A Parallel Algorithm for Solving a Tridiagonal Linear System with the ADI Method

Lixing Ma
Department of Computer Science
University of Nevada
Reno, NV 89557

Frederick C. Harris, Jr.
Department of Computer Science
University of Nevada
Reno, NV 89557
fredh@cs.unr.edu

Abstract

The Alternating Direction Implicit (ADI) method is widely used in various discretized systems. In this paper, a new parallel algorithm is developed to solve a system of tridiagonal linear equations with the ADI method for a large-scale heat conduction problem. The theoretical analysis on the speedup and scalability is also presented. When compared with Thomas's algorithm, this algorithm shows a good improvement for parallel machines.

keywords: parallel algorithm, Alternating Direction Implicit method, ADI

1 Introduction

The unsteady, three-dimensional heat conduction equation, can be written in three spatial dimensions as follows:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) \qquad (1)$$

If the Crank-Nicolson method is used to discretize the above equation, it will contain seven unknowns $T^{n+1}_{i,j,k}$, $T^{n+1}_{i+1,j,k}$, $T^{n+1}_{i-1,j,k}$, $T^{n+1}_{i,j+1,k}$, $T^{n+1}_{i,j-1,k}$, $T^{n+1}_{i,j,k+1}$ and $T^{n+1}_{i,j,k-1}$, where the last four unknowns prevent us from putting it into a standard tridiagonal form [9], and Thomas's algorithm can not be directly used. There are some matrix methods and iterative methods that can be used to solve this discretization equation, but the computation time is much longer than that of a tridiagonal system. As a result, there is a distinct advantage in developing a scheme that will allow Equation (1) to be solved by means of tridiagonal form. Such a scheme is the Alternating Direction Implicit (ADI) method [1].

The ADI method usually introduces three time step terms $(t+\frac{1}{3}\Delta t, t+\frac{2}{3}\Delta t, t+\Delta t)$ for a three-dimensional problem and two time step terms $(t+\frac{1}{2}\Delta t, t+\Delta t)$ for two-dimensional problem. In this manner, Equation (1) can be discretized as three tridiagonal forms:

$$\frac{T^{n+\frac{1}{3}}_{i,j,k} - T^{n}_{i,j,k}}{\frac{\Delta t}{3}} = \alpha \left(\frac{\partial^2 T^{n+\frac{1}{3}}}{\partial x^2} + \frac{\partial^2 T^{n}}{\partial y^2} + \frac{\partial^2 T^{n}}{\partial z^2} \right) \qquad (2)$$

$$\frac{T^{n+\frac{2}{3}}_{i,j,k} - T^{n+\frac{1}{3}}_{i,j,k}}{\frac{\Delta t}{3}} = \alpha \left(\frac{\partial^2 T^{n+\frac{1}{3}}}{\partial x^2} + \frac{\partial^2 T^{n+\frac{2}{3}}}{\partial y^2} + \frac{\partial^2 T^{n+\frac{1}{3}}}{\partial z^2} \right) \qquad (3)$$

$$\frac{T^{n+1}_{i,j,k} - T^{n+\frac{2}{3}}_{i,j,k}}{\frac{\Delta t}{3}} = \alpha \left(\frac{\partial^2 T^{n+\frac{2}{3}}}{\partial x^2} + \frac{\partial^2 T^{n+\frac{2}{3}}}{\partial y^2} + \frac{\partial^2 T^{n+1}}{\partial z^2} \right) \qquad (4)$$

where $\partial^2/\partial x^2$, $\partial^2/\partial y^2$, and $\partial^2/\partial z^2$ are the central differences in space. Equations (2), (3), and (4) always have only three unknowns, which allows for construction of a tridiagonal matrix. First, Equation (2) is used to solve $T^{n+1/3}_{i,j,k}$ in the x-direction according to the initial condition or the results of the last time step. Then, $T^{n+1/3}_{i,j,k}$ is used in Equation (3) to solve $T^{n+2/3}_{i,j,k}$ in the y-direction. Finally, $T^{n+1}_{i,j,k}$ in the z-direction is solved according to Equation (4). This is an overview of the Alternating Direction Implicit (ADI) method. Generally, these three equations can be represented in a linear algebraic equation, namely:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = f_i \qquad i = 1, ..., n-1 \quad (5)$$

The ADI scheme is unconditionally stable and has second-order accuracy with a truncation error of $\mathcal{O}[(\Delta t)^2, (\Delta x)^2, (\Delta y)^2]$ for two dimensions and is conditionally stable and with a truncation error of $\mathcal{O}[(\Delta t), (\Delta x)^2, (\Delta y)^2, (\Delta z)^2]$ for three dimensions [6]. The other advantage of the ADI scheme is that Thomas's algorithm can be used to find exact solution of Equation (5).

There are two major parallel algorithms that are widely used for the ADI scheme. One is the domain decomposition algorithm[12], and the other uses higher

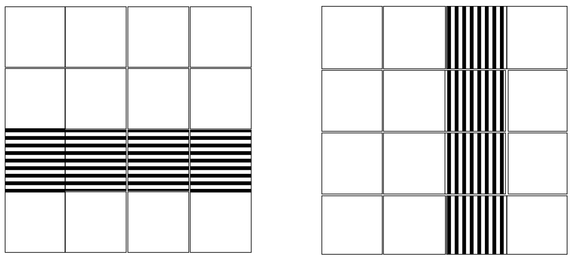

Figure 1: Domain decomposition based on localized inversion.

dimensional linear recurrences to solve the diatriangular linear system Equation (5)[13].

In the domain decomposition algorithm, the main domain is divided into many subdomains, where the number of subdomains is usually chosen to be a power of 2. The data in a subdomain depend on other domains only through the inner boundaries between those domains. Within a subdomain, the nodal data have a well-defined interdependency. A more effective approach is to localize the line inversion algorithm and implement a bona fide domain decomposition. If one can break the line inversion across the whole domain into local line inversions within the subdomains and remove the interprocessor data dependency, then each processor can work on the local line segment and the execution can be carried out in parallel. With such a parallel line inversion algorithm, it is possible to achieve domain decomposition and parallel processing by dividing the computational domain into subdomains such as the division presented in Figure 1.

Figure 1 shows a domain decomposed into 4×4 subdomains with four processors in parallel. When scanning in the horizontal direction, each processor combines four horizontal subdomains to solve Equation (5) to obtain a new x_i or T_i. Based on those values, Equation (5) is then solved in the vertical direction in parallel.

Because the boundary node value of each horizontal domain or vertical domain is always taken from previous data, it will take more iterations to converge if massively parallel processors are used. This is the same problem that causes the Gauss-Siedel method to be replaced by SOR.

In the second algorithm, the main domain is not divided into subdomains. The goal is to provide a parallel algorithm for solving the diatriangular linear system in Equation (5), which also can be presented in matrix form as:

$$Ax = f \tag{6}$$

where x and f are n-dimensional vectors and A is the following $n \times n$ tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & \dots & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \tag{7}$$

Matrix A can be decomposed into an LDU factorization form. That is, you can determine a unit lower triangular matrix L, a diagonal matrix D, and a unit upper triangular matrix U such that $A = LDU$. The matrices L, D and U can be specified as follows:

$$L = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ l_2 & 1 & 0 & \dots & 0 \\ 0 & l_3 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & l_n & 1 \end{bmatrix} \quad D = \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & d_n \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & u_1 & 0 & \dots & \dots & 0 \\ 0 & 1 & u_2 & 0 & \dots & 0 \\ 0 & 0 & 1 & u_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & 0 & 1 & u_{n-1} \\ 0 & \dots & \dots & \dots & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} d_1 &= b_1 \\ d_j &= b_j - a_j c_{j-1}/d_{j-1}, \quad && 2 \le j \le n \\ l_j &= a_j/d_{j-1}, && 2 \le j \le n \\ u_j &= c_j/d_j, && 1 \le j \le n-1 \end{aligned} \tag{8}$$

From Equation (8), it is clear that the values of l_j and u_j can be deduced immediately from the values of d_j. For convenience of parallelization, let $d_j = w_j/w_{j-1}$, where $w_0 = 1$ and$w_1 = b_1$. Substituting $d_j = w_j/w_{j-1}$ and $d_{j-1} = w_{j-1}/w_{j-2}$ in the equation $d_j = b_j - a_j c_{j-1}/d_{j-1}$, the sequence w_j is given by the following second-order linear recurrence:

$$\begin{aligned} w_0 &= 1 \\ w_1 &= b_1 \\ w_j &= b_j w_{j-1} - a_j c_{j-1} w_{j-2}, \quad && 2 \le j \le n \end{aligned} \tag{9}$$

After the sequence w_j is obtained, all of the values in matrices L, D, and U can be obtained through the relation in Equation (8). The solution to the linear system $Ax = b$ can be expressed by the following recurrence:

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} \\ x_i &= \sum_{j=i-m+1}^{i-1} -\frac{a_{ij}}{a_{ii}} x_j + \frac{b_i}{a_{ii}}, \quad && 2 \le i \le n \end{aligned} \tag{10}$$

Solving this system of equations with a Divide and Conquer approach (DAC) improved this algorithm. Using DAC, the whole linear tridiagonal equation system is divided into k subsystems with a matrix transformation on matrix A. Then P processors are assigned to deal with each subsystem in parallel. This method is complicated and needs many transforms to take into account the effect of the internal boundary node. To find the final solution of one linear system, you have to solve several linear systems first. Therefore in this paper, a new parallel algorithm is introduced to solve the linear system $Ax = b$.

2 Modified Cyclic Reduction

2.1 Cyclic Reduction

Harold Stone presents the Cyclic Reduction method for solving tridiagonal equations in his book [11]. His method works extremely well for the Poisson matrix whose diagonals contain only 1s and -2s. The idea behind cyclic reduction is to sum three consecutive equations as indicated here:

$$\begin{aligned} x_{i-2} - 2x_{i-1} + x_i &= f_{i-1} \\ x_{i-1} - 2x_i + x_{i+1} &= f_i \\ x_i - 2x_{i+1} + x_{i+2} &= f_{i+1} \end{aligned} \tag{11}$$

Adding the first and third equations with the second equation multiplied by two, x_{i-1} and x_{i+1} are eliminated, and Equation (11) becomes one equation as indicated here:

$$x_{i-2} - 2x_i - x_{i+2} = f_{i-1} + 2f_i + f_{i+1} \tag{12}$$

To solve the full system, all of the equations can be reduced to Equation (12) for the first kind of boundary condition or into the three equations in (11) for the second kind of boundary condition. Then, back substitution is used to solve all unknowns. Both the reduction and back substitution steps can be done in parallel.

Unfortunately, this scheme works only for the steady, isotropic and uniform space step problem as described in Equation (1), because the parameter matrix does not contain just 1s and -2s. The important thing to note is that the cyclic reduction method can be modified and the idea can be generalized to other types of matrices.

2.2 Modified Cyclic Reduction(MCR)

The general form of a tridiagonal matrix used in the ADI method is represented in Equation (7). The three consecutive equations are as follows:

$$\begin{aligned} a_{i-1}x_{i-2} + (-b_{i-1}x_{i-1}) + c_{i-1}x_i &= f_{i-1} \\ a_i x_{i-1} - b_i x_i + c_i x_{i+1} &= f_i \\ x_{i+1}x_i - b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= f_{i+1} \end{aligned} \tag{13}$$

In order to eliminate two unknowns x_{i-1} and x_{i+1}, we sum the first equation multiplied by a_i/b_{i-1}, the third equation multiplied by c_i/b_{i+1} and the second equation. After rearranging, Equation (13) becomes

$$a_i^{(1)}x_{i-2} - b_i^{(1)}x_i + c_i^{(1)}x_{x+2} = f_i^{(1)} \tag{14}$$

where

$$\begin{aligned} a_i^{(1)} &= a_{i-1}a_i/b_{i-1}, \\ b_i^{(1)} &= b_i + a_i c_{i-1}/b_{i-1}, \\ c_i^{(1)} &= c_i c_{i+1}/b_{i+1}, \quad \text{and} \\ f_i^{(1)} &= (a_i/b_{i-1})f_{i-1} + f_i + (c_i/b_{i+1})f_{i+1}. \end{aligned}$$

In order to obtain only one equation, the space steps Δx, Δy, and Δz have to be adjusted in the discretization of the Equation (1) so that the total number of unknowns is equal to $2^n - 1$. In addition, it is strongly recommended that you normalize the coefficient b_i. The first advantage of normalization is that the memory can be saved for storing b_i. The second advantage is that it helps you to avoid overflow if the problem has huge numbers. For an isotropic and uniform space step problem, a_i and c_i are usually identical and are normalized more naturally. This is the correct choice, but it only works for $b_i = 2$. Because the matrix A is diagonal dominant and $b_i^{(0)} = b$, and $b_i^{(i+1)} = b_i^{i^2} - 2$, it will overflow after several steps of reduction. For a typical Poisson equation, $b_i^{(0)}$ is usually taken as 4 or 6.

2.3 Middle Node Algorithm for a Parallel MCR

Let us consider the first kind of boundary condition for the ADI method while solving Equation (1). In a specific direction, the number of nodes is equal to $2n + 1$. The first and the last nodes are known because of the first boundary condition, therefore the total number of unknown nodes is equal to $2n-1$. This number is also equal to the dimension of the matrix A in Equation (5). Obviously, $n - 1$ is the total number of times you perform cyclic reduction, at which time you have only one equation left. Three nodes remain in the last equation, x_0, $x_{2^{n-1}}$, and x_{2^n}, where x is counted from 0. These are the first node, the middle

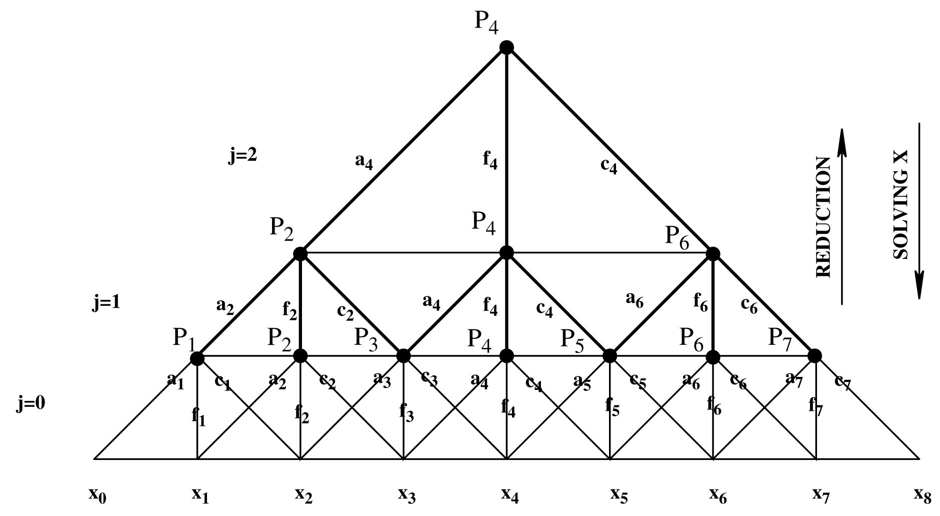

Figure 2: Scheme of the Middle Node Algorithm.

node, and the last node. The first node and the last node are two known boundary values; therefore, the middle node can be found through the last equation. At the middle point, the line is cut in two parts and the same method is used to find two middle node values. In a similar manner, find four, eight, sixteen,..., middle nodes until all the unknown node values are obtained. We call this algorithm the Middle Node Algorithm. The basic formula of this algorithm is based on the following equations

$$a_i^{(j)} x_{i-2^j} - x_i + c_i^{(j)} x_{i+2^j} \quad = \quad f_i^{(j)}$$

$$a_i^{(0)} = \frac{a_i}{b_i},\ a_i^{(j)} \quad = \quad \frac{a_{i-2^{j-1}}^{(j-1)} a_t^{(j-1)}}{1 - a_i^{(j-1)} c_{i-2^{j-1}}^{(j-1)} - a_{i+2^{j-1}}^{(j-1)} c_i^{(j-1)}}$$

$$c_i^{(0)} = \frac{c_i}{b_i},\ c_i^{(j)} \quad = \quad \frac{c_i^{(j-1)} c_{t+2^{j-1}}^{(j-1)}}{1 - a_i^{(j-1)} c_{i-2^{j-1}}^{(j-1)} - a_{i+2^{j-1}}^{(j-1)} c_i^{(j-1)}}$$

$$f_i^{(0)} = \frac{f_i}{b_i},\ f_i^{(j)} \quad = \quad \frac{a_i^{(j-1)} f_{t-2^{j-1}}^{(j-1)} + f_t^{(j-1)} + c_i^{(j-1)} f_{t+2^{j-1}}^{(j-1)}}{1 - a_i^{(j-1)} c_{i-2^{j-1}}^{(j-1)} - a_{i+2^{j-1}}^{(j-1)} c_i^{(j-1)}} \tag{15}$$

Where j is the order of reduction step, $0 \le j \le n-1$. At $j = 0$, the equation is almost same as Equation (5), the only difference is that each b_i is normalized. There are 2^{n-1} nodes at $j = 0$. Each node also represents a processor, and the processors execute code for Equation (15). Each node or processor is assumed to have its own memory to store the information contained in Equation (15), *e.g.* the parameters(a, c and f) and boundary values or neighbor data. For convenience, the parallel algorithm is illustrated through the seven node example in Figure 2.

There are two main procedures in the Middle Node Algorithm: one is reduction and the other is middle node solving. Let us look at the reduction procedure first. At the level $j = 0$, seven processors normalize their own equations in parallel, and a new a, c, and f are obtained. Before entering the second level, this data (a, c, and f) is sent in parallel from P1, P3, P5 and P7 to all their neighbor processors. P1 and P7 have only one neighbor, and P3 and P5 have two neighbors each. In level $j = 1$, P2, P4 and P6 calculate the new a, c, and f based on the data from level $j = 0$. Then, P2 and P6 send a, c, and f to P4. At the highest level $j = 2$, the procedure of reduction is completed after a, c, and f are calculated by P4.

In most implementations of the ADI method, the boundary values x_0 and x_8 are usually combined into the right hand side (the f term) of Equation (5) in the discretization of node 1 and node 7. But x_0, x_8 and their coefficients still remain in the left hand side of the Equation (5). The advantage is to guarantee the last equation is solvable. Otherwise, we need to solve three equations at the highest level.

In the middle node solving procedure, P4 uses the first equation of Equation (15) to find x_4 and sends the value to P2, P6, P3, and P5 at the level $j = 2$. P2 and P6 find x_2 and x_6, respectively, in parallel. Then P2 sends x_2 to its neighbors P1 and P3, and P6 sends x_6 to P5 and P7. At the lowest level $j = 0$, P1, P3, P5 and P7 find x_1, x_3, x_5, and x_7 in parallel. At this point, all the unknowns in Equation (5) are solved.

For the second kind of boundary condition, the linear system is described in Equation (5). Because the first and last nodes are also unknowns, three equations have to be solved in the final step. To do so, we use the Crown Algorithm. The Crown Algorithm uses 2^{n+1} processors and is illustrated in Figure 3. The difference between the Middle Node Algorithm and the Crown Algorithm is in the reduction procedure. They are exactly same after the first three equations are solved. The Crown Algorithm will not be discussed further in this paper.

Sometimes, the number of processors is less than the number of unknown nodes. This is often encountered when we use PVM [3] or MPI [4, 8, 10] to link several workstations together as a parallel virtual machine. Figure 4 contains the pseudo code for when there is a difference between the unknown node number and the processor number.

3 Performance

Speedup is an important parameter to consider when evaluating the performance of a parallel algorithm. It is normally defined as the ratio of the executing time for one processor to the executing time for P processors. The executing time is the sum of computation time, data communication time, and idle time. The computation time of an algorithm (T_{cal}) is the time spent performing computation rather than communicating or idling. The communication time of an algorithm (T_{com}) is the time that its tasks spend sending and receiving messages. In the idealized multicomputer architecture, the cost of sending a message between two tasks located on different processors can be represented by two parameters: the message startup time, which is the time required to initiate the communication, and the transfer time per word. For speedup analysis in this paper, we consider only computation time and communication time. The operations of addition and multiplication are also assumed to have the same operation time. In addition, because there is no big data set that needs to be communicated in the Middle Node Algorithm, the startup time is the main part for T_{com}. We can roughly take the value of T_{com} as some constant times T_{cal} [2].

For the Middle Node Algorithm, six operations are needed to calculate $a_i^{(j)}$ and $c_i^{(j)}$, nine operations are needed to calculate $f_i^{(j)}$, and four operations are needed to calculate x_i. If the number of unknown nodes N is equal to $2^L - 1$, $a_i^{(j)}$, $c_i^{(j)}$, and $f_i^{(j)}$ will be calculated $2N - \log_2(N+1)$ times and

```
input:
    N — number of unknown nodes
    P — number of processors
    a[i], c[i], f[i] — normalized matrix parameters for unknown node i. i=1,2 ... N
    x[0], x[N+1] — boundary node value from boundary condition
output:
   find x[1], x[2], ..., x[N]

step 1:
   create P tasks(procs) and get their task_id[i], i=1, 2, ..., P
    find the level of unknown nodes $ln$ and level of tasks $lp$
      $l_n = \log_2(N+1)$
      $l_P = \log_2(P+1)$
    assign a, c, f and x[0], x[N+1] to all tasks
       for $i = 1$ to $i = P$, do
          for $j = (i-1) * 2^{ln-lp} + 1$ to $(i+1) * 2^{ln-lp} - 1$, do
             assign a[j], c[j] and f[j] to task_id[i]
          send x[0], x[N+1] to task_id[i].
step 2:
    computing a, c, and f parallelly in each task_id[p],
          p=1, ..., P without data communication
      count = 1
      while $count < ln - lp$ do
         for $i = (p-1) * 2^{ln-lp} + 2^{count}$ to $(p-1) * 2^{ln-lp}$
               $+2^{ln-lp+1}$, $step = 2^{count}$ do
            compute a, c, and f according to Equation (15)
         count = count + 1
      end do

    send a, c and f to task_id[p-1](if $p > 1$)
       and task_id[p+1]($p < P$)
step 3:
    computing a, c and f parallelly in each task_id[p],
          p=1, ... ,P with data communication
       count = 1
       while $count < lp$ do
          if $p\%2^{count} = 0$ do
             receive a, c, and f from $p - 2^{count-1}$ and $p + 2^{count-1}$
             compute a, c, and f according to Equation (15)
             if $p\%2^{count+1}! = 0$ send a, c and f
                to task_id[$p - 2^{count}$] and task_id[$p + 2^{count}$]
          end do
          count = count + 1
       end while
step 4:
    use Equation (15) to solve unknown nodes
```

Figure 4: Pseudo Code of the Middle Node Algorithm

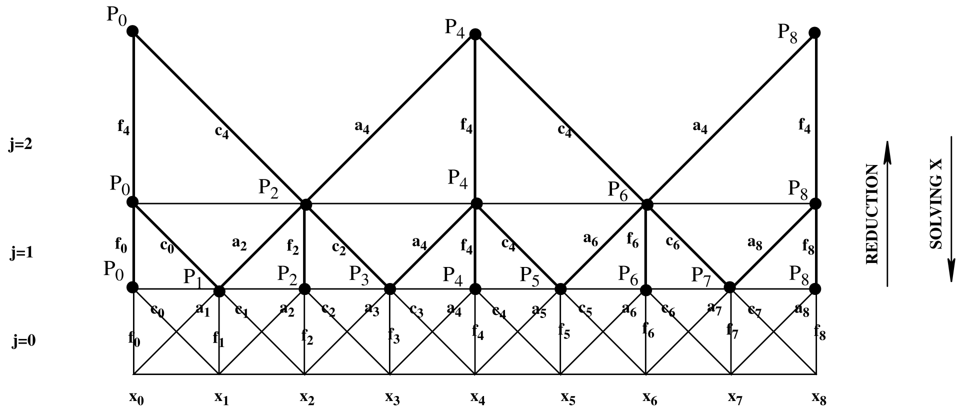

Figure 3: Scheme of the Crown Algorithm for secondary boundary condition.

$a_i^{(j)}$ has to be calculated N times. Therefore, the total operations needed in this algorithm is equal to $46N - 21\log_2(N+1)$, and the total calculation time is equal to $[46N - 21\log_2(N+1)]T_{cal}$, where T_{cal} is the time needed for a unique addition or multiplication operation.

For the parallel algorithm, P processors are used on N unknown nodes, where $P = 2^{Lp} - 1$. If P is less than N, there are $4(n+1)/(P+1) - \log_2((N+1)/(P+1)) - 3$ times to calculate $a_i^{(j)}$, $c_i^{(j)}$, and $f_i^{(j)}$ in each processor. Also there are $(N+1)/2^{Lp} - 1$ times to calculate x_i. Those operations are executed independently–that is, without data communication. Therefore, the total calculation time is $(92(N+1)/(P+1) - 2\log_2[(N+1)/(P+1)])T_{cal}$.

In the top $L_p - 1$ layer, processors calculate $a_i^{(j)}$, $c_i^{(j)}$, $f_i^{(j)}$, and x_i with data communication. Each processor calculates only one node on one level. The total time for calculation is equal to $25(\log_2(P+1) - 1)T_{cal}$. For each node, the processor needs two sets of $a_i^{(j)}$, $c_i^{(j)}$, and $f_i^{(j)}$, from both the left and the right neighbor processors and sends x_i to those neighbors. If the receiving procedure and sending procedure are assumed to take the same time, the total time spent on data communication is $3(\log_2(P+1) - 1)T_{com}$. According to the definition we have speedup equal to:

$$\frac{[46N - 21\log_2(N+1)]T_{cal}}{(92\frac{N+1}{P+1} - 21\log_2(\frac{N+1}{P+1}) + 25\log_2(\frac{P+1}{2}))T_{cal} + 3(\log_2(\frac{P+1}{2}))T_{com}}$$

Figure 5 is the speedup graph with total unknown node size equal to 2047 and three different constants chosen to set communication speeds. It can be seen from the figure that the Middle Node Algorithm has a very good performance for the multiprocessor machine with intra-processor communication.

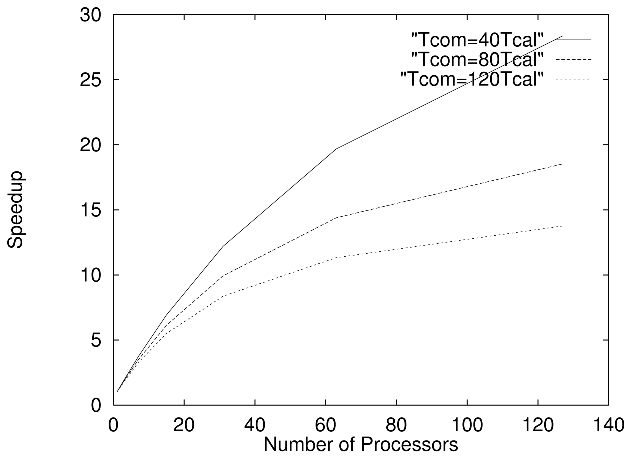

Figure 5: Speedup Curves

4 Conclusion

When compared with the SOR method, our algorithm with the ADI method is always stable even when the matrix A in Equation (7) is not predominate. The ADI method can be used for all kinds of thermal and fluid problems while SOR can work only for a Poison equation. Similar to SOR, the ADI method is also a numerical method and needs a number of iterations to converge to an exact solution. Because the ADI method finds an exact linear equation solution in one dimension, it can save iteration times to find a convergent solution in the whole domain. Therefore, a good parallel ADI algorithm can be used in many engineering and scientific computation aspects.

Thomas's algorithm has been widely used in other ADI method algorithms to solve the tridiagonal linear equation on serial machines. The Middle Node Algorithm presented in this paper is slower than Thomas's algorithm when only one processor is used. Both algorithms have same order notation, $\mathcal{O}(n)$, in general case, i.e. a_i, b_i and c_i have different values for $i = 1, ..., n$. But in many cases, they have identical value except for the two equations at boundary nodes, *i.e.* $i = 1$ and n. In this case, the calculation operations to find a_i, b_i, and c_i have to be taken only once on each level, and the Middle Node Algorithm will be much faster than Thomas's algorithm. Our algorithm is the best case with order $\mathcal{O}(\log(n))$ when we limit the input to isotropic problems with unique space step.

For parallel processing, the Middle Node Algorithm has the same speedup as the DAC algorithm when the total number of processor is less than 63. In this range, the effective processing speed will increase at the rate of about 33%. There are two factors that affect the speedup. One is the communication startup time. At the current time, even for IBM SP2, the startup time is almost 40 times of the calculation speed, which is why this number was chosen for Figure 5. For the Middle Node Algorithm, the speedup is cut mainly by the second factor, which is the idle processor. The executing processor number is cut in half as the level goes up. To improve the performance of the algorithm, it is good to choose the number of unknown nodes to be much larger than the total number of processors. If the problem is of the best case type, the speedup will increase significantly. In addition, the scalability will also improve, because the ratio of the data communication to the data calculation is much smaller.

The evaluation of parallel methods is a complicated project [5, 7]. Both speedup and scalability are two of the main parameters taken into account during the evaluation. Unfortunately, they are only the parallelization performance description of a specific algorithm. Now people usually take the speed of the same algorithm with one processor as the reference for speedup calculation. This leads to some problems in the final results. For example, the serial SOR method could be 64 times fast than the serial Gauss-Siedel method for some large-scale linear problem. A good parallel Gauss-Siedel algorithm with 64 processors can get a speedup of 64. It is still difficult to say whether or not it is a good parallel algorithm. Therefore, a good parallel algorithm for SOR must have a good order and good convergence speed, in addition to speedup and scalability.

In conclusion, we have developed a new parallel algorithm that matches or outperforms the other methods in our comparison group. We have shown that communication costs are the major part of its performance impact. This leads us to believe that our algorithm will be more significant in the future with faster communication networks.

References

[1] John D. Anderson. *Computational Fluid Dynamics.* McGraw Hill, 1995.

[2] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, 1995.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A User's guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, 1994.

[4] Willaim Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[5] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Reading, MA, 1992.

[6] Wolfgang Kollman. *Computation Fluid Dynamics.* Hemisphere Publishing Corporation, 1980.

[7] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[8] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kauffman, San Francisco, CA, October 1996.

[9] Suhas V. Patankar. *Numerical Hear Transfer and Fluid Flow.* Hemisphere Publishing Corporation, 1980.

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongerra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, 1996.

[11] Harold S. Stone. *High Performance Computer Architecture.* Addison-Wesley Publishing Company, 3rd edition, 1993.

[12] M. Wang and J.G. Georgiadis. Parallel computation of forced convection using domain decomposition. *Numerical Heat Transfer, Part B*, 20:41–59, 1991.

[13] M. Wang and S.P. Vanka. A parallel algorithm for high order finite-difference solution of the unsteady heat conduction equation, and its implementation on the CM-5. *Numerical Heat Transfer, Part B*, **24**:143–159, 1993.