]). The SGEFA algorithm factors a real matrix by gaussian elimination.

We compare methods of solving a system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

The original version of this code comes from John Burkardt and is published at his web-site [9].

4.1.1 SGEFA-versions

Burkardt's code solves the problem in five slightly different manners, one after the other. The first one is a standard sequential version of the SGEFA-algorithm. The second one is a modification of the first one taking advantage of Fortran's possibilities of updating a subset of an array by just one command line. In this case the update is done column-vice. (The range of array indices in each dimension must be specified. E.g $a(k+1:n,k) = -a(k+1:n,k)/a(k,k)$) It is tempting to call this *a vectorization of the do-loop.* In this text the expression will be used, even though it might refer to other things elsewhere. The third one is a parallelized version (OpenMP) of the second one. The fourth one also exploits vectorization of the do-loop, but the update is done in a row-vice order. The last one is a parallel version of the fourth one.

4.1.2 Restructuring of the code

The five different versions described above were implemented as five different (and quite long) subroutines despite the fact that only a few lines differ between the different versions (A.1). As a consequence there was a high amount of redundant code. A modified and synthesized version was made which consist of only one subroutine (A.2). Two parameters were added. The first one describing the sgefa-method, normal, columnoriented or roworiented. The second one telling if it was to bee run sequentially or in parallel. Then at execution time by the means of if-tests on the two parameters, the appropriate lines of code were chosen.

4.1.3 OpenMP work

The *sgefa_c_omp* and *sgefa_r_omp* subroutines were already partially parallelized with the use of an `!$omp parallel workshare` and an `!$omp parallel do` OpenMP directive. With this method a new parallel region is created twice for each iteration of the loop k over the columns. The creation of a parallel region is a rather expensive operation. In one way or another there will be necessary to synchronize some information between the threads. The master thread automatically get extra coordination work by setting up (and ultimately destroying) the threads.

It is desirable to create parallel regions as seldom as possible. The first obvious thing to do in this case, is to move the creation of the parallel region out of the loop, i.e. using the `!$omp parallel` directive. (See listing line number 73 in the A.2 appendix.) Hence the costly operation is done only once. At the respective code blocks the work-sharing constructs `!$omp workshare` and `!$omp do` are used instead of their combined parallel-work-sharing counterparts. The unparallelized regions inside the loop were protected by the `!$omp single` directive. In this way only the first thread arriving at these parts, in each iteration, executes the given code. And since the `!$omp end single` directive contains an implicit synchronization barrier the correctness of the execution is guaranteed, at the potential cost of some extra waiting time.

A new section of the subroutine was also parallelized. The section in question is the one labeled *Interchange rows K and L* see the two code samples included below 1 and 2. The OpenMP construct chosen was the `!$omp workshare` directive. In addition an index-vectorization technique similar to the one explained in 4.1.1 was added to the block of code.

Listing 1: Original version

```
! Interchange rows K and L.
  if ( l /= k ) then
    do j = k, n
      t      = a(l,j)
      a(l,j) = a(k,j)
      a(k,j) = t
    end do
  end if
```

Listing 2: New version

```
! Interchange rows K and L.
  if ( l /= k ) then
    !$omp workshare
    temp(k:n) = a(l,k:n)
    a(l,k:n) = a(k,k:n)
    a(k,k:n) = temp(k:n)
    !$omp end workshare
  end if
```

4.1.4 Scheduling of do-loops

Let us consider a *do-loop* with N iterations. If this structure is to be parallelized on P processors using the `!$omp do` directive, the default way of sharing the work is to give an equal number (N/P) consecutive iterations to each processor. This is not always the best method. If for instance the amount of work is not constant between the iterations, one might witness an important difference in the workload on some processors. To get around this problem, OpenMP allows the programmer to distribute the iterations in different ways. A rule for distributing the iterations is called a schedule. See for instance Miguel Hermanns compendium [13] for an introduction to the other possibilities for scheduling.

Some initial and not very thorough experimentation was done concerning scheduling of the do-loop in the parallelized sgefa code. The testing was mainly done on the desktop PC. There was not recorded any

major differences in runtime for different scheduling types with the actual problem sizes ($N = 10, 100, 1000$). It is possible that more thorough testing might have shown a difference in run time. The tests were performed at an early stage of the process of getting acquainted with OpenMP, and the understanding of how things work was not as good as they are to day.

In retrospect, a better understanding of the theoretical differences has been obtained. Inside an iteration of the othermost loop (of index k), the workload is perfectly balanced between the iterations of the inner loop of index j. Therefor a standard static schedule seems to be the best choice. The only problem is that for each iteration of k, less and less work is distributed in the inner loop j. Which means that sooner or later it will be better to hand out the work to fewer processors. One might try the `static` schedule with a minimal `chunk` size. The latter should be determined experimentally.

Ideally it would be desirable to be able to use less processors after a wile. With the OpenMP version 2.0 this is not possible. It is only possible to choose the number of threads upon the creation of the parallel region. After this choice is made all threads are stuck with that parallel region upon destruction of all the threads at the same time. It seems though that v 3.0 provides some tools bether suited for this kind of idea.

4.1.5 RNG

The A matrix in this problem was a full matrix. The subroutine *matgen* fills this matrix with (pseudo) random numbers. The algorithm for doing this was simple and fast. For small problem sizes the method worked ok, but for bigger problems it broke down. (Larger then 1000 unknowns.) The reason for this was that a simple modulo generator without seed and with to short period was used. Sooner or later the start of a new period will fall in the first column of a line. Since the generator is pseudo random, the same sequence of numbers comes up each period. This line will then be identical to the first line of the matrix. Two identical lines in the system causes a breakdown in the Gaussian elimination.

Some work was done to improve the RNG. Larger sets were generated correctly, but at a tremendous performance cost. The modified routine does not have the limitation of max 1000 unknowns. But it is 100 times slower. Originally the RNG routine was negligible compared to the forward elimination process from a performance point of view. The new version became equally important.

4.2 The Poisson equation

We parallelize a Fortran90 program which solves the 2D Poisson equation using the finite element method [6],[7],[8].

The computational region is a rectangle, with homogeneous Dirichlet boundary conditions applied along the boundary. The state variable $u(x, y)$ is then constrained by:

$$\begin{aligned} -(u_{xx} + u_{yy}) &= f(x,y) \text{ in the rectangle} \\ u(x,y) &= g(x,y) \text{ on the rectangle boundary} \end{aligned}$$

The computational region is first covered with an $NX \cdot NY$ rectangular array of points, creating $(NX-1) \cdot (NY-1)$ sub-rectangles. Each sub-rectangle is divided into two triangles, creating a total of $2 \cdot (NX-1) \cdot (NY-1)$ geometric "elements". Because quadratic basis functions are to be used, each triangle will be associated not only with the three corner nodes that defined it, but with three extra mid-side nodes. If we include these additional nodes, there are now a total of $(2 \cdot NX - 1) \cdot (2 \cdot NY - 1)$ nodes in the region.

The original version of this code comes from Janet Peterson by was modified by John Burkardt and published at his web-site [9].

4.2.1 Profiling of the code

Burkardt's original version was analyzed with the profiling tool *gprof.* A sample of the output listing (3) is displayed below. The entire profile, along with an explication of the different pieces of information, is available in the appendix B.1. The table below announce the three most time consuming subroutines in the program. As we can see from the profiler listing, one subroutine *dgb_fa* completely dominates the run time by using 96.37% of the execution time.

Listing 3: Flat profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
96.37     33.47    33.47        1   33.47    33.47  dgb_fa_
 1.47     33.98     0.51  1942488    0.00     0.00  qbf_
 1.21     34.40     0.42        1    0.42     0.74  assemble_
```

An extract of the call graph (4)is also presented in this section. This piece shows that the subroutine is only called once (by the *main* routine), and *dgb_fa* does not call any other subroutines itself. This fact is very promessing when it comes to parallelization. Because it is in general easier to get speedup from a larger section of code which is run a few times, then from a short section which is run many times. On the other side, this is not a sufficient condition for large gains. It all depends on the structure of the code how well it parallelizes.

Listing 4: Call graph

index	% time	self	children	called	name
		33.47	0.00	1/1	MAIN__ [1]
[3]	96.4	33.47	0.00	1	dgb_fa_ [3]

4.2.2 What is going on in the dgb_fa routine?

The profiling tells us that the dgb_fa subroutine is by far the most interesting routine from a performance point of view. The routine is included in the appendix B.2.

The routine performs a Gaussian elimination on a banded matrix. The storage format used for the matrices is called *dgb*, and handles M by$N$ banded matrices with lower bandwidth ML and upper bandwidth MU. During the Gaussian elimination nonzero entries might be generated on ML extra superdiagonals. This must be taken account of in the storage format, and the total bandwidth is then $BW = 2*ML+MU+1$. The original diagonals are collapsed downward, so that diagonals become rows and columns are preserved.

4.2.3 OpenMP parallelization

Burkardt's original version of the *dgb_fa* subroutine is given in the appendix B.2. The parallelized version is also given in its' totality in the appendix B.3.

To get a more detailed picture of the dgb_fa subroutine, manual timers were inserted into the code. Only the two most time consuming blocks of code in the routine are presented here. The timing results for a sequence of different problem sizes of $NX = NY = 10, 20, 40, 80, 160, 320$ run on $P = 1, 2, 4, 8, 16$ processors can be found in the appendix B.4.

These two most important blocks are labeled *TzeroOut* and *TloopK*. The first block zeros out the extra fill inn band of the matrix. The second block performs the Gaussian elimination on the banded matrix. This last one is implemented as a *do-loop* that loops over the columns k of the matrix. Inside this loop there are some negligible parts, and only one important part when it comes to consuming time. The important subblock of code performes the row elimination, see the column *Tinner*. In the sequential case, TloopK is in general about 100 times more important then the TzeroOut. From a performance part of view it is not much to gain from parallelizing the last one. Never the less, it was done just for the sake of training.

Tinner A small sample of the routine containing the subblock "Tinner" is presented both for the sequential case (listing 5) and the parallel case (listing 6).

As it is a gaussian elimination that is going on, it is not possible to parallelize the othermost do-loop. But as mentioned in the sgefa-case, the parallelization process can be divided in two steps. First create the parallel region, then share the work. Even though the othermost do-loop is not parallelizable per-se, the parallel region can be created outside of this loop and then create worksharing constructs inside the loop.

The inner do-loop over the variable j can be parallelized with an `!$omp do` directive. Notice that the vectorized update of the a-matrix conceals a sort of an extra implied do-loop. The update of the a matrix in one iteration of j does not depend on any value of a from a different iteration. There are a couple of index variables inside the j-loop, called l and mm. These take care of some swapping of values in a. In the sequential version these are just decremented by one for each iteration. In order to parallelize correctly this feature, l and mm must be expressed explicitly as a function of j instead as a simple decrement. If not the iterations would have to be executed in increasing order. It was quite trivial to do this.

Problem size	$P=1$	$P=2$	$P=4$	$P=8$	$P=16$
$NX=10$	1.00	0.91	0.87	0.62	0.55
$NX=20$	1.00	1.39	1.88	1.63	1.69
$NX=40$	1.00	1.76	2.93	3.62	4.41
$NX=80$	1.00	2.09	3.93	6.44	9.56
$NX=160$	1.00	2.18	4.94	9.52	16.34
$NX=320$	1.00	2.09	4.30	8.87	16.19

Table 1: Speedup in the `Tinner` subblock of the `dgb_fa` routine, for different problem sizes $NX = NY$ and numbers of processors P used.

Listing 5: Tinner sequential

```
!
  do k = 1, n-1    !loop over columns

    execute introblock

!   Row elimination with
!   column indexing.
    ju = max(ju , mu + pivot(k))
    ju = min(ju , n)
    mm = m



    do j = k+1, ju
      l = l - 1
      mm = mm - 1

      if ( l /= mm ) then
        temp    = a(l , j)
        a(l , j)  = a(mm, j)
        a(mm, j) = temp
      end if

      a(mm+1 : mm+lm, j) = a(mm+1 : mm+lm, j) &
              & + a(mm, j) * a(m+1 : m+lm, k)

    end do

  end do
!
```

Listing 6: Tinner parallel

```
!$omp parallel
do k = 1, n-1 !loop over columns
   !$omp single
   execute introblock

!  Row elimination with
!  column indexing.
   ju = max(ju , mu + pivot(k))
   ju = min(ju , n)
   mm = m
   !$omp end single

   !$omp do private(ll ,mm,temp)
   do j = k+1, ju
     ll = l + k - j
     mm = m + k - j

     if ( ll /= mm ) then
       temp    = a(ll , j)
       a(ll , j)  = a(mm, j)
       a(mm, j) = temp
     end if

     a(mm+1 : mm+lm, j) = a(mm+1 : mm+lm, j) &
             & + a(mm, j) * a(m+1 : m+lm, k)
   end do
   !$omp end do

end do
!$omp end parallel
```

TzeroOut In the sequential version the zero out process is done in two stages (listing 7). The first one before the elimination process starts and the second one after. In the parallel version everything is done at the same stage (listing 8). An analysis of the indices used in the elimination loop showed that this was a safe approach. Output of the residuals in the sequential and in the parallel case supported it as well.

Listing 7: TzeroOut sequential

```
! Zero out the initial fill in columns.
  j0 = mu + 2
  j1 = min ( n, m ) - 1
  do jz = j0 , j1
    i0 = m + 1 - jz
    a(i0:ml, jz) = 0.0D+00
  end do
  jz = j1
  ju = 0

  do k = 1, n-1
!
! Zero out the next fill-in column.
!
    jz = jz + 1
    if ( jz <= n ) then
      a(1:ml, jz) = 0.0D+00
    end if
! ...more work...
  end do
```

Listing 8: TzeroOut parallel

```
j0 = mu + 2
j1 = min ( n, m ) - 1
!Last initial fill in column.

!$omp parallel do, private(i0)
do jz = j0 , n
   i0 = m + 1 - jz
   i0 = max(i0 ,1)
   a(i0:ml, jz) = 0.0D+00
end do
!$omp end parallel do
```

Problem size	$P=1$	$P=2$	$P=4$	$P=8$	$P=16$
$NX=10$	1.00	0.35	0.47	0.27	0.16
$NX=20$	1.00	0.83	0.98	0.97	0.79
$NX=40$	1.00	1.34	2.69	3.22	3.23
$NX=80$	1.00	2.04	3.93	7.13	10.77
$NX=160$	1.00	1.95	3.21	5.85	6.12
$NX=320$	1.00	1.66	3.11	4.67	5.16

Table 2: Speedup in the `TzeroOut` subblock of the `dgb_fa` routine, for different problem sizes $NX = NY$ and numbers of processors P used.

Results The complete runtime output is given in the appendix B.4. The output shows timing results, $L2$ error and speedup for different problemsizes and different number of processors. A synthesis of all these results facilitates the interpretation. The tables 1 and 2 gives the calculated speedups for respectively the `Tinner` and `TzeroOut` subblocks, while the table 3 gives the speedup for the entire program.

Problem size	$P=1$	$P=2$	$P=4$	$P=8$	$P=16$
$NX=10$	1.00	1.07	1.03	0.98	0.97
$NX=20$	1.00	1.17	1.16	1.08	1.16
$NX=40$	1.00	1.48	1.86	1.95	2.07
$NX=80$	1.00	1.89	2.89	3.73	4.39
$NX=160$	1.00	2.11	4.29	7.02	9.77
$NX=320$	1.00	2.08	4.13	7.99	13.21

Table 3: Global speedup in the Poisson equation solver, for different problem sizes $NX = NY$ and numbers of processors P used.

5 Parallelizing a CFD code (SIMRA)

SIMRA is a computer code for the simulation of incompressible, inelastic and turbulent flow developed by Torbjørn Utnes at SINTEF. The code solves for averaged (filtered) values of velocity ($\mathbf{u}$), pressure (p), temperature (θ), turbulent kinetic energy (k) and dissipation of energy (ϵ), using a set of conservation and turbulence modeling (closure) equations.

CONSERVATION OF MOMENTUM (MOMENTUM EQUATIONS):

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = \nabla \cdot \mathbf{R} - \nabla(\frac{p'}{\rho}) + \frac{\theta'}{\theta}\mathbf{g}$$

where $\theta = \theta_s + \theta'$ and $p = p_s + p'$. With given θ_s can (p_s, ρ_s) be computed from a hydrostatic relation and equation-of-state.

CONSERVATION OF MASS (CONTINUITY EQUATION) (inelastic):

$$\nabla \cdot (\rho \mathbf{u}) = 0$$

CONSERVATION OF ENERGY (EQUATION OF POTENTIAL TEMPERATURE):

$$\frac{\partial \theta}{\partial t} + (\mathbf{u} \cdot \nabla)\theta = \nabla \cdot \left(\frac{\nu_T}{\sigma_T}\nabla\theta\right) + S_\theta, \quad \theta = T(\frac{p_0}{p})^{R/C_p}$$

EQUATIONS OF TURBULENCE:

$$\frac{\partial k}{\partial t} + (\mathbf{u} \cdot \nabla)k = \nabla \cdot (\nu_T \nabla k) + P_k + G_\theta - \epsilon$$

$$\frac{\partial \epsilon}{\partial t} + (\mathbf{u} \cdot \nabla)\epsilon = \nabla \cdot \left(\frac{\nu_T}{\sigma_T}\nabla\epsilon\right) + (C_1 P_k + C_3 G_\theta)\frac{\epsilon}{k} - C_2\frac{\epsilon^2}{k}$$

$$\nu_T = C_\mu \frac{k^2}{\epsilon}$$

The latter equations being equations of *turbulent kinetic energy* and *dissipation of turbulent kinetic energy* and the model of *eddy viscosity*, respectively.

EQUATIONS OF REYNOLDS STRESSES:

$$R_{ij} = \nu_T \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{2}{3}k\delta_{ij}$$

For a detailed description of the FEM-based approximation and solution procedure used in SIMRA it is referred to [15].

5.1 Profiling

The main goal of this project was to do some parallelization work on Simra by using the OpenMP-standard. The previous chapter describes some initial work on basic problems. The work on these easier cases was carried out in order to get a bether understanding of the OpenMP-framework and to get the necessary training in using it efficiently.

A profile of Simra in its serial version was obtained with the program gprof. The aim was twofold. The profile is a good way to get a first view of how the program is organized by the call graph. The profile also points out the percentage of the total runtime used by the different subroutines.

A simple input file was given to Simra. The input data describes a so called *Hunt mountain*. Simra was used to predict the wether around this mountain. The first test run used $n = 100$ time steps of $dT = 0.05s$. The data set contained $npoin = 84035$ points in the mesh.

5.1.1 Flat profile

The following extract of the flat profile displays all the subroutines with a "self-time" of at least 1% of the total runtime of the simulation.

Listing 9: Flat profile of Simra

```
ngranularity: Each sample hit covers 4 bytes. Time: 125.46 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.8     32.35    32.35     3726     8.68     8.68  .__solver_NMOD_matvec[8]
 15.6     51.88    19.53     1701    11.48    11.48  .__solver_NMOD_pssor[10]
 12.4     67.38    15.50      100   155.00   350.30  .__solver_NMOD_pcg[6]
  9.8     79.64    12.26      100   122.60   271.34  .__turbke_NMOD_turb2[9]
  9.1     91.02    11.38      100   113.80   335.38  .__momentum_NMOD_veloc[7]
  6.9     99.71     8.69  7833600     0.00     0.00  .__turbke_NMOD_elmat_k[13]
  5.9    107.16     7.45  7833600     0.00     0.00  .__momentum_NMOD_elmat_u[14]
  3.2    111.16     4.00      500     8.00    73.86  .__solver_NMOD_bicgstab2[4]
  2.7    114.50     3.34      100    33.40   120.30  .__turbke_NMOD_assemk[11]
  2.0    117.02     2.52      100    25.20    99.70  .__momentum_NMOD_assemu[12]
  1.1    118.34     1.32                             .__mcount[15]
  1.0    119.58     1.24      100    12.40   362.70  .__pressure_NMOD_dpres[5]
```

It is important to point out that the percentages spend by each routine depend on both the size of the data set, and the number of time steps. Which means that the relative importance of the different routines might change with different input. The flat profile is therefore not an absolute quantity but should be taken as a hint of relative performance.

5.1.2 Call graph

An extract of the call graph can be found in the appendix C.1.1. The *total time* spend within a routine is the self time plus the time spend in the descendents of the routine. Only the routines with a total time of at least 1% of the runtime spend in the entire program are included in the presented listing.

The call graph given by the profiler is text-based. Each routine is given a node number. Each node got a list of its parents, and another list of all its descendants. Much information is given by this approach, but a graph is best visualized when the nodes and edges are drawn. There are several tools to draw graphs from the text-based call graph. The figure 1 was obtained with the use of the program *Kprof* together with the program *dot*.

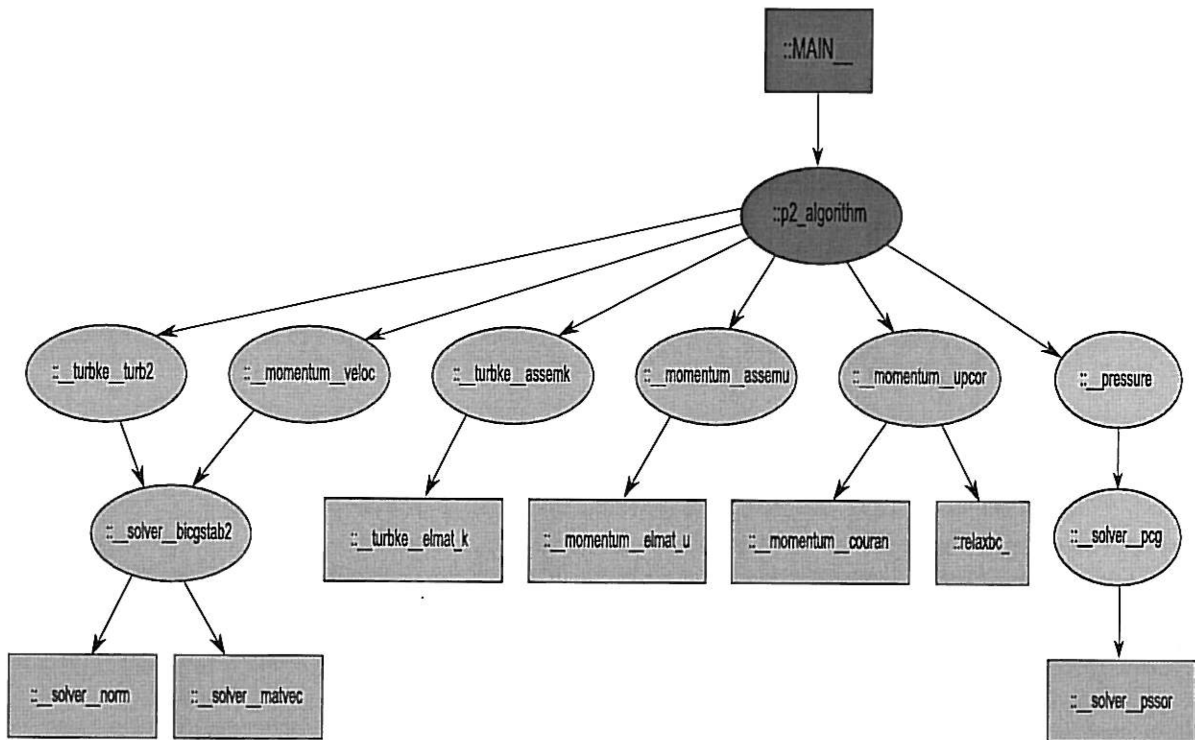

Figure 1: Part of the call graph for Simra. All the mayor routines from a performance point of view are included. A couple of not so important routines are included as well because of limitations in the graph drawing program.

5.1.3 Decisions made from the profiling

The flat profile together with the call graph tell us that the two most time consuming functions, `matvec` (25.8%) and `pssor` (15.6%), do not have any descendants. Consequently they are a obvious place to start the parallelization process as they together spend 41.4% of the run time.

In descending order, the next three most important routines are `pcg`, `turb2` and `veloc`. Together they spend 31.3% of the run time. It is expected that the parallelization might be harder because they got descendants. Since nesting of parallel regions is not mastered at the moment, the code lines containing calls to their descendants could not be parallelized. Therefore it is expected that the parallelization of the three mentioned routines might also be less effective then the first two routines. On the other hand, all of the three routines got a much higher self-time per call then the first two. Which might indicate that the potential for parallelization is quite high afterall.

The profiler results just tell us where the potential might be the biggest for gains by parallelization. It does not tell anything about how difficult it might be to actually obtain that gain. (Or if it is possible at all to parallelize.)

5.2 Parallelizing

5.2.1 Method

The profilling gave us information on which subroutines that would be most interesting to parallelize. To get a more detailed view, of the respective importance of different subblocks of code inside the subroutines, manual timers were added inside the subroutines.

A wrapper program was made, that permits Simra to be run multiple times in a row on the same problem but with different number of processors P used. The wrapper took care of handling the timeres, calculated speedup for each case and printed a summary of the results. The same data set were used with three different resolutions, having respectively $nPts = 339521$, $nPts = 173225$ and $nPts = 84035$ number of points in the mesh. Simra were run with all the three data sets for 200 time steps with a time step resolution $dT = 0.05s$. The run time output from the detailed timers can be found in the appendix C.2.

The approach chosen was to start with *loop level parallelization* based on the results from the detailed timers. The most time consuming loops were considered first. Among them, the most easy ones to parallelize were chosen as a starting point. When parallelizing a loop, it is usually best to start by getting it right,

and then afterwards fast. As already stated, creating a parallel region is a costly operation. When several loops are nested, it is preferable to create the parallel region outside the outhermost loop as long as it is algorithmically possible. In practice it is often easier to to start the parallelization process on the innermost loop. Once that loop is correctly parallelized, the second one can be attacked.

Limitations by timer type The timers used were of the type *wall clock time*. The time measured is just the clock time that has elapsed since the last time the clock was watched. In practice when a program is run we want to minimize the wall clock time, i.e we want the program to run as fast as possible.

When analyzing programs from a performance point of view it is usually better to use a timer that measures *cpu time*. These types of timeres counts the number of cpu-cycles used by a program. Unless the program got exclusive and absolute priority to the system resources, cpu-time is more accurate. On a normal desktop PC, if a program is analyzed with wall clock timers, and for instance the mail client starts downloading incoming mails the performance of the program will seem to drop. The observed performance drop down is not real, it is just a question about reduced access to system resources. On the Njord supercomputer similar problems were encountered, and some of the timing results are inaccurate because of this. If all computing nodes are busy when the national weather forcast calculations are submitted some of the other users are temporarily put aside. The larger the data set, the longer time is needed to run the program. And hence a higher probability of being put on the sideline for a while by the wether forcast.

In the OpenMP standard version 2.0 only a wall clock timer is included. Fortran 90 got the same problem. In Fortran 95 on the other hand cpu time might be measured by an intrinsic function. OpenMP will probably get the possibility of measuring cpu time at a later stage.

Limitations by time and money During this project we had access to an account on the Njord supercomputer. The number of available cpu-hours were however limited. A full run of the three data sets takes 2-3 hours of computing time. As a result of these two limitations only two complete runs were done with the so far final version. The problem with wall clock timers and lack of priority was present in most of the runs. Only one run of each data set were judged to be of acceptable quality. It would be advantageous to run more test runs and calculate some mean values.

5.2.2 The matvec-routine

The profiling told us that the `matvec` routine used 25.8% of the run time in the initial case. The routine performs a standard matrix-vector product. It is very easy and safe to parallelize. The original version of the code is stated in the listing 10 . The parallel version is stated in the listing 11

Listing 10: Original version

```
subroutine matvec &
        &(n,mxnz,amat,ia,ja,x,y)
  integer :: n,mxnz
  integer :: ia(n+1),ja(mxnz)
  real    :: amat(mxnz),x(n),y(n)

  y=0.0

  DO i=1,n
     do k=ia(i),ia(i+1)-1
        y(i)=y(i)+amat(k)*x(ja(k))
     enddo
  END DO

END subroutine matvec
```

Listing 11: Parallel version

```
subroutine matvec &
        &(n,mxnz,amat,ia,ja,x,y)
  integer :: n,mxnz
  integer :: ia(n+1),ja(mxnz)
  real    :: amat(mxnz),x(n),y(n)

  y=0.0
  !$omp parallel do
  DO i=1,n
     do k=ia(i),ia(i+1)-1
        y(i)=y(i)+amat(k)*x(ja(k))
     enddo
  END DO
  !$omp end parallel do
END subroutine matvec
```

Results The speedup Sp of the `matvec` routine was quite good as shown by the table 4.

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.97	3.26	6.40	11.95
$nPts = 173225$	1.00	1.96	3.44	6.81	12.57
$nPts = 84035$	1.00	1.78	3.51	6.73	11.97

Table 4: Speedup in the `matvec` routine for different problem sizes $nPts$ and numbers of processors nP used.

5.2.3 The pssor-routine

The profiling told us that the `pssor` routine used 15.6% of the run time in the initial case. The routine is an implementation of a *preconditioner* that performs a *symmetric successive over relaxation* on the input matrix. It turned out to be much harder to parallelize than the `matvec` routine. At the present moment we have not been able to implement an efficient way of parallelizing the routine.

The routine consists of two blocks of code. The first one does a forward substitution, the second one a backward substitution. Both are implemented as two nested do-loops. The difficulty of parallelizing these nested loops arise from the fact that the the iterations of the outheremost loop is dependant of a certain number of the previous iterations. A satisfying speedup demands that the iterations of the outhermost loop can be safely distribute onto the different processors. It seems that this implies that some change must be made to the nested loops. This should really be considered because the potential gain is quite large.

5.2.4 The pcg-routine

The profiling told us that the `pcg` routine used 15.6% of the run time in the initial case. The routine is an implementation of a *preconditioned conjugated gradient* method to solve a linear system $Ax = b$.

The routine consist of two blocks (B_1 and B_2) of code. Their respective percentages of the run time depends on the problem size as stated by the table below:

Case	B_1	B_2
Small	5.7%	94.3%
Medium	4.4%	95.6%
Large	3.3%	96.7%

The first block calculates the initial residual. The second block is a while-loop which is repeated until the residual is less then the desired precision, $r < eps$, or until the maximum number of iterations is reached.

The while-loop can be further decomposed into three mayor subblocks of code B_{2a}, B_{2b} and B_{2c}. The table below gives the respective percentage of the total runtime of the `pcg` routine in function of the problem size

The B_{2a} subblock consist only of the call to the `pssor` routine, and is therefor out of the scope when it comes to parallelizing the `pcg` routine. The B_{2b} subblock is a matrix-vector product. As it was show above, the parallelization of this block of code is trivial. The resulting speedup from this subblock is given in the table below:

Case	B_{2a}	B_{2b}	B_{2c}
Small	46.9%	41.9%	5.5%
Medium	50.1%	39.6%	5.9%
Large	52.1%	39.1%	5.5%

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	2.01	2.85	6.18	15.40
$nPts = 173225$	1.00	2.05	4.09	8.26	15.77
$nPts = 84035$	1.00	1.97	4.58	8.20	17.42

Table 5: Speedup in the B_{2b} subblock of the pcg routine for different problem sizes $nPts$ and numbers of processors nP used.

The B_{2c} subblock is the unconnected set of all other calculations within the loop. Most of these are quite easy to parallelize. The speedup is about the same as for the the matrix-vector product up till 4 processors, and a bit less for 8 and 16. One important difference might be pointed out. The speedup from sharing a piece of work depends on the *workload balancing*. It always takes some extra time to coordinate the threads. If the time needed to execute the work distributed to each processor is not large compared to the set up time then the speedup will not be good. The amount of work in the third subblock is considerably less then in the first two subblocks. With the problem sizes used in these experiments the extra speedup per extra processor added was a bit less in the third subblock then with the matrix-vector product.

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.91	2.47	3.92	6.33
$nPts = 173225$	1.00	2.06	3.93	6.78	9.57
$nPts = 84035$	1.00	2.01	4.45	7.29	12.39

Table 6: Speedup in the B_{2c} subblock of the pcg routine for different problem sizes $nPts$ and numbers of processors nP used.

How it was done Some examples on how the parallelizations from the third block was carried out is included in the listings 12 and 13.

In this routine a maximum of 50% could be parallelized. According to Amdahl's law, the maximal theoretical speedup would then be $SpMax = 2$. The total speedups in the pcg routine are given by the table below:

Listing 12: Original version

```
bcoef=0.0


do ip=1,np
   bcoef=bcoef+rk(ip)*zk(ip)
enddo


if (iter.eq.1) then

   pk(1:np) = zk(1:np)

else
   beta=bcoef/bcoefm

   pk(1:np) = zk(1:np) &
          & + beta*pk(1:np)

endif
```

Listing 13: Parallel version

```
bcoef=0.0
!$omp parallel
!$omp do reduction(+:bcoef)
do ip=1,np
   bcoef=bcoef+rk(ip)*zk(ip)
enddo
!$omp end do

if (iter.eq.1) then
   !$omp workshare
   pk(1:np) = zk(1:np)
   !$omp end workshare
else
   beta=bcoef/bcoefm
   !$omp workshare
   pk(1:np) = zk(1:np) &
          & + beta*pk(1:np)
   !$omp end workshare
endif
!$omp end parallel
```

5.2.5 The bicgstab routine

The routine is an implementation of a preconditioned bi-conjugate gradient (stable?) algorithm.

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.38	1.21	1.48	1.98
$nPts = 173225$	1.00	1.36	1.63	1.83	1.97
$nPts = 84035$	1.00	1.39	1.88	1.79	2.09

Table 7: Speedup in the `pcg` routine for different problem sizes $nPts$ and numbers of processors nP used.

About 86% of the time spend in the routine is used by calls to the `matvec` routine. About 14% is left to parallelize. (This was the case for all the three problem sizes.)

Some parallelization were implemented. Mostly by the means of the `!$omp workshare` directive, but also some `!$omp do` directives. More can probably be done. The parallelizations already done seem to be correct, but might hopefully be more effective. A closer attention should be given to the eventual possibility of creating a bit fewer parallel regions. This might be possible by splitting the parallel constructs and the work-sharing constructs. Some better tuning of the existing parallel-do-loops are also probably possible.

The speedups measured were as stated by the table below:

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.23	1.48	1.56	1.58
$nPts = 173225$	1.00	1.35	1.47	1.61	1.66
$nPts = 84035$	1.00	1.43	1.42	1.55	1.61

Table 8: Speedup in the `bicgstab` routine, excluding the calls for the `matvec` routine, for different problem sizes $nPts$ and numbers of processors nP used.

5.2.6 The turb2 routine

The routine solves a k-ϵ turbulence model for high Reynolds numbers.

Four subblocks were identified. The first one consists of some initializations. The second one calculates the right hand side (rhs) for the k-ϵ equations. The third and fourth deal with the solution of respectively the k- and the ϵ-equation. Both these subblocks do first a update of the boundary conditions, and then they make a call for the `bicgstab` routine that actually solves the equation.

The timing results from the medium sized data set gave the following respective percentages of run time for the four subblocks: 2.10%, 42.38%, 32.36% and 22.74%. The initialization block is quite small, the expected gains from parallelizing it is not very big. At the moment this block has been untouched. The third and fourth block have not been parallelized either. Almost all the time spend in these two blocks is spend in the `bicgstab` routine. The only truly interesting subblock to investigate was the second one.

The 'rhs'-subblock consists of three nested do-loops. The first one is over the number of elements in the mesh, and the two others over the nodes in each element. For the purpose of explaining the parallelization done, it is not necessary to include all the code (it is about 100 code lines) nor an thourough explanation of the algorithm. The serial and parallel code are presented in condensed versions in respectively listing 14 and 15. An explanation of the parallelization approach is also included after the listings.

Listing 14: Original version

```
do ie=1,nelem
   calculate lots of stuff &
   & for the element considered

   do i=1,8
      ip = lnode(i,ie)
      do j=1,8
         jp = lnode(j,ie)
         calculate stuff in &
         & function of ie,i,j,ip,jp
         & gives contrip(jp)

         !add contribution to rhs
         rhs(ip) = rhs(ip) &
               & + contrib(jp)
      end do
   end do
end do
```

Listing 15: Parallel version

```
!$omp parallel do default(none) &
!$omp private(list of all private variables) &
!$omp shared(list of all shared variables)
do ie=1,nelem
   calculate lots of stuff &
   & for the element considered

   do i=1,8
      ip = lnode(i,ie)
      do j=1,8
         jp = lnode(j,ie)
         calculate stuff in &
         & function of ie,i,j,ip,jp
         & gives contrip(jp)

         !add contribution to rhs
         rhsLocal(ip) = rhsLocal(ip) &
                   & + contrib(jp)
      end do
   end do

   !$omp critical
   rhs( lnode(1:8,ie) ) = rhs( lnode(1:8,ie) ) &
                 & + rhsLocal(1:8)
   !$omp end critical
end do
!$omp end parallel do
```

In the real code the 'rhs' variable does not appear. Instead there are two components called 'rsk' and 'rsd', one for each equation (k and ϵ). The 'lnode' variable is an array which lists the nodes that make up each of the elements in the mesh.

The iterations of the othermost loop does not have to be executed in increasing order, as long as the update of the 'rhs' from each node is done correctly. This loop can be parallelized with an `!$omp parallel do` directive. The mayor challenge with this particular loop is the very high number of variables that appear in it. At least half of them need to be *private* to each thread.

The default behavior is to consider all variables as *shared* unless explicitly declared private at the creation of a parallel region. It is possible to change the behavior for a specific parallel region. Two possibilities exist. The first is to use the `default(private)` clause. Which means that all variables are private unless explicitly declared shared. In this particular loop it was judged usefull to use the second possibility, the `default(none)`. This implies that all variables that appear in the loop must be declared either in the list of private variables or in the list of shared variables. If a variable in the loop is not present in exactly one of the lists, a compilation error occurs. This approach decreases the chances of forgetting to add a variable that should have been part of the private list in the case of `default(shared)` or the other way around.

In the original version the following update scheme for the rhs variable occur: For a specific element ie, and for each of the nodes i in that element, the value of $rhs(ie)$ is updated eight times. Each node appear in general in several elements, which means that for a given value $k \in 1, ..., nbrOfNodes$, $rhs(k)$ will be updated multiple times and by different iterations ie of the element loop. The update of $rhs(k)$ constitute a *critical region*.

The `!$omp critical` - `!$omp end critical` directive pair could have been placed directly around the code line that updates the rhs variable. However this approached is problematic in practice. Each time a critical region construct is used, huge waiting times will be added. It was desirable to find a solution where each iteration only updates the rhs variable once, and thereof only needing one critical region. The chosen solution was to use a local array, $rhsLocal(1:8)$, which the two innermost do-loops work on.

It should be pointed out that the solution with the local array, gives a speedup even in a strictly serial environment. The reason for this is that the *cache hierarchy* is used more efficiently.

Results The speedups resulting from the parallelization of the `rhs` subblocks are given by the table below:

An interesting observation can be made. For all the three problem sizes it seems to be optimal to use four processors. Unless superlinear speedup occurs as a consequence of better use of the cache, the maximal speedup using four processors is $Sp = 4$. Considering that the parallelization of this loop contains a critical region, a speedup of more than 3 in all cases is good. The reason that the speedup decreases when more processors are added is that the waiting time at the critical region barrier increases faster than the gain from less work done by each processor. As a consequence this particular loop should be parallelized with the `NUM_THREADS(4)` clause.

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.80	3.38	3.07	2.79
$nPts = 173225$	1.00	1.81	3.41	3.23	2.94
$nPts = 84035$	1.00	1.74	3.24	3.13	2.83

Table 9: Speedup in the `rhs`-subblock in the `turb2` routine for different problem sizes $nPts$ and numbers of processors nP used.

5.2.7 The veloc routine

The purpose of the `veloc` routine is to compute the complete momentum equation. The internal structure of the routine is very much similar to the `turb2` routine. The `veloc` routine can be divided into four subblocks. The first one does the calculation of the 'rhs' for the three components of the momentum equation. The three other subblocks take care of solving one component each, including boundary condition handling.

The 'rhs' subblock was parallelized in the same manner as explained for the 'rhs' in the `turb2` case. The other three subblocks spend the mayor part of their time in the `bicgstab` routine. More or less the rest of the code lines in these blocks were parallelized with `!$omp workshare` and `!$omp do` directives. No particular problems were encountered.

Results The speedups resulting from the parallelization of the `rhs` subblocks are given by the table below:

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.50	1.96	1.57	1.60
$nPts = 173225$	1.00	1.47	2.01	1.65	1.68
$nPts = 84035$	1.00	1.45	1.88	1.57	1.59

Table 10: Speedup in the `rhs`-subblock in the `veloc` routine for different problem sizes $nPts$ and numbers of processors nP used.

Compared to the speedups measured in the 'rhs' subblock in the `turb2` routine the speedup in this case is not as good. Two possible explanations are proposed. In the `turb2` routine two components must be updated in the critical region. Here it is three components to update, which lead to 50% more time spend in the critical region each time, which again leads to a higher probability of several threads arriving at the critical region more or less at the same time. And hence longer mean waiting times. A second possible explanation is that in the 'rhs' subblock of the `veloc` routine, there are in fact two similar nested do-loops after each other. Both are parallelized the same way as stated above. But the second one contains considerably less work. It should be tried to remove the parallelization of this second loop to see if it runs faster on a single processor.

5.2.8 The elmat_k and elmat_u routines

Both these two routines use $6 - 7\%$ of the global run time according to the profiling of the initial test case. They were examined for possible parallelization. Nothing were implemented. It is not impossible that there might be gains from parallelization, just improbable. Both routines got a very low self time. Their importance on the global scale arise from the very high number of times these two routines are called. This fact seems to suggest that the parent routines, `assem_k` and `assem_u` should be investigated instead.

5.2.9 The assem_u routine

The purpose of the `assem_u` routine is to assemble the coefficient matrix for the velocity components. This routine was not measured with detailed timers. The main reason was that when the routine was considered, the experience from other routines seemed to indicate were and how the parallelization should be done. If more details on the actual effect is wanted at a later stage, for instance if better tuning of `!$omp parallel do` loops is explored, timers might and should be added.

Two techniques were used. First a few initializations were parallelized with the `!$omp workshare` directive. Then a triple nested do-loop were parallelized with the technique described for the 'rhs' subblock in the `turb2` routine.

The speedups of the entire routine are given in the table below.

As already seen with the parallelization technique used and on the problem sizes considered the speedup is best when four processors are used. We would hope for a speedup between 3 and 4 in this case. Since the maximum is only about 2.5, some further tuning and experimentation could be good.

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.83	2.33	2.18	2.15
$nPts = 173225$	1.00	1.77	2.48	2.38	2.35
$nPts = 84035$	1.00	1.72	2.37	2.23	2.21

Table 11: Speedup in the `assem_u` routine for different problem sizes $nPts$ and numbers of processors nP used.

5.2.10 The assem_k routine

The purpose of the `assem_k` routine is to assemble the coefficient matrix for the turbulence components. This routine is almost identical to the `assem_u` routine. It has not yet been parallelized, but that should be trivial to do.

5.2.11 The dpres routine

The purpose of the `dpres` routine is to perform a projection step as part of the solution of the pressure correction. Most of the time spend in the routine is for the call for the `pcg` routine. For the three problemsizes used (small, medium and large), the respective percentages of the time spend in `pcg` are 94%, 95% and 96%. So far there has not been done any parallelization work on the last few percentages. A simple inspection seems to indicate that it is possible. The suggested method to use is again the same as for the 'rhs' case.

5.3 Evaluation

A look back on the flat profile Comparing the flat profile of the initial test case, in listing 9, with a conservative estimate of the percentage of parallelization in the considered routines, gives an approximative percentage of untouched code of 35%. This percentage is relative to the total run time for the smallest data set and with only 100 time steps. With other inputs the percentage will change, but so will the serial profile.

Out of these 35% of untouched code, four cases can be named. First of all, there is the `pssor` routine (about 15%) which present an algorithmic challenge. Then there are maybe about 5% that is just a trivial deal to do. (Same as somewhere else.) Further there are about 5% that are totally unknown. And finally about 10% that is considered hard, if at all possible to parallelize.

Global speedups For the three data sets used in the detailed tests the following speedups were measured:

Problem size	$nP = 1$	$nP = 2$	$nP = 4$	$nP = 8$	$nP = 16$
$nPts = 339521$	1.00	1.48	1.86	2.00	2.20
$nPts = 173225$	1.00	1.52	2.00	2.19	2.30
$nPts = 84035$	1.00	1.51	1.99	2.10	2.27

Table 12: Global speedup in the `Simra` program for different problem sizes $nPts$ and numbers of processors nP used.

The measured speedup is higher for the medium sized data set than for the large one. That is not as expected. We believe the reason for this comes from the problem of mixing 'wall-clock-time' with 'non-exclusive-non-absolute-priority-access' to a computer system.

Amdahl Even though the percentages from the flat profile can not be directly compared to the three data sets used later on, it is just to temptating to do it. If we stick with the estimate of 35% of the code left to parallelize, Amdahl's law tell us that the maximal possible speedup $maxSp$ using $nP = 16$ processors is,

$$maxSp = \frac{1}{0.35 + \frac{0.65}{16}} = 2.56. \tag{1}$$

This seems to indicate that the obtained global speedup is not that bad. To get a much better estimate, the percentage of the global run time should be calculated for each routine and each subblock in the timer-wrapper. It is very easy to do this the next time the tests are run. (But it is quite cumbersome to calculate it retrospectively.)

5.4 Future work

There are still lots of work to do. It might be divided in four different threads.

5.4.1 The pssor routine

The `pssor` routine is untouched. The gain from parallelizing this one on loop level will be big. Investigate what can be done. If the algorithm used does not parallelize well, it should be considered to maybe use another one that does so.

5.4.2 Fill the holes and fine tune loops

In some of the routines that have already been parallelized, there are some percentages of the code that have not been touched. Not all of these pieces parallelize well, some might even get mayor slowdowns. Some of the loops already parallelized, might gain quite a bit performance from some fine tuning. For instance, the ideal number of processors to use in a given parallel region might depend on the problem size. Both of these approaches will be much more successful if the two following tools are developed: better timers and a small program that measures the time used by the OpenMP-directives themselves.

First of all, "cpu-timers" should absolutely be added one way or another. There exist libraries to properly measure "cpu-time" on the different threads. A search on the Internet revealed for instance a library called *TIM* [14]. It is probable that the program has to be compiled in Fortran 95. However the differences between Fortran 90 and 95 are quite small. The final version might be compiled in Fortran 90, since the timers are only for the development face.

A small program should be made, or found on the Internet, that measures how the different OpenMP-directives behave on a specific machine. Some simple loops are sufficient to test for instance how much time it takes to create a parallel region and close it down with increasing number of processors. Below a small suggestion is presented for how this might be solved.

```
do i = 0,4
   nP = 2^i
   wtime = omp_get_wtime()
   do j = 1,1000
      !$omp parallel NUM_THREADS(nP)

      !$omp end parallel
   end do
   timer(i ) = omp_get_wtime() - wtime
end do
```

The idea is to compare these measured times with how many floating point operations that can be done in the same time. And from there draw some conclusion on how specific loops might be tuned. The small program might even be included in the Simra source code. In that way a short test at the beginning of the execution of the program might permit to tune loops at run time.

The list of compiler options should be studied more in detail. The version run on Njord was tuned for the type of processor and architecture on that specific supercomputer. The compiler options are included in the *Makefile* and permits different tuning on different platforms. There might be other options available that could speedup the program.

The Xprofiler on Njord can also be used for parallel profiling. This should be explored. Parallel profiling permits among other things to identify (too) long waiting times in the program, for instance at synchronize barriers.

5.4.3 Nesting of parallel regions and equation level parallelization

There has already been identified two places were equation level parallelization might be possible in Simra. For a given time step, the three velocity components dU, dV and dW are independent and could in principle be solved in parallel. The same case holds for the two turbulence components $d\epsilon$ and dk.

The program should be examined with the aim of identifying more regions that could be parallelized on a higher level.

At the moment nesting of parallel regions has not been mastered. That means that one has to chose either to parallelize on equation level or on loop level. It might depend from case to case which technique that gives the highest speedup. Detailed and accurate timing should be done to decide which one to chose in a give setting.

The ideal solution would be to get access to a system that supports nesting of two parallel regions. Take for instance the case of the turbulence equation. The mayor part of the speedup comes from the parallelization of the `matvec` routine. The speedup is not doubled when going from 8 to 16 processors, even though it is not very far from being the case. It might happend that solving the two components in parallel with 8 processors on each would be faster. OpenMP v2.0 supports nesting of parallel regions. But the underlaying system must do it as well. At the moment this has not been mastered on Njord. It is not clear if this lack of support is a question about hardware or software. If the latter is the case, maybe it could be fixed in the future.

Nesting of parallel regions could also be implemented as an hybrid OpenMP-MPI solution. In this case one node with 16 cores could be used to solve one component. The hybrid approach is on the other hand much more complicated to implement.

5.4.4 New software

The v3.0 of the OpenMP standard was released in summer 2008. Gnu's compiler *gfortran* already supports the newest standard. IBM's Fortran compiler, *xlf*, also does so in the newest version. At the moment this version is not installed on Njord. Maybe it is a good idea to put some pressure on Itea to buy and install it?

6 Summary

Profiling is a very good tool to get an indication on which subroutines to concentrate on when parallelizing scientific codes. In general there will be necessary to get more detailed information regarding the relative importance of different sub-blocks inside an interesting subroutine. This information might be obtained by the use of manually inserted timers inside the subroutine. In order to get more reliable results *cpu-timers* should be used instead of *wall-clock-timers*.

The main goal was to parallelize the *Simra* CFD-code as much as possible. Some initial work on less complex and smaller programs did undoubtedly lead to better results for the Simra code. At the moment about 65% of the program has been parallelized. When using 16 cores on the Njord supercomputer, the global speedup resulting from this work is between 2.2 and 2.3 depending on the problem size.

There is one subroutine that is very important from an performance point of view, about 15% of the runtime, that has not been touched at all. The algorithm is not trivial to parallelize. It might be possible, but if not one should consider changing it with another that does the same job. If the subroutine in question is parallelized successfully, a global speedup between 3 and 4 should be expected.

References

[1] Peter Arbenz, Wesley Petersen, Introduction to Parallel Computing - A practical guide with examples in C, Oxford University Press.

[2] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, Parallel Programming in OpenMP, Morgan Kaufmann, 2001.

[3] Barbara Chapman, Gabriele Jost, Ruud vanderPas, David Kuck, Using OpenMP: Portable Shared Memory Parallel Processing, MIT Press, 2007.

[4] Jack Dongarra, Jim Bunch, Cleve Moler, Pete Stewart, LINPACK User's Guide, SIAM, 1979.

[5] Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh, Algorithm 539: Basic Linear Algebra Subprograms for Fortran Usage, ACM Transactions on Mathematical Software, Volume 5, Number 3, September 1979, pages 308-323.

[6] Hans Rudolf Schwarz, Finite Element Methods, Academic Press, 1988.

[7] Gilbert Strang, George Fix, An Analysis of the Finite Element Method, Cambridge, 1973.

[8] Olgierd Zienkiewicz, The Finite Element Method, Sixth Edition, Butterworth-Heinemann, 2005.

[9] John Burkardt,

`http://people.scs.fsu.edu/~burkardt`

[10] NOTUR - THE NORWEGIAN METACENTER FOR COMPUTATIONAL SCIENCE,

`http://www.notur.no/hardware/njord/`

[11] Wikipedia article on Profiling aka. Performance analysis,

`http://en.wikipedia.org/wiki/Performance_analysis`

[12] Jay Fenlason and Richard Stallman, GNU gprof - The GNU Profiler,

`http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`

[13] Miguel Hermanns, Parallel Programming in Fortran 95 using OpenMP School of Aeronautical Engineering. Polytechnical University of Madrid, Spain April 2002.

[14] A Fortran 95 library for timing of parallel programs. Autor: Jalal Chergui, IDRIS-CNRS.

`http://www.idris.fr/data/publications/TIM/`

[15] Torbjørn Utnes, A Segregated Implicit Pressure Projection Method for Incompressible Flows, J. of Computational Physics 227 (4), 2198–2211, 2008.

A SGEFA

A.1 Original code

In the original code five slightly different versions of the SGEFA routine are beeing used. Here only the first one is presented in its' totality. The only parts that differ among the five, are the two blocks labeled *Compute column K of the lower triangular factor* (starting at code line 87) and *Add multiplies of the pivot row to the remaining rows* (starting at code line 93). These two blocks are the only parts presented from the other subroutines.

```
1    subroutine sgefa ( a, lda, n, ipvt, info )
2
3   !*********************************************************************
4   !
5   !! SGEFA factors a real matrix by gaussian elimination.
6   !
7   !  Discussion:
8   !
9   !    This is a version of the LINPACK routine SGEFA which has been
10  !    simplified by replacing all the calls to BLAS routines by
11  !    equivalent code.
12  !
13  !     on entry
14  !
15  !        a       real(lda, n)
16  !                the matrix to be factored.
17  !
18  !        lda     integer
19  !                the leading dimension of the array  a .
20  !
21  !        n       integer
22  !                the order of the matrix  a .
23  !
24  !     on return
25  !
26  !        a       an upper triangular matrix and the multipliers
27  !                which were used to obtain it.
28  !                the factorization can be written  a=l*u  where
29  !                l  is a product of permutation and unit lower
30  !                triangular matrices and  u  is upper triangular.
31  !
32  !        ipvt    integer(n)
33  !                an integer vector of pivot indices.
34  !
35  !        info    integer
36  !                = 0  normal value.
37  !                = k  if  u(k,k) .eq. 0.0 .  this is not an error
38  !                     condition for this subroutine, but it does
39  !                     indicate that sgesl or sgedi will divide by zero
40  !                     if called.  use  rcond  in sgeco for a reliable
41  !                     indication of singularity.
42  !
```

```
43      implicit none
44
45      integer lda
46      integer n
47
48      real a(lda,n)
49      integer i
50      integer info
51      integer ipvt(n)
52      integer j
53      integer k
54      integer l
55      real t
56
57      info = 0
58      do k = 1, n - 1
59    !
60    !  Find the pivot index L.
61    !
62        l = k
63        t = abs ( a(k,k) )
64        do i = k + 1, n
65          if ( t < abs ( a(i,k) ) ) then
66            l = i
67            t = abs ( a(i,k) )
68          end if
69        end do
70        ipvt(k) = l
71
72        if ( a(l,k) == 0.0E+00 ) then
73          info = k
74          return
75        end if
76    !
77    !  Interchange rows K and L.
78    !
79        if ( l /= k ) then
80          do j = k, n
81            t      = a(l,j)
82            a(l,j) = a(k,j)
83            a(k,j) = t
84          end do
85        end if
86    !
87    !  Compute column K of the lower triangular factor.
88    !
89        do i = k + 1, n
90          a(i,k) = - a(i,k) / a(k,k)
91        end do
92    !
```

```
 93  !  Add multiples of the pivot row to the remaining rows.
 94  !
 95      do j = k + 1, n
 96        do i = k + 1, n
 97          a(i,j) = a(i,j) + a(i,k) * a(k,j)
 98        end do
 99      end do
100
101    end do
102
103    ipvt(n) = n
104    if ( a(n,n) == 0.0E+00 ) then
105      info = n
106    end if
107
108    return
109  end subroutine sgefa
110  !**********************************************************************
111  subroutine sgefa_c ( a, lda, n, ipvt, info )
112  ! start identic part as sgefa
113  .
114  .
115  .
116  ! end  identic part as sgefa
117
118  !
119  !  Compute column K of the lower triangular factor.
120  !
121      a(k+1:n,k) = - a(k+1:n,k) / a(k,k)
122  !
123  !  Add multiples of the pivot row to the remaining rows.
124  !
125      do j = k+1, n
126        a(k+1:n,j) = a(k+1:n,j) + a(k+1:n,k) * a(k,j)
127      end do
128
129  ! start identic part as sgefa
130  .
131  .
132  .
133  ! end  identic part as sgefa
134
135  end subroutine sgefa_c
136  !**********************************************************************
```

```
137  subroutine sgefa_c_omp ( a, lda, n, ipvt, info )
138  ! start identic part as sgefa
139  .
140  .
141  .
142  ! end  identic part as sgefa
143
144  !
145  !  Compute column K of the lower triangular factor.
146  !
147  !$omp workshare
148        a(k+1:n,k) = - a(k+1:n,k) / a(k,k)
149  !$omp end workshare
150  !
151  !  Add multiples of the pivot row to the remaining rows.
152  !
153  !$omp parallel do shared ( a, k, n ) private ( j )
154       do j = k+1, n
155         a(k+1:n,j) = a(k+1:n,j) + a(k+1:n,k) * a(k,j)
156       end do
157  !$omp end parallel do
158
159  ! start identic part as sgefa
160  .
161  .
162  .
163  ! end  identic part as sgefa
164
165  end subroutine sgefa_c_omp
166  !*******************************************************************
167  subroutine sgefa_r ( a, lda, n, ipvt, info )
168  ! start identic part as sgefa
169  .
170  .
171  .
172  ! end  identic part as sgefa
173
174  !  Compute column K of the lower triangular factor.
175  !
176       a(k+1:n,k) = - a(k+1:n,k) / a(k,k)
177  !
178  !  Add multiples of the pivot row to the remaining rows.
179  !
180     do i = k+1, n
181       a(i,k+1:n) = a(i,k+1:n) + a(i,k) * a(k,k+1:n)
182     end do
183
184  ! start identic part as sgefa
185  .
186  .
187  .
188  ! end  identic part as sgefa
189
190  end subroutine sgefa_r
```

```
191  subroutine sgefa_r_omp ( a, lda, n, ipvt, info )
192  ! start identic part as sgefa
193  .
194  .
195  .
196  ! end  identic part as sgefa
197
198  !  Compute column K of the lower triangular factor.
199  !
200  !$omp workshare
201        a(k+1:n,k) = - a(k+1:n,k) / a(k,k)
202  !$omp end workshare
203  !
204  !  Add multiples of the pivot row to the remaining rows.
205  !
206  !$omp parallel do shared ( a, k, n ) private ( i )
207     do i = k+1, n
208       a(i,k+1:n) = a(i,k+1:n) + a(i,k) * a(k,k+1:n)
209     end do
210  !$omp end parallel do
211
212  ! start identic part as sgefa
213  .
214  .
215  .
216  ! end  identic part as sgefa
217
218  end subroutine sgefa_r_omp
```

A.2 Modified code

Two if-tests (see listing line 73 and 109) permit to decide respectivly if sequential or parallel version is to be run, and wheter normal, columnvice or rowvice version is to be run.

```
1   subroutine sgefa ( a, lda, n, ipvt, info, version, mode )
2
3   !*************************************************************************
4   !
5   !! SGEFA factors a real matrix by gaussian elimination.
6   !
7   !  Discussion:
8   !
9   !    This is a version of the LINPACK routine SGEFA which has been
10  !    simplified by replacing all the calls to BLAS routines by
11  !    equivalent code.
12  !
13  !     on entry
14  !
15  !        a        real(lda, n)
16  !                 the matrix to be factored.
17  !
18  !        lda      integer
19  !                 the leading dimension of the array  a .
20  !
21  !        n        integer
22  !                 the order of the matrix  a .
23  !
24  !      version    indicates which version of the algorithme to choose.
25  !                 Possible values are 'standard', 'column' and 'row'
26  !
27  !       mode      indicates the modus. 'sequential' or 'parallel'
28  !
29  !     on return
30  !
31  !        a        an upper triangular matrix and the multipliers
32  !                 which were used to obtain it.
33  !                 the factorization can be written  a=l*u  where
34  !                 l  is a product of permutation and unit lower
35  !                 triangular matrices and  u  is upper triangular.
36  !
37  !        ipvt     integer(n)
38  !                 an integer vector of pivot indices.
39  !
40  !        info     integer
41  !                 = 0, this is the normal value, and the algorithm succeeded.
42  !                 = k  if  u(k,k) .eq. 0.0 .
43  !                   If K, then on the K-th elimination step,
44  !                   a zero pivot was encountered.
45  !                   The matrix is numerically not invertible.
46  !                       (this is not an error
47  !                       condition for this subroutine, but it does
48  !                       indicate that sgesl or sgedi will divide by zero
49  !                       if called.  use  rcond  in sgeco for a reliable
50  !                       indication of singularity.)
51  !
52  !------------------------------------------------
53
54   implicit none
55
56  !------------------------------------------------
57  !-Input parameters
58   integer, intent(in) :: lda
59   integer, intent(in) ::  n
60   real, intent(inout) ::  a(lda,n)
61   integer, intent(out) ::  info
62   integer, intent(out) ::  ipvt(n)
63
64   character(len=*), intent(in) :: version
65   character(len=*), intent(in) :: mode
66  !------------------------------------------------
67   integer i , j, k, l
68   real temp(n)
69   real t
70
71   info = 0
72
```

```
73    !$omp parallel IF( mode == 'parallel ')
74    do k = 1, n - 1
75      !$omp single
76  !
77  !  Find the pivot index L.
78  !
79      l = k
80      t = abs ( a(k,k) )
81      do i = k + 1, n
82        if ( t < abs ( a(i,k) ) ) then
83          l = i
84          t = abs ( a(i,k) )
85        end if
86      end do
87      ipvt(k) = l
88
89      if ( a(l,k) == 0.0E+00 ) then
90        info = k
91        !return !can't 'return' because invalid exit from omp structured block
92        write (*,*) 'Critical error!!!!. Pivot breakdown at row k'
93      end if
94      !$omp end single
95  !
```

```
 96  !  Interchange rows K and L.
 97  !
 98       if ( l /= k ) then
 99          !$omp workshare !if (mode == 'parallel')
100          temp(k:n) = a(l,k:n)
101          a(l,k:n) = a(k,k:n)
102          a(k,k:n) = temp(k:n)
103          !$omp end workshare
104       end if
105
106  !
107  !  Compute column K of the lower triangular factor.
108  !
109       if (version == 'standard') then
110          do i = k + 1, n
111             a(i,k) = - a(i,k) / a(k,k)
112          end do
113       else ! version == 'column' or version == 'row'
114          !$omp workshare !IF(mode == 'parallel')
115          a(k+1:n,k) = - a(k+1:n,k) / a(k,k)
116          !$omp end workshare
117       end if
118
119  !
120  !  Add multiples of the pivot row to the remaining rows.
121  !
122       if (version == 'standard') then
123          do j = k + 1, n
124             do i = k + 1, n
125                a(i,j) = a(i,j) + a(i,k) * a(k,j)
126             end do
127          end do
128
129       else if (version == 'column' ) then
130          !$omp do schedule(guided) !IF(mode == 'parallel')
131          do j = k+1, n
132             a(k+1:n,j) = a(k+1:n,j) + a(k+1:n,k) * a(k,j)
133          enddo
134          !$omp end do
135
136       else  ! (version == 'row' )
137          !$omp do schedule(guided) !IF(mode == 'parallel')
138          do i = k+1, n
139             a(i,k+1:n) = a(i,k+1:n) + a(i,k) * a(k,k+1:n)
140          enddo
141          !$omp end do
142
143       end if
144
145     end do
146     !$omp end parallel
147
```

```
148
149    ipvt(n) = n
150    if ( a(n,n) == 0.0E+00 ) then
151      info = n
152    end if
153
154    return
155  end subroutine sgefa
```

B Poisson equation

B.1 Profiler listings

```
 1  Flat profile:
 2
 3  Each sample counts as 0.01 seconds.
 4    %   cumulative   self              self     total
 5   time   seconds   seconds    calls  s/call   s/call  name
 6   96.37     33.47    33.47        1   33.47    33.47  dgb_fa_
 7    1.47     33.98     0.51  1942488    0.00     0.00  qbf_
 8    1.21     34.40     0.42        1    0.42     0.74  assemble_
 9    0.58     34.60     0.20        1    0.20     0.20  dgb_sl_
10    0.17     34.66     0.06        1    0.06     0.28  errors_
11    0.06     34.68     0.02   143659    0.00     0.00  exact_
12    0.03     34.69     0.01   171396    0.00     0.00  rhs_
13    0.03     34.70     0.01     9522    0.00     0.00  quad_e_
14    0.03     34.71     0.01        1    0.01     0.01  bandwidth_
15    0.03     34.72     0.01        1    0.01     0.01  boundary_
16    0.03     34.73     0.01        1    0.01     0.01  solution_write_
17    0.00     34.73     0.00        3    0.00     0.00  get_unit_
18    0.00     34.73     0.00        3    0.00     0.00  r8vec_print_some_
19    0.00     34.73     0.00        2    0.00     0.00  dgb_print_some_
20    0.00     34.73     0.00        2    0.00     0.00  timestamp_
21    0.00     34.73     0.00        2    0.00     0.00  timestring_
22    0.00     34.73     0.00        1    0.00    34.73  MAIN__
23    0.00     34.73     0.00        1    0.00     0.00  area_set_
24    0.00     34.73     0.00        1    0.00     0.00  compare_
25    0.00     34.73     0.00        1    0.00     0.00  element_write_
26    0.00     34.73     0.00        1    0.00     0.00  grid_t6_
27    0.00     34.73     0.00        1    0.00     0.00  indx_set_
28    0.00     34.73     0.00        1    0.00     0.00  nodes_write_
29    0.00     34.73     0.00        1    0.00     0.00  quad_a_
30    0.00     34.73     0.00        1    0.00     0.00  xy_set_
31
32   %          the percentage of the total running time of the
33  time        program used by this function.
34
35  cumulative  a running sum of the number of seconds accounted
36   seconds    for by this function and those listed above it.
37
38   self       the number of seconds accounted for by this
39  seconds     function alone.  This is the major sort for this
40              listing.
41
42  calls       the number of times this function was invoked, if
43              this function is profiled, else blank.
44
45   self       the average number of milliseconds spent in this
46  ms/call     function per call, if this function is profiled,
47              else blank.
48
49   total      the average number of milliseconds spent in this
50  ms/call     function and its descendents per call, if this
51              function is profiled, else blank.
52
53  name        the name of the function.  This is the minor sort
54              for this listing. The index shows the location of
55              the function in the gprof listing. If the index is
56              in parenthesis it shows where it would appear in
57              the gprof listing if it were to be printed.
58
59                       Call graph (explanation follows)
60
```

```
61
62  granularity: each sample hit covers 4 byte(s) for 0.03% of 34.73 seconds
63
64  index % time    self  children    called     name
65                  0.00   34.73       1/1           main [2]
66  [1]    100.0    0.00   34.73       1         MAIN__ [1]
67                 33.47    0.00       1/1           dgb_fa_ [3]
68                  0.42    0.33       1/1           assemble_ [4]
69                  0.06    0.22       1/1           errors_ [6]
70                  0.20    0.00       1/1           dgb_sl_ [7]
71                  0.01    0.00       1/1           boundary_ [9]
72                  0.01    0.00       1/1           bandwidth_ [12]
73                  0.01    0.00       1/1           solution_write_ [13]
74                  0.00    0.00       1/1           compare_ [14]
75                  0.00    0.00       3/3           r8vec_print_some_ [16]
76                  0.00    0.00       2/2           timestamp_ [18]
77                  0.00    0.00       2/2           dgb_print_some_ [17]
78                  0.00    0.00       1/1           xy_set_ [26]
79                  0.00    0.00       1/1           grid_t6_ [22]
80                  0.00    0.00       1/1           quad_a_ [25]
81                  0.00    0.00       1/1           area_set_ [20]
82                  0.00    0.00       1/1           indx_set_ [23]
83                  0.00    0.00       1/1           nodes_write_ [24]
84                  0.00    0.00       1/1           element_write_ [21]
85  -----------------------------------------------
86                                                   <spontaneous>
87  [2]    100.0    0.00   34.73                 main [2]
88                  0.00   34.73       1/1           MAIN__ [1]
89  -----------------------------------------------
90                 33.47    0.00       1/1           MAIN__ [1]
91  [3]     96.4   33.47    0.00       1         dgb_fa_ [3]
92  -----------------------------------------------
93                  0.42    0.33       1/1           MAIN__ [1]
94  [4]      2.1    0.42    0.33       1         assemble_ [4]
95                  0.32    0.00 1199772/1942488     qbf_ [5]
96                  0.01    0.00  171396/171396      rhs_ [10]
97  -----------------------------------------------
98                  0.20    0.00  742716/1942488     errors_ [6]
99                  0.32    0.00 1199772/1942488     assemble_ [4]
100 [5]      1.5    0.51    0.00 1942488         qbf_ [5]
101 -----------------------------------------------
102                 0.06    0.22       1/1           MAIN__ [1]
103 [6]      0.8    0.06    0.22       1         errors_ [6]
104                 0.20    0.00  742716/1942488     qbf_ [5]
105                 0.02    0.00  123786/143659      exact_ [8]
106                 0.01    0.00    9522/9522        quad_e_ [11]
107 -----------------------------------------------
108                 0.20    0.00       1/1           MAIN__ [1]
109 [7]      0.6    0.20    0.00       1         dgb_sl_ [7]
110 -----------------------------------------------
111                 0.00    0.00     552/143659      boundary_ [9]
112                 0.00    0.00   19321/143659      compare_ [14]
113                 0.02    0.00  123786/143659      errors_ [6]
114 [8]      0.1    0.02    0.00  143659         exact_ [8]
115 -----------------------------------------------
116                 0.01    0.00       1/1           MAIN__ [1]
117 [9]      0.0    0.01    0.00       1         boundary_ [9]
118                 0.00    0.00     552/143659      exact_ [8]
119 -----------------------------------------------
120                 0.01    0.00  171396/171396      assemble_ [4]
121 [10]     0.0    0.01    0.00  171396         rhs_ [10]
122 -----------------------------------------------
123                 0.01    0.00    9522/9522        errors_ [6]
124 [11]     0.0    0.01    0.00    9522         quad_e_ [11]
125 -----------------------------------------------
126                 0.01    0.00       1/1           MAIN__ [1]
127 [12]     0.0    0.01    0.00       1         bandwidth_ [12]
128 -----------------------------------------------
129                 0.01    0.00       1/1           MAIN__ [1]
130 [13]     0.0    0.01    0.00       1         solution_write_ [13]
131                 0.00    0.00       1/3           get_unit_ [15]
132 -----------------------------------------------
133                 0.00    0.00       1/1           MAIN__ [1]
134 [14]     0.0    0.00    0.00       1         compare_ [14]
135                 0.00    0.00   19321/143659      exact_ [8]
136 -----------------------------------------------
```

```
137                     0.00      0.00          1/3             element_write_ [21]
138                     0.00      0.00          1/3             nodes_write_ [24]
139                     0.00      0.00          1/3             solution_write_ [13]
140   [15]      0.0     0.00      0.00          3           get_unit_ [15]
141   -----------------------------------------------
142                     0.00      0.00          3/3             MAIN__ [1]
143   [16]      0.0     0.00      0.00          3           r8vec_print_some_ [16]
144   -----------------------------------------------
145                     0.00      0.00          2/2             MAIN__ [1]
146   [17]      0.0     0.00      0.00          2           dgb_print_some_ [17]
147   -----------------------------------------------
148                     0.00      0.00          2/2             MAIN__ [1]
149   [18]      0.0     0.00      0.00          2           timestamp_ [18]
150                     0.00      0.00          2/2             timestring_ [19]
151   -----------------------------------------------
152                     0.00      0.00          2/2             timestamp_ [18]
153   [19]      0.0     0.00      0.00          2           timestring_ [19]
154   -----------------------------------------------
155                     0.00      0.00          1/1             MAIN__ [1]
156   [20]      0.0     0.00      0.00          1           area_set_ [20]
157   -----------------------------------------------
158                     0.00      0.00          1/1             MAIN__ [1]
159   [21]      0.0     0.00      0.00          1           element_write_ [21]
160                     0.00      0.00          1/3             get_unit_ [15]
161   -----------------------------------------------
162                     0.00      0.00          1/1             MAIN__ [1]
163   [22]      0.0     0.00      0.00          1           grid_t6_ [22]
164   -----------------------------------------------
165                     0.00      0.00          1/1             MAIN__ [1]
166   [23]      0.0     0.00      0.00          1           indx_set_ [23]
167   -----------------------------------------------
168                     0.00      0.00          1/1             MAIN__ [1]
169   [24]      0.0     0.00      0.00          1           nodes_write_ [24]
170                     0.00      0.00          1/3             get_unit_ [15]
171   -----------------------------------------------
172                     0.00      0.00          1/1             MAIN__ [1]
173   [25]      0.0     0.00      0.00          1           quad_a_ [25]
174   -----------------------------------------------
175                     0.00      0.00          1/1             MAIN__ [1]
176   [26]      0.0     0.00      0.00          1           xy_set_ [26]
177   -----------------------------------------------
178
179    This table describes the call tree of the program, and was sorted by
180    the total amount of time spent in each function and its children.
181
182    Each entry in this table consists of several lines.  The line with the
183    index number at the left hand margin lists the current function.
184    The lines above it list the functions that called this function,
185    and the lines below it list the functions this one called.
186    This line lists:
187        index       A unique number given to each element of the table.
188                    Index numbers are sorted numerically.
189                    The index number is printed next to every function name so
190                    it is easier to look up where the function in the table.
191
192        % time      This is the percentage of the 'total'␣time␣that␣was␣spent
193   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in␣this␣function␣and␣its␣children.␣␣Note␣that␣due␣to
194   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣different␣viewpoints,␣functions␣excluded␣by␣options,␣etc,
195   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣these␣numbers␣will␣NOT␣add␣up␣to␣100%.
196
197   ␣␣␣␣␣self␣␣␣␣␣␣␣␣This␣is␣the␣total␣amount␣of␣time␣spent␣in␣this␣function.
198
199   ␣␣␣␣␣children␣␣␣␣This␣is␣the␣total␣amount␣of␣time␣propagated␣into␣this
200   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣function␣by␣its␣children.
201
202   ␣␣␣␣␣called␣␣␣␣␣␣This␣is␣the␣number␣of␣times␣the␣function␣was␣called.
203   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣If␣the␣function␣called␣itself␣recursively,␣the␣number
204   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣only␣includes␣non-recursive␣calls,␣and␣is␣followed␣by
205   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a␣'+' and the number of recursive calls.
206
207        name        The name of the current function.  The index number is
208                    printed after it.  If the function is a member of a
209                    cycle, the cycle number is printed between the
210                    function's␣name␣and␣the␣index␣number.
211
212
```

```
213  For the function's parents, the fields have the following meanings:
214
215       self        This is the amount of time that was propagated directly
216                   from the function into this parent.
217
218       children    This is the amount of time that was propagated from
219                   the function's children into this parent.
220
221       called      This is the number of times this parent called the
222                   function '/' the total number of times the function
223                   was called. Recursive calls to the function are not
224                   included in the number after the '/'.
225
226       name        This is the name of the parent. The parent's index
227                   number is printed after it. If the parent is a
228                   member of a cycle, the cycle number is printed between
229                   the name and the index number.
230
231  If the parents of the function cannot be determined, the word
232  '<spontaneous>' is printed in the 'name' field, and all the other
233  fields are blank.
234
235  For the function's children, the fields have the following meanings:
236
237       self        This is the amount of time that was propagated directly
238                   from the child into the function.
239
240       children    This is the amount of time that was propagated from the
241                   child's children to the function.
242
243       called      This is the number of times the function called
244                   this child '/' the total number of times the child
245                   was called. Recursive calls by the child are not
246                   listed in the number after the '/'.
247
248       name        This is the name of the child. The child's index
249                   number is printed after it. If the child is a
250                   member of a cycle, the cycle number is printed
251                   between the name and the index number.
252
253  If there are any cycles (circles) in the call graph, there is an
254  entry for the cycle-as-a-whole. This entry shows who called the
255  cycle (as parents) and the members of the cycle (as children.)
256  The '+' recursive calls entry shows the number of function calls that
257  were internal to the cycle, and the calls entry for each member shows,
258  for that member, how many times it was called from other members of
259  the cycle.
260
261
262  Index by function name
263
264    [1] MAIN__                  [21] element_write_        [11] quad_e_
265   [20] area_set_                [6] errors_               [16] r8vec_print_some_
266    [4] assemble_                [8] exact_                [10] rhs_
267   [12] bandwidth_              [15] get_unit_             [13] solution_write_
268    [9] boundary_               [22] grid_t6_              [18] timestamp_
269   [14] compare_                [23] indx_set_             [19] timestring_
270    [3] dgb_fa_                 [24] nodes_write_          [26] xy_set_
271   [17] dgb_print_some_          [5] qbf_
272    [7] dgb_sl_                 [25] quad_a_
```

B.2 Burkardt's original version

```
 1  subroutine dgb_fa ( n, ml, mu, a, pivot, info )
 2
 3  !*****************************************************************************80
 4  !
 5  !! DGB_FA performs a LINPACK-style PLU factorization of an DGB matrix.
 6  !
 7  !  Discussion:
 8  !
 9  !    The DGB storage format is for an M by N banded matrix, with lower
10  !    bandwidth ML and upper bandwidth MU.  Storage includes room for ML
11  !    extra superdiagonals, which may be required to store nonzero entries
12  !    generated during Gaussian elimination.
```

```
13  !
14  !    The original M by N matrix is "collapsed" downward, so that diagonals
15  !    become rows of the storage array, while columns are preserved.  The
16  !    collapsed array is logically 2*ML+MU+1 by N.
17  !
18  !    The following program segment will set up the input.
19  !
20  !      m = ml + mu + 1
21  !      do j = 1, n
22  !        i1 = max ( 1, j-mu )
23  !        i2 = min ( n, j+ml )
24  !        do i = i1, i2
25  !          k = i - j + m
26  !          a(k,j) = afull(i,j)
27  !        end do
28  !      end do
29  !
30  !    This uses rows ML+1 through 2*ML+MU+1 of the array A.
31  !    In addition, the first ML rows in the array are used for
32  !    elements generated during the triangularization.
33  !
34  !    The ML+MU by ML+MU upper left triangle and the
35  !    ML by ML lower right triangle are not referenced.
36  !
37  !  Modified:
38  !
39  !    04 March 1999
40  !
41  !  Author:
42  !
43  !    FORTRAN90 version by John Burkardt
44  !
45  !  Reference:
46  !
47  !    Jack Dongarra, Jim Bunch, Cleve Moler, Pete Stewart,
48  !    LINPACK User's Guide,
49  !    SIAM, 1979
50  !
51  !  Parameters:
52  !
53  !    Input, integer N, the order of the matrix.
54  !    N must be positive.
55  !
56  !    Input, integer ML, MU, the lower and upper bandwidths.
57  !    ML and MU must be nonnegative, and no greater than N-1.
58  !
59  !    Input/output, real ( kind = 8 ) A(2*ML+MU+1,N), on input, the matrix
60  !    in band storage, on output, information about the LU factorization.
61  !
62  !    Output, integer PIVOT(N), the pivot vector.
63  !
64  !    Output, integer INFO, singularity flag.
65  !    0, no singularity detected.
66  !    nonzero, the factorization failed on the INFO-th step.
67  !
68    implicit none
69
70    integer ml
71    integer mu
72    integer n
73
74    real ( kind = 8 ) a(2*ml+mu+1,n)
75    integer i0, info, pivot(n), j, j0, j1, ju, jz, k, l, lm, m, mm
76    real ( kind = 8 ) temp
77
78    m = ml + mu + 1
79    info = 0
80  !
81  !  Zero out the initial fill-in columns.
82  !
83    j0 = mu + 2
84    j1 = min ( n, m ) - 1
85
86    do jz = j0, j1
87      i0 = m + 1 - jz
88      a(i0:ml,jz) = 0.0D+00
```

```
 89     end do
 90
 91     jz = j1
 92     ju = 0
 93
 94     do k = 1, n-1
 95   !
 96   !  Zero out the next fill-in column.
 97   !
 98       jz = jz + 1
 99       if ( jz <= n ) then
100         a(1:ml,jz) = 0.0D+00
101       end if
102   !
103   !  Find L = pivot index.
104   !
105       lm = min ( ml, n-k )
106
107       l = m
108       do j = m+1, m+lm
109         if ( abs ( a(l,k) ) < abs ( a(j,k) ) ) then
110           l = j
111         end if
112       end do
113
114       pivot(k) = l + k - m
115   !
116   !  Zero pivot implies this column already triangularized.
117   !
118       if ( a(l,k) == 0.0D+00 ) then
119         info = k
120         write ( *, '(a)' ) '␣'
121         write ( *, '(a)' ) 'DGB_FA␣-␣Fatal␣error!'
122         write ( *, '(a,i8)' ) '␣␣Zero␣pivot␣on␣step␣', info
123         return
124       end if
125   !
126   !  Interchange if necessary.
127   !
128       if ( m /= l ) then
129         temp   = a(l,k)
130         a(l,k) = a(m,k)
131         a(m,k) = temp
132       end if
133   !
134   !  Compute multipliers.
135   !
136       a(m+1:m+lm,k) = - a(m+1:m+lm,k) / a(m,k)
137   !
138   !  Row elimination with column indexing.
139   !
140       ju = max ( ju, mu+pivot(k) )
141       ju = min ( ju, n )
142       mm = m
143
144       do j = k+1, ju
145
146         l = l - 1
147         mm = mm - 1
148
149         if ( l /= mm ) then
150           temp    = a(l,j)
151           a(l,j)  = a(mm,j)
152           a(mm,j) = temp
153         end if
154
155         a(mm+1:mm+lm,j) = a(mm+1:mm+lm,j) + a(mm,j) * a(m+1:m+lm,k)
156
157       end do
158
159     end do
160
161     pivot(n) = n
162     if ( a(m,n) == 0.0D+00 ) then
163       info = n
164       write ( *, '(a)' ) '␣'
```

```
165       write ( *, '(a)' ) 'DGB_FA␣-␣Fatal␣error!'
166       write ( *, '(a,i8)' ) '␣␣Zero␣pivot␣on␣step␣', info
167     end if
168
169     return
170   end subroutine dgb_fa
```

B.3 Modified version

```
1    subroutine dgb_fa ( n, ml, mu, a, pivot, info, timerLen, timer, mode, &
2                       & extraArg1 )
3
4     use omp_lib
5     implicit none
6
7     ! -Calling
8     integer, intent(in) :: n
9     integer, intent(in) :: ml
10    integer, intent(in) :: mu
11    real ( kind = 8 ), intent(inout) ::  a(2*ml+mu+1,n)
12    integer, intent(out) :: pivot(n)
13    integer, intent(out) :: info
14
15    integer, intent(in) :: timerLen
16    double precision, intent(inout), dimension(timerLen) :: timer
17    character(len=*), intent(in) :: mode
18    character(len=*), intent(in) :: extraArg1
19
20    ! --Locals
21    integer i0, j ,j0, j1, ju, jz, k, l, ll, lm, m, mm, index
22    real ( kind = 8 ) temp
23
24    double precision :: wtime1, wtime2, wtime3 , wtime4, wtime5, wtime6
25
26    m = ml + mu + 1
27    info = 0
28    !
29    ! Zero out the  fill-in columns.
30    !
31    j0 = mu + 2
32    j1 = min ( n, m ) - 1 !Last initial fill in column.
33
34    wtime1 = omp_get_wtime()
35    !$omp parallel do IF(mode == 'parallel'),private(i0)
36    do jz = j0, n
37       i0 = m + 1 - jz
38       i0 = max(i0,1) !BG add: comb all zero out in one loop
39       a(i0:ml,jz) = 0.0D+00
40    end do
41    !$omp end parallel do
42    wtime2 = omp_get_wtime()
43    timer(1) = wtime2 - wtime1
44
45    jz = j1
46    ju = 0
47
48    timer(3) = 1.0
49    timer(4:6) = 0.00000000000000000
50    wtime5 = omp_get_wtime()
51
52    !$omp parallel IF(mode == 'parallel')
53    do k = 1, n-1 !loop over columns
54       !$omp single
55
56       !
57       ! Find L = pivot index.
58       !
59       wtime3 = omp_get_wtime()
60       lm = min ( ml, n-k )
61       l = m
62
63       do j = m+1, m+lm
64          if ( abs ( a(l,k) ) < abs ( a(j,k) ) ) then
65             l = j
66          end if
```

```
 67         end do
 68
 69         wtime4 = omp_get_wtime ()
 70         timer(4) = timer(4) + (wtime4 - wtime3)
 71
 72         pivot(k) = l + k - m
 73   !
 74   !  Zero pivot implies this column already triangularized.
 75   !
 76         if ( a(l,k) == 0.0D+00 ) then
 77            info = k
 78            write ( *, '(a)' ) '␣'
 79            write ( *, '(a)' ) 'DGB_FA␣-␣Fatal␣error!'
 80            write ( *, '(a,i8)' ) '␣␣Zero␣pivot␣on␣step␣', info
 81            !return
 82         end if
 83   !
 84   !  Interchange if necessary.
 85   !
 86         if ( m /= l ) then
 87            temp   = a(l,k)
 88            a(l,k) = a(m,k)
 89            a(m,k) = temp
 90         end if
 91   !
 92   !  Compute multipliers.
 93   !
 94         wtime3 = omp_get_wtime ()
 95         a(m+1:m+lm,k) = - a(m+1:m+lm,k) / a(m,k)
 96         wtime4 = omp_get_wtime ()
 97         timer(5) = timer(5) + (wtime4 - wtime3)
 98   !
 99   !  Row elimination with column indexing.
100   !
101         ju = max ( ju, mu+pivot(k) )
102         ju = min ( ju, n )
103         mm = m
104
105         wtime3 = omp_get_wtime ()
106         !$omp end single
107
108         !$omp do private(ll,mm,temp)
109         do j = k+1, ju
110            ll = l + k - j
111            mm = m + k - j
112            if ( ll /= mm ) then
113               temp    = a(ll,j)
114               a(ll,j) = a(mm,j)
115               a(mm,j) = temp
116            end if
117            a(mm+1:mm+lm,j) = a(mm+1:mm+lm,j) + a(mm,j) * a(m+1:m+lm,k)
118         end do
119         !$omp end do
120
121         !$omp single
122         wtime4 = omp_get_wtime ()
123         timer(6) = timer(6) + (wtime4 - wtime3)
124         !$omp end single
125
126      end do
127      !$omp end parallel
128
129      wtime6 = omp_get_wtime ()
130      timer(2) = wtime6 - wtime5
131      timer(7) = wtime6 - wtime1
132
133      pivot(n) = n
134      if ( a(m,n) == 0.0D+00 ) then
135         info = n
136         write ( *, '(a)' ) '␣'
137         write ( *, '(a)' ) 'DGB_FA␣-␣Fatal␣error!'
138         write ( *, '(a,i8)' ) '␣␣Zero␣pivot␣on␣step␣', info
139      end if
140
141      return
142   end subroutine dgb_fa
```

B.4 Runtime output

Legend:

1. *Ttotal* refers to the total execution time for a the entire program for a given proble size and a given number of processors.
2. *Tdgb* refers to the run time of the dgb-subroutine.
3. *TzeroOut* and *TloopK* design the execution time for two blocks of the dgb-subroutine.
4. *Tinner* is the run time for a subblock of the TloopK-block.
5. *BandWidth* refers to the total bandwidht of the banded matrix.

```
 1  June  4 2008   10:31:59.025 AM
 2   maxNp =  16
 3
 4    Dimensions are
 5
 6       NX =        10          NY =        10
 7    #nodes =      361  #elements =        162  BandWidth = 115
 8
 9     mode              Ttotal      Tdgb       TzeroOut    TloopK      Tinner      L2 error
10
11   sequential( 1)     2.204E-02   3.206E-03   1.478E-05   3.191E-03   1.586E-03   3.883E-04
12    Speedup           1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
13    Speedup/proc      1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
14  ------------------------------------------------------------------------------------------
15   parallel  ( 2)     2.069E-02   2.496E-03   4.196E-05   2.454E-03   1.745E-03   3.883E-04
16    Speedup           1.065E+00   1.285E+00   3.523E-01   1.301E+00   9.093E-01
17    Speedup/proc      5.327E-01   6.423E-01   1.761E-01   6.503E-01   4.546E-01
18  ------------------------------------------------------------------------------------------
19   parallel  ( 4)     2.130E-02   2.771E-03   3.147E-05   2.739E-03   1.822E-03   3.883E-04
20    Speedup           1.034E+00   1.157E+00   4.697E-01   1.165E+00   8.706E-01
21    Speedup/proc      2.586E-01   2.893E-01   1.174E-01   2.913E-01   2.176E-01
22  ------------------------------------------------------------------------------------------
23   parallel  ( 8)     2.251E-02   3.985E-03   5.386E-05   3.931E-03   2.560E-03   3.883E-04
24    Speedup           9.791E-01   8.045E-01   2.743E-01   8.118E-01   6.196E-01
25    Speedup/proc      1.224E-01   1.006E-01   3.429E-02   1.015E-01   7.745E-02
26  ------------------------------------------------------------------------------------------
27   parallel  (16)     2.284E-02   4.432E-03   9.465E-05   4.337E-03   2.901E-03   3.883E-04
28    Speedup           9.651E-01   7.235E-01   1.562E-01   7.358E-01   5.468E-01
29    Speedup/proc      6.032E-02   4.522E-02   9.761E-03   4.599E-02   3.418E-02
30  ------------------------------------------------------------------------------------------
31
32    Dimensions are
33
34       NX =        20          NY =        20
35    #nodes =     1521  #elements =        722  BandWidth = 235
36
37     mode              Ttotal      Tdgb       TzeroOut    TloopK      Tinner      L2 error
38
39   sequential( 1)     5.887E-02   2.411E-02   9.108E-05   2.402E-02   2.127E-02   4.114E-05
40    Speedup           1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
41    Speedup/proc      1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
42  ------------------------------------------------------------------------------------------
43   parallel  ( 2)     5.037E-02   1.927E-02   1.094E-04   1.916E-02   1.528E-02   4.114E-05
44    Speedup           1.169E+00   1.251E+00   8.322E-01   1.253E+00   1.392E+00
45    Speedup/proc      5.843E-01   6.255E-01   4.161E-01   6.267E-01   6.959E-01
46  ------------------------------------------------------------------------------------------
47   parallel  ( 4)     5.076E-02   1.596E-02   9.322E-05   1.586E-02   1.133E-02   4.114E-05
48    Speedup           1.160E+00   1.511E+00   9.770E-01   1.514E+00   1.877E+00
49    Speedup/proc      2.899E-01   3.777E-01   2.442E-01   3.785E-01   4.692E-01
50  ------------------------------------------------------------------------------------------
51   parallel  ( 8)     5.471E-02   1.985E-02   9.346E-05   1.976E-02   1.308E-02   4.114E-05
52    Speedup           1.076E+00   1.214E+00   9.745E-01   1.215E+00   1.626E+00
53    Speedup/proc      1.345E-01   1.518E-01   1.218E-01   1.519E-01   2.032E-01
54  ------------------------------------------------------------------------------------------
55   parallel  (16)     5.088E-02   1.966E-02   1.154E-04   1.954E-02   1.265E-02   4.114E-05
56    Speedup           1.157E+00   1.226E+00   7.893E-01   1.229E+00   1.681E+00
57    Speedup/proc      7.232E-02   7.665E-02   4.933E-02   7.682E-02   1.051E-01
58  ------------------------------------------------------------------------------------------
59
60    Dimensions are
61
62       NX =        40          NY =        40
63    #nodes =     6241  #elements =       3042  BandWidth = 475
64
65     mode              Ttotal      Tdgb       TzeroOut    TloopK      Tinner      L2 error
66
67   sequential( 1)     4.839E-01   3.209E-01   1.134E-03   3.198E-01   3.019E-01   4.754E-06
68    Speedup           1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
69    Speedup/proc      1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
70  ------------------------------------------------------------------------------------------
71   parallel  ( 2)     3.260E-01   1.947E-01   8.478E-04   1.939E-01   1.715E-01   4.754E-06
72    Speedup           1.484E+00   1.648E+00   1.338E+00   1.650E+00   1.760E+00
73    Speedup/proc      7.421E-01   8.241E-01   6.690E-01   8.248E-01   8.802E-01
74  ------------------------------------------------------------------------------------------
75   parallel  ( 4)     2.605E-01   1.294E-01   4.220E-04   1.290E-01   1.032E-01   4.754E-06
76    Speedup           1.858E+00   2.480E+00   2.688E+00   2.479E+00   2.925E+00
77    Speedup/proc      4.644E-01   6.200E-01   6.720E-01   6.199E-01   7.313E-01
78  ------------------------------------------------------------------------------------------
79   parallel  ( 8)     2.485E-01   1.173E-01   3.529E-04   1.169E-01   8.342E-02   4.754E-06
80    Speedup           1.947E+00   2.737E+00   3.215E+00   2.735E+00   3.619E+00
81    Speedup/proc      2.434E-01   3.421E-01   4.019E-01   3.419E-01   4.524E-01
82  ------------------------------------------------------------------------------------------
83   parallel  (16)     2.339E-01   1.024E-01   3.510E-04   1.021E-01   6.842E-02   4.754E-06
84    Speedup           2.069E+00   3.134E+00   3.232E+00   3.133E+00   4.412E+00
85    Speedup/proc      1.293E-01   1.959E-01   2.020E-01   1.958E-01   2.758E-01
86  ------------------------------------------------------------------------------------------
87
88    Dimensions are
89
90       NX =        80          NY =        80
91    #nodes =    25281  #elements =      12482  BandWidth = 955
92
93     mode              Ttotal      Tdgb       TzeroOut    TloopK      Tinner      L2 error
94
95   sequential( 1)     6.019E+00   5.137E+00   1.242E-02   5.125E+00   4.998E+00   5.718E-07
96    Speedup           1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
```

```
 97   Speedup/proc       1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
 98  ------------------------------------------------------------------------------------
 99  parallel   ( 2)     3.187E+00   2.540E+00   6.091E-03   2.534E+00   2.388E+00   5.718E-07
100   Speedup            1.889E+00   2.023E+00   2.040E+00   2.023E+00   2.093E+00
101   Speedup/proc       9.443E-01   1.011E+00   1.020E+00   1.011E+00   1.046E+00
102  ------------------------------------------------------------------------------------
103  parallel   ( 4)     2.079E+00   1.435E+00   3.159E-03   1.432E+00   1.271E+00   5.718E-07
104   Speedup            2.894E+00   3.581E+00   3.933E+00   3.580E+00   3.932E+00
105   Speedup/proc       7.236E-01   8.952E-01   9.833E-01   8.950E-01   9.831E-01
106  ------------------------------------------------------------------------------------
107  parallel   ( 8)     1.614E+00   9.707E-01   1.744E-03   9.690E-01   7.760E-01   5.718E-07
108   Speedup            3.728E+00   5.293E+00   7.126E+00   5.289E+00   6.441E+00
109   Speedup/proc       4.660E-01   6.616E-01   8.908E-01   6.612E-01   8.051E-01
110  ------------------------------------------------------------------------------------
111  parallel   (16)     1.372E+00   7.278E-01   1.154E-03   7.266E-01   5.226E-01   5.718E-07
112   Speedup            4.388E+00   7.059E+00   1.077E+01   7.053E+00   9.564E+00
113   Speedup/proc       2.742E-01   4.412E-01   6.731E-01   4.408E-01   5.977E-01
114  ------------------------------------------------------------------------------------
115
116   Dimensions are
117
118      NX =        160          NY =        160
119   #nodes =    101761   #elements =     50562   BandWidth = 1915
120
121    mode               Ttotal      Tdgb        TzeroOut    TloopK      Tinner      L2 error
122
123  sequential( 1)      1.012E+02   9.611E+01   9.094E-02   9.602E+01   9.503E+01   7.014E-08
124   Speedup            1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
125   Speedup/proc       1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
126  ------------------------------------------------------------------------------------
127  parallel   ( 2)     4.801E+01   4.479E+01   4.657E-02   4.475E+01   4.369E+01   7.014E-08
128   Speedup            2.107E+00   2.146E+00   1.953E+00   2.146E+00   2.175E+00
129   Speedup/proc       1.054E+00   1.073E+00   9.763E-01   1.073E+00   1.088E+00
130  ------------------------------------------------------------------------------------
131  parallel   ( 4)     2.357E+01   2.034E+01   2.834E-02   2.032E+01   1.924E+01   7.014E-08
132   Speedup            4.292E+00   4.724E+00   3.209E+00   4.727E+00   4.939E+00
133   Speedup/proc       1.073E+00   1.181E+00   8.022E-01   1.182E+00   1.235E+00
134  ------------------------------------------------------------------------------------
135  parallel   ( 8)     1.441E+01   1.121E+01   1.555E-02   1.119E+01   9.986E+00   7.014E-08
136   Speedup            7.020E+00   8.577E+00   5.849E+00   8.581E+00   9.517E+00
137   Speedup/proc       8.775E-01   1.072E+00   7.312E-01   1.073E+00   1.190E+00
138  ------------------------------------------------------------------------------------
139  parallel   (16)     1.036E+01   7.122E+00   1.486E-02   7.107E+00   5.816E+00   7.014E-08
140   Speedup            9.765E+00   1.350E+01   6.120E+00   1.351E+01   1.634E+01
141   Speedup/proc       6.103E-01   8.435E-01   3.825E-01   8.444E-01   1.021E+00
142  ------------------------------------------------------------------------------------
143
144   Dimensions are
145
146      NX =        320          NY =        320
147   #nodes =    408321   #elements =    203522   BandWidth = 3835
148
149    mode               Ttotal      Tdgb        TzeroOut    TloopK      Tinner      L2 error
150
151  sequential( 1)      1.642E+03   1.610E+03   6.228E-01   1.610E+03   1.602E+03   8.685E-09
152   Speedup            1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
153   Speedup/proc       1.000E+00   1.000E+00   1.000E+00   1.000E+00   1.000E+00
154  ------------------------------------------------------------------------------------
155  parallel   ( 2)     7.906E+02   7.743E+02   3.750E-01   7.739E+02   7.660E+02   8.685E-09
156   Speedup            2.077E+00   2.080E+00   1.661E+00   2.080E+00   2.092E+00
157   Speedup/proc       1.038E+00   1.040E+00   8.303E-01   1.040E+00   1.046E+00
158  ------------------------------------------------------------------------------------
159  parallel   ( 4)     3.973E+02   3.807E+02   2.000E-01   3.805E+02   3.725E+02   8.685E-09
160   Speedup            4.133E+00   4.230E+00   3.114E+00   4.231E+00   4.301E+00
161   Speedup/proc       1.033E+00   1.058E+00   7.785E-01   1.058E+00   1.075E+00
162  ------------------------------------------------------------------------------------
163  parallel   ( 8)     2.056E+02   1.890E+02   1.335E-01   1.888E+02   1.805E+02   8.685E-09
164   Speedup            7.986E+00   8.521E+00   4.665E+00   8.524E+00   8.874E+00
165   Speedup/proc       9.983E-01   1.065E+00   5.831E-01   1.065E+00   1.109E+00
166  ------------------------------------------------------------------------------------
167  parallel   (16)     1.243E+02   1.079E+02   1.206E-01   1.078E+02   9.895E+01   8.685E-09
168   Speedup            1.321E+01   1.492E+01   5.164E+00   1.493E+01   1.619E+01
169   Speedup/proc       8.256E-01   9.323E-01   3.227E-01   9.330E-01   1.012E+00
170  ------------------------------------------------------------------------------------
171
172  FEM2D_POISSON:
173    Normal end of execution.
174
175  June  4 2008  11:28:12.311 AM
```

C Simra

C.1 Profile

C.1.1 Call graph

An extract of the call graph obtained by the profiler is presented here. In total 155 different functions and subroutine are called during the simulation, including system calls and intrinsic Fortran functions. The total time spend in a routine/function is the "self-time" plus the time spend in the other units called from that spesific unit. There are only 17 of the 155 units that are listed with at least 1% of the runtime as their total time. In this appendix it is only the partial call graph for these 17 units that is presented. In the appendix B.1 a complete profiler listing is presented for the basic poisson problem. In that listing it is also included an explication of the terms used.

Listing 16: Extract of the call graph of Simra

```
ngranularity: Each sample hit covers 4 bytes. Time: 125.45 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                0.00      123.66      1/1            .__start [2]
[1]     98.6    0.00      123.66      1          .main [1]
                0.00      120.28      1/1            .p2_algorithm [3]
                0.65        0.61      1/1            .__pressure_NMOD_storage_cppe [17]
                0.62        0.46      1/1            .storage_q1 [18]
                0.80        0.00      1/1            .__pressure_NMOD_assemp [20]
                0.05        0.15      1/1            .calcul [25]
                0.01        0.03      1/1            .input [36]
                0.00        0.00      6/88           .__malloc [78]

-----------------------------------------------

6.6s                                               <spontaneous>
[2]     98.6    0.00      123.66                 .__start [2]
                0.00      123.66      1/1            .main [1]

-----------------------------------------------

                0.00      120.28      1/1            .main [1]
[3]     95.9    0.00      120.28      1          .p2_algorithm [3]
                1.24       35.03    100/100          .__pressure_NMOD_dpres [5]
               11.38       22.16    100/100          .__momentum_NMOD_veloc [7]
               12.26       14.87    100/100          .__turbke_NMOD_turb2 [9]
                3.34        8.69    100/100          .__turbke_NMOD_assemk [11]
                2.52        7.45    100/100          .__momentum_NMOD_assemu [12]
                0.98        0.31    100/100          .__momentum_NMOD_upcor [16]
                0.01        0.04      2/2            .result [29]
                0.00        0.00    100/100          .time_step [77]
                0.00        0.00      3/88           .__malloc [78]

-----------------------------------------------

                1.60       13.17    200/500          .__turbke_NMOD_turb2 [9]
                2.40       19.76    300/500          .__momentum_NMOD_veloc [7]
[4]     29.4    4.00       32.93    500          .__solver_NMOD_bicgstab2 [4]
               32.35        0.00   3726/3726         .__solver_NMOD_matvec [8]
                0.58        0.00   2112/2112         .__solver_NMOD_norm [22]

-----------------------------------------------

                1.24       35.03    100/100          .p2_algorithm [3]
[5]     28.9    1.24       35.03    100          .__pressure_NMOD_dpres [5]
               15.50       19.53    100/100          .__solver_NMOD_pcg [6]

-----------------------------------------------

               15.50       19.53    100/100          .__pressure_NMOD_dpres [5]
[6]     27.9   15.50       19.53    100          .__solver_NMOD_pcg [6]
               19.53        0.00   1701/1701         .__solver_NMOD_pssor [10]

-----------------------------------------------

               11.38       22.16    100/100          .p2_algorithm [3]
[7]     26.7   11.38       22.16    100          .__momentum_NMOD_veloc [7]
                2.40       19.76    300/500          .__solver_NMOD_bicgstab2 [4]

-----------------------------------------------

               32.35        0.00   3726/3726         .__solver_NMOD_bicgstab2 [4]
[8]     25.8   32.35        0.00   3726          .__solver_NMOD_matvec [8]

-----------------------------------------------

               12.26       14.87    100/100          .p2_algorithm [3]
[9]     21.6   12.26       14.87    100          .__turbke_NMOD_turb2 [9]
                1.60       13.17    200/500          .__solver_NMOD_bicgstab2 [4]
                0.03        0.02  240100/324135      .__pow [28]
                0.03        0.00  240100/240100      .__log [38]
                0.02        0.00  240100/240100      .__exp [42]

-----------------------------------------------

               19.53        0.00   1701/1701         .__solver_NMOD_pcg [6]
[10]    15.6   19.53        0.00   1701          .__solver_NMOD_pssor [10]

-----------------------------------------------

                3.34        8.69    100/100          .p2_algorithm [3]
[11]     9.6    3.34        8.69    100          .__turbke_NMOD_assemk [11]
                8.69        0.00 7833600/7833600     .__turbke_NMOD_elmat_k [13]

-----------------------------------------------

                2.52        7.45    100/100          .p2_algorithm [3]
[12]     7.9    2.52        7.45    100          .__momentum_NMOD_assemu [12]
                7.45        0.00 7833600/7833600     .__momentum_NMOD_elmat_u [14]
```

```
-----------------------------------------------------------
                8.69          0.00 7833600/7833600      .__turbke_NMOD_assemk [11]
[13]      6.9   8.69          0.00 7833600            .__turbke_NMOD_elmat_k [13]
-----------------------------------------------------------
                7.45          0.00 7833600/7833600      .__momentum_NMOD_assemu [12]
[14]      5.9   7.45          0.00 7833600            .__momentum_NMOD_elmat_u [14]
-----------------------------------------------------------
6.6s                                                     <spontaneous>
[15]      1.1   1.32          0.00                    .__mcount [15]
-----------------------------------------------------------
                0.98          0.31     100/100          .p2_algorithm [3]
[16]      1.0   0.98          0.31     100            .__momentum_NMOD_upcor [16]
                0.31          0.00     100/100          .__momentum_NMOD_couran [24]
-----------------------------------------------------------
                0.65          0.61       1/1            .main [1]
[17]      1.0   0.65          0.61       1            .__pressure_NMOD_storage_cppe [17]
                0.03          0.40   78336/162371       .__sorting_NMOD_sortheap [19]
                0.15          0.02       1/1            .__macro_el_NMOD_elnab [26]
                0.01          0.00   78336/78336        .__pressure_NMOD_elmat_p [55]
                0.00          0.00       1/1            .__pressure_NMOD_calc_p [63]
                0.00          0.00     13/88            .__malloc [78]
-----------------------------------------------------------
```

C.2 Timer results

The same model was used with three different resoulutions with respectively 84035, 173225 and 339521 number of points in the mesh. This sections gives the run time output from the detailed timers and the corresponding speedups.

C.2.1 Smal size data set

Listing 17: Run time output for the smal size data set

```
Core values
npoin  nelem   nz_u     nz_p   niter  nstep    eps       dl
84035  78336  2268945  2115072  200    200   1.00E-03  5.00E-02

*******************************************************************************

Run times and Speed Up in MAIN
mode     Ttotal       Tinput       Tcalcul       Tcppe        TassemP       Tq1          Tp2
--------------------------------------------------------------------------------------------

  nP =  1
runTime 2.3446E+02  2.2902E-01  2.1683E-01  1.2256E+00  7.8764E-01  1.1583E+00  2.3084E+02
% Ttot  100.00        0.10        0.09        0.52        0.34        0.49       98.46

Speedup    1.00       1.00        1.00        1.00        1.00        1.00        1.00
Speedup/P  1.00       1.00        1.00        1.00        1.00        1.00        1.00
--------------------------------------------------------------------------------------------

  nP =  2
runTime 1.5826E+02  2.4952E-01  2.1631E-01  1.1931E+00  7.6815E-01  1.1108E+00  1.5472E+02
% Ttot  100.00        0.16        0.14        0.75        0.49        0.70       97.76

Speedup    1.48       0.92        1.00        1.03        1.03        1.04        1.49
Speedup/P  0.74       0.46        0.50        0.51        0.51        0.52        0.75
--------------------------------------------------------------------------------------------

  nP =  4
runTime  1.2638E+02  1.9398E-01  2.0503E-01  1.1802E+00  7.6662E-01  1.1354E+00  1.2290E+02
% Ttot   100.00        0.15        0.16        0.93        0.61        0.90       97.25

Speedup    1.86       1.18        1.06        1.04        1.03        1.02        1.88
Speedup/P  0.46       0.30        0.26        0.26        0.26        0.26        0.47
--------------------------------------------------------------------------------------------

  nP =  8
runTime  1.1699E+02  1.9002E-01  2.0882E-01  1.1769E+00  7.6973E-01  1.1367E+00  1.1351E+02
% Ttot   100.00        0.16        0.18        1.01        0.66        0.97       97.02

Speedup    2.00       1.21        1.04        1.04        1.02        1.02        2.03
Speedup/P  0.25       0.15        0.13        0.13        0.13        0.13        0.25
--------------------------------------------------------------------------------------------

  nP = 16
runTime  1.0636E+02  1.8949E-01  2.0456E-01  1.1726E+00  7.6878E-01  1.1461E+00  1.0288E+02
% Ttot   100.00        0.18        0.19        1.10        0.72        1.08       96.73

Speedup    2.20       1.21        1.06        1.05        1.02        1.01        2.24
Speedup/P  0.14       0.08        0.07        0.07        0.06        0.06        0.14
--------------------------------------------------------------------------------------------
```

Run times and Speedup in P2–algo

measure	Tp2alg	TassemU	Tveloc	Tdpres	TassemK	Tturb2
nP = 1						
runTime	2.3084E+02	2.6016E+01	8.5119E+01	3.8619E+01	2.4118E+01	5.4038E+01
% of Tp2alg	100.00	11.27	36.87	16.73	10.45	23.41
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2						
runTime	1.5472E+02	1.4191E+01	5.0683E+01	2.8422E+01	2.3828E+01	3.4761E+01
% of Tp2alg	100.00	9.17	32.76	18.37	15.40	22.47
Speedup	1.49	1.83	1.68	1.36	1.01	1.55
Speedup/P	0.75	0.92	0.84	0.68	0.51	0.78
nP = 4						
runTime	1.2290E+02	1.1164E+01	3.2952E+01	3.2143E+01	2.3477E+01	2.0329E+01
% of Tp2alg	100.00	9.08	26.81	26.15	19.10	16.54
Speedup	1.88	2.33	2.58	1.20	1.03	2.66
Speedup/P	0.47	0.58	0.65	0.30	0.26	0.66
nP = 8						
runTime	1.1351E+02	1.1924E+01	3.0639E+01	2.6849E+01	2.3779E+01	1.7504E+01
% of Tp2alg	100.00	10.50	26.99	23.65	20.95	15.42
Speedup	2.03	2.18	2.78	1.44	1.01	3.09
Speedup/P	0.25	0.27	0.35	0.18	0.13	0.39
nP = 16						
runTime	1.0288E+02	1.2128E+01	2.7207E+01	2.0631E+01	2.3265E+01	1.6827E+01
% of Tp2alg	100.00	11.79	26.45	20.05	22.61	16.36
Speedup	2.24	2.15	3.13	1.87	1.04	3.21
Speedup/P	0.14	0.13	0.20	0.12	0.06	0.20

****

Run times and Speedup in veloc (called from p2algo)

measure	Tveloc	Trhs	Tsolve	(TdU	TdV	TdW)
nP = 1						
runTime	8.5118E+01	2.9937E+01	5.5181E+01	1.8483E+01	1.7912E+01	1.8398E+01
% of Tveloc	100.00	35.17	64.83	21.71	21.04	21.61
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2						
runTime	5.0682E+01	1.9918E+01	3.0764E+01	9.9851E+00	9.9881E+00	1.0431E+01
% of Tveloc	100.00	39.30	60.70	19.70	19.71	20.58
Speedup	1.68	1.50	1.79	1.85	1.79	1.76
Speedup/P	0.84	0.75	0.90	0.93	0.90	0.88
nP = 4						
runTime	3.2951E+01	1.5300E+01	1.7651E+01	5.6101E+00	5.6695E+00	5.9952E+00
% of Tveloc	100.00	46.43	53.57	17.03	17.21	18.19
Speedup	2.58	1.96	3.13	3.29	3.16	3.07
Speedup/P	0.65	0.49	0.78	0.82	0.79	0.77
nP = 8						
runTime	3.0638E+01	1.9122E+01	1.1516E+01	3.6134E+00	3.6749E+00	3.8306E+00
% of Tveloc	100.00	62.41	37.59	11.79	11.99	12.50
Speedup	2.78	1.57	4.79	5.12	4.87	4.80
Speedup/P	0.35	0.20	0.60	0.64	0.61	0.60
nP = 16						
runTime	2.7207E+01	1.8690E+01	8.5166E+00	2.6735E+00	2.6607E+00	2.7882E+00
% of Tveloc	100.00	68.70	31.30	9.83	9.78	10.25
Speedup	3.13	1.60	6.48	6.91	6.73	6.60
Speedup/P	0.20	0.10	0.40	0.43	0.42	0.41

Run times and Speedup in turb2 (called from p2algo)

measure	Tturb2	Tinit	Trhs	TdK	TdE
nP = 1					
runTime	5.4036E+01	1.3724E+00	2.5839E+01	1.5676E+01	1.0866E+01
% of Tturb2	100.00	2.54	47.82	29.01	20.11
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	3.4760E+01	1.3627E+00	1.4377E+01	1.0788E+01	7.9523E+00
% of Tturb2	100.00	3.92	41.36	31.03	22.88
Speedup	1.55	1.01	1.80	1.45	1.37
Speedup/P	0.78	0.50	0.90	0.73	0.68
nP = 4					
runTime	2.0327E+01	1.3669E+00	7.6350E+00	6.5408E+00	4.5038E+00
% of Tturb2	100.00	6.72	37.56	32.18	22.16
Speedup	2.66	1.00	3.38	2.40	2.41
Speedup/P	0.66	0.25	0.85	0.60	0.60
nP = 8					
runTime	1.7503E+01	1.3621E+00	8.4215E+00	4.5176E+00	2.9216E+00
% of Tturb2	100.00	7.78	48.12	25.81	16.69
Speedup	3.09	1.01	3.07	3.47	3.72
Speedup/P	0.39	0.13	0.38	0.43	0.46
nP = 16					
runTime	1.6826E+01	1.3673E+00	9.2649E+00	3.6418E+00	2.2547E+00
% of Tturb2	100.00	8.13	55.06	21.64	13.40
Speedup	3.21	1.00	2.79	4.30	4.82
Speedup/P	0.20	0.06	0.17	0.27	0.30

****

Run times and Speedup in bicgstab (called from dU,dV,dW,dK,dE).

measure	Tbicgstb	Tinit	TinitMv	Tloop-MV	TloopMv
nP = 1					
runTime	7.9844E+01	9.9050E-01	9.2858E+00	1.0078E+01	5.9500E+01
% of Tbicg	100.00	1.24	11.63	12.62	74.52
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	4.7811E+01	6.6485E-01	4.6451E+00	8.4243E+00	3.4082E+01
% of Tbicg	100.00	1.39	9.72	17.62	71.28
Speedup	1.67	1.49	2.00	1.20	1.75
Speedup/P	0.83	0.74	1.00	0.60	0.87
nP = 4					
runTime	2.7149E+01	4.9756E-01	2.3175E+00	7.0610E+00	1.7276E+01
% of Tbicg	100.00	1.83	8.54	26.01	63.63
Speedup	2.94	1.99	4.01	1.43	3.44
Speedup/P	0.74	0.50	1.00	0.36	0.86
nP = 8					
runTime	1.7446E+01	4.2696E-01	1.2132E+00	6.7930E+00	9.0142E+00
% of Tbicg	100.00	2.45	6.95	38.94	51.67
Speedup	4.58	2.32	7.65	1.48	6.60
Speedup/P	0.57	0.29	0.96	0.19	0.83
nP = 16					
runTime	1.2906E+01	4.2115E-01	8.1815E-01	6.7369E+00	4.9304E+00
% of Tbicg	100.00	3.26	6.34	52.20	38.20
Speedup	6.19	2.35	11.35	1.50	12.07
Speedup/P	0.39	0.15	0.71	0.09	0.75

Run times and Speedup In matvec (called from bicgstab)

measure	Tmatvec
nP = 1	
runTime	6.8786E+01
Speedup	1.00
Speedup/P	1.00
nP = 2	
runTime	3.8727E+01
Speedup	1.78
Speedup/P	0.89
nP = 4	
runTime	1.9593E+01
Speedup	3.51
Speedup/P	0.88
nP = 8	
runTime	1.0227E+01
Speedup	6.73
Speedup/P	0.84
nP = 16	
runTime	5.7485E+00
Speedup	11.97
Speedup/P	0.75

****

Run times and Speedup In pcg (called from dpres)

measure	Tpcg	Trhs	Tloop	Tpssor	TrestPC	TmatVect	Tcoef
nP = 1							
runTime	3.6164E+01	2.0605E+00	3.4103E+01	1.6963E+01	7.0161E−01	1.5153E+01	1.2835E+00
% Tpcg	100.00	5.70	94.30	46.91	1.94	41.90	3.55
Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2							
runTime	2.6121E+01	1.0760E+00	2.5045E+01	1.6453E+01	3.8350E−01	7.5492E+00	6.5784E−01
% Tpcg	100.00	4.12	95.88	62.99	1.47	28.90	2.52
Speedup	1.38	1.91	1.36	1.03	1.83	2.01	1.95
Speedup/P	0.69	0.96	0.68	0.52	0.91	1.00	0.98
nP = 4							
runTime	2.9791E+01	5.6729E−01	2.9224E+01	2.3096E+01	3.3170E−01	5.3106E+00	4.8340E−01
% Tpcg	100.00	1.90	98.10	77.53	1.11	17.83	1.62
Speedup	1.21	3.63	1.17	0.73	2.12	2.85	2.66
Speedup/P	0.30	0.91	0.29	0.18	0.53	0.71	0.66
nP = 8							
runTime	2.4471E+01	3.0653E−01	2.4164E+01	2.1170E+01	2.6040E−01	2.4518E+00	2.7957E−01
% Tpcg	100.00	1.25	98.75	86.51	1.06	10.02	1.14
Speedup	1.48	6.72	1.41	0.80	2.69	6.18	4.59
Speedup/P	0.18	0.84	0.18	0.10	0.34	0.77	0.57
nP = 16							
runTime	1.8242E+01	2.0997E−01	1.8032E+01	1.6599E+01	2.9755E−01	9.8368E−01	1.5097E−01
% Tpcg	100.00	1.15	98.85	90.99	1.63	5.39	0.83
Speedup	1.98	9.81	1.89	1.02	2.36	15.40	8.50
Speedup/P	0.12	0.61	0.12	0.06	0.15	0.96	0.53

Run times and Speedup in pssor (called from pcg).

measure	Tpssor	Tforward	Tbackwrd
nP = 1			
runTime	1.6962E+01	8.1157E+00	8.8461E+00
% of Tpssor	100.00	47.85	52.15
Speedup	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00
nP = 2			
runTime	1.6451E+01	8.0087E+00	8.4424E+00
% of Tpssor	100.00	48.68	51.32
Speedup	1.03	1.01	1.05
Speedup/P	0.52	0.51	0.52
nP = 4			
runTime	2.3094E+01	1.1192E+01	1.1902E+01
% of Tpssor	100.00	48.46	51.54
Speedup	0.73	0.73	0.74
Speedup/P	0.18	0.18	0.19
nP = 8			
runTime	2.1169E+01	1.0254E+01	1.0914E+01
% of Tpssor	100.00	48.44	51.56
Speedup	0.80	0.79	0.81
Speedup/P	0.10	0.10	0.10
nP = 16			
runTime	1.6597E+01	8.1176E+00	8.4795E+00
% of Tpssor	100.00	48.91	51.09
Speedup	1.02	1.00	1.04
Speedup/P	0.06	0.06	0.07

****

Residuals for the last time step.

residual	dU	dV	dW	dP	dK	dE
nP = 1	1.9540E-05	1.3674E-05	2.9765E-05	6.9696E-04	1.6023E-05	2.1428E-04
nP = 2	1.5267E-04	1.2032E-04	1.8982E-04	6.8967E-04	1.2792E-04	7.5973E-04
nP = 4	1.5422E-04	1.2398E-04	1.9243E-04	6.9857E-04	1.2640E-04	7.6308E-04
nP = 8	1.5007E-04	1.5920E-04	1.9930E-04	6.9257E-04	1.2527E-04	7.6337E-04
nP = 16	1.3732E-04	1.0257E-04	2.8165E-04	6.3466E-04	1.4747E-04	7.6918E-04

C.2.2 Medium size data set

Listing 18: Run time output for the medium size data set

```
Core values
npoin   nelem   nz_u     nz_p     niter  nstep  eps       dt
173225  163840  4677075  4423680  200    200    1.00E-03  5.00E-02
*****************************************************************************************
 Run times and Speed Up in MAIN
mode      Ttotal     Tinput      Tcalcul     Tcppe       TassemP     Tq1         Tp2
---------------------------------------------------------------------------------------
 nP =  1
runTime   5.6898E+02 4.8176E-01 4.5468E-01 2.5984E+00 1.7598E+00 2.4302E+00 5.6125E+02
% Ttot    100.00          0.08        0.08        0.46        0.31        0.43       98.64

Speedup     1.00          1.00        1.00        1.00        1.00        1.00        1.00
Speedup/P   1.00          1.00        1.00        1.00        1.00        1.00        1.00
---------------------------------------------------------------------------------------
 nP =  2
runTime   3.7464E+02 4.0930E-01 4.4625E-01 2.4919E+00 1.7076E+00 2.3281E+00 3.6726E+02
% Ttot    100.00          0.11        0.12        0.67        0.46        0.62       98.03

Speedup     1.52          1.18        1.02        1.04        1.03        1.04        1.53
Speedup/P   0.76          0.59        0.51        0.52        0.52        0.52        0.76
---------------------------------------------------------------------------------------
 nP =  4
runTime   2.8510E+02 3.8252E-01 4.2869E-01 2.4901E+00 1.7040E+00 2.3852E+00 2.7771E+02
% Ttot    100.00          0.13        0.15        0.87        0.60        0.84       97.41

Speedup     2.00          1.26        1.06        1.04        1.03        1.02        2.02
Speedup/P   0.50          0.31        0.27        0.26        0.26        0.25        0.51
---------------------------------------------------------------------------------------
 nP =  8
runTime   2.6038E+02 3.8284E-01 4.3703E-01 2.4859E+00 1.7019E+00 2.3832E+00 2.5299E+02
% Ttot    100.00          0.15        0.17        0.95        0.65        0.92       97.16

Speedup     2.19          1.26        1.04        1.05        1.03        1.02        2.22
Speedup/P   0.27          0.16        0.13        0.13        0.13        0.13        0.28
---------------------------------------------------------------------------------------
 nP = 16
runTime   2.4694E+02 3.8240E-01 4.2803E-01 2.4857E+00 1.7485E+00 2.3826E+00 2.3952E+02
% Ttot    100.00          0.15        0.17        1.01        0.71        0.96       96.99

Speedup     2.30          1.26        1.06        1.05        1.01        1.02        2.34
Speedup/P   0.14          0.08        0.07        0.07        0.06        0.06        0.15
---------------------------------------------------------------------------------------

*****************************************************************************************
 Run times and Speedup in P2-algo.
   measure      Tp2alg      TassemU     Tveloc      Tdpres      TassemK     Tturb2
---------------------------------------------------------------------------------------
  nP =  1
     runTime   5.6125E+02  6.0230E+01  1.9904E+02  1.0542E+02  5.5615E+01  1.3441E+02
 % of Tp2alg       100.00       10.73       35.46       18.78        9.91       23.95
     Speedup         1.00        1.00        1.00        1.00        1.00        1.00
   Speedup/P         1.00        1.00        1.00        1.00        1.00        1.00
---------------------------------------------------------------------------------------
  nP =  2
     runTime   3.6726E+02  3.3991E+01  1.1600E+02  7.8750E+01  5.4302E+01  7.7870E+01
 % of Tp2alg       100.00        9.26       31.59       21.44       14.79       21.20
     Speedup         1.53        1.77        1.72        1.34        1.02        1.73
   Speedup/P         0.76        0.89        0.86        0.67        0.51        0.86
---------------------------------------------------------------------------------------
  nP =  4
     runTime   2.7771E+02  2.4256E+01  7.9025E+01  6.6488E+01  5.3753E+01  4.7883E+01
 % of Tp2alg       100.00        8.73       28.46       23.94       19.36       17.24
     Speedup         2.02        2.48        2.52        1.59        1.03        2.81
   Speedup/P         0.51        0.62        0.63        0.40        0.26        0.70
---------------------------------------------------------------------------------------
  nP =  8
     runTime   2.5299E+02  2.5344E+01  6.8898E+01  5.9644E+01  5.3620E+01  3.9227E+01
 % of Tp2alg       100.00       10.02       27.23       23.58       21.19       15.51
     Speedup         2.22        2.38        2.89        1.77        1.04        3.43
   Speedup/P         0.28        0.30        0.36        0.22        0.13        0.43
---------------------------------------------------------------------------------------
  nP = 16
     runTime   2.3952E+02  2.5682E+01  6.0187E+01  5.5952E+01  5.4656E+01  3.6688E+01
 % of Tp2alg       100.00       10.72       25.13       23.36       22.82       15.32
     Speedup         2.34        2.35        3.31        1.88        1.02        3.66
   Speedup/P         0.15        0.15        0.21        0.12        0.06        0.23
---------------------------------------------------------------------------------------
```

Run times and Speedup in veloc (called from p2algo).

measure	Tveloc	Trhs	Tsolve	(TdU	TdV	TdW)
nP = 1						
runTime	1.9904E+02	6.6154E+01	1.3288E+02	4.3836E+01	4.3190E+01	4.4746E+01
% of Tveloc	100.00	33.24	66.76	22.02	21.70	22.48
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2						
runTime	1.1600E+02	4.4890E+01	7.1108E+01	2.2030E+01	2.1860E+01	2.6246E+01
% of Tveloc	100.00	38.70	61.30	18.99	18.84	22.63
Speedup	1.72	1.47	1.87	1.99	1.98	1.70
Speedup/P	0.86	0.74	0.93	0.99	0.99	0.85
nP = 4						
runTime	7.9024E+01	3.2976E+01	4.6048E+01	1.5342E+01	1.3354E+01	1.6474E+01
% of Tveloc	100.00	41.73	58.27	19.41	16.90	20.85
Speedup	2.52	2.01	2.89	2.86	3.23	2.72
Speedup/P	0.63	0.50	0.72	0.71	0.81	0.68
nP = 8						
runTime	6.8897E+01	4.0114E+01	2.8783E+01	9.5317E+00	7.9316E+00	1.0446E+01
% of Tveloc	100.00	58.22	41.78	13.83	11.51	15.16
Speedup	2.89	1.65	4.62	4.60	5.45	4.28
Speedup/P	0.36	0.21	0.58	0.57	0.68	0.54
nP = 16						
runTime	6.0186E+01	3.9265E+01	2.0922E+01	6.8947E+00	5.8874E+00	7.2661E+00
% of Tveloc	100.00	65.24	34.76	11.46	9.78	12.07
Speedup	3.31	1.68	6.35	6.36	7.34	6.16
Speedup/P	0.21	0.11	0.40	0.40	0.46	0.38

****

Run times and Speedup in turb2 (called from p2algo).

measure	Tturb2	Tinit	Trhs	TdK	TdE
nP = 1					
runTime	1.3441E+02	2.8239E+00	5.6957E+01	4.3489E+01	3.0560E+01
% of Tturb2	100.00	2.10	42.38	32.36	22.74
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	7.7868E+01	2.8192E+00	3.1409E+01	2.3031E+01	2.0037E+01
% of Tturb2	100.00	3.62	40.34	29.58	25.73
Speedup	1.73	1.00	1.81	1.89	1.53
Speedup/P	0.86	0.50	0.91	0.94	0.76
nP = 4					
runTime	4.7881E+01	2.8208E+00	1.6688E+01	1.5362E+01	1.2443E+01
% of Tturb2	100.00	5.89	34.85	32.08	25.99
Speedup	2.81	1.00	3.41	2.83	2.46
Speedup/P	0.70	0.25	0.85	0.71	0.61
nP = 8					
runTime	3.9226E+01	2.7990E+00	1.7623E+01	1.0481E+01	7.7568E+00
% of Tturb2	100.00	7.14	44.93	26.72	19.77
Speedup	3.43	1.01	3.23	4.15	3.94
Speedup/P	0.43	0.13	0.40	0.52	0.49
nP = 16					
runTime	3.6687E+01	2.8123E+00	1.9371E+01	8.2025E+00	5.7347E+00
% of Tturb2	100.00	7.67	52.80	22.36	15.63
Speedup	3.66	1.00	2.94	5.30	5.33
Speedup/P	0.23	0.06	0.16	0.33	0.33

Run times and Speedup in bicgstab (called from dU,dV,dW,dK,dE).

measure	Tbicgstb	Tinit	TinitMv	Tloop-MV	TloopMv
nP = 1					
runTime	2.0183E+02	2.2491E+00	2.0691E+01	2.5524E+01	1.5339E+02
% of Tbicg	100.00	1.11	10.25	12.65	76.00
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	1.0961E+02	1.3816E+00	1.0119E+01	1.9200E+01	7.8916E+01
% of Tbicg	100.00	1.26	9.23	17.52	72.00
Speedup	1.84	1.63	2.04	1.33	1.94
Speedup/P	0.92	0.81	1.02	0.66	0.97
nP = 4					
runTime	6.9802E+01	9.9881E-01	5.1109E+00	1.8179E+01	4.5519E+01
% of Tbicg	100.00	1.43	7.32	26.04	65.21
Speedup	2.89	2.25	4.05	1.40	3.37
Speedup/P	0.72	0.56	1.01	0.35	0.84
nP = 8					
runTime	4.3213E+01	8.4740E-01	2.5754E+00	1.6814E+01	2.2980E+01
% of Tbicg	100.00	1.96	5.96	38.91	53.18
Speedup	4.67	2.65	8.03	1.52	6.67
Speedup/P	0.58	0.33	1.00	0.19	0.83
nP = 16					
runTime	3.1073E+01	7.8313E-01	1.5326E+00	1.6447E+01	1.2312E+01
% of Tbicg	100.00	2.52	4.93	52.93	39.62
Speedup	6.50	2.87	13.50	1.55	12.46
Speedup/P	0.41	0.18	0.84	0.10	0.78

****

Run times and Speedup in matvec (called from bicgstab).

measure	Tmatvec
nP = 1	
runTime	1.7408E+02
Speedup	1.00
Speedup/P	1.00
nP = 2	
runTime	8.9035E+01
Speedup	1.96
Speedup/P	0.98
nP = 4	
runTime	5.0630E+01
Speedup	3.44
Speedup/P	0.86
nP = 8	
runTime	2.5555E+01
Speedup	6.81
Speedup/P	0.85
nP = 16	
runTime	1.3844E+01
Speedup	12.57
Speedup/P	0.79

Run times and Speedup in pcg (called from dpres).

measure	Tpcg	Trhs	Tloop	Tpssor	TrestPC	TmatVect	Tcoef
nP = 1							
runTime	1.0013E+02	4.4188E+00	9.5710E+01	5.0184E+01	1.9549E+00	3.9618E+01	3.9504E+00
% Tpcg	100.00	4.41	95.59	50.12	1.95	39.57	3.95
Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2							
runTime	7.3586E+01	2.3593E+00	7.1227E+01	4.9003E+01	1.0214E+00	1.9349E+01	1.8507E+00
% Tpcg	100.00	3.21	96.79	66.59	1.39	26.29	2.52
Speedup	1.36	1.87	1.34	1.02	1.91	2.05	2.13
Speedup/P	0.68	0.94	0.67	0.51	0.96	1.02	1.07
nP = 4							
runTime	6.1392E+01	1.2471E+00	6.0145E+01	4.8940E+01	5.7598E-01	9.6847E+00	9.4161E-01
% Tpcg	100.00	2.03	97.97	79.72	0.94	15.78	1.53
Speedup	1.63	3.54	1.59	1.03	3.39	4.09	4.20
Speedup/P	0.41	0.89	0.40	0.26	0.85	1.02	1.05
nP = 8							
runTime	5.4679E+01	6.8295E-01	5.3996E+01	4.8266E+01	4.2932E-01	4.7975E+00	5.0134E-01
% Tpcg	100.00	1.25	98.75	88.27	0.79	8.7	0.92
Speedup	1.83	6.47	1.77	1.04	4.55	8.26	7.88
Speedup/P	0.23	0.81	0.22	0.13	0.57	1.03	0.98
nP = 16							
runTime	5.0880E+01	4.2686E-01	5.0453E+01	4.7182E+01	4.2951E-01	2.5117E+00	3.2788E-01
% Tpcg	100.00	0.84	99.16	92.73	0.84	4.94	0.64
Speedup	1.97	10.35	1.90	1.06	4.55	15.77	12.05
Speedup/P	0.12	0.65	0.12	0.07	0.28	0.99	0.75

***

Run times and Speedup in pssor (called from pcg)

measure	Tpssor	Tforward	Tbackwrd
nP = 1			
runTime	5.0181E+01	2.5567E+01	2.4614E+01
% of Tpssor	100.00	50.95	49.05
Speedup	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00
nP = 2			
runTime	4.8999E+01	2.3631E+01	2.5368E+01
% of Tpssor	100.00	48.23	51.77
Speedup	1.02	1.08	0.97
Speedup/P	0.51	0.54	0.49
nP = 4			
runTime	4.8937E+01	2.3490E+01	2.5447E+01
% of Tpssor	100.00	48.00	52.00
Speedup	1.03	1.09	0.97
Speedup/P	0.26	0.27	0.24
nP = 8			
runTime	4.8263E+01	2.3207E+01	2.5056E+01
% of Tpssor	100.00	48.08	51.92
Speedup	1.04	1.10	0.98
Speedup/P	0.13	0.14	0.12
nP = 16			
runTime	4.7179E+01	2.2323E+01	2.4856E+01
% of Tpssor	100.00	47.32	52.68
Speedup	1.06	1.15	0.99
Speedup/P	0.07	0.07	0.06

```
Residuals for the last time step.

 residual     dU          dV          dW          dP          dK          dE
--------------------------------------------------------------------------------
 nP =  1    4.9415E-04  1.0776E-04  5.0114E-04  8.7938E-04  1.2053E-04  5.6341E-04
--------------------------------------------------------------------------------
 nP =  2    7.2539E-04  7.1126E-04  1.7789E-04  8.8457E-04  4.5587E-04  8.8265E-05
--------------------------------------------------------------------------------
 nP =  4    1.3797E-04  7.9228E-04  1.7658E-04  8.7963E-04  6.2053E-04  1.0040E-04
--------------------------------------------------------------------------------
 nP =  8    1.3832E-04  9.3070E-04  2.6666E-04  9.3472E-04  6.1828E-04  1.3663E-04
--------------------------------------------------------------------------------
 nP = 16    1.2355E-04  7.7583E-04  1.7029E-04  8.8173E-04  2.7057E-04  1.1406E-04
--------------------------------------------------------------------------------
```

C.2.3 Large size data set

Listing 19: Run time output for the large size data set

```
 Core values

 npoin   nelem   nz_u     nz_p     niter  nstep    eps       dt
339521  324000  9167067  8748000    200    200   1.00E-03  5.00E-02

*********************************************************************************

 Run times and Speed Up in MAIN


 mode    Ttotal     Tinput     Tcalcul    Tcppe      TassemP    Tq1        Tp2
--------------------------------------------------------------------------------
  nP =  1
runTime 1.1785E+03 8.2021E-01 9.0001E-01 5.1696E+00 3.2966E+00 4.8148E+00 1.1635E+03
% Ttot  100.00     0.07       0.08       0.44       0.28       0.41       98.73

Speedup   1.00     1.00       1.00       1.00       1.00       1.00        1.00
Speedup/P 1.00     1.00       1.00       1.00       1.00       1.00        1.00
--------------------------------------------------------------------------------
  nP =  2
runTime 7.7897E+02 8.1574E-01 8.7131E-01 4.9582E+00 3.2795E+00 4.6129E+00 7.6443E+02
% Ttot  100.00     0.10       0.11       0.64       0.42       0.59       98.13

Speedup   1.51     1.01       1.03       1.04       1.01       1.04        1.52
Speedup/P 0.76     0.50       0.52       0.52       0.50       0.52        0.76
--------------------------------------------------------------------------------
  nP =  4
runTime 5.9188E+02 7.3772E-01 8.4874E-01 4.9581E+00 3.2766E+00 4.7258E+00 5.7733E+02
% Ttot  100.00     0.12       0.14       0.84       0.55       0.80       97.54

Speedup   1.99     1.11       1.06       1.04       1.01       1.02        2.02
Speedup/P 0.50     0.28       0.27       0.26       0.25       0.25        0.50
--------------------------------------------------------------------------------
  nP =  8
runTime 5.6151E+02 7.3707E-01 8.4855E-01 4.9564E+00 3.2745E+00 4.7235E+00 5.4697E+02
% Ttot  100.00     0.13       0.15       0.88       0.58       0.84       97.41

Speedup   2.10     1.11       1.06       1.04       1.01       1.02        2.13
Speedup/P 0.26     0.14       0.13       0.13       0.13       0.13        0.27
--------------------------------------------------------------------------------
  nP = 16
runTime 5.1903E+02 7.3438E-01 8.4839E-01 4.9486E+00 3.2793E+00 4.7253E+00 5.0449E+02
% Ttot  100.00     0.14       0.16       0.95       0.63       0.91       97.20

Speedup   2.27     1.12       1.06       1.04       1.01       1.02        2.31
Speedup/P 0.14     0.07       0.07       0.07       0.06       0.06        0.14
--------------------------------------------------------------------------------
```

Run times and Speedup in P2–algo.

measure	Tp2alg	TassemU	Tveloc	Tdpres	TassemK	Tturb2
nP = 1						
runTime	1.1635E+03	1.1100E+02	3.9621E+02	2.7422E+02	1.0229E+02	2.6751E+02
% of Tp2alg	100.00	9.54	34.05	23.57	8.79	22.99
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2						
runTime	7.6443E+02	6.4536E+01	2.3214E+02	1.9963E+02	1.0286E+02	1.5289E+02
% of Tp2alg	100.00	8.44	30.37	26.12	13.46	20.00
Speedup	1.52	1.72	1.71	1.37	0.99	1.75
Speedup/P	0.76	0.86	0.85	0.69	0.50	0.87
nP = 4						
runTime	5.7733E+02	4.6801E+01	1.6447E+02	1.5059E+02	1.0156E+02	1.0155E+02
% of Tp2alg	100.00	8.11	28.49	26.08	17.59	17.59
Speedup	2.02	2.37	2.41	1.82	1.01	2.63
Speedup/P	0.50	0.59	0.60	0.46	0.25	0.66
nP = 8						
runTime	5.4697E+02	4.9780E+01	1.4161E+02	1.5786E+02	1.0394E+02	8.1556E+01
% of Tp2alg	100.00	9.10	25.89	28.86	19.00	14.91
Speedup	2.13	2.23	2.80	1.74	0.98	3.28
Speedup/P	0.27	0.28	0.35	0.22	0.12	0.41
nP = 16						
runTime	5.0449E+02	5.0245E+01	1.2328E+02	1.3645E+02	1.0685E+02	7.5388E+01
% of Tp2alg	100.00	9.96	24.44	27.05	21.18	14.94
Speedup	2.31	2.21	3.21	2.01	0.96	3.55
Speedup/P	0.14	0.14	0.20	0.13	0.06	0.22

****

Run times and Speedup in veloc (called from p2algo).

measure	Tveloc	Trhs	Tsolve	(TdU	TdV	TdW)
nP = 1						
runTime	3.9620E+02	1.2535E+02	2.7085E+02	8.8763E+01	9.0082E+01	8.9875E+01
% of Tveloc	100.00	31.64	68.36	22.40	22.74	22.68
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2						
runTime	2.3214E+02	8.6263E+01	1.4588E+02	4.8698E+01	4.6770E+01	4.8392E+01
% of Tveloc	100.00	37.16	62.84	20.98	20.15	20.85
Speedup	1.71	1.45	1.86	1.82	1.93	1.86
Speedup/P	0.85	0.73	0.93	0.91	0.96	0.93
nP = 4						
runTime	1.6447E+02	6.6711E+01	9.7761E+01	3.3565E+01	2.9274E+01	3.2957E+01
% of Tveloc	100.00	40.56	59.44	20.41	17.80	20.04
Speedup	2.41	1.88	2.77	2.64	3.08	2.73
Speedup/P	0.60	0.47	0.69	0.66	0.77	0.68
nP = 8						
runTime	1.4161E+02	7.9650E+01	6.1959E+01	2.1041E+01	1.8290E+01	2.0751E+01
% of Tveloc	100.00	56.25	43.75	14.86	12.92	14.65
Speedup	2.80	1.57	4.37	4.22	4.93	4.33
Speedup/P	0.35	0.20	0.55	0.53	0.62	0.54
nP = 16						
runTime	1.2328E+02	7.8975E+01	4.4305E+01	1.4747E+01	1.3056E+01	1.4645E+01
% of Tveloc	100.00	64.06	35.94	11.96	10.59	11.88
Speedup	3.21	1.59	6.11	6.02	6.90	6.14
Speedup/P	0.20	0.10	0.38	0.38	0.43	0.38

Run times and Speedup in turb2 (called from p2algo).

measure	Tturb2	Tinit	Trhs	TdK	TdE
nP = 1					
runTime	2.6750E+02	5.6038E+00	1.0844E+02	8.9931E+01	6.2394E+01
% of Tturb2	100.00	2.09	40.54	33.62	23.32
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	1.5289E+02	5.5801E+00	6.2270E+01	5.0036E+01	3.3880E+01
% of Tturb2	100.00	3.65	40.73	32.73	22.16
Speedup	1.75	1.00	1.74	1.80	1.84
Speedup/P	0.87	0.50	0.87	0.90	0.92
nP = 4					
runTime	1.0154E+02	5.5860E+00	3.3479E+01	3.6069E+01	2.5297E+01
% of Tturb2	100.00	5.50	32.97	35.52	24.91
Speedup	2.63	1.00	3.24	2.49	2.47
Speedup/P	0.66	0.25	0.81	0.62	0.62
nP = 8					
runTime	8.1555E+01	5.5892E+00	3.4681E+01	2.4134E+01	1.6046E+01
% of Tturb2	100.00	6.85	42.53	29.59	19.68
Speedup	3.28	1.00	3.13	3.73	3.89
Speedup/P	0.41	0.13	0.39	0.47	0.49
nP = 16					
runTime	7.5387E+01	5.5946E+00	3.8329E+01	1.8777E+01	1.1580E+01
% of Tturb2	100.00	7.42	50.84	24.91	15.36
Speedup	3.55	1.00	2.83	4.79	5.39
Speedup/P	0.22	0.06	0.18	0.30	0.34

****

Run times and Speedup in bicgstab (called from dU,dV,dW,dK,dE).

measure	Tbicgstb	Tinit	TinitMv	Tloop-MV	TloopMv
nP = 1					
runTime	4.1330E+02	4.6506E+00	4.1569E+01	5.2428E+01	3.1470E+02
% of Tbicg	100.00	1.13	10.06	12.69	76.14
Speedup	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00
nP = 2					
runTime	2.2060E+02	2.8207E+00	2.0879E+01	3.7182E+01	1.5974E+02
% of Tbicg	100.00	1.28	9.46	16.85	72.41
Speedup	1.87	1.65	1.99	1.41	1.97
Speedup/P	0.94	0.82	1.00	0.71	0.99
nP = 4					
runTime	1.5054E+02	1.9474E+00	1.0420E+01	3.9352E+01	9.8835E+01
% of Tbicg	100.00	1.29	6.92	26.14	65.65
Speedup	2.75	2.39	3.99	1.33	3.18
Speedup/P	0.69	0.60	1.00	0.33	0.80
nP = 8					
runTime	9.4011E+01	1.6181E+00	5.2573E+00	3.6715E+01	5.0427E+01
% of Tbicg	100.00	1.72	5.59	39.05	53.64
Speedup	4.40	2.87	7.91	1.43	6.24
Speedup/P	0.55	0.36	0.99	0.18	0.78
nP = 16					
runTime	6.6719E+01	1.5470E+00	2.9450E+00	3.5364E+01	2.6867E+01
% of Tbicg	100.00	2.32	4.41	53.00	40.27
Speedup	6.19	3.01	14.12	1.48	11.71
Speedup/P	0.39	0.19	0.88	0.09	0.73

Run times and Speedup in matvec (called from bicgstab).

measure	Tmatvec
nP = 1	
runTime	3.5627E+02
Speedup	1.00
Speedup/P	1.00
nP = 2	
runTime	1.8062E+02
Speedup	1.97
Speedup/P	0.99
nP = 4	
runTime	1.0926E+02
Speedup	3.26
Speedup/P	0.82
nP = 8	
runTime	5.5684E+01
Speedup	6.40
Speedup/P	0.80
nP = 16	
runTime	2.9812E+01
Speedup	11.95
Speedup/P	0.75

****

Run times and Speedup in pcg (called from dpres).

measure	Tpcg	Trhs	Tlo	Tpssor	TrestPC	TmatVect	Tcoef
nP = 1							
runTime	2.6376E+02	8.6341E+00	2.5513E+02	1.3753E+02	5.0135E+00	1.0303E+02	9.5498E+00
% Tpcg	100.00	3.27	96.73	52.14	1.90	39.06	3.62
Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00	1.00	1.00	1.00	1.00
nP = 2							
runTime	1.8921E+02	4.4914E+00	1.8472E+02	1.2528E+02	2.5596E+00	5.2191E+01	4.6897E+00
% Tpcg	100.00	2.37	97.63	66.21	1.35	27.58	2.48
Speedup	1.39	1.92	1.38	1.10	1.96	1.97	2.04
Speedup/P	0.70	0.96	0.69	0.55	0.98	0.99	1.02
nP = 4							
runTime	1.4022E+02	2.4240E+00	1.3780E+02	1.1200E+02	1.2407E+00	2.2510E+01	2.0447E+00
% Tpcg	100.00	1.73	98.27	79.87	0.88	16.05	1.46
Speedup	1.88	3.56	1.85	1.23	4.04	4.58	4.67
Speedup/P	0.47	0.89	0.46	0.31	1.01	1.14	1.17
nP = 8							
runTime	1.4774E+02	1.3709E+00	1.4637E+02	1.3176E+02	8.3813E−01	1.2571E+01	1.1975E+00
% Tpcg	100.00	0.93	99.07	89.18	0.57	8.51	0.81
Speedup	1.79	6.30	1.74	1.04	5.98	8.20	7.97
Speedup/P	0.22	0.79	0.22	0.13	0.75	1.02	1.00
nP = 16							
runTime	1.2624E+02	9.1287E−01	1.2533E+02	1.1812E+02	6.4537E−01	5.9131E+00	6.4455E−01
% Tpcg	100.00	0.72	99.28	93.57	0.51	4.68	0.51
Speedup	2.09	9.46	2.04	1.16	7.77	17.42	14.82
Speedup/P	0.13	0.59	0.13	0.07	0.49	1.09	0.93

Run times and Speedup in pssor (called from pcg).

measure	Tpssor	Tforward	Tbackwrd
nP = 1			
runTime	1.3753E+02	6.9332E+01	6.8194E+01
% of Tpssor	100.00	50.41	49.59
Speedup	1.00	1.00	1.00
Speedup/P	1.00	1.00	1.00
nP = 2			
runTime	1.2527E+02	6.0409E+01	6.4863E+01
% of Tpssor	100.00	48.22	51.78
Speedup	1.10	1.15	1.05
Speedup/P	0.55	0.57	0.53
nP = 4			
runTime	1.1199E+02	5.2767E+01	5.9226E+01
% of Tpssor	100.00	47.12	52.88
Speedup	1.23	1.31	1.15
Speedup/P	0.31	0.33	0.29
nP = 8			
runTime	1.3175E+02	6.2108E+01	6.9646E+01
% of Tpssor	100.00	47.14	52.86
Speedup	1.04	1.12	0.98
Speedup/P	0.13	0.14	0.12
nP = 16			
runTime	1.1812E+02	5.6031E+01	6.2085E+01
% of Tpssor	100.00	47.44	52.56
Speedup	1.16	1.24	1.10
Speedup/P	0.07	0.08	0.07

****

Residuals for the last time step.

residual	dU	dV	dW	dP	dK	dE
nP = 1	5.7769E-04	2.0743E-04	3.8769E-04	8.9015E-04	5.1662E-04	2.6069E-04
nP = 2	5.9045E-04	2.7009E-04	6.8717E-05	8.9156E-04	5.1763E-04	5.0155E-04
nP = 4	7.1693E-05	9.2297E-04	1.6717E-04	8.9284E-04	2.4277E-04	4.7776E-04
nP = 8	1.0564E-04	8.0584E-04	1.6520E-04	8.9286E-04	2.4190E-04	3.5336E-04
nP = 16	4.7184E-04	9.5502E-04	1.5876E-04	9.2045E-04	2.4376E-04	1.1259E-04