ELSEVIER

PII: S0965-9978(98)00011-8

*Advances in Engineering Software* Vol. 29, No. 3–6, pp. 325–330, 1998

0965-9978/98/$19.00+0.00

# A three-level parallelization of spatial direct numerical simulation

**Qingxu Hou[a] & Xian-He Sun[b,*]**

[a]*ActaMed Corporation, 7000 Central Parkway, Suite 620, Atlanta, GA 30328, USA*
[b]*Department of Computer Science, Louisiana State University, Baton Rouge, LA 700803-4020, USA*

The parallelization of a NASA Navier–Stokes simulation code is carefully and systematically investigated. Based on numerical, dependence and partition analysis, three different levels of parallelization have been proposed and implemented. The low-level parallelization is at the kernel level. A previously developed parallel tridiagonal solver is used for concurrent processing. The middle-level parallelization solves the multiple right-hand-sides of the tridiagonal systems concurrently. The high-level parallelization is at the level of time step iterations. A PVM implementation of the parallelization has been accomplished. Different communication patterns and different PVM communication calls have been examined for best performance. The parallelized code has been tested on two parallel platforms, a cluster of workstations available at Louisiana State University and the IBM SP2 parallel computer available at Cornell Supercomputing Center. Experimental results confirm our analytical findings: the three-level parallelization is feasible and effective. A linear speedup is measured. In addition to parallelization, through profiling and performance analysis, the original sequential code is optimized as well. © 1998 Published by Elsevier Science Limited. All rights reserved.

## 1 INTRODUCTION

One CFD simulation code which is of interest to many researchers is the code developed by Man M. Rai and his colleagues at NASA.[1,2] This code (we call it Rai's code) adopts fifth-order finite-difference discretizations and Newton–Rapson-type techniques for solving Navier–Stokes equations. In our study, the parallelization of Rai's code is carefully and systematically investigated. Based on numerical, dependence, and partition analysis, three different levels of parallelization have been proposed and implemented on the sequential code. The low-level parallelization is at the kernel level. A previously developed tridiagonal solver[3] is used along the X-dimension. The middle-level parallelization solves the multiple right-hand-sides of the tridiagonal systems concurrently. The high-level parallelization is at the level of time step iterations. Time step iteration has inherited data and flow dependence; the next iteration uses the result of the current iteration. In general, parallelization of time step iteration is not feasible. However, with a thorough dependence analysis, we identified that, with some manipulation, I/O independence can be achieved. Output of different iterations can be conducted concurrently. Parallel I/O is significant. I/O cost appears to be the single most important contributing factor in the performance of Rai's code. Corresponding to Rai's code, the middle-level and high-level parallelization is realized through the concurrent processing of the outermost loop of the procedure LHS1 and an inner loop of the procedure CONTROL, respectively.

Our three-level parallelization is based on formal dependence analysis. The correctness of the parallelization is guaranteed. A PVM implementation of the parallelization has been accomplished. Different communication patterns and different PVM communication calls have been examined for best performance. The parallelized Rai's code has been tested on two parallel platforms, a cluster of workstations available at Louisiana State University and the IBM SP2 parallel computer available at Cornell Supercomputing Center. Experimental results confirm our analytical findings: the three-level parallelization is feasible and effective. A linear speedup is measured. In addition to parallelization, based on profiling and performance analysis, we have optimized the sequential implementation of the original version of Rai's code as well. For instance, with stride minimization and array reconstruction, we have reduced sequential I/O time by 47% for our test cases.

## 2 NUMERICAL MODELING

Rai's code is developed based on a numerical model given

*Author to whom all correspondence should be addressed. E-mail: sun@bit.csc.isu.edu

by Rai and Moin[1] for direct numerical simulation. By their model, the corresponding unsteady, compressible, nonconservative formulation of the Navier–Stokes equation in three spatial dimensions is

$$Q_t = AQ_x + BQ_y + CQ_z = \frac{1}{\rho}(R_x + S_y + T_z) \tag{1}$$

where $Q$ is the vector of dependent variables

$$Q = \begin{pmatrix} \rho \\ u \\ v \\ w \\ p \end{pmatrix} \tag{2}$$

and $\rho$ is the density; $u$, $v$, $w$ are the velocities in the $X$, $Y$, and $Z$ directions, respectively; and $p$ is the pressure. The components in the above Navier–Stokes equations are reflected in modules of the Rai's code. For examples, modules LHS1 and RHS1 represent contribution from the $X$ direction; modules LHS2 and RHS2 represent contribution from the $Y$ direction; and modules LHS3 and RHS3 represent contribution from the $Z$ direction.

In time direction, the second-order-accurate fully upwind (backward) difference for a first derivative is used. Higher order difference is used for the unknown vector $Q$ of dependent variables. Let $Q'$ and $Q''$ be the forward and backward differences of the vector $Q$, respectively. Then the fifth-order-accurate forward- and backward-finite differences using a seven-point stencil are

$$Q_x^- = \frac{-6Q_{i+2} + 60Q_{i+1} + 40Q_i - 120Q_{i-1} + 30Q_{i-2} - 4Q_{i-3}}{120\Delta x} \tag{3}$$

$$Q_x^+ = \frac{4Q_{i+3} - 30Q_{i+2} + 120Q_{i+1} - 40Q_i - 60Q_{i-1} + 6Q_{i-2}}{120\Delta x} \tag{4}$$

Using the Newton–Raphson-type iterative technique and factorization technique, eqn (1) can be expressed as

$$\begin{aligned}
&\left[\alpha I + \beta\Delta t\left(\frac{A + \nabla_x}{\nabla x_i} + \frac{A^-\Delta_x}{\Delta x_i}\right)\right]^p \\
&\times \left[\alpha I + \beta\Delta t\left(\frac{B^+\nabla y}{\nabla y_j} + \frac{B^-\Delta y}{\Delta y_j} - M\left(\frac{\Delta y}{\Delta y_j} + \frac{\nabla y}{\nabla y_j}\right)\right.\right. \\
&\left.\left. - N\left(\frac{\Delta y}{\Delta y_j} - \frac{\nabla y}{\nabla y_j}\right)\right)\right]^p \times \left[\alpha I + \beta\Delta t\left(\frac{C^+\nabla z}{\nabla z_k} + \frac{C^-\Delta_z}{\Delta z_k}\right)\right]^p \\
&\times (Q^{p+1} - Q^p) = -\Delta t\left[\frac{3Q^p - 4Q^n + Q^{n-1}}{2\Delta t}\right. \\
&+ (A^+Q_x^- + A^-Q_x^+ + B^+Q_y^- + B^-Q_y^+ + C^+Q_z^- + C^-Q_z^+)^p \\
&\left. - \left[\frac{1}{\rho}(R_x + S_y + T_z)\right]^p\right]
\end{aligned} \tag{5}$$

where $\alpha = 1.5^{1/3}$, $\beta = 1.5^{2/3}$ and

$$\nabla x_i = x_i - x_{i-1} \tag{6}$$

$$\Delta x_i = x_{i+1} - x_i \tag{7}$$

Eqn (5) (corresponding to Equation 11 in Ref. [1]) represents some of the major modules in the sequential code. The solvers for left-hand-sides are modules LHS1, LHS2 and LHS3 for the $X$, $Y$ and $Z$ directions respectively. The solvers for right-hand-sides are RHS1, RHS2, and RHS3 for the $X$, $Y$, and $Z$ directions respectively. Solver RHSV is for the remaining terms.

The block tridiagonal equations in the $X$ and $Z$ directions can be transformed to scalar tridiagonal equations to reduce about 25% execution time.[1] Utilizing the following equations

$$Q^{p+1} - Q^p = Q^* \tag{8}$$

$$A = P\Lambda P^{-1},\ A^{\pm} = P\Lambda^{\pm}P^{-1},\ I = PP^{-1} \tag{9}$$

where $\Lambda^+$ and $\Lambda^-$ are diagonal matrices containing the positive and negative eigenvalues of $\Lambda$ respectively. The $X$ direction contribution in eqn (5) can be written as

$$\begin{aligned}
&\left[\alpha I + \beta\Delta t\left(\frac{\Lambda^+\nabla_x}{\nabla x_i} + \frac{\Lambda^-\Delta_x}{\Delta x_i} + \frac{\Lambda^-\Delta_x}{\Delta x_i}\right)\right]Q^* \\
&= \left(-\frac{\beta\Delta t}{\nabla x_i}\Lambda^+ \left(\alpha I + \frac{\beta\Delta t}{\nabla x_i}\nabla^+ - \frac{\beta\Delta t}{\Delta x_i}\Lambda^-\right)\frac{\beta\Delta t}{\Delta x_i}\Lambda^-\right) \\
&\times \begin{pmatrix} Q_{i-1}^* \\ Q_i^* \\ Q_{i+1}^* \end{pmatrix} = (D_{i-1}E_iF_{i+1}) \begin{pmatrix} Q_{i-1}^* \\ Q_i^* \\ Q_{i+1}^* \end{pmatrix}
\end{aligned} \tag{10}$$

where $D$, $E$ and $F$ are all $5 \times 5$ blocks. This shows that the equations in $X$ directions can be transformed to scalar tridiagonal equations from the original block tridiagonal equations. A similar formula is valid for the $Z$ direction. In the $Y$ direction, the block tridiagonal equations remain.

A step-by-step approach to the solution of the above system of equations in eqn (5) is given as follows

$$\begin{aligned}
&P[F(x)]P^{-1}[G(y)]T[H(z)]T^{-1}Q^* = R \\
&S_1 = P^{-1}R \Rightarrow S_1 \\
&[F(x)]S_2 = S_1 \Rightarrow S_2 \\
&S_3 = PS_2 \Rightarrow S_3 \\
&[G(y)]S_4 = S_3 \Rightarrow S_4 \\
&S_5 = T^{-1}S_4 \Rightarrow S_5 \\
&[H(z)]S_6 = S_5 \Rightarrow S_6 \\
&Q^* = TS_6 \Rightarrow Q^* \\
&Q^{p+1} = Q^* + Q^p
\end{aligned} \tag{11}$$

Eigenvalue analysis shows that the equations are diagonal dominant. This diagonal dominant property is essential for the Parallel Diagonal Dominant Solver.[3]

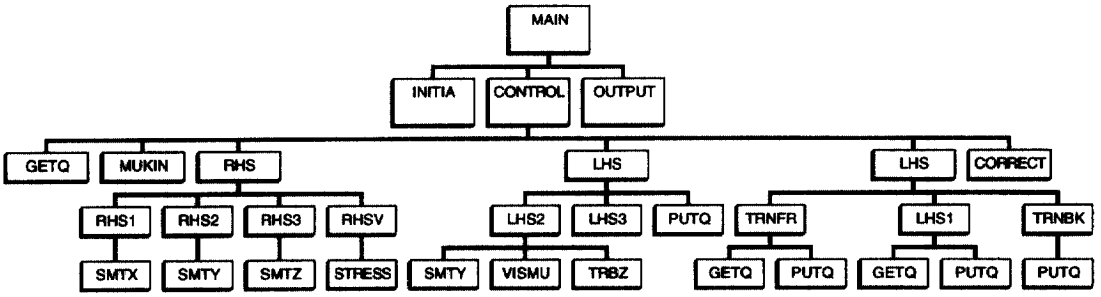


**Fig. 1.** Functional decomposition of sequential code.

## 3 DATA DEPENDENCE AND PARTITION

Functional decomposition is the first step toward analyzing the dependence structure of the parallelization of a sequential program. It breaks a main module into sub-modules and reveals the parent–children relationship within a module. Rai's code has 4K lines of code. It consists of 36 subroutines which can be grouped into several modules. The high level modules are INITIA, DATA, CONTROL and OUTPUT. Routines INITIA and CONTROL are the main modules. They have 12 and 20 children subroutines respectively. Fig. 1 shows the functional decomposition of Rai's code.

While subroutine INITIA is only executed once for each run, the CONTROL module is called at each of the iteration steps. Execution of the CONTROL module dominates the overall execution time. The decomposition is obtained through analysis utilizing Unix utilities.

The sequential code is optimized through loop interchange and array reconstruction. For example, the code to optimize *getq* is listed here.

*getq* (original)

```
      do 100 iy = 1, nyn
      do 100 iz = 1, nzn
      do 100 l = 1,5
      qbuf(iy,iz,l) = qbn(i,iunit,iy,iz,l)
  100 continue
```

*getq* (optimized)

```
      do 100 l = 1,5
      do 100 iz = 1,nzn
      do 100 iy = 1,nyn
      qbuf(iy,iz,l) = qbn(iy,iz,l,i,iunit)
  100 continue
```

For a study case, the overall run time is reduced by 17.6%, from the original 5324 to 4387 s. The percentage of time distribution for *putq* and *getq* is reduced from 14.8~14.9% to 8.1~8.3%, a reduction of 44.8%. Meanwhile, the sec/call for *putq* and *getq* is reduced from 132.2~132.3 to 59.6~61.2, a reduction of 54%. The profiling result is shown in Fig. 2.

### 3.1 Data dependence concept

If the iterations of a loop can be executed in random order and still produce the correct result, it is an independent loop.[4] Independent loops are perfect for parallel execution. But loops are rarely independent. Dependent loops, in which the dependency involves all the statements in the loop, must be executed serially on any machine due to their dependency. When the dependency does not involve all the statements in the loop, partial overlapping, or pipelining of successive iterations may be possible during the execution.

Data dependence is a consequence of the flow of data in a program. A task that uses a variable in an expression is data dependent on the task which computes the value of that variable. If task T1 is data-dependent on task T2, then execution of task T2 must precede execution of task T1.

Data dependence can be further classified into five categories:

- Flow dependence: A statement S2 is flow-dependent on statement S1 if an execution path exists from S1 to S2 and if at least one output of S1 feeds in as input to S2.
- Antidependence: Statement S2 is antidependent on statement S1 if S2 follows S1 in program order and if the output of S2 overlaps the input of S1.
- Output dependence: Two statements are output-dependent if they write the same output variable.
- I/O dependence: Read and write refer to the same file.
- Unknown dependence: the dependence cannot be explicitly determined.

Bernstein's conditions imply that two processes can be executed in parallel if they are flow-independent, anti-independent, and output-independent.[5]

The above concepts of data dependence are used to analyze the CFD program for parallelism. Loops perfect for

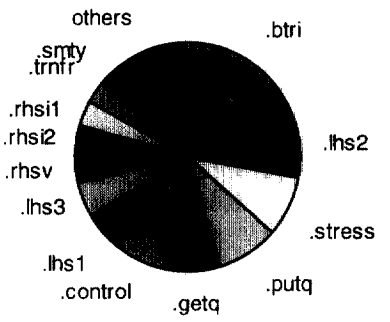


**Fig. 2.** Profiling of the sequential code.

parallelization are discovered at a very high level for both subroutines CONTROL and LHS1. Some careful manipulation of data initialization in CONTROL is done to achieve this goal. Since CONTROL takes more than 90% execution time, its parallelization is very significant to the overall performance.

### 3.2 Data dependence analysis

The module CONTROL does the calculation at each iteration step. It has quite a few nested loops. To get maximum parallelization, a natural approach is to go to the outermost loop. Observing Rai's code, the loop 10 is to get the iterative solution at each iteration step. Since the next iteration always uses the result of the current one, loop dependence is internally determined. Actually, it goes through all the $X$ direction planes to find the solutions along the $Y$ and $Z$ directions in loop 160 and then swaps to all the $Y$ planes for solutions in $X$ direction (this is done by LHS1).

The analysis gives the same result. First, there is a flow dependence from one iteration to the next. While the current iteration writes $Q$ using PUTQ in loop 300, the next iteration reads vector $Q$ in loop 150 and loop 160. Second, there is an anti-flow dependence within each iteration because the input $Q$ in loop 150 is also the output in loop 300. These two types of data dependence determine that it is very unlikely to do parallelization at the loop 10 level. The next highest loop level is loop 160. It has 13 direct subroutine calls and the out loop goes over all the $X$ direction planes. It involves solving solutions along the $Y$ and $Z$ directions and calculating all the right-hand-sides which includes the invocation of subroutines RHS1, RHS2, RHS3 and RHSV. It should be noticed that RHSV calls the time-consuming subroutine STRESS directly.

There are several GETQ calls and one PUTQ call within loop 160. This, however, does not introduce a flow or anti-flow dependence because they operate on different units of the $Q$'s. The call GETQ ($i$fetch, 1, qeq1) seems to be an anti-flow dependence because the $i$th iteration uses the qeq1 from $(i + 4)$th iteration $X$ direction. The boundary elements are exceptions. Two things should be noticed here. First, in sequential calculation, the $(i + 4)$th iteration is done after the $i$th iteration and the values of qeq1 used are from the last time step iteration. Second, if the $i$th and the $(i + 4)$th iterations are grouped into the same process for parallelization, the dependence goes away. This uses the exceptions of boundaries because their elements are not affected by the $(i + 4)$th iteration. This is exactly what comes from the data dependence analysis: loop 160 can be parallelized perfectly.

What about I/O dependence, output dependence and unknown dependence? Since there is no file writing or reading involved within loop 160, the I/O dependence is eliminated. Although each iteration updates or writes the PUTQ, there is no overlapping because they use different $i$ values. This takes care of the output dependence. Finally, the unknown dependence is eliminated through tracing and analyzing all the called subroutines. The fact that the arrays do not involve implicit or nested subscripts makes the analyzing a little easier.

The above description for data dependence analysis seems to be simple, but the actual work is very complicated. When the test using random order for loop 160 is done at the beginning, the results do not agree with the original ones. This denies the parallelization and data dependence analysis and it is hard to find a clue where things go wrong. After several tries, the boundaries and loop bounds (including upper bound and lower bound) are recognized to be the critical points. Then the original code is modified based on this careful observation for correctness and efficiency. It should be emphasized that the bound may involve its own initialization. The unknown dependence is not resolved by itself but by test and manipulation.

The CONTROL module contributes most to the overall execution time. For a study case, in which a detailed timing profiling is obtained for each subroutine using *prof* and *gprof*, it takes about 90% of the total execution time of the sequential code. Although the data independence is not at the highest level, the parallelism achieved above is a critical part of a successful, feasible parallelization for the direct simulation. This justifies the effort for the above data dependence analysis.

The module LHS1 is a tridiagonal solver for solutions in the $X$ direction. It iterates over all the planes in the $Y$ direction. The data dependence analysis for LHS1 is done in the same way as for CONTROL. The analysis gives the conclusions that the highest loop level of LHS1 can be parallelized perfectly. Numerous random order loop tests have been done to confirm that the loop is flow independent, anti-flow independent, and output independent.

Since LHS1 is called within CONTROL and its contribution is not significant (a study case shows about 10% execution time), the improvement through partition in LSH1 is limited. However, the partition of LSH1 and that of CONTROL are independent and can be done together to achieve better performance.

### 3.3 Data partition analysis

Since the loops for perfect parallelization are found in modules CONTROL and LHS1, the data partition is straightforward. From the partition point of view, more sub-tasks correspond to less execution time. However, communication is necessary for parallel processing of the loops. The communication overhead is highly determined by the network conditions and by the whole virtual machine system in the case of PVM. Massive parallelization may not necessarily lead to a good performance. To reduce the communication cost, multicasting mode is used for data communication in the parallel PVM implementation.

## 4 EXPERIMENTAL RESULTS ON SP2

PVM implementation of Rai's code has been tested on an SP2 system and on a cluster of workstations. Here we present only the experimental results on SP2. Results for a

cluster of workstations can be found in Ref. [6]. The parallel implementation on SP2 is carried out with an EASY-LL batch system using architecture RS6K and data encoding PvmDataRaw.

The input data and format are:

**Input Format**

- read(5,*) iread, iwrite
- read(5,*) niter, nprint
- read(5,*) imax, jmax, kmax, iplt, ilcc
- read(5,*) datu, cour, amach, reperin

**Input Data**:

- 0 0
- 1 1
- 768 48 7 728 182
- 0.002 0.25 2.25 635000.0

PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. User programs written in C, C++ or Fortran access PVM through library routines. Daemon programs provide communication and process control between computers.

EASY-LL is a collaboration between EASY (Extensible Argonne Scheduling sYstem) and LoadLeveler which effectively divides the work of job scheduling into two parts. LoadLeveler handles the administrative part: recording information about the resources available on each node, the status of jobs, and the status of nodes. EASY takes care of the job scheduling, that is, when a particular job should run and on which nodes it should run.

Sequential runs and PVM parallel implementations are measured on 1, 2, 4, 8, and 16 thin processors of the SP2 machine, respectively. Rai's original sequential code is used for the sequential run. Execution time is measured for both sequential implementation and PVM implementation with tasks of 2, 4, 8, and 16 on thin nodes. Average overall execution times and speedup are illustrated in Figs 3 and 4, respectively.

As more tasks are used, the overall execution time is reduced. However, the reduction gained from each task tends to get smaller. The most significant reduction from each task is seen when two tasks are used and the least is seen when sixteen tasks are used. This is clearly shown from both figures. As the task number increases, the speedup tends to be constant. This observation justifies Amdahl's Law: for a fixed workload, the total execution time decreases as the processor number increases; eventually the sequential part will dominate the performance because of the sequential bottle neck, which accounts for the sequential fraction of the whole program.[5]

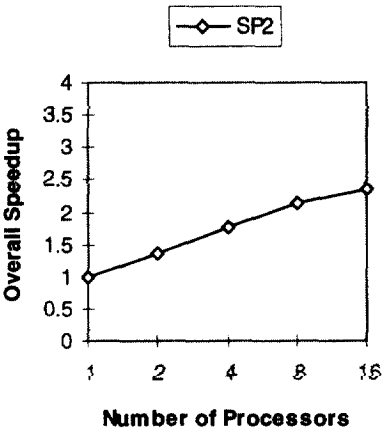


**Fig. 4.** Overall speedup.

Our parallelization is implemented at the CONTROL module. The execution time of the parallelized CONTROL module is also measured for performance study. Five iterations are conducted in the code to reach a satisfactory solution. The execution time of each iteration of CONTROL and the corresponding speedup are presented in Figs 5 and 6, respectively. The parallelized CONTROL module achieves a very good speedup. Its measured speedup is a linear function of the number of processors.

Study cases show that when LHSI is parallelized, the execution time can be reduced by about 10% from that time when only CONTROL is parallelized.

Notice that the measured time of each iteration does not include the communication cost between each iteration.

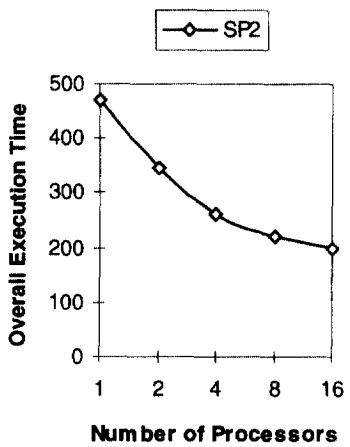


**Fig. 3.** Overall execution time (s).

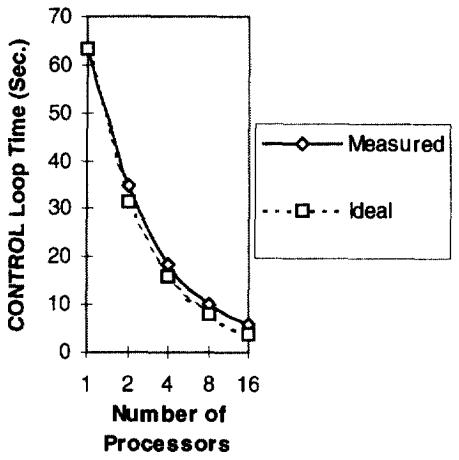


**Fig. 5.** Execution time of CONTROL loop iteration (s).

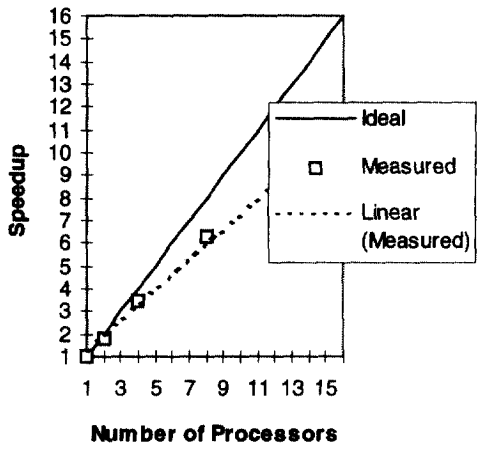


**Fig. 6.** Speedup for CONTROL loop.

Even without considering the communication overhead, due to sequential/parallel portion increases, the measured speedup gets farther away from the ideal speedup when the number of processors increases. The power of parallel computing can be utilized more efficiently if scaled computing is adopted, in which problem size increases with system size.[7]

Due to our limited access to the Cornell SP2 machine, only small-size problems have been tested. However, while our current experiment results do not show the best speedup possible, they clearly demonstrate the correctness and effectiveness of our dependence analysis and partition approach. They have shown the high potential of parallel processing gain in solving Rai's code. Also, our results show that using data encoding PvmDatRaw provides a better performance than using PvmDataDefault in solving Rai's code.

## 5 CONCLUSIONS

A linear speedup has been achieved on parallelizing a direct Navier–Stokes simulation code (Rai's code). The parallelization consists of several steps. First, profiling is used to identify the critical section of the code. Then, data dependence and data partition analysis are conducted to reveal the existence of three levels of concurrency. Finally, a PVM implementation is fine tuned based on the three levels of concurrency for parallel processing.

While the parallelized/optimized Rai's code has its practical usefulness, the parallelization process itself is probably more valuable than its results. Rai's code is a general CFD simulation code. It is reasonably large and lacks documentation. We had no previous knowledge of the code. The parallelization is achieved through the use of a combination of generally available UNIX utilities such as *gprof* and dependence analysis techniques, such as flow dependence, anti-dependence, output dependence, I/O dependence, and unknown dependence analysis, which have been recently developed for parallel compilers. The experience gained and lessons learned during this parallelization project can certainly be applied to parallelize other dust-deck large scientific simulation packages as well.

## ACKNOWLEDGEMENTS

This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-1672 and by the Louisiana Education Quality Support Found (LEQSF). We are also grateful to the Cornell Theory Center for providing access to the SP2 parallel computer.

## REFERENCES

1. Rai, M.M. and Moin, P. Direct simulations of turbulent flow using finite-difference schemes. *Journal of Computational Physics*, 1991, **96**, 15–53.
2. Rai, M.M. and Moin, P. Direct numerical simulation of transition and turbulence in a spatially evolving boundary layer. *AIAA Paper*, 1991, **11**, 1591–1607.
3. Sun, X.-H. Application and accuracy of the parallel diagonal dominant algorithm. *Parallel Computing*, 1995, **8**, 1241–1267.
4. Lewis, T. and El-Rewini, H., *Introduction to Parallel Computing*. Prentice Hall, 1992.
5. Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.
6. Hou, Q., A parallel implementation of a 3D computational fluid dynamics application. Master of Science Project Report, Department of Computer Science, Louisiana State University, Jan. 1997.
7. Sun, X.-H. and Ni, L. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 1993, **9**, 27–37.