A block SPP algorithm for multidimensional tridiagonal equations with optimal message vector length*

Hong Guo[a], Zhao-Hua Yin[b] and Li Yuan[a]

[a] LSEC and Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics & Systems Science, Chinese Academy of Sciences, Beijing 100190, P.R. China

[b] National Microgravity Laboratory, Institute of Mechanics, Chinese Academy of Sciences, Beijing 100190, P.R. China

Received 02 November 2007; Accepted 13 August 2008

ABSTRACT

A parallel strategy for solving multidimensional tridiagonal equations is investigated in this paper. We present in detail an improved version of single parallel partition (SPP) algorithm in conjunction with message vectorization, which aggregates several communication messages into one to reduce the communication cost. We show the resulting block SPP can achieve good speedup for a wide range of message vector length (MVL), especially when the number of grid points in the divided direction is large. Instead of only using the largest possible MVL, we adopt numerical tests and modeling analysis to determine an optimal MVL so that significant improvement in speedup can be obtained.

Key words: Tridiagonal equation, Single Parallel Partition, message vectorization, message vector length.

1. INTRODUCTION

The system of linear tridiagonal equations plays an important role in computational sciences. Many parallel strategies can be used together with serial algorithms for solving tridiagonal equations in multidimensions, e.g. the transpose strategy [1] and the pipelined method [2]. Moreover, some

---

*Email address: guoh@lsec.cc.ac.cn (Hong Guo), zhaohua.yin@imech.ac.cn (Zhao-hua Yin), lyuan@lsec.cc.ac.cn (Li Yuan).

optimization techniques like message vectorization [3, 4] can also be used together with direct parallel algorithms to solve multidimensional tridiagonal equations.

The first parallel algorithm for solving a tridiagonal system referred to as cyclic reduction was presented by Hockney in 1965 [5]. Stone introduced his recursive doubling algorithm in 1973 [6]. Among other parallel tridiagonal solvers, most are build on partitioning methods. One of them called P-scheme is very attractive for solving a very large system on a parallel computer [7]. However, the P-scheme is unstable when any off-diagonal element is close to zero. Of other direct parallel algorithms based on partitioning methods, the Single Parallel Partition (SPP) algorithm [8], which was modified from Wang's partition method [9], is also noticeable. Compared with other parallel algorithms, the SPP algorithm leads to a big reduction in data transport without any significant increase in the number of computational operations. However, the SPP could not yet achieve satisfactory speedup relative to the serial pursuit method on a single processor. This is because the computational count of the SPP is far more than that of the pursuit method.

When many unrelated (or, multidimensional) tridiagonal equations are to be solved, the performance of many parallel algorithms mentioned above can be improved more or less with the idea of message vectorization, which aggregates several data sending instead of "one by one" sending. In Ref.[10], Wakatani combined message vectorization with his parallel tridiagonal solver, the P-scheme, to solve two-dimensional ADI equations with a wide range of problem sizes, and super-linear speedup was observed when the size of the problem was 16386×16386.

Although message vectorization has proved useful when applied to the P-scheme, to the best knowledge of us, there are yet efforts so far to combine message vectorization with the SPP algorithm. In this paper, we combine message vectorization with the SPP. Section 2 briefly describes the original SPP algorithm, and its computational complexity is also presented. In section 3, we firstly provide the implementation of SPP with message vectorization, then present description of the resulting block SPP algorithm, and then analyze and test its parallel efficiency and performance on a parallel computer. We have found the best speedup does not correspond to the maximum MVL, and the block SPP algorithm performs well in high dimensional problems when the number of grid points in the divided direction is large enough. Conclusions are given finally.

2. THE ORIGINAL SPP ALGORITHM

2.1. A Brief Description of SPP

We consider a tridiagonal system of equations of order n:

$$Ax = \begin{pmatrix} d_1 & c_1 & & & \\ a_2 & d_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & d_{n-1} & c_{n-1} \\ & & & a_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} = b, \quad (1)$$

and we restrict to the situation where there exists a nonsingular coefficient matrix A. The matrix A is subdivided in p (the number of processors available) groups of k rows, and we assume $n = p \times k$. All processors have a local memory and the data have been spread over the local memories. The local memory of each processor contains only matrix- and vector-elements of k rows of the ith group.

SPP algorithm can be described as follows:

1. Each processor (denote as N_0, N_1, …, N_{p-1}) reads its own data;
2. For N_0, reduce d_1 to 1, then eliminate a_2, then reduce d_2 to 1, and go on until a_i, $i = 2, …, k$ are all eliminated;
   For N_{p-1}, reduce d_n to 1, then eliminate c_{n-1}, then reduce d_{n-1} to 1, and go on until c_i, $i = n - k + 1, …, n$ are all eliminated;
   For N_i, $i = 1, …, p - 2$, first do the reduction as N_0, after all a_{ik+j}, $j = 1, …, k$ are eliminated, do the reduction as N_{p-1}, until c_{ik+j}, $j = 1, …, k$ are all eliminated;
3. For N_{p-1}, send $a_{(p-1)k+1}$ and $b_{(p-1)k+1}$ to N_{p-2};
   For N_i, $i = p - 2, …, 1$, receive elements sent from N_{i+1}, eliminate $a_{(i+1)k+1}$ on N_i, reduce the resulting $d_{(i+1)k+1}$ to 1 using the elements of $(i + 1)k$th line, and then eliminate c_{ik+1} on the first line. Send the new a_{ik+1} and b_{ik+1} to N_{i-1};
   For N_0, once receiving element a_{k+1} sent from N_1, eliminate a_{k+1} on N_0, and then reduce d_{k+1} to 1 and eliminate c_k. After the communication of data, eliminate c_i left on $N_1, …, N_{p-2}$;
4. For N_0, send b_k to N_1;
   For N_i, $i = 1, …, p - 2$, receive elements sent from N_{i-1}, then eliminate $a_{(i+1)k}$, and then send $b_{(i+1)k}$ to N_{i+1};
   For N_{p-1}, receive elements sent from N_{p-2}, and then eliminate $a_{(p-1)k+1}$.

5. When communications are completed, eliminate the nondiagonal elements remained on each processor. b_i, $i = 1, \ldots, n$, are the answers to eqn (1).

Here, it should be noted that computation in step 2 can be fully parallelized on p processors.

2.2. Computation and Communication Counts

The total time (T_{sum}) for each processor can be expressed as

$$T_{sum} = T_{comp} + T_{comm} = T_{comp} + T_{sendrecv} + T_{delay}, \tag{2}$$

where T_{comp} is the sum of the computation time and T_{comm} the communication time. T_{comm} can be divided into two parts: transmission time $(T_{sendrecv})$ and latency time (T_{delay}) (e.g. see [11]). According to [8], we can obtain the total time on multiprocessors for SPP:

$$T_{spp} = \left(21\frac{n}{p} - 26\frac{n}{p^2}\right)t_c + 6(p-1)t_{sendrecv} + 4(p-1)t_{delay}, \qquad p > 1. \tag{3}$$

Here, t_c is the per-element computational time in a single processor, $t_{sendrecv}$ is the time to transmit an element between processors, and t_{delay} is the latency time for message passing.

According to [8], SPP has operation counts of $\mathrm{O}(n)$ but the actual counts are smaller than most earlier parallel tridiagonal algorithms such as the cyclic reduction $(\mathrm{O}(n \log n))$[5]. The communication times between processors in the Cyclic Reduction are 2 log n, while in SPP they are $2(p-1)$. Therefore, SPP is more suitable for coarse-grained parallel computation, where $p \ll n$.

Fig. 1 shows speedups of SPP and the Cyclic Reduction for solving Eq. (1). The speedup factor is relative to the computing time of the pursuit method on a single processor. We can see SPP has larger speedups than the Cyclic Reduction.

It should be noted that the total time for the pursuit method on one computer is

$$T_{purs} = (8n - 7)t_c. \tag{4}$$

It is clear that although the operation counts of SPP are slightly smaller than those of other algorithms such as the P-scheme [7], they are still larger than those in pursuit method. It is difficult to reduce the computation time of SPP, so the only way to make SPP efficient is to lower the communication cost.

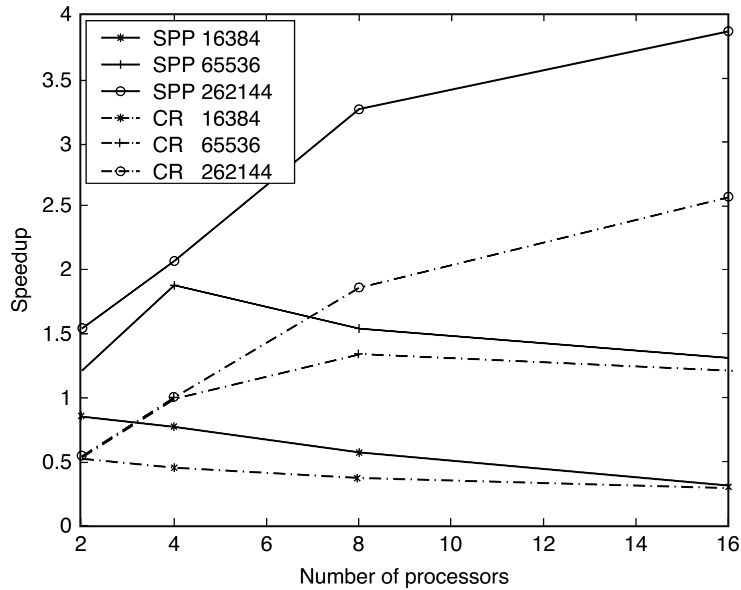

Figure 1. Speedups of the SPP and the cyclic reduction with varying system size n for Eq.(1).

3. THE BLOCK SPP ALGORITHM

In this section, we will try to use both modeling analysis and real runs on parallel computers to reduce T_{comm} on multidimensional tridiagonal systems. On modern parallel computers, $T_{sendrecv}$ is generally expected to be very low and T_{delay} should be reduced to make SPP efficient.

3.1. Message Vectorization

For multidimensional tridiagonal systems, SPP can be applied aggregately to several data instead of the "one by one" approach. Several data sent in one time can reduce the frequency for message passing, and the latency cost can be reduced dramatically. By aggregating m data into one message, the communication cost is reduced to $t_{delay} + mt_{sendrecv}$ instead of $m(t_{delay} + t_{sendrecv})$ [10]. In this paper, we denote SPP with message vectorization as the block SPP algorithm.

Without losing generality, we assume that only one dimension of arrays is distributed among processors, and the SPP scheme can be implemented in the divided dimension. We take three dimensional case as example, and let (i_{dm}, j_{dm}, k_{dm}) denote the number of points in x, y and z coordinate directions, respectively. If x direction of the grid is divided across the number of

processors, the size of message transmitted from one processor to another in one communication (i.e. MVL) can be arranged from 1 to $j_{dm} \times k_{dm}$.

3.2. Description of Block SPP

In the present program structure of the block SPP, we assume that the x direction of the grid is evenly divided by p processors. For simplicity, here we use a 2D problem to present the block SPP algorithm.

Denote MVL as m. For $1 \leq m \leq j_{dm}$ and $i_{dm} = p \times k$, the process of the block SPP is

$$for\ L = 1,\ \ldots,\ \frac{j_{dm}}{m}\ \ do$$

$$for\ J = m(L-1)+1,\ \ldots,\ mL\quad do$$

$$\{\{$$

$$g_{ik+1,J} \leftarrow \frac{c_{ik+1,J}}{d_{ik+1,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$b_{ik+1,J} \leftarrow \frac{b_{ik+1,J}}{d_{ik+1,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$f_{ik+1,J} \leftarrow \frac{a_{ik+1,J}}{d_{ik+1,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$for\ j = 2,\ \ldots,\ k\quad do$$

$$\{$$

$$df_{ik+j,J} \leftarrow d_{ik+j,J} - g_{ik+j-1,J} \times a_{ik+j,J}, \qquad i = 0,\ \ldots,\ p-2$$

$$g_{ik+j,J} \leftarrow \frac{c_{ik+j,J}}{df_{ik+j,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$b_{ik+j,J} \leftarrow \frac{b_{ik+j,J} - b_{ik+j-1,J} \times a_{ik+j,J}}{df_{ik+j,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$f_{ik+j,J} \leftarrow \frac{-f_{ik+j-1,J} \times a_{ik+j,J}}{df_{ik+j,J}}, \qquad i = 0,\ \ldots,\ p-2$$

$$\}$$

$$f_{ik+k,J} \leftarrow \frac{a_{ik+k,J}}{d_{ik+k,J}}, \qquad i = p-1$$

$$f_{ik+k,J} \leftarrow \frac{f_{ik+k,J}}{d_{ik+k,J}}, \qquad i = 1,\ \ldots,\ p-2$$

$b_{ik+k,J} \leftarrow \dfrac{b_{ik+k,J}}{d_{ik+k,J}}, \qquad i = 1, \ldots, p-1$

$g_{ik+k,J} \leftarrow \dfrac{g_{ik+k,J}}{d_{ik+k,J}}, \qquad i = 1, \ldots, p-2$

for $j = k-1, \ldots, 1$ *do*

{

$df_{ik+j,J} \leftarrow d_{ik+j,J} - c_{ik+j,J} \times f_{ik+j+1,J}, \qquad i = p-1$

$f_{ik+j,J} \leftarrow \dfrac{a_{ik+j,J}}{df_{ik+j,J}}, \qquad i = p-1$

$b_{ik+j,J} \leftarrow \dfrac{b_{ik+j,J} - b_{ik+j+1,J} \times g_{ik+j,J}}{df_{ik+j,J}}, \qquad i = p-1$

$f_{ik+j,J} \leftarrow f_{ik+j,J} - f_{ik+j+1,J} \times g_{ik+j,J}, \qquad i = 1, \ldots, p-2$

$b_{ik+j,J} \leftarrow b_{ik+j,J} - b_{ik+j+1,J} \times g_{ik+j,J}, \qquad i = 1, \ldots, p-2$

$g_{ik+j,J} \leftarrow -g_{ik+j,J} \times g_{ik+j+1,J}, \qquad i = 1, \ldots, p-2$

}

}

do the latter from N_{p-1} *to* N_0 *one by one*

{

Recv $f_{(i+1)k+1,J}$, $b_{(i+1)k+1,J}$, *from* N_{i+1},

$J = m(L-1)+1, \ldots, mL; \quad i = p-2, \ldots, 0$

for $J = m(L-1)+1, \ldots, mL$ *do*

{

$df_{(i+1)k+1,J} \leftarrow 1 - f_{(i+1)k+1,J} \times g_{(i+1)k,J}, \qquad i = p-2, \ldots, 0$

$b_{(i+1)k+1,J} \leftarrow \dfrac{b_{(i+1)k+1,J} - b_{(i+1)k,J} \times f_{(i+1)k+1,J}}{df_{(i+1)k+1,J}}, \qquad i = p-2, \ldots, 0$

$f_{(i+1)k+1,J} \leftarrow \dfrac{-f_{(i+1)k,J} \times f_{(i+1)k+1,J}}{df_{(i+1)k+1,J}}, \qquad i = p-2, \ldots, 1$

$b_{ik+1,J} \leftarrow b_{ik+1,J} - b_{(i+1)k+1,J} \times g_{ik+1,J}, \qquad i = p-2, \ldots, 1$

$f_{ik+1,J} \leftarrow f_{ik+1,J} - f_{(i+1)k+1,J} \times g_{ik+1,J}, \qquad i = p-2, \ldots, 1$

$$b_{k,J} \leftarrow b_{k,J} - b_{k+1,J} \times g_{k,J}, \qquad i = 0$$

}

Send $f_{ik+1,J}$, $b_{ik+1,J}$ *to* N_{i-1},

$$J = m(L-1)+1, \ldots, mL; \quad i = p-1, \ldots, 1$$

for $J = m(L-1)+1, \ldots, mL$ *do*

for $j = 2, \ldots, k$ *do*

{{

$$b_{ik+j,J} \leftarrow b_{ik+j,J} - b_{(i+1)k+j,J} \times g_{k+j,J}, \qquad i = 1, \ldots, p-2$$

$$f_{ik+j,J} \leftarrow f_{ik+j,J} - f_{(i+1)k+j,J} \times g_{k+j,J}, \qquad i = 1, \ldots, p-2$$

}}

}

do the latter from N_0 *to* N_{p-1} *one by one*

{

Recv $b_{(i+1)k+1,J}$ *from* N_{i+1},

$$J = m(L-1)+1, \ldots, mL; \quad i = 1, \ldots, p-1$$

for $J = m(L-1)+1, \ldots, mL$ *do*

{

$$b_{(i+1)k,J} \leftarrow b_{(i+1)k,J} - b_{ik,J} \times f_{(i+1)k,J}, \qquad i = 1, \ldots, p-2$$

}

Send $b_{(i+1)k,J}$ *to* N_{i+1},

$$J = m(L-1)+1, \ldots, mL; \quad i = 0, \ldots, p-2$$

}

for $J = m(L-1)+1, \ldots, mL$ *do*

for $j = 1, \ldots, k-1$ *do*

{{

$$b_{ik+j,J} \leftarrow b_{ik+j,J} - b_{(i+1)k+j,J} \times f_{ik+j,J}, \qquad i = 1, \ldots, p-2$$

}}

for $J = m(L-1)+1, \ldots, mL$ *do*

$$
\begin{array}{l}
for \;\; j = 1, \ldots, k \quad do \\
\quad \{\{ \\
\qquad b_{ik+j,J} \leftarrow b_{ik+j,J} - b_{ik+j-1,J} \times f_{ik+j,J}, \qquad\qquad i = p-1 \\
\quad \}\} \\
\}
\end{array}
$$

It should be noted here that j and J correspond to different variables.

In the following, we will try to follow the structure of the program described above to analyze the parallel characteristics of the block SPP. The performance of a parallel algorithm is affected by many factors such that it is diffcult to develop an exact model. What we do here is to simply employ an approximate parallel model which has been used before in the block pipelined method [2].

The number of iterations for all the message vectors to be sent is

$$l = \frac{j_{dm}}{m} \tag{5}$$

In the data propagation process of the block SPP algorithm, all processors except the first one must wait for the data to be sent by the previous processor, and the time for the last processor to receive the message will be the time for the first processor to begin its pth iteration sending. The job is not done until all processors finish their own iterations. Therefore, the whole number of sending iterations is $l + p - 1$. Fig. 2 shows an example for $l = p = 4$. From Fig. 2 we can see, the computation and data sending with the same number can be done simultaneously.

In three dimensional cases, $l = \frac{j_{dm} \times k_{dm}}{m}$, we can get the total time used for solving all $j_{dm} \times k_{dm}$ equations in x direction:

$$
\begin{aligned}
T_{sum} &= (l + p - 1)\left[6\frac{j_{dm}k_{dm}}{l} t_{sendrecv} + 4t_{delay} + \varepsilon(i_{dm}, p)\frac{j_{dm}k_{dm}}{l} t_c\right] \\
&\quad + \varepsilon l\left[\frac{j_{dm}k_{dm}}{l}\left(21\frac{i_{dm}}{p} - 26\frac{i_{dm}}{p^2}\right)t_c\right] \\
&= a / l + b \times l + c
\end{aligned}
\tag{6}
$$

where

$$
\begin{aligned}
a &= 6j_{dm}k_{dm}(p-1)t_{sendrecv} + \varepsilon(i_{dm}, p)j_{dm}k_{dm}(p-1)t_c, \\
b &= 4t_{delay},
\end{aligned}
$$

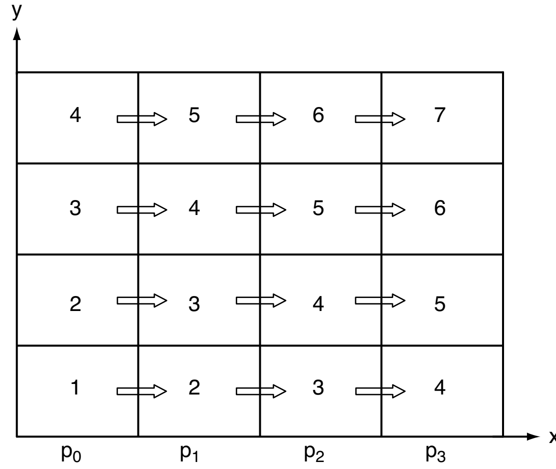

Figure 2. Data sending iterations.

$$c = 6j_{dm}k_{dm}t_{sendrecv} + 4(p-1)t_{delay} + \varepsilon i_{dm}\, j_{dm}k_{dm}\left(\frac{21}{p}-\frac{26}{p^2}\right)t_c + \varepsilon(i_{dm},\ p)\ j_{dm}k_{dm}t_c,$$

and $\varepsilon(i_{dm},\ p) < 1$ is a factor representing the influence of the cache hit rate on computational time. $(i_{dm},\ p)$ means ε depends on the the grid numbers, number of processors and maybe other factors. It should be mentioned that $\varepsilon(i_{dm},\ p)$ $\frac{j_{dm}k_{dm}}{l}t_c$ is an idle time term for waiting in receive for other processors to complete their computations before they can send. While in Eq. (3) there is no such term because the idle time is so small that can be neglected. When the grid number is large enough, for the fixed p, $\varepsilon(i_{dm},\ p)$ will be larger when i_{dm} is larger. It is easy to show there is an equilibrium l, which makes T_{sum} minimal:

$$l_{opt} = \sqrt{\frac{a}{b}}. \tag{7}$$

Thus we have the optimal message vector length:

$$\begin{aligned} m_{opt} &= j_{dm} \times k_{dm}\sqrt{\frac{b}{a}} \\ &= \sqrt{\frac{4_{jdm}k_{dm}t_{delay}}{(6t_{sendrecv} + \varepsilon(i_{dm}, p)t_c)(p-1))}}. \end{aligned} \tag{8}$$

The idea of message vectorization gives people the feeling that larger MVL will lead to better parallel efficiency since coarser grained parallelism may save the latency time more. However, this is not always the case since normally we have

$$\frac{4t_{delay}}{(6t_{sendrecv} + \varepsilon(i_{dm}, p)t_c)(p-1))} < j_{dm}k_{dm}, \qquad (9)$$

therefore $1 < m_{opt} < j_{dm}k_{dm}$.

Moreover, from Eq. (8), we can easily get the following conclusions:

1. For the fixed p and $j_{dm} \times k_{dm}$, m_{opt} is smaller when i_{dm} is larger.
2. For the fixed $i_{dm} \times j_{dm} \times k_{dm}$, m_{opt} is smaller when p is larger.

These conclusion will be confirmed in the following numerical tests.

3.3. Experiment with the Block SPP and Optimal Message Vector Length in 3D Tridiagonal Linear Systems

We firstly apply the block SPP algorithm presented in section 3.2 with MPI FORTRAN on Lenovo DeepComp 1800 cluster, which has two Intel 2GHz Xeon CPUs on each node of total 256 nodes and a 512KB cache size, connected with Myrinet 2000 with 10 microseconds latency and 2Gbps bandwidth. The block SPP is implemented in the x direction which has the largest number of grid points. We measure the wall time for executing only the x direction sweep. The speedup factor is the wall clock time of the pursuit method divided by that of the block SPP algorithm in the same resolution. Three different grid resolutions are used: 64×64^2, 256×64^2 and 1024×64^2, and the possible values of MVL are from 1 to 4096.

Fig. 3 shows speedups for the 64×64^2 resolution. We can see that although the speedup of the block SPP is better than the original SPP without any message vectorization (or MVL equals to 1), it is far lower than the ideal speedup for such small-scale problems. This is due to larger computational complexities of the SPP compared with the pursuit method. For the second resolution problem, the speedup is slightly better (Fig. 4).

The situation is quite different when i_{dm} is larger. For the 1024×64^2 problem (Fig. 5), we see super-linear speedups are achieved for MVL in the range of 2 to 4096 on 2 processors and 64 to 256 on 4 processors, respectively. When the size of array is large, the effect of the computational complexity is counteracted by the improved cache hit rate. Although there is no super-linear speedup achieved on 8 processors, the parallel efficiency of 1024×64^2 is much better than that of the 256×64^2 problem.

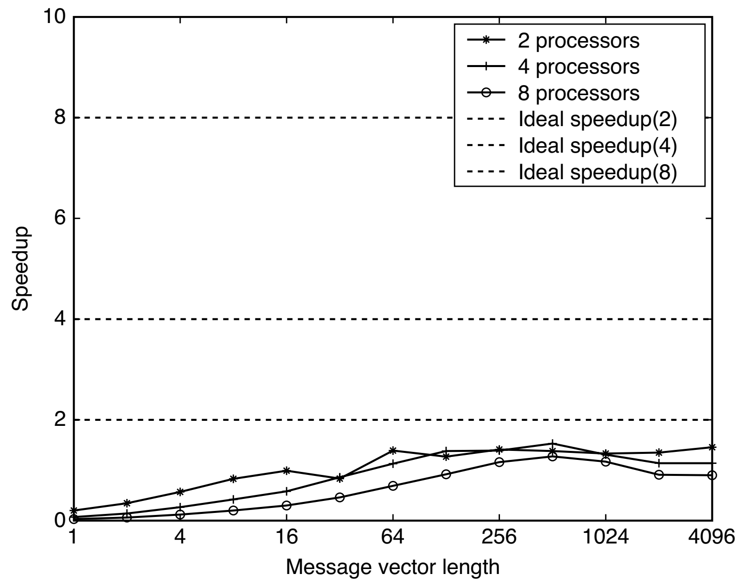

Figure 3. Speedups of the SPP with message vectorization for a 64×64^2 problem on Lenovo DeepComp 1800 cluster.

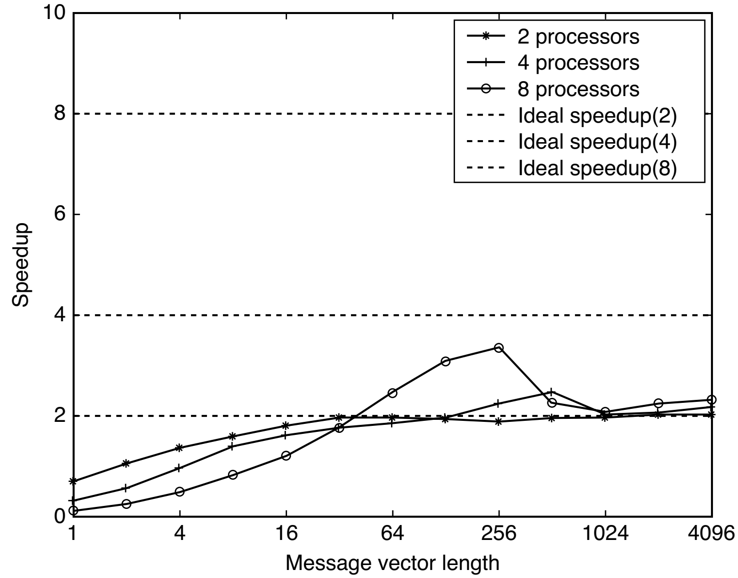

Figure 4. Speedups of the SPP with message vectorization for a 256×64^2 problem on Lenovo DeepComp 1800 cluster.

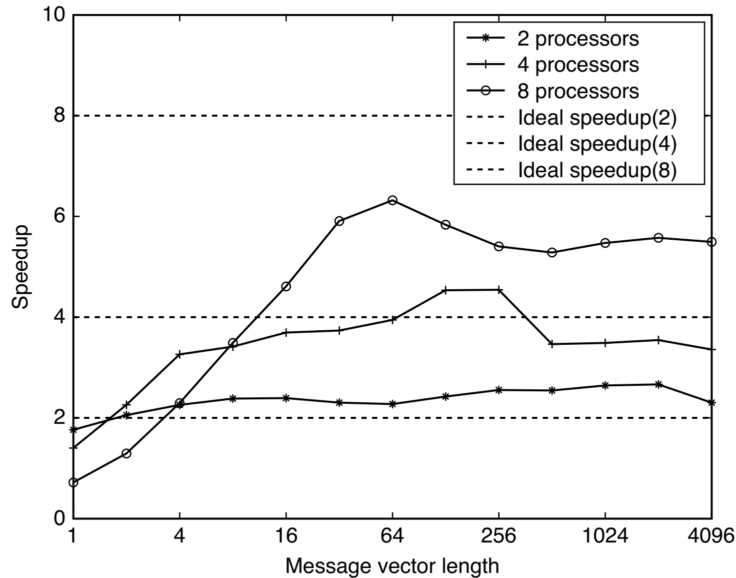

Figure 5. Speedups of the SPP with message vectorization for a 1024×64^2 problem on Lenovo DeepComp 1800 cluster.

As predicted in the previous subsection, the maximum speedup does not always occur at the largest MVL but rather at some intermediate MVL. In the case of the 64^3 problem, the optimal MVL is 4096 only in the case of 2 processors. For 4 or 8 processors, the optimal MVL is 512. In the 256×64^2 and 1024×64^2 problems, the optimal MVL never occurs at the largest MVL. Moreover, as predicted by our model, the value of optimal MVL decreases when p becomes larger. In the case of 1024×64^2 problem, the value of optimal MVL will be divided by four when p is doubled each time.

Table 1. shows the relation of p and optimal MVL for three resolutions, and we can see that m_{opt} becomes smaller when idm gets larger for fixed p and $j_{dm} \times k_{dm}$. Thus, conclusions drawn from Eq. (8) are verified.

Fig. 6 shows the optimal speedups of our code vs. CPU numbers on Lenovo DeepComp 1800 cluster. The values adopted here are the speedup factors when optimal MVL is applied for certain p and resolution. We see good parallel efficiency is obtained with the optimal MVL for the 1024×64^2 resolution.

We also test the block SPP algorithm on SGI Origin3800 which has a large cache (8M) and fast interconnect (0.5GB/s of data bandwidth). From Table 2, we can observe the same varying trends as in Table 1. It is again show that our

Table 1. The values of optimal MVL for different resolutions and processor numbers on Lenovo DeepComp 1800 cluster.

	2 Processors	**4 Processors**	**8 Processors**
64×64^2	MVL = 4096	MVL = 512	MVL = 512
256×64^2	MVL = 2048	MVL = 512	MVL = 256
1024×64^2	MVL = 2048	MVL = 256	MVL = 64

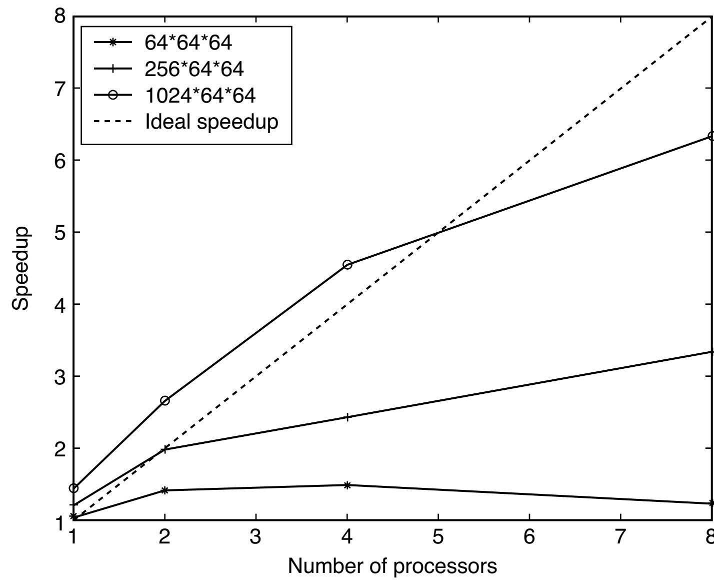

Figure 6. Optimal Speedups on Lenovo DeepComp 1800 cluster.

Table 2. The values of optimal MVL for different resolutions and processor numbers on SGI Origin 3800.

	2 Processors	**4 Processors**	**8 Processors**
64×64^2	MVL = 4096	MVL = 1024	MVL = 512
256×64^2	MVL = 2048	MVL = 1024	MVL = 256
1024×64^2	MVL = 2048	MVL = 512	MVL = 128

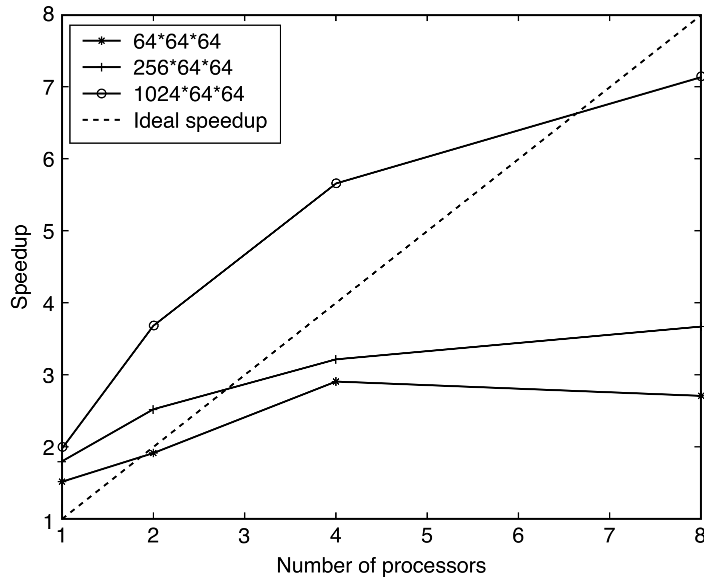

Figure 7. Optimal Speedups on SGI Origin 3800.

analyzing model (Eq. (8)), although simple, is effective in explaining the parallel behavior of our block SPP algorithm.

Fig. 7 also shows an optimal speedup figure as Fig. 6 on SGI Origin 3800. The speedups are a little better than those on Lenovo DeepComp 1800 cluster because of the large cache on SGI Origin 3800.

4. APPLICATION IN FLUID FLOW SIMULATION

We implement the present block SPP algorithm into a finite-difference flow solver [12] where the artificial compressibility method is adopted to solve the 3D incompressible Navier-Stokes equations. We compute the spherical Couette flow between two concentric rotating spheres (the inner one rotating and the outer one stationary). The resulting discretized linear algebraic equations are solved with a diagonalized approximate factorization scheme, leading to a 3D ADI scheme (see [12] for more details).

The computation adopts $360 \times 153 \times 34$ grid points on 24 CPUs on Lenovo DeepComp 1800 cluster. The block SPP is only implemented in the first dimension, and the optimal MVL is found to be 153 by numerical test. Fig. 8 shows computed

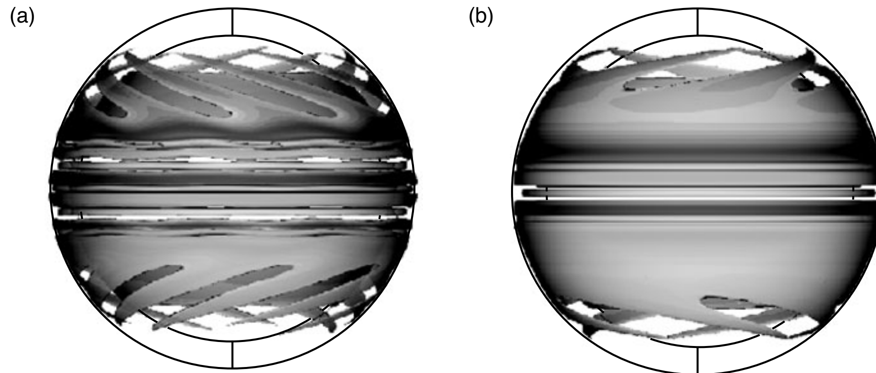

Figure 8. Isocontours of meridional velocity v_m = 0.11 for the spherical Couette flow at Re = 7200 (a) and Re = 8500 (b). The resolution is 360 $\times$ 153 $\times$ 34.

spiral vortex patterns. Fig. 8(a) shows nine spiral vortices in the polar region and two pairs of toroidal Taylor vortices near the equator, and Fig. 8(b) shows five spiral vortices near the polar region and one pair of Taylor vortex near the equator. These newly simulated flow patterns agree with experiments in Ref. [13].

5. CONCLUSIONS

In this paper, we have presented an improved version of the SPP algorithm with message vectorization for solving multidimensional tridiagonal equations, and demonstrated that good speedup for 3D problems can be obtained with the block SPP algorithm. We have also presented a simple parallel model which can forecast the optimal MVL. The use of optimal MVL leads to better speedup. Because we only use the block SPP in the longest dimension of a 3D problem, current work has good speedup only for small number of processors (e.g., less than 24 CPUs). Other directions should be divided to reach good parallel efficiency for larger-scale computations, and this will be done in the future.

ACKNOWLEDGEMENTS

This project is supported by NSF of China (G10502054, G10432060) and CAS Innovation Program. LY is supported by NSF of China (G10476032, G10531080).

REFERENCES

[1] Edison, T. W. and Erlebacher, G., Implementation of a fully-balanced periodic tridiagonal solver on a parallel distributed memory architecture, *Concurrency-pract EX 7*, 1995, 4, 273–302.

[2] Zhang, L. B., On pipelined computation of a set of recurrences on distributed memory systems, *Journal on Numerical Methods and Computer Applications*, 1999, 3, 184–191.

[3] Balasundaram, V., Fox, G., Kennedy, K. and Kremer, U., An Interactive Environment for Data Partitioning and Distribution, in: Walker, D.W. and Stout, Q.F. ed., *Architectures, Software Tools and Other General Issues (Vol. 2): Proceedings of the 5th Distributed Memory Computing Conference*, ACM Press, New York, 1990, 1160–1170.

[4] Hall, M. W., Hiranandani, S., Kennedy, K. and Tseng, C. W., Interprocedural compilation of fortran d for mimd distributed-memory machines, in: Werner, R., ed., *Conference on High Performance Networking and Computing: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, 522–534.

[5] Hockney, R. W., A fast direct solution of poissons equation using fourier analysis, Appendix Golub, G. H., *Journal of the Association for Computing Machinery*, 1965, 12, 95–113.

[6] Hockney, R. W., An effcient parallel algorithm for the solution of a tridiagonal linear system of equations, *Journal of the Association for Computing Machinery*, 1973, 20, 27–38.

[7] Wakatani, A., A parallel scheme for solving a tridiagonal matrix with prepropagation, *LECTURE NOTES IN COMPUTER SCIENCE*, 2003, 2840, 222–226.

[8] Wang, C. R., Wang, Z. H. and Yang, X. H., *Computational Fluid Dynamics and Parallel Algorithms*, National University of Defence Technology Press, Changsha, CHINA, 2000.

[9] Wang, H. H., A Parallel Method for Tridiagonal Equations, *ACM Transactions on Mathematical Software*, 1981, 7, 170–183.

[10] Wakatani, A., A parallel and scalable algorithm for ADI method with pre-propagation and message vectorization, *Parallel Computing*, 2004, 30, 1345–1359.

[11] Yin, Z., Yuan, L. and Tang, T., A new parallel strategy for two-dimensional incompressible flow simulations using pseudo-spectral methods, *Journal of Computational Physics*, 2005, 210, 325–341.

[12] Yuan, L., Comparison of implicit multigrid schemes for three-dimensional incompressible flows, *Journal of Computational Physics*, 2002, 177, 134–155.

[13] Wimmer, M., Experiments on a viscous fluid flow between concentric rotating spheres, *Journal of Fluid Mechanics*, 1976, 78, 317–335.