Applied Mathematics and Computation 215 (2009) 2565–2577

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

On optimal message vector length for block single parallel partition algorithm in a three-dimensional ADI solver

Li Yuan [a,*], Hong Guo [a], Zhaohua Yin [b]

[a] LSEC and Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, PR China
[b] National Microgravity Laboratory, Institute of Mechanics, Chinese Academy of Sciences, Beijing 100190, PR China

ARTICLE INFO

Keywords:
Tridiagonal equation
Single parallel partition
ADI scheme
Message vectorization
Optimal message vector length

ABSTRACT

It has long been recognized that many direct parallel tridiagonal solvers are only efficient for solving a single tridiagonal equation of large sizes, and they become inefficient when naively used in a three-dimensional ADI solver. In order to improve the parallel efficiency of an ADI solver using a direct parallel solver, we implement the single parallel partition (SPP) algorithm in conjunction with message vectorization, which aggregates several communication messages into one to reduce the communication costs. The measured performances show that the longest allowable message vector length (MVL) is not necessarily the best choice. To understand this observation and optimize the performance, we propose an improved model that takes the cache effect into consideration. The optimal MVL for achieving the best performance is shown to depend on number of processors and grid sizes. Similar dependence of the optimal MVL is also found for the popular block pipelined method.

1. Introduction

The alternating direction implicit (ADI) method is widely used for solving multidimensional partial differential equations in computational sciences such as computational fluid dynamics [1–6]. The ADI method is an iterative algorithm, which divides every iteration step into the same number of sub-steps as the number of dimensions of the problem and updates the variable by solving a tridiagonal (or multidiagonal) system of equations every sub-step [7]. ADI method can be easily implemented on shared-memory parallel computers, but it is hard to be parallelized on distributed-memory parallel computers since heavy communications are required. The efficiency of a 2D or 3D parallel ADI solver depends partially on the 1D tridiagonal solver used, and partially on the overall parallelization strategy of the ADI solver. Intensive research has been done to develop better parallel tridiagonal solvers as well as overall parallel strategies. A good survey can be found in references [8,9]. Here we give a brief review of some developments in the two aspects.

In the aspect of direct parallel tridiagonal solvers, a large number of algorithms with different performances were developed in the past four decades. Duff and van der Vorst [10] divided these parallel algorithms into four classes based on the factorization of the tridiagonal matrix used:

1. Twisted factorization, first introduced by Babuska [11], then discussed by van der Vorst [12].
2. Recursive doubling, proposed by Stone [13]. The more recent P-scheme (pre-propagation scheme) proposed by Wakatani [14,15] also used double recurrences but no LU decomposition.

* Corresponding author.
E-mail addresses: lyuan@lsec.cc.ac.cn (L. Yuan), guoh@lsec.cc.ac.cn (H. Guo), zhaohua.yin@imech.ac.cn (Z. Yin).

0096-3003/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2009.08.052

3. Cyclic reduction, originally appeared in Hockney's paper [16], later elaborated in [17].
4. Divide and conquer (DAC) approaches, also fall into two classes for different factorizations of the tridiagonal matrix A:
   - $A = L \times R$ (left matrix multiply right matrix), the well known factorization presented by Wang [18]; later a modification was made by Michielse and van der Vorst [19]; parallel Cholesky factorization was presented by Bar-on [20], and block partitioned (Cholesky) factorization algorithm by Lin and Roest [21]; Lawrie and Samehs' algorithm [22], Johnsson's algorithm [23] and Mattor's algorithm [24] should also be mentioned here;
   - $A = \Delta A + A_0$ (where ΔA is the aggregation of some upper and lower diagonal elements cut from A, and A_0 is the residual), the method proposed in [25,26] and also named divide and conquer by Bondeli [26], and after that DAC is used to refer to a strategy. The method of Bondeli can also be combined with recursive doubling, cyclic reduction or Wang's partition method. Later on DAC was applied for solving block tridiagonal systems [27]. Sun proposed the parallel partition LU (PPT) algorithm, the parallel partition hybrid (PPH) algorithm and the parallel diagonal dominant (PDD) algorithm [28,29].

In terms of generality and simplicity, Wang's partition method [18] is very attractive for solving a very large system on a parallel computer. However, it needs large data transport, which costs too much time in communication. Later, a method modified from Ref. [18] and named single parallel partition algorithm (SPP) was introduced in Ref. [30]. This algorithm, by modifying the sequence of elimination and reducing the diagonal elements to one, results in considerable reduction in waiting time and transported data, whereas the number of operation for executing the complete algorithm does not increase significantly.

With aspect to the overall parallel strategy for an ADI solver, many methods such as the transpose strategy [31], the pipelined method [32], the skewed block data partition [33], and the message vectorization [15,34,35], were developed. The transpose strategy can take full advantage of the classic pursuit method (Gauss elimination specialized for linear tridiagonal equations), but it is not efficient because of huge communication requests. It was shown that the transpose strategy is slower than the message vectorization scheme [31]. The skewed block data partition [33] seems a bit more complicated.

The message vectorization [15,34,35] is aggregate data sending instead of "one by one" sending. Wakatani [15] combined message vectorization with his parallel tridiagonal solver, the P-scheme [14], to solve two-dimensional ADI equations with a wide range of problem sizes, and obtained super-linear speedup for a $16,386 \times 16,386$ problem. It has long been recognized that message vectorization can improve the efficiency of a parallel ADI solver, but how the message vector length (MVL) affects the performance is not well understood.

The pipelined method in Ref. [32] used the pursuit method in a pipelined fashion. Good parallel efficiency was obtained in 2D and 3D problems. Later an improved version named the block pipelined method was developed in Ref. [36]. Actually, the main idea of blocking the pipelined method is similar to that of the message vectorization, which is to apply the basic tridiagonal-system solver aggregately to a block instead of a single line. The results in Ref. [36] shown that the block pipelined method has higher parallel efficiency than the "one by one" pipelined method.

In this paper, we study a parallel ADI solver which combines the above mentioned SPP algorithm [30] with message vectorization technique, and analyze its performance with emphasis on the role of the optimal MVL in maximizing the computational efficiency. Because previous studies on parallelization of ADI solvers are numerous, it is worthwhile to point out the peculiarity of the present work. First, after analyzing and testing several typical parallel tridiagonal solvers, i.e., the SPP [30] (DAC), the P-scheme [15] (recursive doubling), and the cyclic reduction [16], we find that both communication and computation costs of the SPP are the least of them. Second, the SPP is combined with message vectorization for an ADI solver, which has not been done in literature. Third, we present an improved parallel model to predict the optimal MVL. Finally, we show that other parallel ADI solvers like the block pipelined method can also benefit from using an optimal MVL.

The paper is organized as follows: Section 2 briefly describes our ADI scheme used for solving the incompressible Navier–Stokes equations, Section 3 provides the description of the original SPP algorithm and a comparison with the P-scheme. Section 4 presents implementation of the SPP with message vectorization, analysis with an existing model, and a refined model to predict the optimal MVL. The parallel performance of the resulting block SPP is also shown. Extension of the optimal MVL concept to the block pipelined method is given in Section 5. Conclusions are given finally.

2. The ADI scheme for the 3D Navier–Stokes equations

Our ADI scheme is constructed for finite difference solution to the three-dimensional unsteady incompressible Navier–Stokes equations in artificial compressibility formulation, which is written in generalized curvilinear coordinates (ξ, η, ζ) as

$$\frac{\partial \mathbf{Q}}{\partial \tau} + \mathbf{I}_m \frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial (\mathbf{E} - \mathbf{E}_v)}{\partial \xi} + \frac{(\mathbf{F} - \mathbf{F}_v)}{\partial \eta} + \frac{\partial (\mathbf{G} - \mathbf{G}_v)}{\partial \zeta} = \mathbf{0}, \tag{1}$$

where $\mathbf{Q} = \frac{1}{J}(p, u, v, w)^T$, with p being the static pressure, and u, v and w the velocity components in Cartesian coordinates, respectively. J is the Jacobian of coordinate transformation, $\mathbf{I}_m$ is a modified identity matrix whose first element is zero, $\mathbf{E}, \mathbf{F}$ and $\mathbf{G}$ are the convective flux vectors, and $\mathbf{E}_v, \mathbf{F}_v$ and $\mathbf{G}_v$ are the viscous flux vectors. By using the dual-time stepping technique, the Beam–Warming approximate factorization [2], and the diagonalized form as given in Ref. [6], we have

$$\mathbf{P}_1[\mathbf{D}+h\delta_\xi(\mathbf{\Lambda}_1-\gamma_1\mathbf{I}\delta_\xi)]\mathbf{P}_1^{-1}\mathbf{D}^{-1}\mathbf{P}_2\left[\mathbf{D}+h\delta_\eta\left(\mathbf{\Lambda}_2-\gamma_2\mathbf{I}\delta_\eta\right)\right]\mathbf{P}_2^{-1}\mathbf{D}^{-1}\mathbf{P}_3[\mathbf{D}+h\delta_\zeta(\mathbf{\Lambda}_3-\gamma_3\mathbf{I}\delta_\zeta)]\mathbf{P}_3^{-1}\Delta\mathbf{Q}^{n+1,m}=\mathbf{R}(\mathbf{Q}^{n+1,m}), \tag{2}$$

where $h=\Delta\tau J$, $\Delta\mathbf{Q}^{n+1,m}=\mathbf{Q}^{n+1,m+1}-\mathbf{Q}^{n+1,m}$, $\mathbf{D}=(1+1.5\frac{\Delta\tau}{\Delta t})\mathbf{I}$, $\delta_\xi,\delta_\eta,\delta_\zeta$ are the finite difference operators, n is the physical time level, m is the pseudo-time level, γ_1,γ_2 and γ_3 are viscous coefficients, $\mathbf{\Lambda}_1,\mathbf{\Lambda}_2$ and $\mathbf{\Lambda}_3$ are the diagonal matrices whose elements are eigenvalues of Jacobian matrices of the convective flux vectors in the ξ,η, and ζ directions, respectively, $\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3$ and $\mathbf{P}_1^{-1},\mathbf{P}_2^{-1},\mathbf{P}_3^{-1}$ are corresponding right and left eigenvector matrices in the three directions, respectively, and $\mathbf{R}$ is the residual operator. For more detail, see Ref. [6]. Numerical solution to Eq. (2) is equivalent to solving three one-dimensional linear systems of equations in sequence

$$[\mathbf{D}+h\delta_\xi(\mathbf{\Lambda}_1-\gamma_1\mathbf{I}\delta_\xi)]\Delta\mathbf{Q}^{***}=\mathbf{P}_1^{-1}\mathbf{R}(\mathbf{Q}^{n+1,m}), \tag{3a}$$

$$\left[\mathbf{D}+h\delta_\eta\left(\mathbf{\Lambda}_2-\gamma_2\mathbf{I}\delta_\eta\right)\right]\Delta\mathbf{Q}^{**}=\mathbf{P}_2^{-1}\mathbf{D}\mathbf{P}_1\Delta\mathbf{Q}^{***}, \tag{3b}$$

$$[\mathbf{D}+h\delta_\zeta(\mathbf{\Lambda}_3-\gamma_3\mathbf{I}\delta_\zeta)]\Delta\mathbf{Q}^{*}=\mathbf{P}_3^{-1}\mathbf{D}\mathbf{P}_2\Delta\mathbf{Q}^{**}, \tag{3c}$$

and then compute

$$\Delta\mathbf{Q}^{n+1,m}=\mathbf{P}_3\Delta\mathbf{Q}^{*},\qquad \mathbf{Q}^{n+1,m+1}=\mathbf{Q}^{n+1,m}+\Delta\mathbf{Q}^{n+1,m}. \tag{4}$$

If we choose the first-order upwind scheme for the convective terms and the second-order central scheme for the viscous terms in the left-hand sides of Eqs. (3a)–(3c) but use high-order finite difference schemes in the residual operator $\mathbf{R}$ to obtain desired high-order accuracy, then linear tridiagonal systems of equations can be obtained. Due to the use of diagonalization, only four independent scalar tridiagonal systems of equations will be solved along each grid line. The SPP algorithm to be described below is then used to solve each scalar tridiagonal system of equations.

3. The SPP algorithm

3.1. A brief description of the algorithm

Consider a tridiagonal system of linear equations of order n

$$Ax=\begin{pmatrix} d_1 & c_1 & & & & & \\ a_2 & d_2 & c_2 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & a_{n-1} & d_{n-1} & c_{n-1} \\ & & & & a_n & d_n \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}=\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}=b. \tag{5}$$

We restrict to the situation where there exists a unique solution for given right-hand side b and nonsingular coefficient matrix A. The matrix A is subdivided into p groups of k rows (p equals the number of processors denoted as $N_0,\ldots,N_{p-1}$, and for convenience we assume $n=p\times k$). All processors have a local memory and the data have been spread over the local memories so that the local memory of processor N_{i-1} contains only the matrix-elements and vector-elements of k consecutive rows of the ith group: $a_j,d_j,c_j,b_j,j=ik+1,\ldots,(i+1)k$.

The SPP algorithm [30] can be described in four steps:

Step 1. **Eliminate lower entries a_i and upper entries c_i using local data**.
For $N_i,i=0,\ldots,p-2$, reduces d_{ik+1} to 1, then eliminates a_{ik+2}, then reduces d_{ik+2} to 1, and go on forward until $a_{ik+j},j=2,\ldots,k$ are all eliminated;For $N_i,i=1,\ldots,p-1$, reduces d_{ik+k} to 1, then eliminates c_{ik+k-1}, then reduces d_{ik+k-1} to 1, and go on backward until $c_{ik+j},j=1,\ldots,k-1$ are all eliminated. Now we have all $d_i=1,i=1,\ldots,n$. This step can be readily parallelized in $p-1$ processors.

Step 2. **Eliminate c_i entries using non-local data**.
For N_{p-1}, send first-row elements $a_{(p-1)k+1}$ and $b_{(p-1)k+1}$ to N_{p-2}.
For $N_i,i=p-2,\ldots,1$, receive elements sent from N_{i+1}, use the elements of $(i+1)k$-th row to eliminate the newly received $a_{(i+1)k+1}$, reduce resulting $d_{(i+1)k+1}$ to 1, and then eliminate c_{ik+1} on the first row. After that, send the new a_{ik+1} and b_{ik+1} to N_{i-1}.For N_0, upon receiving element a_{k+1} and b_{k+1} sent from N_1, eliminate c_k on N_0, then reduce resulting d_k to 1. When the sending and receiving are completed, eliminate all c_i left on $N_1,\ldots,N_{p-2}$.

Step 3. **Eliminate a_i entries using non-local data**. For N_0, send b_k to N_1.
For $N_i,i=1,\ldots,p-2$, receive elements b_{ik} sent from N_{i-1}, eliminate last row's $a_{(i+1)k}$, 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 first row's $a_{(p-1)k+1}$. When the sending and receiving are completed, eliminate the left offdiagonal elements remained on each processor.

At the end of Step 3, $b_i,i=1,\ldots,n$ are the answers to Eq. (5).

3.2. Comparison with the P-scheme

According to [18,30], the SPP has fewer operation counts than most of earlier parallel tridiagonal algorithms such as the cyclic reduction [16]. It is emphasized that the diagonal elements need not be transferred after they are reduced to 1 in the SPP. However, data propagation stages in Steps 2 and 3 are still sequential.

For each processor on multiprocessors, the total time (T_{sum}) is the sum of the computation time (T_{comp}) and the communication time (T_{comm}), where (T_{comm}) consists of transmission time (T_{sendrecv}) and latency time (T_{delay}) [37]

$$T_{\text{sum}} = T_{\text{comp}} + T_{\text{comm}} = T_{\text{comp}} + T_{\text{sendrecv}} + T_{\text{delay}}. \tag{6}$$

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

For each data transport in one direction among processors one by one, processors should receive data first and then send it to the next one. Therefore, every processor will do transportation twice. Step 2 has 2 data transferred backward, and Step 3 has 1 data transferred forward, so the total transmitted data are 3. Furthermore, we assume computation is not overlapped with communication, and the computational time is equal for plus, minus, multiply and divide operations. Then we made a detailed count for all computational operations in the algorithm. In that way, we can get the total time for one processor on multiprocessors for the SPP

$$T_{\text{spp}} = \left(26\frac{n}{p} + 15p\right)t_c + 6(p-1)t_{\text{sendrecv}} + 4(p-1)t_{\text{delay}}, \quad p > 1. \tag{7}$$

Similarly, we can obtain the total time for one processor on multiprocessors for the P-scheme [14,15]

$$T_{\text{psch}} = \left(32\frac{n}{p} + 17p\right)t_c + 6(p-1)t_{\text{sendrecv}} + 6(p-1)t_{\text{delay}}, \quad p > 1. \tag{8}$$

Comparing Eqs. (7) and (8), we can find both the computation cost and the latency cost of the SPP algorithm are less than those of the P-scheme. To further demonstrate the superiority of the SPP over the P-scheme, we implement the SPP and the P-scheme for solving a scalar tridiagonal system of equations using MPI on LSSC2 Lenovo DeepComp 1800 cluster. This supercomputer has 512 computing nodes with each node having 2 Intel 2.0 GHz Xeon processors and 1.0 Gigabyte memory. Different nodes are connected through Myrinet 2000 network. A careful measurement on this machine shows that $t_{\text{delay}} \simeq 5 \times 10^{-6}$ s, $t_{\text{sendrecv}} \simeq 2 \times 10^{-8}$ s, and t_c is around 10^{-9} s.

Table 1 shows the measured total times and predicted ones based on Eq. (7) for the SPP algorithm on LSSC2. Table 2 shows the measured total times and the predicted by Eq. (8) for the P-scheme algorithm. We can see that the SPP algorithm needs less wall time than the P-scheme for the same n and p. Thus we show that the SPP algorithm is faster than the P-scheme under the same condition.

The computational cost of the pursuit method (Gauss elimination) on one processor is (cf. [26]):

$$T_{\text{purs}} = (8n - 7)t_c. \tag{9}$$

Table 1
Timing results of the SPP algorithm.

p	$n = 10^4$		$n = 10^5$		$n = 10^6$	
	Measured	Predicted	Measured	Predicted	Measured	Predicted
2	1.99E−4	2.15E−4	1.83E−3	1.97E−3	1.21E−2	1.95E−2
4	1.37E−4	1.58E−4	1.20E−3	1.04E−3	7.52E−3	9.81E−3
8	2.13E−4	1.90E−4	8.29E−4	6.29E−4	4.93E−3	5.02E−3
10	2.45E−4	2.20E−4	7.01E−4	5.71E−4	4.14E−3	4.08E−3
16	3.66E−4	3.27E−4	7.56E−4	5.46E−4	3.03E−3	2.74E−3

Table 2
Timing results of the P-scheme algorithm.

p	$n = 10^4$		$n = 10^5$		$n = 10^6$	
	Measured	Predicted	Measured	Predicted	Measured	Predicted
2	2.35E−4	2.70E−4	2.15E−3	2.43E−3	2.11E−2	2.40E−2
4	1.88E−4	2.11E−4	1.53E−3	1.29E−3	1.53E−2	1.21E−2
8	4.42E−4	2.71E−4	9.31E−4	8.11E−4	8.12E−3	6.21E−3
10	5.14E−4	3.19E−4	7.43E−4	7.51E−4	6.87E−3	5.07E−3
16	6.65E−4	4.82E−4	8.27E−4	7.52E−4	4.69E−3	3.45E−3

It is easy to see that either the SPP or the P-scheme has much more redundant computational counts compared with the pursuit method.

4. Optimal message vector length in the block SPP algorithm

4.1. Message vectorization

For multidimensional systems, both the size of transported data and the number of times for message passing become larger than one dimensional system of the same size, especially the number of times for message passing. In solving multiple tridiagonal systems, SPP can be applied aggregately to several data instead of the "one by one" approach, as shown in Fig. 1. By aggregating m values into one message, the communication cost for m values is reduced to $t_{\text{delay}} + mt_{\text{sendrecv}}$ instead of $m(t_{\text{delay}} + t_{\text{sendrecv}})$. It is called "message vectorization" strategy in Ref. [15]. In this paper, we call the SPP combined with message vectorization *the block SPP algorithm*.

Let (i_{dm}, j_{dm}, k_{dm}) denote the number of points in the ξ, η and ζ directions, respectively. If the ξ direction of the grid is divided across the number of processors, then the size of message transmitted from one processor to another in one communication, m, could be arranged from 1 to $j_{dm} \times k_{dm}$. The message vector length (MVL) used in this paper refers to this number of grid points, rather than the number of transmitted data which may be several times larger than MVL.

4.2. Experiments with the block SPP algorithm in the ADI scheme

We implement the block SPP algorithm into the solution of the first direction of the ADI scheme, i.e. Eq. (4a). The ξ direction is divided by p processors. For example, for a 64^3 problem, each processor is in charge of a $64/p \times 64 \times 64$ area. Other two directions have no datum transport and the pursuit method can be used.

In our study, every grid point has 4 unknown values (p, u, v, w), which should be combined into one message to decrease the number of communications. In the following description, we will treat a combined message of the four values as one message.

For the 64^3 problem, a tridiagonal system of equations with a size of 64 should be solved 64×64 times on a single computer. On p processors, the SPP with a size of $64/p$ should be iterated 64×64 times when the message vector length is 1, but it should be iterated 2×64 times when the message vector length is 32, thus in Steps 2 and 3 of the SPP, the message with a length of 32 should be sent or received only 2×64 times.

We measure the wall time for solving Eq. (3a) in the ξ direction. Here the wall time includes those for the block SPP algorithm and for generating matrices P_1^{-1}, Λ_1 and the right-hand-side vector. Detailed formulas of these matrices [6] can be used to give the computation counts for generating matrices P_1^{-1}, Λ_1, and the right-hand-side vector:

$$O_{\text{others}} \simeq 6\frac{i_{dm}j_{dm}k_{dm}}{p}n_{\text{eq}}^3, \tag{10}$$

where $n_{\text{eq}} = 4$ represents (p, u, v, w) in the Navier–Stokes equations.

From Eqs. (7) and (10), the total computation counts for the ξ direction are:

$$O_\xi = j_{dm}k_{dm}n_{\text{eq}}\left(26\frac{i_{dm}}{p} + 15p\right) + O_{\text{others}} = j_{dm}k_{dm}n_{\text{eq}}\left(26\frac{i_{dm}}{p} + 15p + 6n_{\text{eq}}^2\frac{i_{dm}}{p}\right). \tag{11}$$

For convenience, performance measure will be presented in Gflops for the above computational counts:

$$P_\xi = \frac{O_\xi}{T_{m\xi}} \times 10^{-9}, \tag{12}$$

where $T_{m\xi}$ is the measured execution time in seconds for the ξ direction.

Fig. 2 shows measured performances of the ξ direction sweep in the ADI scheme with different MVLs for different problem sizes. We can see that the performance improves when MVL increases from 1, and attains a peak at some intermediate values for each fixed number of processors. However, the performance will decrease and remain flat when MVL is large en-

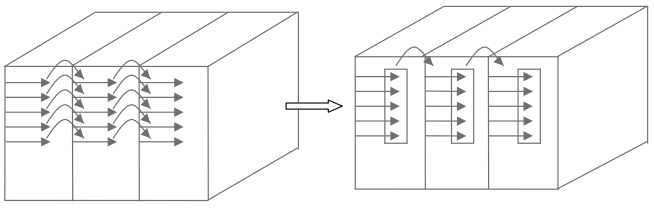

Fig. 1. Aggregation of communication messages.

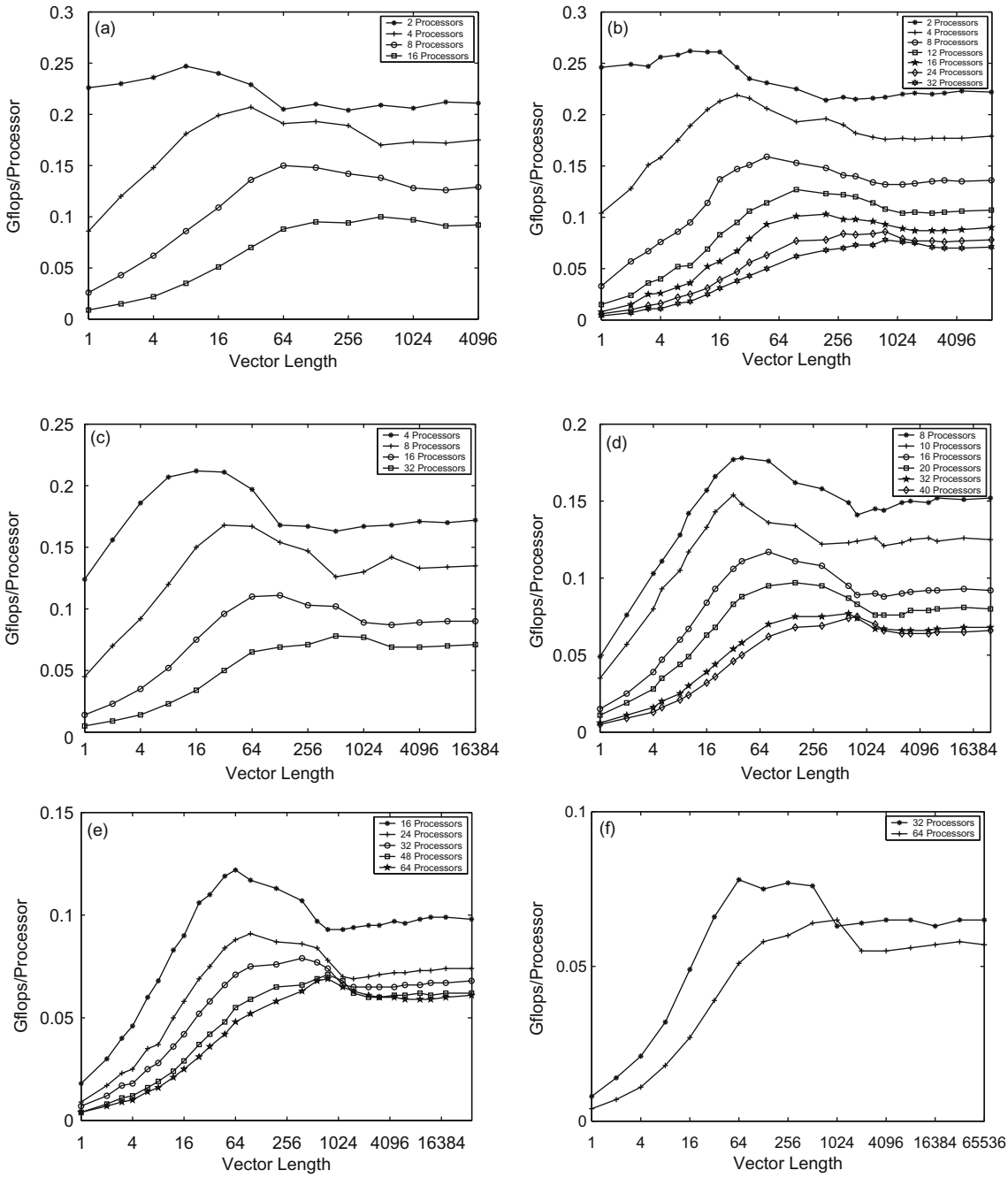

Fig. 2. Measured performances in the ξ direction vs. MVL for different problem sizes: (a) 64^3; (b) 96^3; (c) 128^3; (d) 160^3; (e) 192^3; (f) 256^3.

ough, and the lower the i_{dm}/p, the worse the performance. The latter may be explained by the dominance of communication costs associated with large MVL over computation costs that is proportional to i_{dm}/p on each processor. The occurrence of optimal MVL and its variation with the problem size and number of processors will be examined in following subsections. It will be meaningful if we can predict varying trends of MVL based on an analytical model.

4.3. A parallel model for the ξ-direction sweep

The performance of a parallel algorithm is affected by many factors which are difficult to account for, so we can only use an approximate parallel model to analyze its performance. We first apply the parallel model for the block pipelined method

[32] to a pure block SPP algorithm alone, and then make modification by taking into account variations of the cache effect with the MVL when applied to the ξ-direction sweep of the ADI scheme.

4.3.1. Original model for the block SPP alone

In a pure block SPP alone, $j_{dm} \times k_{dm}$ lines of unrelated scalar tridiagonal systems are to be solved. We assume the size of the MVL is m and the ξ direction of the grid is evenly divided by p processors. Then we have the number of iterations for all of the message vectors to be sent

$$l = \frac{j_{dm} \times k_{dm}}{m}. \tag{13}$$

In the data propagation process in the block SPP algorithm, because all processors except the first one must wait for the data to be sent by the previous processor, 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 iterations of sending throwing off the overlapped ones is $l + p - 1$. Moreover, the operation counts in Step 1, which can be completely parallelized, are much larger than the serial operation counts in Steps 2 and 3. Therefore, these serial operation counts will be ignored here.

From the above description and Eq. (7), we can get the total time used for solving all $j_{dm} \times k_{dm}$ scalar systems by the block SPP:

$$\begin{aligned} T &= (l + p - 1)\left[6 \times \frac{j_{dm} \times k_{dm}}{l} \times t_{\text{sendrecv}} + 4 \times t_{\text{delay}}\right] \\ &\quad + l \times \left[\frac{j_{dm} \times k_{dm}}{l} \times \left(26\frac{i_{dm}}{p} + 15p\right) \times \varepsilon t_c\right] \qquad (14) \\ &= a/l + b \times l + c, \qquad (15) \end{aligned}$$

where

$$\begin{aligned} a &= 6j_{dm}k_{dm}(p-1)t_{\text{sendrecv}}, \\ b &= 4t_{\text{delay}}, \\ c &= 6j_{dm}k_{dm}t_{\text{sendrecv}} + 4(p-1)t_{\text{delay}} + j_{dm}k_{dm}\left(26\frac{i_{dm}}{p} + 15p\right)\varepsilon t_c, \end{aligned}$$

and $\varepsilon \leqslant 1$ is a factor representing the influence of the cache hit rate on computational time. In this subsection, we assume ε is constant. From Eq. (15), it can be shown that there is an equilibrium l that makes T minimal:

$$l_{\text{opt}} = \sqrt{\frac{a}{b}}. \tag{16}$$

Thus we have the optimal MVL for the block SPP algorithm:

$$m_{\text{opt}} = j_{dm} \times k_{dm} \times \sqrt{\frac{b}{a}} = \sqrt{\frac{2 \times j_{dm} \times k_{dm} \times t_{\text{delay}}}{3(p-1)t_{\text{sendrecv}}}}. \tag{17}$$

For Lenovo Deepcomp 1800 supercomputer we used, $t_{\text{delay}}/t_{\text{sendrecv}} \approx 250$, and in practice $j_{dm} \times k_{dm}$ is normally larger than 3600. Thus from Eq. (17), we can easily see that m_{opt} is in the range of $1 < m_{\text{opt}} < j_{dm} \times k_{dm}$, and this proves the existence of an "optimal MVL" within practically available range.

Table 3 shows comparison of the measured optimal MVLs with those predicted by Eq. (17) for the block SPP algorithm alone. We can see that the measured optimal MVL *increases* with increasing value of $j_{dm} \times k_{dm}$ for the same number of processors except $p = 2$ (note that Eq. (7) is not valid for two-processor case), and it *decreases* with increasing number of processors for the same value of $j_{dm} \times k_{dm}$. These two measured trends are in agreement with what Eq. (17) predicts, even though the numbers differ.

Table 3
Measured optimal MVLs for the block SPP algorithm alone in comparison with predictions by Eq. (17).

n	p							
	2		4		8		16	
	Measured	Predicted	Measured	Predicted	Measured	Predicted	Measured	Predicted
64^3	4096	*826*	512	*477*	512	*312*	512	*213*
96^3	1536	*1239*	1152	*716*	1152	*468*	768	*320*
160^3	2560	*2066*	1280	*1193*	1280	*781*	800	*533*

Next, we turn back to check the varying trends in Fig. 2 for the ξ-direction sweep in our ADI solver. We can see from Fig. 2 that the optimal MVL *decreases* with increasing value of $j_{dm} \times k_{dm}$ for the same number of processors in most cases, and it *increases* with increasing number of processors for the same value of $j_{dm} \times k_{dm}$. Both trends are contrary to what Eq. (17) predicts. The reason may be that the cache effect was not considered to vary in the model Eq. (14). We will try to consider it in following modified model.

4.3.2. A modified model for the ξ-direction sweep

We notice that in the ξ direction sweep in Eq. (3a), extra computation counts (Eq. (10)) in addition to the block SPP will be undertaken, and $j_{dm} \times k_{dm} \times n_{eq}$ tridiagonal matrix equations instead of $j_{dm} \times k_{dm}$ equations will be solved in the ξ direction sweep. In the following, we will improve the model Eq. (14) to describe the ξ direction sweep of our ADI solver by taking varying cache effect into account.

We try to find relationship between the computation time and the communication time. Table 4 shows the timing results corresponding to Fig. 2a. We can see that the total execution times on 4 processors and 8 processors are equal at $m = 16$ (corresponding $l = 64^2/m = 256$). The whole number of iterations of communication on 8 processors, $l + p - 1 = 256 + 7$, is 4 more than $l + p - 1 = 256 + 3$ on 4 processors, but $(i_{dm}/4 - i_{dm}/8) \times j_{dm} \times k_{dm} = 32768$ fewer grid points should be computed per processor on 8 processors than on 4 processors. If we substitute numbers of (l, p, j_{dm}, k_{dm}) for $p = 4$ and $p = 8$, respectively into a modified formula similar to Eq. (14):

$$T = (l + p - 1)\left[6 \times \frac{j_{dm} \times k_{dm}}{l} \times n_{eq} \times t_{sendrecv} + 4 \times t_{delay}\right] + j_{dm} \times k_{dm} \times \frac{i_{dm}}{p} \times \widehat{T}_{comp}, \tag{18}$$

and subtract, we can obtain

$$4 \times \left[6m \times n_{eq} t_{sendrecv} + 4t_{delay}\right] = 32768 \times \widehat{T}_{comp}, \tag{19}$$

where we have assumed the computation time for each grid point per processor, $\widehat{T}_{comp}$, is the same on 4 processors and 8 processors. Eq. (19) gives a relationship between the computation time and the communication time. Similar relationship between the computation time and the communication time can also be found from other timing results for cases in Fig. 2b–f.

As a generalization from above case study, we assume $\widehat{T}_{comp}$ is proportional to the communication time between any two processors

$$\widehat{T}_{comp} = \delta(6m \times n_{eq} t_{sendrecv} + 4t_{delay}), \tag{20}$$

where δ depends on the cache effect. By using (20), Eq. (18) can be rewritten as:

$$\begin{aligned}
T &= (l + p - 1)\left[6 \times \frac{j_{dm} \times k_{dm}}{l} \times n_{eq} t_{sendrecv} + 4t_{delay}\right] \\
&\quad + l \times \left[\frac{j_{dm} \times k_{dm}}{l} \times \frac{i_{dm}}{p} \times \widehat{T}_{comp}\right] \\
&= (l + p - 1)\left[6n_{eq} \frac{j_{dm} \times k_{dm}}{l} t_{sendrecv} + 4t_{delay}\right] \\
&\quad + \left(\frac{i_{dm} \times j_{dm} \times k_{dm} \times \delta}{p}\right)\left[6n_{eq} \frac{j_{dm} \times k_{dm}}{l} t_{sendrecv} + 4t_{delay}\right] \\
&= \left(l + p - 1 + \frac{i_{dm} \times j_{dm} \times k_{dm} \times \delta}{p}\right)\left[6n_{eq} \frac{j_{dm} \times k_{dm}}{l} t_{sendrecv} + 4t_{delay}\right] \qquad (21) \\
&= 6n_{eq}\left(p - 1 + \frac{i_{dm} \times j_{dm} \times k_{dm} \times \delta}{p}\right) \frac{j_{dm} \times k_{dm}}{l} t_{sendrecv} + 4t_{delay} \times l \\
&\quad + 4\left(p - 1 + \frac{i_{dm} \times j_{dm} \times k_{dm} \times \delta}{p}\right) t_{delay} + 6n_{eq} j_{dm} \times k_{dm} \times t_{sendrecv} \\
&= a/l + b \times l + c, \qquad (22)
\end{aligned}$$

which represents the total time for executing the ξ direction sweep of the ADI scheme. Therefore, the optimal MVL is:

$$m_{opt} = j_{dm} \times k_{dm} \times \sqrt{\frac{b}{a}} = \sqrt{\frac{2 \times j_{dm} \times k_{dm} \times t_{delay}}{3n_{eq}\left[p - 1 + \frac{i_{dm} \times j_{dm} \times k_{dm} \times \delta}{p}\right] t_{sendrecv}}} \simeq \sqrt{\frac{2 \times p \times t_{delay}}{3n_{eq} \times i_{dm} \times \delta \times t_{sendrecv}}}$$
$$(\text{if } p^2 \ll i_{dm} j_{dm} k_{dm} \times \delta). \tag{23}$$

It is clear from Eq. (23) that m_{opt} *decreases* with increasing i_{dm} for the same number of processors, and it *increases* with increasing number of processors for the same i_{dm}. This is in agreement with the varying trends shown in Fig. 2.

Table 4
Timing results corresponding to Fig. 2a.

MVL	p			
	2	4	8	16
1	0.285	0.382	0.691	1.450
2	0.279	0.274	0.424	0.816
4	0.273	0.222	0.292	0.522
8	0.265	0.183	0.207	0.343
16	0.268	**0.165**	**0.165**	0.233
32	0.281	0.159	0.132	0.172
64	0.315	0.172	0.120	0.136
128	0.286	0.170	0.121	0.125
256	0.317	0.175	0.126	0.127
512	0.307	0.194	0.131	0.122
1024	0.312	0.191	0.140	0.124
2048	0.312	0.191	0.141	0.130
4096	0.310	0.190	0.140	0.129

As seen from Eq. (20), larger δ corresponds to less cache effect and longer computation time. To determine one value of δ, we fitted Eq. (23) to a measured optimal MVL = 20 for the 160^3 problem with $p = 8$. In this test, 90% memory on every processor was used, and the cache effect was assumed to be little. The fitted value was

$$\delta = 0.005. \tag{24}$$

Using Eq. (23) with $\delta = 0.005$, we can predict optimal MVLs for other cases with different memory occupations on each processor. Table 5 shows comparison of measured and predicted optimal MVLs. We can see that the predicted values generally match the measured data better for smaller number of processors (say $p = 4$), but underestimates the measured optimal MVL much for larger number of processors, especially $p = 32$. This is because the value $\delta = 0.005$ represents large-memory occupation cases, and is inappropriately large for the same mesh size to be distributed on larger number of processors ($p = 32$) where each processor will use a small fraction of its total memory so that the cache effect should be expected larger. In such a situation, δ should be adjusted to smaller value, hence making m_{opt} predicted from Eq. (23) larger to better match the measured value. Our new model can predict the varying trend and give crude estimation of optimal MVLs. It should be remarked that exact optimal MVLs can only be found via numerical tests.

4.4. Performance of the 3D parallel ADI solver

As stated in Section 4.2, our parallel ADI solver consists of the block SPP for the divided ξ direction and the pursuit method for the undivided η and ζ directions. We measure the wall time for executing one iteration of the ADI solution.

Most programs will have better performance per processor when they are run on processors with large memory capacity. Therefore, a more useful measure is the performance versus processors at a fixed memory utilization [31]. In this study, for each set of tests which were run on the same number of processors, linear extrapolation was used to estimate the performance when 90% memory of each processor was utilized. The curve marked " 90% mem" are quadratic least squares fits through these 90% estimate points. A constant value of the performance per processor over a range of processors implies scalability for that range.

Fig. 3 is a plot of the ξ direction results with measured optimal MVLs. For a fixed grid size, the performance deteriorates with increasing number of processors. However, the scalability marked by the 90% memory begins to level above 32 processors, indicating the scalability is good.

Fig. 4 shows the performances in the η and ζ directions. In these two directions, the pursuit method is used to solve the tridiagonal systems. The timings include all operation counts in the η and ζ directions like those in the ξ direction, thus extra operation counts in addition to those of a pure pursuit method are included. The performances in the η and ζ directions are

Table 5
Measured optimal MVLs for the ξ direction sweep in comparison with predictions by Eq. (23) with $\delta = 0.005$ to account for cache effect.

n	p							
	4		8		16		32	
	Measured	Predicted	Measured	Predicted	Measured	Predicted	Measured	Predicted
128^3	16	*16*	32	*23*	128	*32*	512	*46*
160^3	–	*14*	40	*20*	64	*29*	512	*41*
192^3	–	*13*	–	*19*	64	*26*	384	*37*
256^3	–	*11*	–	*16*	–	*23*	64	*32*

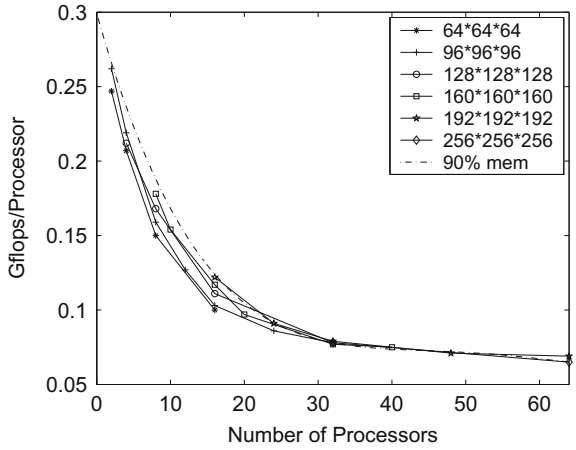

Fig. 3. Optimal performances in the ξ direction.

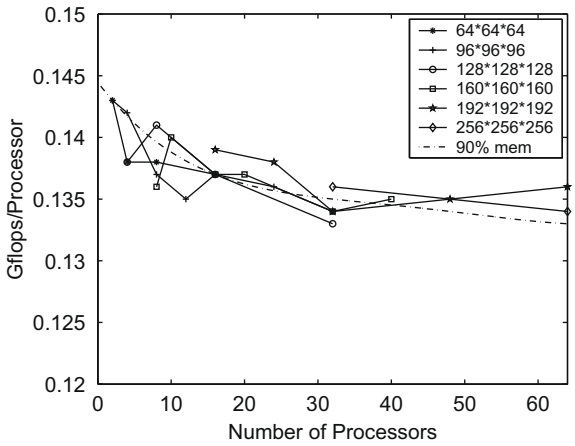

Fig. 4. Optimal performances in the η and ζ directions.

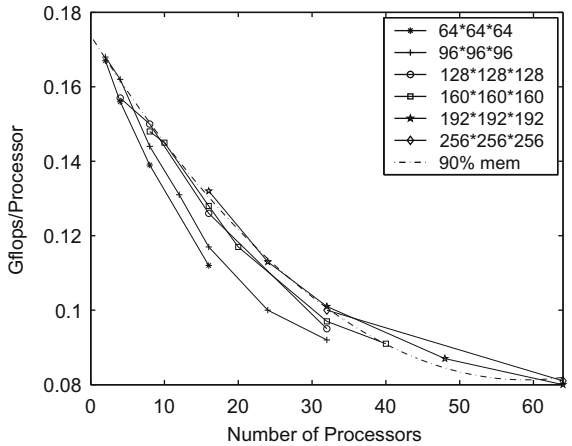

Fig. 5. Optimal performances of the 3D ADI with the block SPP implemented only in the ξ direction.

almost constant, and this is expected since there is no communication between processors in the η and ζ directions for the present implementation. The performance drops down slightly as the number of processors increases, and remains around 0.135 when $p > 32$.

Fig. 5 shows three-direction performances of the ADI solver with the bock SPP implemented in the ξ direction only. It looks like Fig. 3. This is as excepted since the ξ-direction performance with greater variation will dominate the overall performance of the ADI solver.

5. Extension of optimal message vector length to the block pipelined method

Vectorization is widely used as a parallel technique. Here we will show that the block pipelined method can also benefit from utilizing an optimal MVL. In implementing the block pipelined method, we only distribute grid points in the ξ direction among processors.

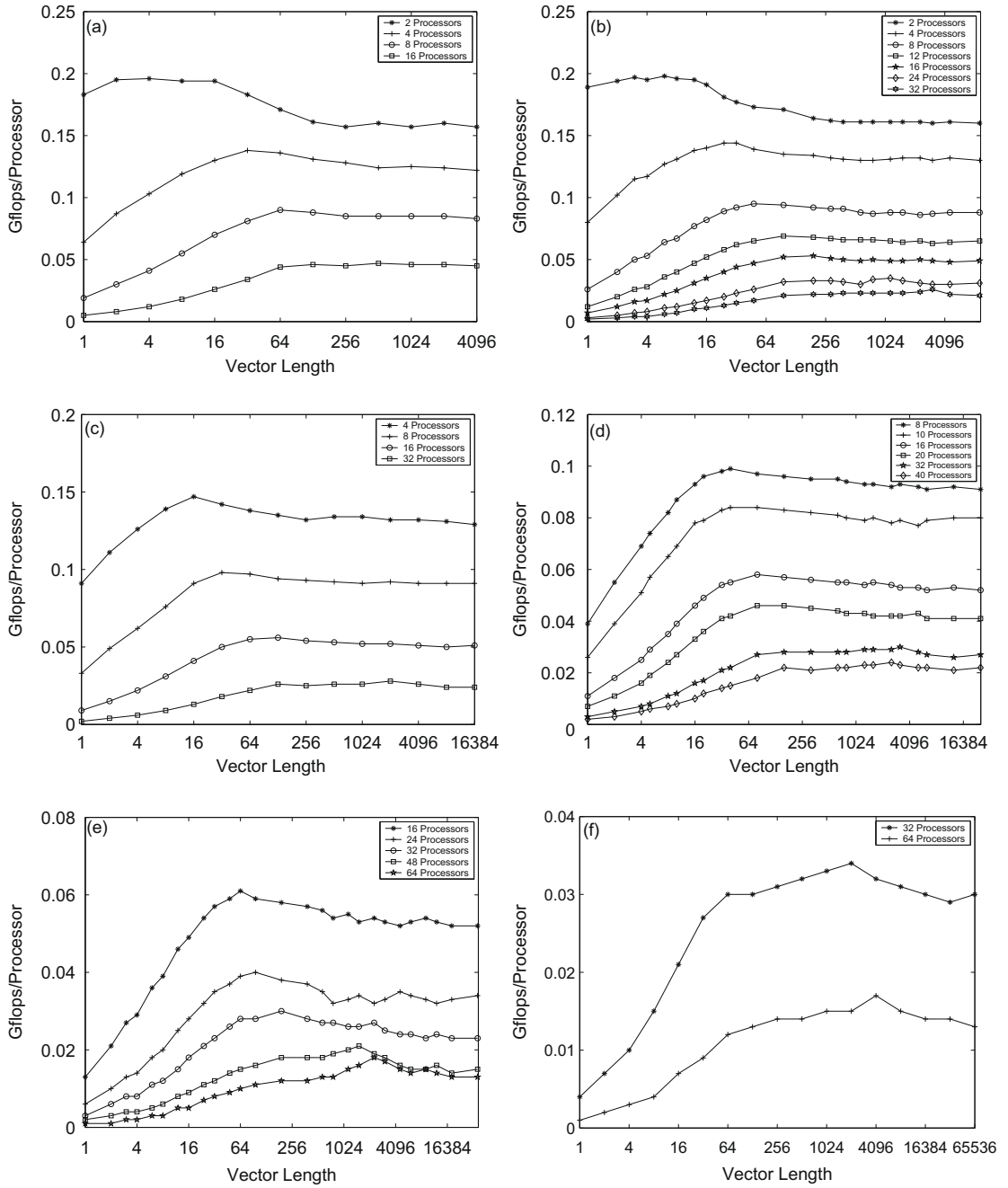

Fig. 6. Performances of the block pipelined method in the ξ direction for different problem sizes: (a) 64^3; (b) 96^3; (c) 128^3; (d) 160^3; (e) 192^3; (f) 256^3.

First, we check under what conditions the present block SPP algorithm is faster than the block pipelined method. The SPP has 2 communications and 3 values to be transmitted between two processors. The pipelined method has 3 communications (2 in the elimination step and 1 in the back substitution step) and each communication has 1 value to be transmitted between any two processors. But 2 communications in the elimination step can be packed together and 1 communication can be saved. Thus the communication counts of the pipelined method are the same as those of the SPP. However, the block pipelined method uses the pursuit method whose computation counts are much smaller than those of the SPP. According to Ref. [32] and Eq. (14), the total wall time for the block pipelined method and the block SPP is:

$$T_{\text{bpip}} = (l+p-1)\left(T_{\text{comm}} + T_{\text{comp}}\right), \tag{25}$$

$$T_{\text{bspp}} = (l+p-1)T_{\text{comm}} + lQT_{\text{comp}}, \tag{26}$$

where Q is the ratio of the computational complexity of the SPP algorithm to that of the pursuit method. For most parallel tridiagonal solvers, $Q > 2$ [15]. Assuming that the computational time per block data, T_{comp}, is the same for both algorithms, it can be easily shown from above formulas that if

$$l(Q-1) < (p-1), \tag{27}$$

then the block SPP will be faster than the block pipelined method, otherwise, the block SPP will be slower.

Next we measured the wall time for the ADI scheme with the block pipeline method implemented in the ξ direction only. The timings include extra operations for generating matrices P_1, A_1, and the right-hand-side vector, just the same as in the block SPP.

Fig. 6 shows performances of the block pipelined method in the ξ direction sweep in the ADI scheme versus different MVLs for different problem sizes. We can see that the optimal MVL *decreases* with increasing grid size for the same number of processors in most cases, and it *increases* with increasing number of processors for the same grid size. These trends are in agreement with what our new model Eq. (23) predicts.

6. Conclusions

We have presented an improved version of SPP algorithm and demonstrated that a 3D ADI solver implemented with this block SPP can obtain better performance with appropriate choice of the message vector length (MVL). The existence and varying trends of the optimal MVL with different number of processors and problem sizes are predicted based on a refined parallel model. We have shown that the optimal MVL can also benefit the block pipelined method. In this paper, fewer than 64 processors were used and only one direction was divided. For more processors, 2 or 3 directions need to be divided to get better performances. This is the work we will do in the future.

In practice, it is a challenging task to tune a code on any parallel computer to the optimal efficiency. For the ADI solver discussed in this paper, we propose the following procedure:

1. If the tridiagonal system will be solved only for several times, the analytical models developed in this paper can be used to guess an optimal MVL.
2. If the tridiagonal system will be solved for thousands of times, it is worth measuring the computing time for all possible MVL values and a number of processors to decide an optimal MVL precisely for a given problem size in preliminary runs. This evaluating process can be facilitated with the present modified model.

Actually, this is also the procedure adopted in some of the most popular calculating packages like FFTW (http://www.fftw.org).

Acknowledgments

This work was supported by Natural Science Foundation of China (G10476032, G10531080) and state key program for developing basic sciences (2005CB321703). The computation was conducted on LSSC2 cluster at LSEC, Institute of Computational Mathematics, Chinese Academy of Sciences. ZY was supported by National Nature Science Foundation of China (G10502054).

References

[1] W. Briley, H. McDonald, Solution of the three-dimensional Navier–Stokes equations by an implicit technique, in: Proceedings of the Fourth International Conference on Numerical Methods in Fluid Dynamics, Lecture Notes in Physics, vol. 35, Springer-Verlag, Berlin, 1975.
[2] R. Beam, R. Warming, An implicit finite-difference algorithm for hyperbolic systems in conservation law form, J. Comput. Phys. 22 (1976) 87–110.
[3] A. Averbuch, L. Ioffe, M. Israeli, L. Vozovoi, Two-dimensional parallel solver for the solution of the Navier–Stokes equations with constant and variable coefficients using ADI on cells, Parallel Comput. 24 (1998) 673–699.
[4] U.W. Rathe, P. Sanders, P.L. Knight, A case study in scalability: an ADI method for the two-dimensional time-dependent Dirac equation, Parallel Comput. 25 (1999) 525–533.
[5] P.E. Morgan, M.R. Visbal, P. Sadayappan, Development and application of a parallel implicit solver for unsteady viscous flows, Int. J. Comput. Fluid Dyn. 16 (2002) 21–36.

[6] L. Yuan, Comparison of implicit multigrid schemes for three-dimensional incompressible flows, J. Comput. Phys. 177 (2002) 134–155.
[7] W. Press, Numerical Recipes in C, Cambridge, UK, 1988.
[8] J. Ortega, R. Voigt, Solution of partial differential equations on vector and parallel computers, SIAM Rev. 27 (1985) 149–240.
[9] C. Ho, S. Johnsson, Optimizing tridiagonal solvers for alternating direction methods on boolean cube multiprocessors, SIAM J. Sci. Stat. Comput. 11 (3) (1990) 563–592.
[10] I.S. Duff, H.A. van der Vorst, Developments and trends in the parallel solution of linear systems, Parallel Comput. 25 (1990) 1931–1970.
[11] I. Babuska, Numerical stability in problems of linear algebra, SIAM J. Numer. Anal. 9 (1972) 53–77.
[12] Henk A. van der Vorst, Large tridiagonal and block tridiagonal linear systems on vector and parallel computers, Parallel Comput. 5 (1987) 45–54.
[13] H.S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, J. Assoc. Comput. Mach. 20 (1973) 27–38.
[14] A. Wakatani, A parallel scheme for solving a tridiagonal matrix with pre-propagation, in: Proceedings of the 10th Euro PVM/MPI Conference, 2003.
[15] A. Wakatani, A parallel and scalable algorithm for ADI method with pre-propagation and message vectorization, Parallel Comput. 30 (2004) 1345–1359.
[16] R.W. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, J. ACM. 12 (1965) 95–113.
[17] J.J. Lambiotte, R.G. Voigt, The solution of tridiagonal linear systems on the CDC STAR-100 computer, ACM Trans. Math. Soft. 1 (1975) 308–329.
[18] H.H. Wang, A Parallel method for tridiagonal equations, ACM Trans. Math. Soft. 7 (1981) 170–183.
[19] P.H. Michielse, H.A. van der Vorst, Data transport in Wang's partition method, Parallel Comput. 7 (1988) 87–95.
[20] I. Bar-On, Efficient logarithmic time parallel algorithms for the Cholesky decomposition and Gram–Schmidt process, Parallel Comput. 17 (1991) 409–417.
[21] H.X. Lin, M.R.T. Roest, Parallel solution of symmetric banded systems. in: G.R. Joubert, D. Trystram, F.J. Peters, D.J. Evans (Eds.), Parallel Computing: Trends and Applications. 1994, pp. 537–540.
[22] D.H. Lawrie, A.H. Sameh, The computation and communication complexity of a parallel banded system solver, ACM Trans. Math. Soft. 10 (1984) 185–195.
[23] S.L. Johnsson, Solving tridiagonal systems on ensemble architectures, SIAM J. Sci. Stat. Comput. 8 (1987) 354–392.
[24] N. Mattor, T.J. Williams, D.W. Hewett, Algorithm for solving tridiagonal matrix problems in parallel, Parallel Comput. 21 (1995) 1769–1782.
[25] X.H. Sun, H. Zhang, L.M. Ni, Parallel algorithms for solution of tridiagonal systems on multicomputers, in: Proceedings of the 1989 ACM Int. Conf. on Supercomputing, Crete, Greece, June 1989.
[26] S. Bondeli, Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations, Parallel Comput. 17 (1991) 419–434.
[27] V. Mehemann, Divide and conquer methods for block tridiagonal systems, Parallel Comput. 19 (1993) 257–279.
[28] X.H. Sun, H. Zhang, L.M. Ni, Efficient tridiagonal solvers on multicomputers, IEEE Trans. Comput. 41 (1992) 286–296.
[29] X.H. Sun, Application and accuracy of the parallel diagonal dominant algorithm, Parallel Comput. 21 (1995) 1241–1267.
[30] C.R. Wang, Z.H. Wang, X.H. Yang, Computational Ffluid Dynamics and Parallel Algorithms, first ed., National University of Defence Technology Press, Changsha, China, 2000 (in Chinese).
[31] T.M. Edison, G. Erlebacher, Implementation of a fully-balanced periodic tridiagonal solver on a parallel distributed memory architecture, Concurrency-pract EX7 (4) (1995) 273–302.
[32] L.B. Zhang, On pipelined computation of a set of recurrences on distributed memory systems, J. Numer. Methods Comput. Appl. 3 (1999) 184–191 (in Chinese).
[33] M. Smith, R. Wijngaart, M. Yarrow, Improved multi-partition method for line-based iteration schemes, in: Computational Aerosciences Workshop 95, NASA Ames Reserach Center, Moffett Field, CA, USA, 1995.
[34] V. Balasundaram, G. Fox, K. Kennedy, U. Kremer, An interactive environment for data partitioning and distribution, in: Proceedings Fifth Distributed Memory Computing Conference, April 1990, pp. 1160–1170.
[35] M.W. Hall, S. Hiranandani, K. Kennedy, C.W. Tseng, Interprocedural compilation of Fortran d for MIMD distributed-memory machines, in: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, 1992, pp. 522–534.
[36] X.H. Guo, Research on symmetric super compact difference methods and their parallel algorithm, Doctoral Dissertation, Institute of Computational Mathematics, Chinese Academy of Sciences, Beijing, 2004, (in Chinese).
[37] Z. Yin, Li Yuan, Tao Tang, A new parallel strategy for two-dimensional incompressible flow simulations using pseudo-spectral methods, J. Comput. Phys. 210 (2005) 325–341.