Parallelizing Sylvester-like Operations on a Distributed Memory Computer

Dan Y. Hu
Danny C. Sorensen

February 1994

TR94-10
Parallelizing Sylvester-like Operations on a Distributed Memory Computer*

Dan Y. Hu, Danny C. Sorensen
Department of Computational and Applied Mathematics
Rice University
P.O. Box 1892
Houston, TX 77251

Abstract
A SIMD scheme for parallelization of the 2-D array operation $\tilde{M}(x) = (D \otimes A + B \otimes I + \tilde{V})x$ is developed for a distributed memory computer with 2-D mesh processor network. This scheme is carefully designed in order to approach the best possible speed-up by avoiding unnecessary message communication. The numerical results on a Touchstone DELTA machine are presented to demonstrate the effectiveness of the scheme. This scheme can be generalized to one that would deal with the 3-D array operation $\tilde{M}_{3D}(x) = (I \otimes I \otimes A + I \otimes B \otimes I + C \otimes I \otimes I + \tilde{V})x$ on a distributed memory computer with 3-D torus processor network.

1 Introduction

Discretization of linear operators arising in applied mathematics often leads to matrices with the following structure:

$$\tilde{M}(x) = (D \otimes A + B \otimes I + \tilde{V})x,$$

where $x \in \mathbb{R}^{mn}$, $B, D \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times m}$ and $\tilde{V} \in \mathbb{R}^{mn \times mn}$, both $D$ and $\tilde{V}$ are diagonal. For the notational convenience, we assume that both $A$ and $B$ are symmetric. All the results through this paper can be easily extended to the cases with general $A$ and $B$.

The linear operator on $\mathbb{R}^{mn}$ defined by (1) can be viewed as a generalization of the Sylvester operator:

$$\tilde{S}(x) = (I \otimes I \otimes A + B \otimes I_n)x.$$

We therefore refer it as a Sylvester-like operator. The schemes discussed in this paper therefore also apply to Sylvester operator.

Similar to the matrix form of the Sylvester operator:

$$S(X) = AX + XB,$$

we have the matrix form equivalent to (1) as the follows:

$$M(X) = AXD + XB + V \otimes X$$

where $X \in \mathbb{R}^{m \times n}$, $V \in \mathbb{R}^{mn \times mn}$. $V$ is defined as: let $V_{\text{diag}}$ be the $mn$ vector consisting of the diagonal elements of $\tilde{V}$, then $V(:, i) = V_{\text{diag}}((i-1)m+1: nm, (i-1)m+1: nm)$, for $i = 1, \ldots, n$. The operation denoted by '$\otimes$' here is the element-by-element multiplication. More precisely, $V \otimes X \in \mathbb{R}^{m \times n}$, and

$$(V \otimes X)(i, j) = V(i, j)X(i, j).$$

*Financial support for this work was provided in part by the National Science Foundation contract ASC-9112693, National Science Foundation cooperative agreement CCR-9120008, by the Department of Energy contract DE-FG02-91ER25183 and by DARPA through U.S. Army OMAA44666.01.
The equivalence between the matrix form (2) and the 'tensor-product' form (1) is based on the following one-to-one correspondence from $R^{m \times n}$ to $R^{m n}$:

$$
X(:, 1) \\
X(:, 2) \\
\vdots \\
X(:, n)
$$

One example of the Sylvester-like operator is from the reactive scattering model in quantum chemistry (see [2], and [7]). In the study of the accurate state-to-state reaction cross sections for three-atom systems with modest number of energetically open quantum states, the following 2-D Schrödinger equation is concerned:

$$
H(\theta, \chi; \rho_\xi) \Phi_\xi(\theta, \chi; \rho_\xi) = E_\xi(\rho_\xi) \Phi_\xi(\theta, \chi; \rho_\xi),
$$

where $\rho_\xi$ is a fixed center of a sphere in the space, and $\theta, \chi$ are the polar and azimuthal angles, respectively. The Hamiltonian $H$ in the equation can be written as the sum of the following three operators: the kinetic energy operator for the $\chi$ degree of freedom

$$
\hat{T}_\chi = -\frac{k_1}{\sin^2 \theta/2} \frac{\partial^2}{\partial \chi^2},
$$

the kinetic energy operator for the $\theta$ degree of freedom

$$
\hat{T}_\theta = -\frac{k_2}{\sin \theta} \frac{\partial}{\partial \theta} \sin \theta \frac{\partial}{\partial \theta},
$$

and the potential energy term

$$
\hat{V} = V(\rho_\xi, \theta, \chi) + k_3,
$$

where $k_1, k_2$ and $k_3$ are constant numbers. With the discrete variable representation consisting of the Legendre polynomials in $\cos \theta$ and trigonometric functions in $\chi$ the Hamiltonian can be written in the form

$$
\hat{H} = h_\theta \otimes \mathbf{I}_\chi + f_\theta \otimes h_\chi + \hat{V},
$$

which has the same form as (1).

For many iterative algorithms of solving the linear system of equation

$$
\tilde{M} \mathbf{x} = \mathbf{b},
$$

or of solving the eigenvalue problem

$$
\tilde{M} \mathbf{x} = \lambda \mathbf{x},
$$

the linear operation $\tilde{M} \mathbf{x}$ are performed repeatedly. Therefore in the implementation of an iterative algorithm on a certain computer system, the efficiency of computing the following operation

$$
y = \tilde{M} \mathbf{x}
$$

will effect the efficiency of the whole algorithm.

In this paper, we present the SIMD scheme for parallelization of the Sylvester-like operator on a distributed memory computer. This scheme is designed to approach the best possible efficiency by avoiding unnecessary communication among processors. Throughout this paper we use form (2) and form (1) alternately in our presentation and analysis. Hereafter, the integers $m$ and $n$ are reserved for the sizes of the Sylvester-like operator in either (2) or (1).
2 The Standard Systolic Model on a Ring Structure

The matrix $\tilde{M}$ in (1) has a very special structure. In general, we can write $\tilde{M}$ as

$$\tilde{M} = M_{\text{diag}} + M_{\text{off}},$$

where

$$M_{\text{diag}} = \begin{pmatrix}
M_1 & 0 & \ldots & 0 \\
0 & M_2 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & M_n
\end{pmatrix},$$

and

$$M_{\text{off}} = \begin{pmatrix}
0 & \alpha_{12}I_m & \ldots & \alpha_{1m}I_m \\
\alpha_{21}I_m & 0 & \ldots & \alpha_{2m}I_m \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{n1}I_m & \alpha_{n2}I_m & \ldots & 0
\end{pmatrix}.$$  

For a distributed memory system with $n$ processors connected in a ring network (see Figure 1), the matrix-vector product $y := Mx$ can be naturally carried out by the standard Systolic Procedure (see [4]).

First, we denote the $k$-th processor in the ring by $P_{k-1}$ as shown in Figure 1. For a certain processor, left and right refer to the indices of its left and right neighbors, respectively. Thus in the $n = 4$ example from the figure, $P_3$ has $P_2$ and $P_0$ as $P_{\text{left}}$ and $P_{\text{right}}$, respectively.

As usual, we distribute the vector $x$ to the local array $x_{\text{loc}}(1 : m)$ of each processor. For $P_k$, $x_{\text{loc}} = x(km + 1 : (k + 1)m)$. Similarly, the result vector $y$ will be distributed to the local array $y_{\text{loc}}(1 : m)$ with $y_{\text{loc}} = y(km + 1 : (k + 1)m)$. $M_{k+1}$ and $\alpha_{k+1,i}(i = 1, \ldots, n; i \neq k + 1)$ are also saved in $M_{\text{loc}}(1 : m, 1 : m)$ and $a(1 : n)$, respectively, such that $M_{\text{loc}} = M_{k+1}$, and $a(i) = \alpha_{k+1,i}(i = 1, \ldots, n; i \neq k + 1)$.

The procedure is described as the follows:

Algorithm 1 (for $P_k$)

1. $y_{\text{loc}} = M_{\text{loc}}x_{\text{loc}}$
2. $\text{ind} = k$
3. for $i = 1 : n - 1$
   1. send $x_{\text{loc}}$ to $P_{\text{left}}$
   2. recv $x_{\text{loc}}$ from $P_{\text{right}}$
   3. $\text{ind} = \text{ind} + 1$
   4. if $\text{ind} = n, \text{ind} = 0$
   5. $y_{\text{loc}} = y_{\text{loc}} + a(\text{ind} + 1)x_{\text{loc}}$
4. end
To complete the procedure, each processor needs to send and receive

\[ \text{commu}_1 = (n - 1)m \]

floating point numbers and perform

\[ \text{flops}_1 = 2m(m + n - 1) \]

floating point operations (flops). The total memory required is

\[ \text{memor}_1 = m^2 + 3m + n \]

including an \( m \) buffer for message passing.

Algorithm 1 is easy to implement and perfectly load balanced. However, it does have some disadvantages. First, the number of processors has to agree with \( n \). It may be difficult or impossible to arrange this scheme on many applications. Secondly, the communication load on each processor is heavy. In fact, upon completion of the procedure, each processor has contacted information from \( x_{loc} \)'s of all other processors.

### 3 A New Systolic Model on a 2-D Mesh

Our new scheme is designed to reduce communication among processors by taking advantages of both special structure of the operator and the 2-D processor network structure of the computer system.

The cause of heavy communication involved in Algorithm 1 can be traced to the fact that in operation \( y = Mx \), all components of the vector \( x \) are needed to compute any component of \( y \). In contrast with \( y = Mx \), the equivalent matrix operation \( Y = M(X) \) imposes a different dependency of \( Y \) on \( X \). Only two ‘strips’ of \( X \), namely, \( X(i_1 : i_2, :) \) and \( X(:, j_1 : j_2) \), are needed to obtain a submatrix \( Y(i_1 : i_2, j_1 : j_2) \) of the result matrix \( Y \). Figure 2 shows the dependency of \( Y \) on \( X \) with an example of sizes \( m = 8, n = 4 \) and \( i_1 = 3, i_2 = 4, j_1 = 3, j_2 = 4 \). This special dependency relationship provides a motivation to distribute \( X \) and \( Y \) on a 2-D mesh processor network with a
natural 2-D partition. In brief, we partition the matrices $X$ according to the 2-D mesh so that an equal-sized block of $X$ can be distributed to every processor. The same rule applies to $Y$.

Before we discuss the scheme in more detail, we introduce the way of labeling the 2-D mesh of processors. We assume that in a mesh there are $p_c$ columns and $p_r$ rows of processors, and assume that $p = p_c p_r$ is the total number of processors in the mesh. Following the similar way used for the Intel Touchstone DELTA system, (see [5]) we identify each processor with a logical processor number and corresponding 2-D Cartesian coordinates. The one in the upper left corner of the mesh has processor number 0 and denoted by $P_0$. The processor numbers then increase from left to right and top to bottom. The processor in the lower right corner has processor number $p - 1$ and is denoted by $P_{p-1}$. Figure 3 shows a 3 by 4 mesh and the way to label each processor with both logical processor number $i$ and 2-D coordinates $(i_x, i_y)$.

For a processor located at a certain position in the mesh, left, right, up and down refer to the indices of its left, right, upward and downward neighbors, respectively. If a processor has its first coordinate 0, we define its left processor be the one with the same second coordinate and the first coordinate $p_r - 1$. Similar definitions are given for other ‘edge’ processors. Thus in the example shown in Figure 3, $P_6$ has $P_0$ and $P_{11}$ as $P_{down}$ and $P_{left}$, respectively.

On processor $P_k$ with coordinates $(k_x, k_y)$, chunks of matrices $A$, $B$, $D$ and $V$ are distributed in the local arrays $A_{loc}(1 : m_i, 1 : n_i)$, $B_{loc}(1 : m, 1 : n_1)$, $D_{loc}(1 : n_1)$ and $V_{loc}(1 : m_1, 1 : n_1)$ in the following way:

$$A_{loc} = A(k_y m_i + 1 : (k_y + 1)m_i, :),$$

$$V_{loc} = V(k_y n_i + 1 : (k_y + 1)n_i, k_x n_i + 1 : (k_x + 1)n_i),$$

$$B_{loc} = B(:, k_x n_i + 1 : (k_x + 1)n_i),$$

$$D_{loc} = \text{diag}(D)(k_x n_i + 1 : (k_x + 1)n_i).$$

Where we assume that $m = m_i p_r$ and that $n = n_i p_c$. As described before, matrix $X$ has its chunk distributed in the array $X_{loc}(m_i, n_i)$ with the relation

$$X_{loc} = X(k_y m_i + 1 : (k_y + 1)m_i, k_x n_i + 1 : (k_x + 1)n_i).$$

Similarly, the result matrix $Y$ will have its chunk distributed in the array $Y_{loc}(m_i, n_i)$ with the relation

$$Y_{loc} = Y(k_y m_i + 1 : (k_y + 1)m_i, k_x n_i + 1 : (k_x + 1)n_i).$$
The follows is the description of the procedure.

Algorithm 2 (for $P_k$)
{Step 1. applying the operation $Y = V \odot X$}
for $i = 1 : m_1$
  for $j = 1 : n_1$
    $Y_{loc}(i, j) = V_{loc}(i, j) X_{loc}(i, j)$
  end
end

{Step 2. applying the operation $Y = Y + XB$}
$Y_{loc} = Y_{loc} + X_{loc} B_{loc}(k_x n_1 + 1 : (k_x + 1)n_1, 1 : m_1)$
$ind = k_x$
for $i = 1 : p_c - 1$
  send $X_{loc}$ to $P_{left}$
  recv $X_{loc}$ from $P_{right}$
  $ind = ind + 1$
  if $ind = p_c$, $ind = 0$
  $Y_{loc} = Y_{loc} + X_{loc} B_{loc}(ind + 1 : (ind + 1)n_1, 1 : m_1)$
end

{Step 3. recovering the original distribution of $X$}
send $X_{loc}$ to $P_{left}$
recv $X_{loc}$ from $P_{right}$

{Step 4. applying the operation $Y = Y + AXD$}
for $i = 1 : n_1$
  $X_{loc}(:, i) = D_{loc}(i) X_{loc}(:, i)$
end
$Y_{loc} = Y_{loc} + A_{loc}(1 : n_1, k_y m_1 + 1 : (k_y + 1)m_1 ) X_{loc}$
$ind = k_y$
for $i = 1 : p_r - 1$
  send $X_{loc}$ to $P_{up}$
  recv $X_{loc}$ from $P_{down}$
  $ind = ind + 1$
  if $ind = p_r$, $ind = 0$
  $Y_{loc} = Y_{loc} + A_{loc}(1 : n_1, ind m_1 + 1 : (ind + 1)m_1 ) X_{loc}$
end

Algorithm 2 is distinct from Algorithm 1 with its 2-direction message passing pattern. In Step 2 and Step 3 each row of processors in the mesh form a horizontal ring. Message passing is restricted within processors in this row. In Figure 4 the arrow lines show the message passing pattern in these two steps. Similarly, in Step 4 each column of processors in the mesh form a vertical ring. Message passing is restricted within processors in this column. The message passing pattern in this step is shown in Figure 5.

For each processor, in order to complete Algorithm 2, it takes $p_c$ ‘horizontal’ message sending and receiving actions and $p_r - 1$ ‘vertical’ message sending and receiving actions. In each action the message load is $m_1 n_1$. Therefore the total message sending and receiving load is

$$commu_2 = (p_c + p_r - 1)m_1 n_1.$$

The total flops performed in one processor is

$$flops_2 = 2m_1^2 n_1 p_r + 2m_1 n_1^2 p_c + (p_c + p_r + 2)m_1 n_1.$$

The total memory required is

$$memory_2 = m_1 n_1 + mn_1 + 4m_1 n_1 + n_1$$

including a $m_1 n_1$ buffer for message passing.
Figure 4: Message passing pattern in Step 2 and 3

Figure 5: Message passing pattern in Step 4
4 Comparisons

Like Algorithm 1, Algorithm 2 is easy to implement and perfectly load balanced. Furthermore, it has several advantages over Algorithm 1.

First, Algorithm 1 requires that $p$, the number of processors used, must agree with $n$. This would limit its applications on practical problems. For Algorithm 2, the requirement for the size of the processor mesh is much looser. In order to keep the load balance, $p_r$ and $p_c$, the number of rows and the number of columns in the mesh, have to be a factor of $m$ and a factor of $n$, respectively. However, the algorithm can be arranged to deal with a general case where $p_r$ and $p_c$ are arbitrary with imbalanced communication and flops load on processors. Therefore, for many practical cases where Algorithm 1 cannot be implemented, the comparison between the two algorithms is out of the question.

Secondly, even in a case where both algorithms can be used, they apply different message passing strategies which have different communication costs. In Step 2 and Step 3 of Algorithm 2, each row of processors carries out message passing actions independently. Also in Step 4 each column of processors carries out message passing actions independently. This 2-direction message passing pattern makes it more efficient on communications than Algorithm 1. In order to compare the flops and communication load of the two algorithms, we have to fix the sizes of the operators, as well as the numbers of processors being used in the ring network for Algorithm 1 and the mesh network for Algorithm 2. We set $m = n$, so that Algorithm 1 would be well suited to a ring of $n$ processors. Accordingly, we assume Algorithm 2 is adapted on an $\sqrt{n}$ by $\sqrt{n}$ mesh of processors with total $n$ processors. In this case, Algorithm 1 requires each processor to send and receive

$$\text{commu}_1 = n^2 - n$$
floating points and to perform

$$\text{flops}_1 = 4n^2 - 4n$$
flops with a memory space of

$$\text{memor}_1 = n^2 + 3n + n.$$  

Algorithm 2 requires each processor to send and receive

$$\text{commu}_2 = 2n^{3/2} - n$$
floating points and to perform

$$\text{flops}_2 = 4n^2 + 2n^{3/2} + 2n$$
flops with a memory space of

$$\text{memor}_2 = 2n^{3/2} + 4n + \sqrt{n}.$$  

It is clear from these numbers that while Algorithm 2 keeps about the same flops load as Algorithm 1 has (with respect to the leading terms of \text{flops}_1 and \text{flops}_2), it reduces both communication load and memory space dramatically.

Finally, the major computations involved in Algorithm 2 are arisen from matrix-matrix operations, usually handled by Level-3 BLAS routines, while all computations involved in Algorithm 1 are related to Level-2 or lower level operations. This fact provides another advantage of Algorithm 2 over Algorithm 1 because on a high-performance computer system such as Touchstone DELTA, higher level operations usually bring better performance than lower level operations when these operations are executed by assembly-coded BLAS routines. In Figure 6, we show the performances of the highly optimized assembly-coded BLAS routines DGEMV and DGEMM on a single node of Touchstone DELTA [6]. Please notice that the difference in performance between different level operations does not appear when these operations are executed by ordinary BLAS routines. Also in Figure 6 we compare the performances of the FORTRAN-coded BLAS routines DGEMV and DGEMM (see [1]) on a single node of Touchstone DELTA, showing no distinct difference.
5 Discussions on Speed-up

By definition, we say that a parallel algorithm for a particular problem achieves speed-up $S$ if

$$S = T_s / T_p$$

where $T_p$ is the time required for execution of the parallel program on $p$ processors and $T_s$ is the time required by one processor when the best uniprocessor procedure is used (see [4]). We will give the speed-up test data of Algorithm 2 which is computed according to this concept in the next section. However, the statement 'the best uniprocessor procedure' is somewhat confusing in our case. Since our procedure is designed to distribute the original system onto a network of large number of processors, it is likely that a problem of certain sizes can be computed by $p$ processors but cannot be computed by one processor due to insufficient memory on a single processor. In this case, the following ratio should be considered as the 'relative speed-up':

$$\hat{S} = T_{p_{\text{min}}} / T_p,$$

where $p_{\text{min}}$ stands for the least number of processors on which the computation can be carried out. Hereafter, for an integer $i$, $T_i$ stands for the time required for execution of a certain operation on $i$ processors. More generally, we can define relative speed-up $\hat{S}$ as

$$\hat{S} = T_{p_0} / T_{p\mu}$$

where $p_0$ is given and $\mu$ is an integer. In any case, it is important to know, for an operator with given sizes, how run time $T_p$ changes when the processor number $p$ changes. In the following analysis, we measure the change of flops and communication time to 'predict' the change of run time. In next section, we will compare our 'prediction's with the real run time data. An alternative way to measure speed-up would be to use Gustafson's [3] scaled speed-up idea. However, we think the data presented here are informative and they only involve measured quantities.

For simplicity, we assume that our operator has sizes $n$ by $n$ and that in our processor mesh $p_c = p_r$. We first double the sizes of the mesh in both direction, i.e. we enlarge the mesh into $2p_c$ by $2p_c$. Then we enlarge the mesh into $kp_c$ by $kp_c$. Assuming that for each processor $\alpha$ is the
Table 1: Change in flops and communication time caused by changing of processor mesh sizes

<table>
<thead>
<tr>
<th>Problem sizes</th>
<th>Processor number</th>
<th>( t_1 ) (in sec.)</th>
<th>( t_2 ) (in sec.)</th>
<th>( t_1/t_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 x 256 = 65536</td>
<td>16 x 16 = 256</td>
<td>0.2892</td>
<td>0.05249</td>
<td>5.510</td>
</tr>
<tr>
<td>196 x 196 = 38416</td>
<td>14 x 14 = 196</td>
<td>0.1660</td>
<td>0.03431</td>
<td>4.838</td>
</tr>
<tr>
<td>144 x 144 = 20736</td>
<td>12 x 12 = 144</td>
<td>0.1008</td>
<td>0.02279</td>
<td>4.423</td>
</tr>
<tr>
<td>100 x 100 = 10000</td>
<td>10 x 10 = 100</td>
<td>0.05433</td>
<td>0.01500</td>
<td>3.622</td>
</tr>
<tr>
<td>64 x 64 = 4096</td>
<td>8 x 8 = 64</td>
<td>0.01953</td>
<td>0.007810</td>
<td>2.501</td>
</tr>
</tbody>
</table>

Table 2: Comparison of Algorithm 1 and Algorithm 2

\( t_1 \): run time per operation of Algorithm 1.

\( t_2 \): run time per operation of Algorithm 2.

time required to initiate a message and \( \beta \) is the rate that a message can be transferred, the time of sending \( l \) floating point numbers once between two processors is

\[ \alpha + l\beta. \]

Table 1 shows the change in flops and communication time on one processor.

Thus, when the number of processors is raised by \( k^2 \) times, the flops load on each processor is reduced by about \( k^2 \) times. The reduction of the communication time is more complicated to calculate. In the situation when \( \alpha \ll n_1^2\beta \) the reduction is about \( k \) times. However, we can set \( k^2 \) and \( k \) as the ‘upper bound’ and the ‘lower bound’ for the reductions in run time. In the next section, we will refer \( k \) and \( k^2 \) as ‘ideal computation speed-up’ and ‘ideal communication speed-up’ caused by raising the processor sizes in both directions \( k \) times, respectively.

6 Numerical Test Results

In this section, we provide some numerical results to demonstrate the behavior of Algorithm 2 and to compare Algorithm 2 with Algorithm 1. Both algorithms are implemented on the Intel Touchstone DELTA supported by NX/M R 1.5 operating system and PGI R 4.0 compiler with up to 256 i860 processors. All the matrices involved in both matrix form (2) and (1) are formed by random numbers. All programs are composed with the assembly-coded BLAS routines [6] and run in double precision. The single processor program is arranged in order to optimize the run time.

First, we arrange a series of tests to compare Algorithm 1 and Algorithm 2. In each step, we apply Algorithm 1 and Algorithm 2 to the equivalent operators (1) and (2) with the same operator sizes on the same processor mesh. In order to compare our results to the analysis we made in Section 4, we always set \( n = m \) and the processor mesh sizes \( \sqrt{n} \) by \( \sqrt{n} \). Table 6 compares the performances of both algorithms in terms of average time per operation. In the last column of the table, the ratios \( t_1/t_2 \) show the relative speed-ups from Algorithm 1 to Algorithm 2 with respect to different operator sizes and processor mesh sizes. The results indicate that under the above assumptions on the operator sizes and the processor mesh sizes, Algorithm 2 reduced run time dramatically, and when the problem sizes and the processor mesh sizes increase, the amount of the reduction increases too.

We then test the speed-up behavior of Algorithm 2. We first apply Algorithm 2 to the operator with sizes 256 x 256 on processor meshes with different sizes. Then we raise the sizes of the operator to 512 x 512 and 528 x 528, respectively. Figure 7 shows the speed-up in these three cases.
The speed-up curves in Figure 7 implies that when the operator sizes are increased, the speed-up performance is improved. Since a 528 x 528 operator is the largest one we can test on a single node, we have to use other measurement to observe the parallel performance of Algorithm 2.

Following the discussions in Section 4, we arrange tests to observe the ‘relative speed-up’. This time we take the operator with sizes 1024 x 1024. We start from using 4 x 4 = 16 processors and denote the observed run time by $T_{16}$. Then for $k = 2, 4, 8$ we use $4k \times 4k = 16k^2$ processors and denoted the run time by $T_{16k^2}$. Figure 8 shows the ‘relative speed-up’ curve $T_{16}/T_{16k^2}$, and also compares the curve with the ‘ideal computation speed-up’ curve $k^2$ and the ‘ideal communication speed-up’ curve $k$. In order to compare the behaviors of the operators with sizes 256 x 256 and 512 x 512, which we have discussed by terms of speed-up, we also show the curves of these two cases obtained by similar method as the case of 1024 x 1024. However, the curves indicate that $k^2$ and $k$ are good upper bound and lower bound for the real time speed-up. Further more, while operator sizes increase, the relative speed-up curve moves from the curve $k$ towards the curve $k^2$.

In order to measure the performance of Algorithm 2 on larger operators, we observe the relative speed-up caused by raising the processor mesh sizes once. In the tests, we start from using 8 x 8 = 64 processors. Figure 9 shows, for different operator sizes, the speed-ups caused by raising the processor mesh sizes to 16 x 16.

The different behaviors of Algorithm 2 on different operator sizes shown in Figure 9 indicate again that the change of system sizes will effect the speed-up behavior, the larger the system sizes.
are, the closer it looks to 'ideal computation speed-up'.

Finally, for fixed $16 \times 16 = 256$ node mesh, Figure 10 shows the performances of Algorithm 2 on different sized matrices. The figure, in terms of MFLOPS per node, indicates again that the performance becomes better when the sizes of operator becomes larger. For the operator with sizes $2896 \times 2896$, which is the largest one we can test on a $16 \times 16$ node mesh, we can reach 27.6 MFLOPS per node.

7 Future Work: a Systolic Model on a 3-D Torus

The idea of 2-direction message passing in Algorithm 2 can be extended to a 3-direction message passing model which could deal with the 3-D array operator

$$\tilde{M}_{3D}(x) = (I \otimes I \otimes A + I \otimes B \otimes I + C \otimes I \otimes I + \hat{V})x,$$

where $\hat{V}$ is diagonal, on a distributed computer with a 3-D torus network. Similar to the matrix form of the Sylvester-like operator, we can derive an equivalent form of the above operator. Again, for simplicity, we assume that $A, B, C$ are symmetric, and that all matrices involved have the same sizes $n$ by $n$. Let $A = (a_{i,j}), B = (b_{i,j}), C = (c_{i,j})$, we define an operator $\tilde{A}$ on $R^{n \times n \times n}$: for $X \in R^{n \times n \times n}$, $Y = \tilde{A}(X)$ where

$$Y(:, :, j) = \sum_{i=1}^{n} a_{i,j}X(:, :, i).$$

Similarly, define $Y = \tilde{B}(X)$ by

$$Y(:, j, :) = \sum_{i=1}^{n} b_{i,j}X(:, i, :),$$

$$Y(:, :, :) = \sum_{i=1}^{n} c_{i,j}X(:, :, i).$$
Figure 9: Relative speed-up $\hat{S} = T_{64}/T_{256}$ on Touchstone DELTA

and define $Y = \tilde{C}(X)$ by

$$ Y(j,:) = \sum_{i=1}^{n} c_{i,j} X(i,:) . $$

Under the usual one-to-one correspondence from $\mathbb{R}^{n \times n \times n}$ to $\mathbb{R}^{n^3}$:

$$ x = \begin{pmatrix}
X(:,1,1) \\
X(:,1,2) \\
\vdots \\
X(:,1,n) \\
\vdots \\
X(:,n,1) \\
\vdots \\
X(:,n,n)
\end{pmatrix} $$

$\hat{M}_{3D}(x)$ is equivalent to

$$ M_{3D}(X) = \tilde{A}(X) + \tilde{B}(X) + \tilde{C}(X) + V \odot X $$

where $V$ is defined by the inverse image of the one-to-one correspondence of $\text{diag}(\tilde{V})$ and $\odot$ denotes the element-by-element multiplication.

Assume we have a 3-D torus network of processors with sizes $p_1$ by $p_2$ by $p_3$ and the total number of processors $p = p_1 p_2 p_3$. For simplicity we assume $p_1 = p_2 = p_3$ and $n = n p_1$. It is natural to distribute the 3-D array $X = X(1:n,1:n,1:n)$ on the torus in a way such that
Figure 10: MFLOPS on each node for 256 nodes

an equal-sized block of $X$ is stored in the local array $X_{loc} = X_{loc}(1 : n_1, 1 : n_1, 1 : n_1)$. In the example shown in Figure 12, the 8 by 8 by 8 array can be distributed in to a 4 by 4 by 4 torus of processors so that each processor share a 2 by 2 by 2 chunk of the array. The same way can be applied to distribute $Y$ to $Y_{loc}$ and distribute $V$ to $V_{loc}$, respectively. Similar to Algorithm 2, we can label each processor with a logical processor number $\mu$ and corresponding 3-D Cartesian coordinates $(\mu_x, \mu_y, \mu_z)$. Also, similar to Algorithm 2, for a processor located at a certain position in the torus, left, right, up, down, front and back refer to the indices of its left (in -x-axis), right (in -x-axis), upward (in -z-axis), downward (in -z-axis), forward (in y-axis) and backward (in -y-axis) neighbors, respectively.

Matrices $A$, $B$ and $C$ can be distributed in the local arrays $A_{loc}(1 : n_1, 1 : n)$, $B_{loc}(1 : n_1, 1 : n)$ and $C_{loc}(1 : n_1, 1 : n)$ on processor $P_\mu$ with coordinates $(\mu_x, \mu_y, \mu_z)$ in the following way

$A_{loc} = A(\mu_x n_1 + 1 : (\mu_x + 1)n_1, :)$,

$B_{loc} = B(\mu_y n_1 + 1 : (\mu_y + 1)n_1, :)$,

$C_{loc} = C(\mu_z n_1 + 1 : (\mu_z + 1)n_1, :)$,

Similar to the 2-D case, the equivalent operations $y = M_3D(x)$ and $Y = M_3D(X)$ impose different dependencies of $y$ on $x$ and of $Y$ on $X$, respectively. As shown in Figure 11, for computing a 'cubic' $Y(i_1 : i_2, j_1 : j_2, k_1 : k_2)$ in $Y$, only three 'bars' of $X$, namely, $X(i_1 : i_2, j_1 : j_2, :)$, $X(i_1 : i_2, :, k_1 : k_2)$ and $X(:, j_1 : j_2, k_1 : k_2)$ are needed.

Now we are ready to describe our proposed algorithm of computing $Y = M_3D(X)$, which can be viewed as a straightforward generalization of Algorithm 2.

Algorithm 3 (for $P_\mu$)
Figure 11: Dependency of $Y$ on $X$

{Step 1. applying the operation $Y = V \otimes X$}
for $i = 1 : n_1, j = 1 : n_1, k = 1 : n_1$

$Y_{loc}(i, j, k) = V_{loc}(i, j, k)X_{loc}(i, j, k)$
end

{Step 2. applying the operation $Y = Y + \tilde{A}(X)$}

$Y_{loc} = Y_{loc} + \tilde{A}_{loc}(\mu_x n_1 + 1 : (\mu_x + 1)n_1, 1 : n_1)(X_{loc})$

$ind = \mu_x$
for $i = 1 : p_1 - 1$
send $X_{loc}$ to $P_{up}$
recv $X_{loc}$ from $P_{down}$
ind = ind + 1
if ind = $p_1$, ind = 0
$Y_{loc} = Y_{loc} + \tilde{A}_{loc}(\text{ind } n_1 + 1 : (\text{ind } + 1)n_1, 1 : n_1)(X_{loc})$
end

{Step 3. recovering the original distribution of $X$}
send $X_{loc}$ to $P_{up}$
recv $X_{loc}$ from $P_{down}$

{Step 4. applying the operation $Y = Y + \tilde{B}(X)$}

$Y_{loc} = Y_{loc} + \tilde{B}_{loc}(\mu_y n_1 + 1 : (\mu_y + 1)n_1, 1 : n_1)(X_{loc})$

$ind = \mu_y$
for $i = 1 : p_1 - 1$
send $X_{loc}$ to $P_{front}$
recv $X_{loc}$ from $P_{back}$
ind = ind + 1
if ind = $p_1$, ind = 0
$Y_{loc} = Y_{loc} + \tilde{B}_{loc}(\text{ind } n_1 + 1 : (\text{ind } + 1)n_1, 1 : n_1)(X_{loc})$
end

{Step 5. recovering the original distribution of $X$}
send $X_{loc}$ to $P_{front}$
recv $X_{loc}$ from $P_{back}$

{Step 6. applying the operation $Y = Y + \tilde{C}(X)$}

$Y_{loc} = Y_{loc} + \tilde{C}_{loc}(\mu_x n_1 + 1 : (\mu_x + 1)n_1, 1 : n_1)(X_{loc})$

$ind = \mu_x$
Figure 12: A 3-D array, left, and a 3-D torus network, right

for $i = 1 : p_1 - 1$
  send $X_{loc}$ to $P_{left}$
  recv $X_{loc}$ from $P_{right}$
  $ind = ind + 1$
  if $ind = p_1$, $ind = 0$
    $Y_{loc} = Y_{loc} + C_{loc}((ind \cdot n_1 + 1 : (ind + 1)n_1, 1 : n_1)(X_{loc}))$
  end

8 Acknowledgement

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CCR-8809615.

References


