

Minimizing Message Passing in Parallel Iterative Codes

Peiyi Tang

Department of Computer Science
University of Arkansas at Little Rock
2801 S. University Avenue
Little Rock, AR 72204

Abstract

A compiler transformation to minimize the number of messages in PDE parallel iterative codes is presented. For the n -dimensional high-order PDE or second-order PDE using high-order finite differences, this transformation reduces the number of messages from as large as $3^n - 1$ to the minimum $2n$.

1 Introduction

To solve the second-order partial differential equations (PDE) using the second-order finite differences on parallel computers, message passing in the elementary directions such as north, south, east and west, is required and the number of messages at the end of each iteration is small. However, with the increasing computing power of modern processors we would consider using higher-order finite differences for higher precision or faster convergence. The two-dimensional second-order Poisson equation using fourth-order finite differences has a 9-point stencil and its linear system is as follows:

$$\begin{aligned} 20u_{i,j} &- 4(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) \\ &- (u_{i+1,j+1} + u_{i-1,j+1} + u_{i-1,j-1} + u_{i+1,j-1}) \\ &= -6h^2 f_{i,j} \end{aligned} \quad (1)$$

According to the traditional parallel algorithm, each processor needs to send and receive eight messages including the messages in the non-elementary directions such as north-east and north-west and so on.

For the higher-order PDEs, their stencils always include the messages in the non-elementary directions. For example, the 13-point stencil of the two-dimensional fourth-order biharmonic equation [1] would also require eight messages in the parallel code. In general, for the n -dimensional PDE problems the number of messages needed may be as large as $3^n - 1$.

In this paper, we present a transformation for parallel iterative codes to reduce the number of messages from $3^n - 1$ to the minimum $2n$. The technique used is

to piggy-back all the messages in the non-elementary directions to the messages in the $2n$ elementary directions so that the messages in the non-elementary directions are no longer necessary.

It must be emphasized that, although the messages in non-elementary directions are actually routed along the elementary directions by the hardware in 2D- or 3D-mesh supercomputers, the message passing in these non-elementary directions still exist in the parallel iteration codes. Unless they are eliminated as we show in this paper, they will cause significant overhead due the large start-up cost of message passing.

The transformation presented in this paper can be used by parallelizing compilers or domain-specific parallel PDE-compiler to generate efficient parallel iteration codes with minimum message passing.

The organization of this paper is as follows. In Section 2, we provide a formal framework to generate parallel iterative codes for distributed-memory machines. In Section 3, we present the transformation to minimize the number of messages. In Section 4, we further simplify the minimum message passing code. In Section 5, we present the experimental data. Section 6 concludes the paper with related work.

2 PDE Parallel Iterative Codes

The SPMD (Single Program Multiple Data) program of the parallel iterative codes using the Gauss-Seidel or SOR methods is shown in Fig. 1. Array $A'[\]$ is the local data mesh points in each processor. Function $F(\)$ is used to calculate the value of the current data mesh point $A'[\vec{i}]$ using the values of the nearby data mesh points referenced by the *stencil vectors*, $\vec{s}_1, \dots, \vec{s}_r$. These stencil vectors are determined by the stencil of the equation. Vector \vec{i} is the index vector of the nested loop and B_1, \dots, B_n are loop upper bounds determined by the data partition of parallel code. At the end of the each iteration of the iterative **while** loop, each processor needs to send messages to its neighboring processors to update the values of

```

let the current processor be  $\vec{p}$ 
while maximum error  $\geq \epsilon$ 
  for  $i_1 = 1, B_1$ 
  ...
  for  $i_n = 1, B_n$ 
     $A'(\vec{i}) = F(A'(\vec{i}), A'(\vec{i} + \vec{s}_1), \dots, A'(\vec{i} + \vec{s}_r))$ 
    update the maximum error
  endfor
  ...
endfor
for each  $\vec{d} \in \mathcal{D}_s$ 
  if  $\vec{p} + \vec{d} \in \mathcal{P}$  then
    send message  $\mathcal{W}_{\vec{d}}$  to processor  $\vec{p} + \vec{d}$ 
  endif
  if  $\vec{p} - \vec{d} \in \mathcal{P}$  then
    receive a message from processor  $\vec{p} - \vec{d}$ 
    and store it in  $\mathcal{R}_{-\vec{d}}$ 
  endif
endfor
endwhile

```

Figure 1: SPMD Program of Parallel Iterative Code

their mesh points at borders. \mathcal{D}_s is the set of *send direction vectors* along which each processor needs to send messages and can be derived for the stencil vectors. \mathcal{P} of the set of identity vectors of all processors which themselves form an n -dimensional processor array: $\mathcal{P} = \{\vec{p} \in \mathcal{Z}^n \mid \vec{1} \leq \vec{p} \leq \vec{P}\}$. $\mathcal{W}_{\vec{d}}$, called *remote write set*, is the local mesh data points whose values need to be sent to the neighboring processor $\vec{p} + \vec{d}$. Likewise, $\mathcal{R}_{-\vec{d}}$, called *remote read set*, is the local mesh data points where to store the values received from neighboring processor $\vec{p} - \vec{d}$. Fig. 2 shows the layout of the local data mesh for the 2-dimensional problem with the eight send direction vectors. Specifically, the remote write set and remote read set are defined as follows:

Definition 1 (Remote Write Set) *For each processor \vec{p} and a send direction vector $\vec{d} \in \mathcal{D}_s$, the set of data points whose values need to be sent to processor $\vec{p} + \vec{d}$ is called a remote write set and denoted as $\mathcal{W}_{\vec{d}}$. When necessary, it is also denoted as $\mathcal{W}_{\vec{d}}^{\vec{p}}$. In particular,*

$$\mathcal{W}_{\vec{d}} = \{A'[\vec{i}] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

where \vec{l} and \vec{u} are defined by:

$$[l_i, u_i] = \begin{cases} [B_i - \delta_i^- + 1, B_i] & \text{if } d_i > 0 \\ [1, B_i] & \text{if } d_i = 0 \\ [1, \delta_i^+] & \text{if } d_i < 0 \end{cases}$$

for each $1 \leq i \leq n$.

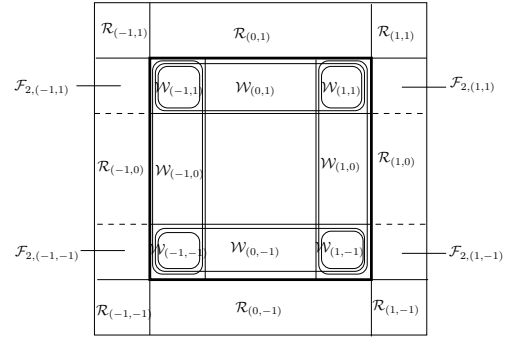


Figure 2: Remote Read and Write Sets and Forward Sets

Definition 2 (Remote Read Set) *For each processor \vec{p} and a direction vector $\vec{d} \in -\mathcal{D}_s$, the set of data points whose values need to be sent from neighbor processor $\vec{p} + \vec{d}$ is called a remote read set and denoted as $\mathcal{R}_{-\vec{d}}$. When necessary, it is also denoted as $\mathcal{R}_{-\vec{d}}^{\vec{p}}$. In particular,*

$$\mathcal{R}_{-\vec{d}} = \{A'[\vec{i}] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

where \vec{l} and \vec{u} are defined by:

$$[l_i, u_i] = \begin{cases} [B_i + 1, B_i + \delta_i^+] & \text{if } d_i > 0 \\ [1, B_i] & \text{if } d_i = 0 \\ [-\delta_i^- + 1, 0] & \text{if } d_i < 0 \end{cases}$$

for each $1 \leq i \leq n$.

Here we used two *boundary vectors*, δ_i^+ and δ_i^- , which quantify the border area of the data mesh. Vector $\delta_i^+ = (\delta_1^+, \dots, \delta_n^+)$ is defined by

$$\delta_i^+ = \max\{s_i \mid (s_1, \dots, s_i, \dots, s_n) \in \mathcal{S} \wedge s_i \geq 0\} \quad (2)$$

for each $1 \leq i \leq n$. Likewise, vector $\delta_i^- = (\delta_1^-, \dots, \delta_n^-)$ is defined by

$$\delta_i^- = \max\{-s_i \mid (s_1, \dots, s_i, \dots, s_n) \in \mathcal{S} \wedge s_i \leq 0\} \quad (3)$$

for each $1 \leq i \leq n$. Obviously, we have

$$-\delta_i^- \leq \vec{s} \leq \delta_i^+ \quad (4)$$

for all $\vec{s} \in \mathcal{S}$, and

$$-\delta_i^- \leq \vec{0} \leq \delta_i^+ \quad (5)$$

The remote write set $\mathcal{W}_{\vec{d}}$ of processor \vec{p} and the remote read set $\mathcal{R}_{-\vec{d}}$ of processor $\vec{p} + \vec{d}$ are the two local realizations of the same region of the global virtual array. However, they may not necessarily have the same values. In order to implement the global virtual array correctly, the new values of $\mathcal{W}_{\vec{d}}^{\vec{p}}$ needs to be sent

from processor \vec{p} to processor $\vec{p} + \vec{d}$ and stored in $\mathcal{R}_{-\vec{d}}^{\vec{p} + \vec{d}}$. After this message in direction \vec{d} is sent and received, $\mathcal{W}_{\vec{d}}^{\vec{p}}$ and $\mathcal{R}_{-\vec{d}}^{\vec{p} + \vec{d}}$ will have the save values and we denote this assertion as

$$\mathcal{W}_{\vec{d}}^{\vec{p}} \stackrel{\vec{d}}{\equiv} \mathcal{R}_{-\vec{d}}^{\vec{p} + \vec{d}}$$

3 Minimizing Message Passing

We divide the direction vectors in \mathcal{D}_s into elementary and non-elementary ones. A direction vector $\vec{d} \in \mathcal{D}_s$ is *elementary* if it has only one non-zero element. A message sent along an elementary direction is called an *elementary* message.

The idea of piggy-backing non-elementary messages and forwarding them through the elementary messages is illustrated in Figure 3. Instead of sending the non-elementary message $\mathcal{W}_{(1,1)}^{\vec{p}}$ to processor $\vec{p} + (1,1)$ directly, we can forward it to processor $\vec{p} + (1,0)$ as part of message $\mathcal{W}_{(1,0)}^{\vec{p}}$ and stored it in the forward set $\mathcal{F}_{2,(-1,1)}$ of processor $\vec{p} + (1,0)$. Then when processor $\vec{p} + (1,0)$ send its $\mathcal{W}_{(0,1)}$ to processor $\vec{p} + (1,1)$, it can piggy-back its $\mathcal{F}_{2,(-1,1)}$ to it and send them as one message. The forwarded data will be stored in $\mathcal{R}_{(-1,-1)}$ of processor $\vec{p} + (1,1)$. Thus, the directy passing of message $\mathcal{W}_{(1,1)}^{\vec{p}}$ from processor \vec{p} to processor $\vec{p} + (1,1)$ is no longer necessary.

The storages to store the non-elementary messages to be forwarded to their final destinations are called *forward sets* and defined as follows:

Definition 3 (Forward Set) *Given an n -vector \vec{g} such that $|\vec{g}| \leq \vec{1} \wedge \vec{g} \neq \vec{0}$ and an integer k such that $1 \leq k \leq n$, the forward set $\mathcal{F}_{k,\vec{g}}$ in each processor is defined as follows:*

$$\mathcal{F}_{k,\vec{g}} = \{A'[\vec{i}] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

where \vec{l} and \vec{u} are defined by:

$$[l_i, u_i] = \begin{cases} [B_i + 1, B_i + \delta_i^+] & \text{if } i < k \wedge g_i = 1 \\ [-\delta_i^- + 1, 0] & \text{if } i < k \wedge g_i = -1 \\ [1, B_i] & \text{if } g_i = 0 \\ [B_i - \delta_i^- + 1, B_i] & \text{if } i \geq k \wedge g_i = 1 \\ [1, \delta_i^+] & \text{if } i \geq k \wedge g_i = -1 \end{cases}$$

for each $1 \leq i \leq n$.

Fig. 2 also shows the four forward sets in the 2-dimensional problem. As usual, we use $\mathcal{F}_{k,\vec{g}}^{\vec{p}}$ to denote $\mathcal{F}_{k,\vec{g}}$ of processor \vec{p} . The definition of $\mathcal{F}_{k,\vec{g}}^{\vec{p}}$ is split to two sections: the first $k-1$ dimensions (i.e. for $i < k$) and the rest $n-k+1$ dimensions (i.e. for $i \geq k$). The definition for the first $k-1$ dimensions is the same

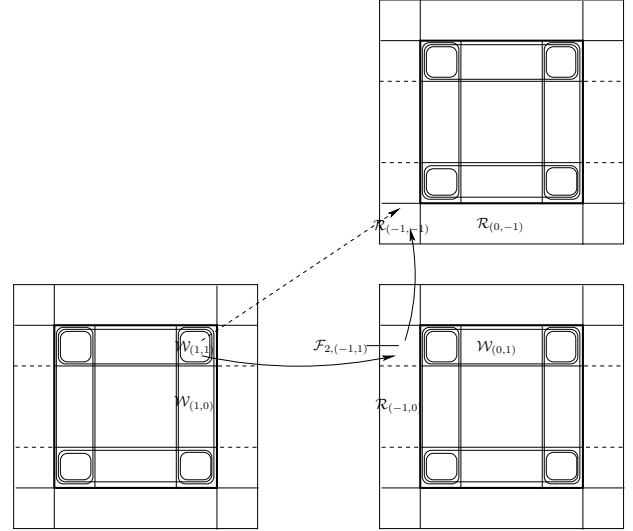


Figure 3: Forwarding $\mathcal{W}_{(1,1)}$

```

for  $k = 1, n$ 
  if  $\vec{p} + \vec{e}_k \in \mathcal{P} \wedge \vec{e}_k \in \mathcal{D}_s$  then
    send message
     $\mathcal{W}_{\vec{e}_k} \cup \bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ 
    to processor  $\vec{p} + \vec{e}_k$ 
  endif
  if  $\vec{p} - \vec{e}_k \in \mathcal{P} \wedge \vec{e}_k \in \mathcal{D}_s$  then
    receive a message from processor  $\vec{p} - \vec{e}_k$ 
    and store the received  $\mathcal{W}_{\vec{e}_k}$  to  $\mathcal{R}_{-\vec{e}_k}$  and
    for each  $(g_1, \dots, g_{k-1}) \neq \vec{0}$ 
      store the received  $\mathcal{F}_{k,(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ 
      in  $\mathcal{R}_{(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$ 
    endifor
  endif
  if  $\vec{p} - \vec{e}_k \in \mathcal{P} \wedge -\vec{e}_k \in \mathcal{D}_s$  then
    send message
     $\mathcal{W}_{-\vec{e}_k} \cup \bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$ 
    to processor  $\vec{p} - \vec{e}_k$ 
  endif
  if  $\vec{p} + \vec{e}_k \in \mathcal{P} \wedge -\vec{e}_k \in \mathcal{D}_s$  then
    receive a message from processor  $\vec{p} + \vec{e}_k$ 
    and store the received  $\mathcal{W}_{-\vec{e}_k}$  to  $\mathcal{R}_{\vec{e}_k}$  and
    for each  $(g_1, \dots, g_{k-1}) \neq \vec{0}$ 
      store the received  $\mathcal{F}_{k,(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$ 
      in  $\mathcal{R}_{(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ 
    endifor
  endif
endifor

```

Figure 4: Code of Minimum Message Passing

as the remote read set, while the definition of the rest $n-k+1$ dimensions is the same as the remote write set. The rationale behind the forward set is that $\mathcal{F}_{k,\vec{g}}^{\vec{p}}$ is the storage to store $\mathcal{W}_{(-g_1, \dots, -g_{k-1}, 0, \dots, 0)}^{\vec{p}+(0, \dots, 0, g_k, \dots, g_n)}$ originated in processor $\vec{p}+(-g_1, \dots, -g_{k-1}, 0, \dots, 0)$. And it will be forwarded further to processor $\vec{p}+(0, \dots, 0, g_k, \dots, g_n)$ and stored in $\mathcal{R}_{(g_1, \dots, g_{k-1}, -g_k, \dots, -g_n)}^{\vec{p}+(0, \dots, 0, g_k, \dots, g_n)}$.

The code for message passing at the end of each iteration of the **while** loop of Fig. 1 now becomes as shown in Fig. 4, where we use \vec{e}_k to denote the k -th elementary vector, i.e. $e_{k,k} = 1$ and $e_{k,j} = 0$ for all $j \neq k$ and $1 \leq j \leq n$. Messages are sent only along at most $2n$ elementary directions in the order of $\vec{e}_1, -\vec{e}_1, \dots, \vec{e}_n, -\vec{e}_n$. Each message is a union of the original elementary message, $\mathcal{W}_{\vec{e}_k}$ or $\mathcal{W}_{-\vec{e}_k}$, with a collection of forward sets, $\bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ or $\bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$, respectively. When $k = 1$, both $\bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ and $\bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k,(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$ are empty, because the $(g_1, \dots, g_{k-1}) \neq \vec{0}$ do not exist.

Notice that we store the received $\mathcal{F}_{k,(g_1, \dots, g_{k-1}, 1, 0, \dots, 0)}$ in $\mathcal{R}_{(g_1, \dots, g_{k-1}, -1, 0, \dots, 0)}$. This is because they are mapped to the same area in the global virtual array, as implied by the following lemma.

Lemma 1 *Given a processor \vec{p} , integer $1 \leq k \leq n$ and vector $(g_1, \dots, g_k, 0, \dots, 0)$ such that $g_k \neq 0$, $\mathcal{F}_{k,(g_1, \dots, g_{k-1}, g_k, 0, \dots, 0)}^{\vec{p}}$ and $\mathcal{R}_{(g_1, \dots, g_{k-1}, -g_k, 0, \dots, 0)}^{\vec{p}+(0, \dots, 0, g_k, 0, \dots, 0)}$ are mapped to the same area of the global virtual array.*

The next theorem establishes the correctness of the message passing code in Fig. 4. That is, at the end of every iteration of the iterative loop, $\mathcal{W}_{\vec{d}}^{\vec{p}} \equiv \mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ is true for every processors $\vec{p}, \vec{p} + \vec{d} \in \mathcal{P}$ and every $\vec{d} \in \mathcal{D}_s$.

Theorem 1 *Given a non-elementary send direction vector $\vec{d} = (\dots, d_{i_1}, \dots, d_{i_2}, \dots, d_{i_k}, \dots)$, where d_{i_j} ($j = 1, \dots, k$) are the only k ($2 \leq k \leq n$) non-zero elements of \vec{d} and $i_1 < \dots < i_k$, after all the elementary messages in Fig. 4 are passed, the following assertions are true:*

$$\begin{aligned} \mathcal{W}_{\vec{d}}^{\vec{p}} &\stackrel{\equiv}{=} \mathcal{F}_{i_1+1, (\dots, -d_{i_1}, \dots, d_{i_2}, \dots, d_{i_k}, \dots)}^{\vec{p}+\vec{d}_{i_1}} \\ &\stackrel{\equiv}{=} \mathcal{F}_{i_2+1, (\dots, -d_{i_1}, \dots, -d_{i_2}, \dots, d_{i_3}, \dots, d_{i_k}, \dots)}^{\vec{p}+\vec{d}_{i_1}+\vec{d}_{i_2}} \\ &\equiv \dots \\ &\stackrel{\equiv}{=} \mathcal{F}_{i_{k-1}+1, (\dots, -d_{i_1}, \dots, -d_{i_{k-1}}, \dots, d_{i_k}, \dots)}^{\vec{p}+\vec{d}_{i_1}+\dots+\vec{d}_{i_{k-1}}} \\ &\stackrel{\equiv}{=} \mathcal{R}_{(\dots, -d_{i_1}, \dots, -d_{i_{k-1}}, \dots, -d_{i_k}, \dots)}^{\vec{p}+\vec{d}_{i_1}+\dots+\vec{d}_{i_k}} \end{aligned}$$

where vector \vec{d}_{i_j} is the elementary direction vector $(0, \dots, 0, d_{i_j}, 0, \dots, 0)$, i.e. $\vec{d}_{i_j} = d_{i_j} \vec{e}_{i_j}$.

This can be proved by mathematical induction. The detail can be found in [2].

According to the way \mathcal{D}_s is obtained (not shown in this paper), if $\vec{d} = (\dots, d_{i_1}, \dots, d_{i_2}, \dots, d_{i_k}, \dots)$ is in \mathcal{D}_s , the elementary send directions \vec{d}_{i_j} ($j = 1, \dots, k$) are also in \mathcal{D}_s . Therefore, each processor will send messages in the directions \vec{d}_{i_j} in the order of $j = 1, \dots, k$. Theorem 1 says that the non-elementary message $\mathcal{W}_{(\dots, d_{i_1}, \dots, d_{i_2}, \dots, d_{i_k}, \dots)}^{\vec{p}}$ will be forwarded by the elementary messages in directions \vec{d}_{i_j} ($j = 1, \dots, k$) and stored in the corresponding forward sets in processors $\vec{p}+\vec{d}_{i_1}, \vec{p}+\vec{d}_{i_1}+\vec{d}_{i_2}, \dots, \vec{p}+\vec{d}_{i_1}+\vec{d}_{i_2}+\dots+\vec{d}_{i_{k-1}}$. It will eventually reach processor $\vec{p}+\vec{d}_{i_1}+\vec{d}_{i_2}+\dots+\vec{d}_{i_{k-1}}+\vec{d}_{i_k}$ and be stored in $\mathcal{R}_{(\dots, -d_{i_1}, \dots, -d_{i_{k-1}}, \dots, -d_{i_k}, \dots)}^{\vec{p}+\vec{d}_{i_1}+\vec{d}_{i_2}+\dots+\vec{d}_{i_{k-1}}+\vec{d}_{i_k}}$.

In other words, after all the messages in the code in Fig. 4 are finished by all processors, we have $\mathcal{W}_{\vec{d}}^{\vec{p}} \equiv \mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ for all $\vec{d} \in \mathcal{D}_s$ and all $\vec{p}, \vec{p} + \vec{d} \in \mathcal{P}$.

It is not difficult to see that the number of messages in Fig. 4 cannot be reduced further. According to Theorem 1, all the non-elementary messages will be piggy-backed and forwarded by the elementary messages to their final destinations. This is because every non-elementary direction can be decomposed to a summation of elementary directions. On the other hand, any elementary direction cannot be decomposed to a summation of other elementary directions. Therefore, every elementary message in Fig. 4 is essential and cannot be piggy-backed and forwarded.

4 Simplify Minimum Message Passing

In this section, we further simplify the minimum message passing code in Fig. 4 by combining the multiple sets of a message to one set to minimize message packing and unpacking.

We define the extended remote write set as follows:

Definition 4 (Extended Remote Write Set)

Given an elementary send direction in the k -th dimension $(0, \dots, 0, d_k, 0, \dots, 0)$, its extended remote write set, denoted as $\mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e$, is

$$\mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e = \{A^{\vec{l}}[\vec{i}] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

where \vec{l} and \vec{u} are defined by:

$$[l_i, u_i] = \begin{cases} [-\delta_i^- + 1, B_i + \delta_i^+] & \text{if } i < k \\ [B_i - \delta_i^- + 1, B_i] & \text{if } i = k \wedge d_k = 1 \\ [1, 1 + \delta_i^+] & \text{if } i = k \wedge d_k = -1 \\ [1, B_i] & \text{if } i > k \end{cases}$$

for each $1 \leq i \leq n$.

The following lemma shows that the extended remote write set $\mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e$ is exactly what needs to be sent in elementary direction $(0, \dots, 0, d_k, 0, \dots, 0)$ in the code in Fig. 4.

Lemma 2 *Given an elementary send direction in the k -th dimension $(0, \dots, 0, d_k, 0, \dots, 0)$, $\mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e = \mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)} \cup \bigcup_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{F}_{k, (g_1, \dots, g_{k-1}, d_k, 0, \dots, 0)}$ is true.*

Therefore, the messages to be sent in the code in Fig. 4 can be simplified to $\mathcal{W}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e$ as one set and no message packing¹ is needed.

Similarly, we can combine all the remote read sets in the message receive code in Fig. 4 into an *extended* remote read set defined as follows to save message unpacking².

Definition 5 (Extended Remote Read Set)

Given an elementary receive direction in the k -th dimension $(0, \dots, 0, d_k, 0, \dots, 0)$, its extended remote read set, denoted as $\mathcal{R}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e$, is

$$\mathcal{R}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e = \{A'[i] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

where \vec{l} and \vec{u} are defined by:

$$[l_i, u_i] = \begin{cases} [-\delta_i^- + 1, B_i + \delta_i^+] & \text{if } i < k \\ [B_i + 1, B_i + \delta_i^+] & \text{if } i = k \wedge d_k = 1 \\ [-\delta_i^- + 1, 0] & \text{if } i = k \wedge d_k = -1 \\ [1, B_i] & \text{if } i > k \end{cases}$$

for each $1 \leq i \leq n$.

The following lemma shows that the extended remote read set $\mathcal{R}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e$ is exactly where to store the message received from elementary direction $(0, \dots, 0, d_k, 0, \dots, 0)$ in the code in Fig. 4.

Lemma 3 *Given an elementary receive direction in the k -th dimension $(0, \dots, 0, d_k, 0, \dots, 0)$, $\mathcal{R}_{(0, \dots, 0, d_k, 0, \dots, 0)}^e = \mathcal{R}_{(0, \dots, 0, d_k, 0, \dots, 0)} + \sum_{(g_1, \dots, g_{k-1}) \neq \vec{0}} \mathcal{R}_{(g_1, \dots, g_{k-1}, d_k, 0, \dots, 0)}$ is true.*

With the definitions of extended remote write and read sets, the message passing code in Fig. 4 can be simplified to the code shown in Fig. 5. According to Lemma 2 and Lemma 3, the code in Fig. 5 is equivalent to that in Fig. 4.

¹Message packing is an operation to copy various data sets to a message before sending.

²Message unpacking is the reverse of message packing.

```

for k = 1, n
  if  $\vec{p} + \vec{e}_k \in \mathcal{P} \wedge \vec{e}_k \in \mathcal{D}_s$  then
    send message
       $\mathcal{W}_{\vec{e}_k}^e$  to processor  $\vec{p} + \vec{e}_k$ 
  endif
  if  $\vec{p} - \vec{e}_k \in \mathcal{P} \wedge \vec{e}_k \in \mathcal{D}_s$  then
    receive a message from processor  $\vec{p} - \vec{e}_k$ 
    and store the received message in  $\mathcal{R}_{-\vec{e}_k}^e$ 
  endif
  if  $\vec{p} - \vec{e}_k \in \mathcal{P} \wedge -\vec{e}_k \in \mathcal{D}_s$  then
    send message  $\mathcal{W}_{-\vec{e}_k}^e$  to processor  $\vec{p} - \vec{e}_k$ 
  endif
  if  $\vec{p} + \vec{e}_k \in \mathcal{P} \wedge -\vec{e}_k \in \mathcal{D}_s$  then
    receive a message from processor  $\vec{p} + \vec{e}_k$ 
    and store the received message in  $\mathcal{R}_{\vec{e}_k}^e$ 
  endif
endif
endfor

```

Figure 5: Simplified Code of Minimum Message Passing

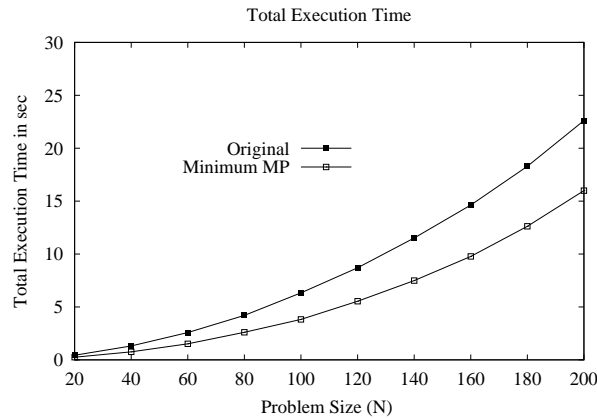
5 Experimental Evaluation

To evaluate the impact of minimizing message passing, we coded and run two versions of the 9-point Jacobi code for Poisson equation on a PC cluster of 16 workstations using MPI. The first version is the traditional parallel code as in Fig. 1. The PC cluster of 16 processors is partitioned to a 4×4 processor array. At the end of each iteration of the iterative loop, each processor sends and receives eight messages to and from its neighbor processors. To avoid the serialization of the message passing, we use the `MPI_SENDRCV()` function for the message passing.

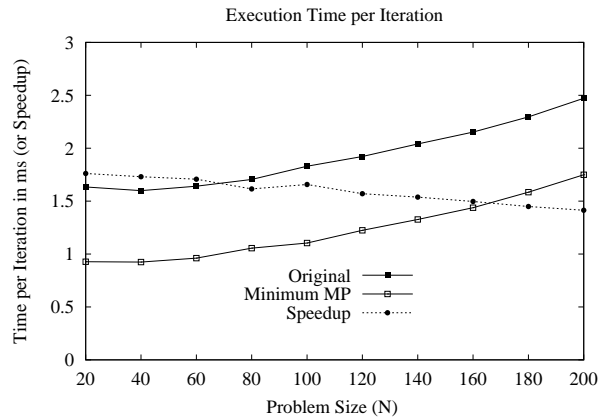
We removed the global reduction to find the maximum error across all processors and turned the **while** loop to a **for** loop with the known number of iterations of convergence. We vary the problem size N from 20 to 200.

The result of experimental evaluation is shown in Fig. 6. Fig. 6(a) shows the completion times in seconds towards the convergence. Fig. 6(b) shows the time of each iteration in micro-seconds. The result clearly shows that minimizing message passing does reduce the time for each iteration. The speedup obtained is between 1.760899 and 1.413513, which is also plotted in Fig. 6(b).

We haven't optimized the cache performance of the computation part yet. If the computation part in each iteration is further reduced by tuning the cache performance, the higher speedup (but less than $2.0=8/4$) would be expected.



(a) Completion Time



(b) Time per Iteration (Speedup)

Figure 6: Experiment Result

6 Related Work and Conclusion

Work on parallelizing iterative codes for PDE or PDE-based applications manually can trace back to the early work of the red/black algorithm by J. M. Ortega and R. G. Voight in 1980's [3]. As pointed by G. Fox et al. in [4], parallelizing the sequential Gauss-Seidel/SOR by partitioning the data space among parallel processors (as we formalized in the paper) would amount to a kind of hybrid between the Jacobi and Gauss-Seidel/SOR methods. Although, the result parallel algorithm is not *pure* Gauss-Seidel/SOR methods anymore, people have been taking this approach to solving PDEs in parallel machines all the time. The more recent work in parallel algorithm of PDE [5] focuses on overlapping the computation with message passing rather than minimizing the number of messages as we do in this paper.

The general idea of using message piggy-backing to reduce the number of messages in distributed-memory machines was first introduced in [6]. But, it did not provide the minimum message passing code.

Project CTADEL [7] is the most recent effort to build parallel code-generator for PDE-based applications. It did not mention any reduction in message passing.

We have shown a compiler transformation that can reduce the number of messages in the traditional parallel PDE codes from $3^n - 1$ to the minimum $2n$. The transformation can be implemented in parallelizing compilers or PDE compilers to generate efficient parallel codes with minimum message passing.

References

- [1] Louis W. Ehrlich. Iterative vs. a directive method for solving fourth order elliptic difference equations. In *Proceedings of the 1966 21st national conference*, January 1966.
- [2] Peiyi Tang. Formal methods to generate parallel iterative codes for PDE-based applications. In <http://titus.compsci.ualr.edu/~ptang/papers/formal.pdf>, 2004.
- [3] J. M. Ortega and R. G. Voight. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27:149–270, 1985.
- [4] Geoffrey C. Fox, Mark A. Johnson, et al. *Solving Problems on Concurrent Processors*, volume 1, General Techniques and Regular Problems. Prentice-Hall, 1988.
- [5] Dexuan Xie and Loyce Adams. New parallel SOR method by domain partitioning. *SIAM Journal on Scientific Computing*, 20(6):2261–2281, 1999.
- [6] Peiyi Tang and John N. Zigman. Reducing data communication overhead for doacross loop nests. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 44–54, Manchester, England, July 1994.
- [7] Robert van Engelen, Lex Wolters, and Gerard Cats. CTADEL: A generator of multi-platform high performance codes for pde-based applications. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 86–93, May 1996.