Formal Methods to Generate Parallel Iterative Codes for PDE-Based Applications

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204
ptang@titus.compsci.ualr.edu

Abstract

Developing parallel software is far more complex than traditional sequential software. An effective approach to deal with the complexity of parallel software is domain-specific programming in an abstraction higher than general-purpose programming languages. In this paper, we focus on the domain of the applications based on partial differential equations (PDE) and provide a formal framework and methods for PDE compilers to generate parallel iterative codes for the domain. We also provide a PDE compiler optimization to minimize the number of messages between parallel processors. Our framework and methods can be used to build PDE compilers to generate efficient parallel software for PDE-based applications automatically.

1. Introduction

One of the directions in which the complexity of software grows is the software for parallel computer systems. Developing parallel software is far more complex than traditional sequential software. Parallel programs are difficult to design, code and debug. There are several reasons behind it:

- Parallel programming is a significant paradigm shift from the traditional programming based on the Von Neumann machine model. Parallel programmers need to program multiple threads on multiple machines operating on multiple objects at the same time.
- Parallel programs share the resources of multiple machines whose access needs to be controlled through synchronization and communication. The share resource management, which is used to be handled by the operating system layer, now finds its way into parallel programs.
- Parallel programs are architecture dependent. Although parallel programming API standards such as MPI [1] and OPENMP [2, 3] helped ease the problem, using theses APIs still requires the understanding of the underlying architecture of the parallel machine.

One approach to deal with the complexity of parallel programming is to let parallelizing compilers to transform sequential programs to parallel programs. Despite the impressive research in parallelizing compilers in the last twenty years or so, commercial parallelizing compilers are still in their infancy.

In the history of computing, abstraction and thus, multi-layer approach, have always been the way to deal with the complexity of computer systems. The abstractions of computer architecture, operating system, programming language, object-oriented programming have served well in the past to reduce the complexity in developing computer applications. The difficulty of parallel programming seems to indicate that we need another layer of abstraction to cope with the complexity of parallel software. There have been efforts to raise the abstraction by adding new layers of (1) visual programming like Computationally Oriented Display Environment (CODE) [4] or (2) new parallel programming languages such as FORTRAN 90 and High-Performance FORTRAN. Both of them are *general-purpose* layers aimed to develop *any* kind of applications. In the real world, parallel computing applications can be divided to many *domains*. The applications in a domain usually share common structures of data and algorithms. For example, the core of PDE-based applications is the iterative loop to update a large array repeatedly, using finite differences to approximate partial differential equations (PDE). For each domain, the level of abstraction can be raised higher than parallel languages or visual programming. The resultant *domain-specific* language will be simple and easy to use. In addition, by taking advantage of the known structure of the computation, domain-specific compilers can optimize and generate effi-

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 IEEE

```
while maximum error ≥ ε do
  for j=1 to N do
    for i=1 to N do
      u_{i,j} ← 1/20 (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})
      update maximum error
    endfor
  endfor
endwhile
```

Figure 1. Iterative Code for 9-Point Poison Equation

cient and reliable parallel codes directly. We believe that the high-level domain-specific approach is a viable way to cope with the complexity of parallel software in the future.

In this paper, we focus on PDE-based applications and provide a formal framework and methods for PDE compilers to generate parallel iterative codes for the domain. The center of the framework is to implement the virtual large array space for the PDE using the local arrays in the local memory of parallel processors. We also provide a PDE compiler optimization to minimize the number of messages between parallel processors. Our framework and methods are the foundation for building PDE compilers to generate efficient parallel software for PDE-based applications automatically.

The rest of the paper is organized as follows. In Section 2, we describe our formal framework and methods to generate parallel iterative codes for PDE-based applications. In Section 3, we present a PDE compiler optimization to minimize the number of messages between parallel processors. In Section 4, we further simplify the minimum message passing parallel code. Section 5 discusses the related work. Section 6 concludes the paper.

2. Formal Framework to Generate Parallel Iterative Codes for PDE

In this section, we provide a formal framework and methods for a PDE compiler to generate the parallel iterative codes for distributed memory machines. We start with the program model of sequential iterative codes.

The data mesh of the linear system for a PDE is implemented with a large array. The iterative code basically changes the points of the array repeatedly until the maximum error becomes less than a predefined small value ϵ. To calculate the new value of a mesh point, we can use the old values of its nearby points calculated in the previous iteration as in the Jacobi method, or a mixture of old and new values of the nearby points as in the Gauss-Seidel or Successive Over-Relaxation (SOR) methods for faster convergence. Fig. 1 shows the Gauss-Seidel iterative code for the Poison equation using the 9-point stencil.

```
while maximum error ≥ ε
  for i_1 = 1, N_1
    ...
      for i_n = 1, N_n
        A[i⃗] =F(A[i⃗], A[i⃗ + s⃗_1], ···, A[i⃗ + s⃗_r])
        update the maximum error
      endfor
    ...
  endfor
endwhile
```

Figure 2. Program Model of Iterative Code

The general program model for the PDE iterative codes is shown in Fig. 2, where $\vec{N} = (N_1, \cdots, N_n)$ is the *size vector* of the problem[1] and $\vec{i} = (i_1, \cdots, i_n)$ the loop index vector such that $\vec{1} \leq \vec{i} \leq \vec{N}$. The outermost **while** loop is called the *iterative loop*. F is the function to calculate the new value of each mesh point. Vectors $\vec{s_1}, \cdots, \vec{s_r}$ are called *stencil vectors* and they can be derived directly from the stencil of the PDE. For example, the stencil vectors for the Poison equation in Fig. 1 are $(1, 0)$, $(-1, 0)$, $(0, 1)$, $(0, -1)$, $(1, 1)$, $(1, -1)$, $(-1, 1)$ and $(-1, -1)$. The stencil vectors are the relative coordinates of the nearby points to be used to calculate the new value of the current point. They should not be confused with the data dependence vectors in the literature of data dependence analysis. The set of the stencil vectors is denoted as $\mathcal{S} = \{\vec{s_1}, \cdots, \vec{s_q}\}$.

From the stencil vectors in $\mathcal{S}$, we derive two *boundary vectors*, $\vec{\delta^+}$ and $\vec{\delta^-}$, to quantify the boundary area of the data mesh. Vector $\vec{\delta^+} = (\delta_1^+, \cdots, \delta_n^+)$ is defined by

$$\delta_i^+ = \max\{s_i \mid (s_1, \cdots, s_i, \cdots, s_n) \in \mathcal{S} \wedge s_i \geq 0\} \quad (1)$$

for each $1 \leq i \leq n$. Likewise, vector $\vec{\delta^-} = (\delta_1^-, \cdots, \delta_n^-)$ is defined by

$$\delta_i^- = \max\{-s_i \mid (s_1, \cdots, s_i, \cdots, s_n) \in \mathcal{S} \wedge s_i \leq 0\} \quad (2)$$

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

$$-\vec{\delta^-} \leq \vec{s} \leq \vec{\delta^+} \quad (3)$$

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

$$-\vec{\delta^-} \leq \vec{0} \leq \vec{\delta^+} \quad (4)$$

[1]In this paper, the elements of any n-vector $\vec{x}$ are always denoted as $x_1, \cdots, x_n$. That is, $\vec{x} = (x_1, \cdots, x_n)$ is always implied.

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

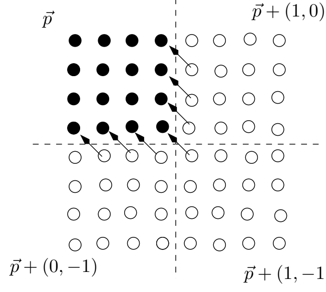

Figure 3. Receive directions for stencil vector $(1,-1)$

In the program model in Fig. 2, A is an n-dimensional array for the data mesh and we represent it as the set of array elements:

$$\mathcal{A} = \{A[\vec{a}] \mid \vec{1} - \vec{\delta^-} \leq \vec{a} \leq \vec{N} + \vec{\delta^+}\} \tag{5}$$

Its subset

$$\mathcal{A}_w = \{A[\vec{a}] \mid \vec{1} \leq \vec{a} \leq \vec{N}\} \tag{6}$$

gives the data points to be updated in each iteration of the iterative loop. The area $\mathcal{A} - \mathcal{A}_w$ is the boundary of the data mesh for the boundary values.

The purpose of using parallel computers for PDE is to solve the large problem with large data mesh. As the problem size vector $\vec{N}$ increases, the requirements of CPU cycle and memory increase in proportion with $\prod_{i=1}^{n} N_i$. Scalable distributed memory machines like Intel Paragon, Fujitsu AP3000, and IBM ASC Blue offer both high computation rate and large virtual memory space through a large number of processors.

A distributed-memory machine is modeled as an n-dimensional array of processors:

$$\mathcal{P} = \{\vec{p} \in Z^n \mid \vec{1} \leq \vec{p} \leq \vec{P}\} \tag{7}$$

where vector $\vec{P} = (P_1, \cdots, P_n)$ is the size vector of the processor array. To simplify the discussion, we assume that $\vec{P}$ divides $\vec{N}$, i.e. $N_i \bmod P_i = 0$ for each $1 \leq i \leq n$.

Each processor $\vec{p}$ in the processor array $\mathcal{P}$ has its own local memory to implement a part of the global virtual data mesh $\mathcal{A}$.

Each processor can send messages to its neighbor processors. Two processor $\vec{p_1}$ and $\vec{p_2}$ are *neighbors* if $|\vec{p_1} - \vec{p_2}| \leq \vec{1}$.

To implement the global virtual data mesh $\mathcal{A}$, each processor declares a small array

$$\mathcal{A}' = \{A'[\vec{a'}] \mid \vec{1} - \vec{\delta^-} \leq \vec{a'} \leq \vec{B} + \vec{\delta^+}\} \tag{8}$$

where $\vec{B} = \vec{N}/\vec{P}$. Its subset

$$\mathcal{A}'_w = \{A'[\vec{a'}] \mid \vec{1} \leq \vec{a'} \leq \vec{B}\} \tag{9}$$

is the area to be updated by the local processor and is called the *write* area. The $\mathcal{A}'_w$ of all processors in $\mathcal{P}$ collectively implement the global virtual data array $\mathcal{A}_w$. Given a processor $\vec{p}$ in $\mathcal{P}$, the relationship between the subscript $\vec{a'}$ of local array element $A'_w[\vec{a'}]$ and the subscript $\vec{a}$ of the corresponding element $A_w[\vec{a}]$ of the global virtual array is

$$\vec{a} = (\vec{p} - \vec{1}) \circ \vec{B} + \vec{a'} \tag{10}$$

where $\circ$ is the dot-product operator between two vectors. This formula can be used to output the final result for the global virtual array from the results of local arrays of the parallel processors.

The area of $\mathcal{A}' - \mathcal{A}'_w$ contains the data points which the local processor may need to read from its neighbor processors. This area is also called *ghost* area.

We now define the local-to-global mapping function $G : \mathcal{P} \times \mathcal{A}' \rightarrow \mathcal{A}$ as follows:

$$G(\vec{p}, A'[\vec{a'}]) = A[(\vec{p} - \vec{1}) \circ \vec{B} + \vec{a'}] \tag{11}$$

The values of the ghost area $A' - A'_w$ of a processor are computed by its neighbor processors. If the processor needs to read these values, they have to be sent from the neighbor processors. The directions from which to receive these values are called *receive directions*. They are determined by the stencil vectors in $\mathcal{S}$. Given a stencil vector $\vec{s}$, the set of receive directions derived from it is

$$\mathcal{D}_{r,\vec{s}} = \{\vec{d} \mid \vec{d} \in \mathcal{C}_1 \times \cdots \times \mathcal{C}_n \wedge \vec{d} \neq \vec{0}\} \tag{12}$$

where C_i for $i = 1, \cdots, n$ is defined by:

$$\mathcal{C}_i = \begin{cases} \{0, 1\} & \text{if } s_i > 0 \\ \{0\} & \text{if } s_i = 0 \\ \{0, -1\} & \text{if } s_i < 0 \end{cases}$$

For example, the receive direction set for the stencil vector $(1, -1)$, $\mathcal{D}_{r,(1,-1)}$, is $\{(1, 0), (0, -1), (1, -1)\}$. Fig. 3 shows the data partition among processors $\vec{p}$, $\vec{p} + (1, 0)$, $\vec{p} + (1, -1)$ and $\vec{p} + (0, -1)$, assuming $B_1 = B_2 = 4$. The $\mathcal{A}'_w$ data points of processor $\vec{p}$ are shown as filled circles, while those of its neighbor processors are shown as hollow circles. The arrows show that due to stencil vector $(1, -1)$ processor $\vec{p}$ needs to receive data values from its three neighbor processors at directions $(1, 0)$, $(1, -1)$ and $(0, -1)$.

The total set of receive directions, denoted as $\mathcal{D}_r$, is the union of the receive direction sets of all stencil vectors:

$$\mathcal{D}_r = \bigcup_{\vec{s} \in \mathcal{S}} \mathcal{D}_{r,\vec{s}} \tag{13}$$

The number of receive directions can be as large as $3^n - 1$, where n is the number of dimensions of the problem. For

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

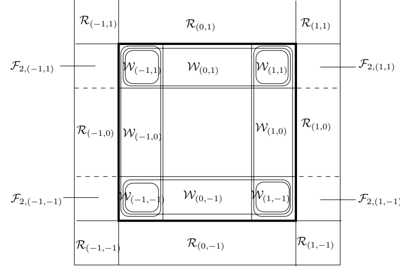

Figure 4. Remote Read and Write Sets and Forward Sets in $\mathcal{A}'_w$

the Poison equation in Fig. 1, $\mathcal{D}_r$ contains 8 receive directions, namely $(1,1)$,$(0,1)$, $(-1,1)$, $(1,0)$, $(-1,0)$, $(1,-1)$, $(0,-1)$, $(-1,-1)$.

Now, we define the remote receive sets in the local data array A' in each processor.

Definition 1 (Remote Read Set) *For each processor $\vec{p}$ and a receive direction vector $\vec{d} \in \mathcal{D}_r$, 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 $\mathcal{R}^{\vec{p}}_{\vec{d}}$. 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 $1 \leq i \leq n$.

If processor $\vec{p}$ needs to receive data from processor $\vec{p} + \vec{d}$ for $\mathcal{R}^{\vec{p}}_{\vec{d}}$, processor $\vec{p} + \vec{d}$ needs to send them in direction $-\vec{d}$. Therefore, the set of directions to send data for each processor, denoted as $\mathcal{D}_s$, can be defined as follows:

$$\mathcal{D}_s = \{-\vec{d} \mid \vec{d} \in \mathcal{D}_r\} \tag{14}$$

Definition 2 (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 $\mathcal{W}^{\vec{p}}_{\vec{d}}$. 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 $1 \leq i \leq n$.

```
let the current processor be p⃗
while maximum error ≥ ϵ
  for i_1 = 1, B_1
    ...
      for i_n = 1, B_n
        A'(i⃗) =F(A'(i⃗), A'(i⃗ + s⃗_1), ···, A'(i⃗ + s⃗_r))
        update the maximum error
      endfor
    ...
  endfor
  for each d⃗ ∈ D_s
    if p⃗ + d⃗ ∈ P then
      send message W_d⃗ to processor p⃗ + d⃗
    endif
    if p⃗ − d⃗ ∈ P then
      receive a message from processor p⃗ − d⃗
        and store it in R_−d⃗
    endif
  endfor
endwhile
```

Figure 5. SPMD Program of Parallel Iterative Code

It is obvious from Definition 2 that $\mathcal{W}_{\vec{d}} \subseteq A'_w$ holds for all $\vec{d} \in \mathcal{D}_s$.

Fig. 4 shows all the eight remote read sets and all the eight remote write sets for the two-dimensional problem. All the remote write sets are within $\mathcal{A}'_w$ enclosed in thick lines. Note that $\mathcal{W}_{(1,1)}$ are the subset of both $\mathcal{W}_{(1,0)}$ and $\mathcal{W}_{(0,1)}$.

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. The following lemma shows that they are mapped to the same region in the global virtual array. Let us extend the memory mapping function in (11) to apply to the remote read and write sets as follows:

$$G(\mathcal{R}^{\vec{p}}_{\vec{d}}) = \{G(\vec{p}, A'[\vec{a'}]) \mid A'[\vec{a'}] \in \mathcal{R}^{\vec{p}}_{\vec{d}}\} \tag{15}$$

and

$$G(\mathcal{W}^{\vec{p}}_{\vec{d}}) = \{G(\vec{p}, A'[\vec{a'}]) \mid A'[\vec{a'}] \in \mathcal{W}^{\vec{p}}_{\vec{d}}\} \tag{16}$$

Given local sets $\mathcal{X}$ and $\mathcal{Y}$, we say $\mathcal{X} \stackrel{G}{=} \mathcal{Y}$ if $G(\mathcal{X}) = G(\mathcal{Y})$.

Lemma 1 *Given a processor $\vec{p} \in \mathcal{P}$ and a send direction vector $\vec{d} \in \mathcal{D}_s$,*

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

is true if $\vec{p} + \vec{d} \in \mathcal{P}$.

0-7695-2284-X/05 $20.00 © 2005 IEEE

```
if $\vec{p}+(1,0) \in \mathcal{P}$ then
  send message $\mathcal{W}_{(1,0)}$ to processor $\vec{p}+(1,0)$
endif
if $\vec{p}-(1,0) \in \mathcal{P}$ then
  receive a message from processor $\vec{p}-(1,0)$
    and store it in $\mathcal{R}_{(-1,0)}$
endif
```

(a) Message Passing in Direction $(1,0)$

```
if $\vec{p}+(-1,0) \in \mathcal{P}$ then
  send message $\mathcal{W}_{(-1,0)}$ to processor $\vec{p}+(-1,0)$
endif
if $\vec{p}-(-1,0) \in \mathcal{P}$ then
  receive a message from processor $\vec{p}-(-1,0)$
    and store it in $\mathcal{R}_{(1,0)}$
```

(b) Message Passing in Direction $(-1,0)$

```
if $\vec{p}+(0,1) \in \mathcal{P}$ then
  send message $\mathcal{W}_{(0,1)} \cup \mathcal{F}_{2,(-1,1)} \cup \mathcal{F}_{2,(1,1)}$
    to processor $\vec{p}+(0,1)$
endif
if $\vec{p}-(0,1) \in \mathcal{P}$ then
  receive a message from processor $\vec{p}-(0,1)$
    and store it in $\mathcal{R}_{(0,-1)} \cup \mathcal{R}_{(-1,-1)} \cup \mathcal{R}_{(1,-1)}$
endif
```

(c) Message Passing in Direction $(0,1)$

```
if $\vec{p}+(0,-1) \in \mathcal{P}$ then
  send message $\mathcal{W}_{(0,-1)} \cup \mathcal{F}_{2,(-1,-1)} \cup \mathcal{F}_{2,(1,-1)}$
    to processor $\vec{p}+(0,-1)$
endif
if $\vec{p}-(0,-1) \in \mathcal{P}$ then
  receive a message from processor $\vec{p}-(0,-1)$
    and store it in $\mathcal{R}_{(0,1)} \cup \mathcal{R}_{(-1,1)} \cup \mathcal{R}_{(1,1)}$
endif
```

(d) Message Passing in Direction $(0,-1)$

Figure 6. Example of Minimized Message Passing

To save the space, the proofs of all the lemmas in this paper are omitted. They can be found in [5].

Although $\mathcal{W}_{\vec{d}}^{\vec{p}}$ and $\mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ are mapped to the same data points in the global virtual array, 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

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

The parallel iterative code to run on each processor $\vec{p} \in \mathcal{P}$ is shown in Fig. 5. This is a SPMD (Simple Program Multiple Data) parallel program and every processor runs the same program with a distinct processor identity vector $\vec{p}$. After the nested **for** loop completes the computation of new values for A'_w, every processor sends the message in each direction $\vec{d} \in \mathcal{D}_s$ (except for the direction such that $\vec{p}+\vec{d}$ does not exist) to make $\mathcal{W}_{\vec{d}}^{\vec{p}} \overset{\vec{d}}{\equiv} \mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ true. After all the messages are sent and received, we will have $\mathcal{W}_{\vec{d}}^{\vec{p}} \overset{\vec{d}}{\equiv} \mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ for all $\vec{d} \in \mathcal{D}_s$ and all $\vec{p}, \vec{p}+\vec{d} \in \mathcal{P}$. Therefore, when the new iteration of the iterative loop starts, each processor has the valid values in all its remote read sets $\mathcal{R}_{\vec{d}}$ ($\vec{d} \in \mathcal{D}_r$) it needs.

3. Minimizing Message Passing

The number of messages passed the parallel iterative code in Fig. 5 is determined by the number of direction vectors in $\mathcal{D}_s$. For the n-dimensional PDE problem, this number can be as large as $3^n - 1$. In this section, we will show a transformation to reduce the number of messages from $3^n - 1$ to $2n$ and $\mathcal{W}_{\vec{d}}^{\vec{p}} \overset{\vec{d}}{\equiv} \mathcal{R}_{-\vec{d}}^{\vec{p}+\vec{d}}$ still hold for all $\vec{d} \in \mathcal{D}_s$ and all $\vec{p}, \vec{p}+\vec{d} \in \mathcal{P}$ at the end of every iteration.

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 basic idea to reduce the number of messages in the parallel iterative codes is to piggy-back the non-elementary messages onto the elementary ones so that non-elementary messages no longer need to be sent directly. There are maximally only $2n$ elementary messages.

Let us use the two-dimensional iterative code to illustrate the idea of piggy-backing non-elementary messages. Suppose that the remote write sets need to be sent in all eight directions, i.e. $\mathcal{D}_s = \{\vec{d} \mid |\vec{d}| \leq \vec{1} \wedge \vec{d} \neq \vec{0}\}$. Fig. 4 shows the eight remote write sets and the eight remote read sets in every processor. The message passing starts with sending the message in the elementary direction $(1,0)$ of the first dimension. Its code is shown in Fig. 6(a). Note

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

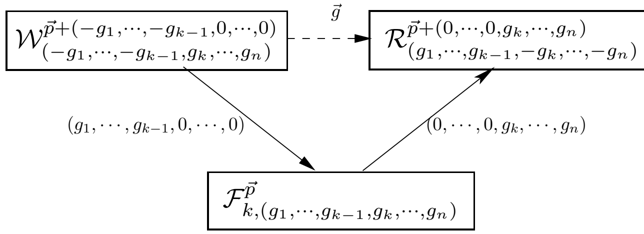

Figure 7. Rationale of Forward Set

let the current processor be $\vec{p}$
while maximum error $\geq \epsilon$
  for i_1 = 1, B_1
    ...
      for i_n = 1, B_n
        $A'(\vec{i}) = F(A'(\vec{i}), A'(\vec{i}+\vec{s_1}), \cdots, A'(\vec{i}+\vec{s_r}))$
        update the maximum error
      endfor
    ...
  endfor
  for $k = 1, n$
    if $\vec{p}+\vec{e_k} \in \mathcal{P} \wedge \vec{e_k} \in \mathcal{D}_s$ **then**
      send message
        $\mathcal{W}_{\vec{e_k}} \cup \bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},1,0,\cdots,0)}$
        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 $(g_1,\cdots,g_{k-1}) \neq \vec{0}$
          store the received $\mathcal{F}_{k,(g_1,\cdots,g_{k-1},1,0,\cdots,0)}$
            in $\mathcal{R}_{(g_1,\cdots,g_{k-1},-1,0,\cdots,0)}$
        endfor
    endif

    if $\vec{p}-\vec{e_k} \in \mathcal{P} \wedge -\vec{e_k} \in \mathcal{D}_s$ **then**
      send message
        $\mathcal{W}_{-\vec{e_k}} \cup \bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},-1,0,\cdots,0)}$
        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 $(g_1,\cdots,g_{k-1}) \neq \vec{0}$
          store the received $\mathcal{F}_{k,(g_1,\cdots,g_{k-1},-1,0,\cdots,0)}$
            in $\mathcal{R}_{(g_1,\cdots,g_{k-1},1,0,\cdots,0)}$
        endfor
    endif
  endfor
endwhile

Figure 8. Parallel Iterative Code with Minimum Message Passing

that both $\mathcal{W}_{(1,1)}$ and $\mathcal{W}_{(1,-1)}$ are subsets of $\mathcal{W}_{(1,0)}$. After $\mathcal{W}_{(1,0)}$ of processor $\vec{p}$ is sent and received by processor $\vec{p}+(1,0)$, $\mathcal{W}^{\vec{p}}_{(1,1)}$ and $\mathcal{W}^{\vec{p}}_{(1,-1)}$ are already in $\mathcal{R}_{(-1,0)}$ of processor $\vec{p}+(1,0)$, i.e. $\mathcal{R}^{\vec{p}+(1,0)}_{(-1,0)}$. The areas in $\mathcal{R}_{(-1,0)}$ of processor $\vec{p}+(1,0)$ which have the values of $\mathcal{W}_{(1,1)}$ and $\mathcal{W}_{(1,-1)}$ of processor $\vec{p}$ are called forward sets and denoted as $\mathcal{F}_{2,(-1,1)}$ and $\mathcal{F}_{2,(-1,-1)}$ in Fig. 4, respectively. Next, each processor sends its $\mathcal{W}_{(-1,0)}$ along another elementary direction $(-1,0)$ of the first dimension as shown in Fig. 6(b). In particular, after processor $\vec{p}+(1,0)$ receives $\mathcal{W}_{(-1,0)}$ from processor $\vec{p}+(2,0)$, the values of $\mathcal{W}^{\vec{p}+(2,0)}_{(-1,1)}$ and $\mathcal{W}^{\vec{p}+(2,0)}_{(-1,-1)}$ are already in the forward sets $\mathcal{F}_{2,(1,1)}$ and $\mathcal{F}_{2,(1,-1)}$ of processor $\vec{p}+(1,0)$, respectively, as parts of its $\mathcal{R}_{(1,0)}$ (see Fig. 4).

Then each processor sends messages along the elementary directions of the second dimension, namely, $(0,1)$ and $(0,-1)$. When sending $\mathcal{W}_{(0,1)}$ along direction $(0,1)$, each processor piggy-backs its $\mathcal{F}_{2,(-1,1)}$ and $\mathcal{F}_{2,(1,1)}$ to $\mathcal{W}_{(0,1)}$ and send the combined message along the direction $(0,1)$ as shown in Fig. 6(c). In particular, processor $\vec{p}+(1,0)$ sends its $\mathcal{W}_{(0,1)}$ as well as its $\mathcal{F}_{2,(-1,1)}$ and $\mathcal{F}_{2,(1,1)}$ (see Fig. 4 for these areas) in one message to processor $\vec{p}+(1,0)+(0,1)$. When processor $\vec{p}+(1,0)+(0,1)$ receives the message, it stores the received $\mathcal{W}_{(0,1)}$ in its $\mathcal{R}_{(0,-1)}$ and the received $\mathcal{F}_{2,(-1,1)}$ and $\mathcal{F}_{2,(1,1)}$ in its $\mathcal{R}_{(-1,-1)}$ and $\mathcal{R}_{(1,-1)}$, respectively. Recall that $\mathcal{F}_{2,(-1,1)}$ carries the values of $\mathcal{W}_{(1,1)}$ from processor $\vec{p}$. Therefore, $\mathcal{W}^{\vec{p}}_{(1,1)}$ has been indirectly passed and stored in $\mathcal{R}^{\vec{p}+(1,1)}_{(-1,-1)}$. This is exactly what the message passing of processor $\vec{p}$ in the non-elementary direction $(1,1)$ is supposed to do. The second message passing in the second dimension is shown in Fig. 6(d).

After all the four messages as above are passed, all the non-elementary messages of every processor have been indirectly forwarded and stored in the corresponding remote read sets of the destination processors.

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:

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

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

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

As usual, we use $\mathcal{F}^{\vec{p}}_{k,\vec{g}}$ to denote the $\mathcal{F}_{k,\vec{g}}$ of processor $\vec{p}$. The definition of $\mathcal{F}^{\vec{p}}_{k,\vec{g}}$ 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 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 $\mathcal{F}^{\vec{p}}_{k,\vec{g}}$ is the storage to store $\mathcal{W}^{\vec{p}+(-g_1,\cdots,-g_{k-1},0,\cdots,0)}_{(-g_1,\cdots,-g_{k-1},g_k,\cdots,g_n)}$ originated in processor $\vec{p}+(-g_1,\cdots,-g_{k-1},0,\cdots,0)$. And it will be forwarded further to processor $\vec{p}+(0,\cdots,0,g_k,\cdots,g_n)$ and stored in $\mathcal{R}^{\vec{p}+(0,\cdots,0,g_k,\cdots g_n)}_{(g_1,\cdots,g_{k-1},-g_k,\cdots,-g_n)}$. This process of message forwarding is illustrated by the solid arrows in Fig. 7. The direct message passing in direction $\vec{g} = (g_1,\cdots,g_{k-1},g_k,\cdots,g_n)$ for $\mathcal{W}^{\vec{p}+(-g_1,\cdots,-g_{k-1},0,\cdots,0)}_{(-g_1,\cdots,-g_{k-1},g_k,\cdots,g_n)}$ to $\mathcal{R}^{\vec{p}+(0,\cdots,0,g_k,\cdots g_n)}_{(g_1,\cdots,g_{k-1},-g_k,\cdots,-g_n)}$ illustrated by the dash arrow in Fig. 7 will not be necessary anymore.

The parallel iterative code with the reduced message passing is shown in Fig. 8, 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 $\vec{e_1}, -\vec{e_1}, \cdots, \vec{e_n}, -\vec{e_n}$. 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, $\bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},1,0,\cdots,0)}$ or $\bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},-1,0,\cdots,0)}$, respectively. When $k = 1$, both $\bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},1,0,\cdots,0)}$ and $\bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},-1,0,\cdots,0)}$ are empty, because the $(g_1,\cdots,g_{k-1}) \neq \vec{0}$ do not exist.

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

Lemma 2 *Given a processor $\vec{p}$, integer $1 \leq k \leq n$ and vector $(g_1,\cdots,g_k,0,\cdots,0)$ such that $g_k \neq 0$,*

$$\mathcal{F}^{\vec{p}}_{k,(g_1,\cdots,g_{k-1},g_k,0,\cdots,0)} \stackrel{G}{\equiv} \mathcal{R}^{\vec{p}+(0,\cdots,0,g_k,0,\cdots,0)}_{(g_1,\cdots,g_{k-1},-g_k,0,\cdots,0)}$$

holds.

The next theorem establishes the correctness of parallel iterative code in Fig. 8. That is, at the end of every iteration of the iterative loop, $\mathcal{W}^{\vec{p}}_{\vec{d}} \equiv \mathcal{R}^{\vec{p}+\vec{d}}_{-\vec{d}}$ 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 $\vec{d} = (\cdots,d_{i_1},\cdots,d_{i_2},\cdots,d_{i_k},\cdots)$, where d_{i_j} ($j = 1,\cdots,k$) are the only k ($2 \leq k \leq n$) non-zero elements of $\vec{d}$ and $i_1 < \cdots < i_k$, after all the elementary messages in Fig. 8 are passed, the following assertions are true:*

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

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

Proof: See the Appendix of this paper. □

According to the definitions of direction vectors in $\mathcal{D}_s$, $\mathcal{D}_r$ and $\mathcal{D}_{r,\vec{s}}$ (see (14), (13) and (12)), if $\vec{d} = (\cdots,d_{i_1},\cdots,d_{i_2},\cdots,d_{i_k},\cdots)$ is in $\mathcal{D}_s$, the elementary send directions $\vec{d_{i_j}}$ ($j = 1,\cdots,k$) are also in $\mathcal{D}_s$. Therefore, each processor will send messages in the directions $\vec{d_{i_j}}$ in the order of $j = 1,\cdots,k$. Theorem 1 says that the non-elementary message $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_2},\cdots,d_{i_k},\cdots)}$ will be forwarded by the elementary messages in directions $\vec{d_{i_j}}$ ($j = 1,\cdots,k$) and stored in the corresponding forward sets in processors $\vec{p}+\vec{d_{i_1}}$, $\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}$, $\cdots$, $\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}+\cdots+\vec{d_{i_{k-1}}}$. It will eventually reach processor $\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}+\cdots+\vec{d_{i_{k-1}}}+\vec{d_{i_k}}$ and be stored in $\mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}+\cdots+\vec{d_{i_{k-1}}}+\vec{d_{i_k}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_{k-1}},\cdots,-d_{i_k},\cdots)}$.

In other words, after all the messages in the code in Fig. 8 are finished by all processors, we have $\mathcal{W}^{\vec{p}}_{\vec{d}} \equiv \mathcal{R}^{\vec{p}+\vec{d}}_{-\vec{d}}$ 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. 8 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 destination. This is because every non-elementory 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. 8 is essential and cannot be piggy-backed and forwarded.

4. Simplify Minimum Message Passing Parallel Iterative Codes

In this section, we further simplify the parallel iterative code in Fig. 8 by combining the multiple sets of a message to one set to minimize message packing and unpacking.

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

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* $(0, \cdots, 0, d_k, 0, \cdots, 0)$*, its extended remote write set, denoted as* $\mathcal{W}^e_{(0,\cdots,0,d_k,0,\cdots,0)}$*, is*

$$\mathcal{W}^e_{(0,\cdots,0,d_k,0,\cdots,0)} = \{A'[\vec{i}] \mid \vec{l} \leq \vec{i} \leq \vec{u}\}$$

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

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

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

The following lemma shows that the extended remote write set $\mathcal{W}^e_{(0,\cdots,0,d_k,0,\cdots,0)}$ is exactly what needs to be sent in elementary direction $(0, \cdots, 0, d_k, 0, \cdots, 0)$ in the code in Fig. 8.

Lemma 3 *Given an elementary send direction in the k-th dimension* $(0, \cdots, 0, d_k, 0, \cdots, 0)$*,* $\mathcal{W}^e_{(0,\cdots,0,d_k,0,\cdots,0)} = \mathcal{W}_{(0,\cdots,0,d_k,0,\cdots,0)} \cup \bigcup_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{F}_{k,(g_1,\cdots,g_{k-1},d_k,0,\cdots,0)}$ *is true.*

Therefore, the messages to be sent in the code in Fig. 8 can be simplified to $\mathcal{W}^e_{(0,\cdots,0,d_k,0,\cdots,0)}$ as one set and no message packing[2] is needed.

Similarly, we can combine all the remote read sets in the message receive code in Fig. 8 into an *extended* remote read set defined as follows to save message unpacking[3].

Definition 5 (Extended Remote Read Set) *Given an elementary receive direction in the k-th dimension* $(0, \cdots, 0, d_k, 0, \cdots, 0)$*, its extended remote read set, denoted as* $\mathcal{R}^e_{(0,\cdots,0,d_k,0,\cdots,0)}$*, is*

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

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

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

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

The following lemma shows that the extended remote read set $\mathcal{R}^e_{(0,\cdots,0,d_k,0,\cdots,0)}$ is exactly where to store the message received from elementary direction $(0, \cdots, 0, d_k, 0, \cdots, 0)$ in the code in Fig. 8.

[2]Message packing is an operation to copy various data sets to a message before sending.

[3]Message unpacking is the reverse of message packing.

Lemma 4 *Given an elementary receive direction in the k-th dimension* $(0, \cdots, 0, d_k, 0, \cdots, 0)$*,* $\mathcal{R}^e_{(0,\cdots,0,d_k,0,\cdots,0)} = \mathcal{R}_{(0,\cdots,0,d_k,0,\cdots,0)} + \sum_{(g_1,\cdots,g_{k-1})\neq\vec{0}} \mathcal{R}_{(g_1,\cdots,g_{k-1},d_k,0,\cdots,0)}$ *is true.*

With the definitions of extended remote write and read sets, the parallel iterative code in Fig. 8 can be simplified to the code shown in Fig. 9. According to Lemma 3 and Lemma 4, the simplified parallel code in Fig. 9 is equivalent to that in Fig. 8.

```
let the current processor be p⃗
while maximum error ≥ ε
  for i_1 = 1, B_1
    ...
      for i_n = 1, B_n
        A'(i⃗) = F(A'(i⃗), A'(i⃗ + s⃗_1), ···, A'(i⃗ + s⃗_r))
        update the maximum error
      endfor
    ...
  endfor
  for k = 1, n
    if p⃗ + e⃗_k ∈ 𝒫 ∧ e⃗_k ∈ 𝒟_s then
      send message
        𝒲^e_{e⃗_k} to processor p⃗ + e⃗_k
    endif
    if p⃗ − e⃗_k ∈ 𝒫 ∧ e⃗_k ∈ 𝒟_s then
      receive a message from processor p⃗ − e⃗_k
        and store the received message in ℛ^e_{−e⃗_k}
    endif

    if p⃗ − e⃗_k ∈ 𝒫 ∧ −e⃗_k ∈ 𝒟_s then
      send message 𝒲^e_{−e⃗_k} to processor p⃗ − e⃗_k
    endif
    if p⃗ + e⃗_k ∈ 𝒫 ∧ −e⃗_k ∈ 𝒟_s then
      receive a message from processor p⃗ + e⃗_k
        and store the received message in ℛ^e_{e⃗_k}
    endif
  endfor
endwhile
```

Figure 9. Simplified Parallel Iterative Code with Minimum Message Passing

We have done prelimary experiemnts to verify the impact of minimizing message passing on parallel execution times. We manually implemented the Poison equation of the 9-point stencil using both codes in Fig. 5 and Fig. 9 and run them on a PC cluster with 4×4 processors. The execution times per iteration in milli-seconds are shown as follows:

size(N)	40	80	120	160	200
Fig. 5	1.600	1.705	1.922	2.152	2.472
Fig. 9	0.924	1.056	1.224	1.438	1.749

0-7695-2284-X/05 $20.00 © 2005 IEEE

which gives the speed-ups between 1.731 and 1.416.

5. Related Work

The work on manually parallelizing the iterative codes for PDE or PDE-based applications can trace back to the early work of the red/black algorithm by J. M. Ortega and R. G. Voight in 1980's [6]. As pointed by G. Fox at. el. in [7], parallelizing the sequential Gauss-Seidel/SOR by partitioning the data space among parallel processors would amount to a kind of hybrid between the Jacobi and Gauss-Seidel/SOR methods. Although, the resultant 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 [8] focuses on overlapping the computation with message passing rather than minimizing the number of messages. All the efforts above are for manual coding of the parallel iterative codes rather than automatic parallel code generation.

The general idea of using message piggy-backing to reduce the number of messages in distributed-memory machines was first introduced in [9], but no formal algorithm and methods were given.

Project CTADEL [10] is the most recent effort to build a PDE compiler for PDE-based applications. It did not provide its formal framework and methods, nor did it mention any message passing optimization.

6. Conclusion

We have shown in this paper a formal framework and methods to generate efficient parallel iterative codes for the domain of PDE-based applications. We have also shown a message passing optimization which can reduce the number of messages from as large as $3^n - 1$ to the minimum $2n$ for the high-order PDE or the second-order PDE using high-order finite differences. The formal framework and methods in this paper laid the foundation for building PDE compilers to generate efficient PDE parallel codes automatically.

References

[1] Mark Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. http://www.openmp.org/drupal/node/view/8, 2004.

[3] Rohit Chandra, Leonardo Dagun, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[4] James C. Brown, Syed I. Hyder, Jack Dongara, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, 3(1):75–83, 1995.

[5] Peiyi Tang. Formal methods to generate parallel iterative codes for PDE-based applications. In http://titus.compsci.ualr.edu/~ptang/papers/formal.pdf, 2004.

[6] J. M. Ortega and R. G. Voight. Solution of partial differentical equations on vector and parallel computers. *SIAM Review*, 27:149–270, 1985.

[7] Geoffrey C. Fox, Mark A. Johnson, et al. *Solving Problems on Concurrent Processors*, volume 1, General Techniques and Regular Problems. Prentice-Hall, 1988.

[8] Dexuan Xie and Loyce Adams. New parallel SOR method by domain partitioning. *SIAM Journal on Scientific Computing*, 20(6):2261–2281, 1999.

[9] 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.

[10] 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.

Appendix (Proof of Theorem 1)

In this Appendix, we provide the proof of Theorem 1. First we need to introduce a few lemmas. Their proofs are omitted and can be found in [5].

Lemma 5 *Given two send direction vectors $\vec{d}$ and $\vec{d}'$, $\mathcal{W}_{\vec{d}} \subset \mathcal{W}_{\vec{d}'}$ is true if $d_j \neq 0 \wedge d'_j = 0$ and $d_k = d'_k$ for all $k \neq j$.*

Lemma 6 *Given a $\vec{p} \in \mathcal{P}$, a $\vec{g}$ such that $|\vec{g}| \leq \vec{1}$ and any $1 \leq k \leq n$,*

$$\mathcal{W}^{\vec{p}}_{(g_1,\cdots,g_k,\cdots,g_n)} \stackrel{G}{=} \mathcal{F}^{\vec{p}+(g_1,\cdots,g_{k-1},0,\cdots,0)}_{k,(-g_1,\cdots,-g_{k-1},g_k,\cdots,g_n)}$$

is true if $\vec{p} + (g_1, \cdots, g_{k-1}, 0, \cdots, 0) \in \mathcal{P}$.

0-7695-2284-X/05 $20.00 © 2005 **IEEE**

Lemma 7 *Given a $\vec{p} \in \mathcal{P}$, a $\vec{g}$ such that $|\vec{g}| \leq \vec{1}$ and any $1 \leq k < n$,*

$$\mathcal{F}^{\vec{p}}_{k,(g_1,\cdots,g_k,\cdots,g_n)} \stackrel{G}{\equiv} \mathcal{F}^{\vec{p}+(0,\cdots,0,g_k,0,\cdots,0)}_{k+1,(g_1,\cdots,g_{k-1},-g_k,g_{k+1},\cdots,g_n)}$$

is true if $\vec{p} + (0, \cdots, 0, g_k, 0, \cdots, 0) \in \mathcal{P}$

Now we prove Theorem 1 by using mathematical induction. Consider the base case of $k = 2$ first. We have $\vec{d} = (\cdots, d_{i_1}, \cdots, d_{i_2}, \cdots)$. In the following, dots $\cdots$ between d_{i_j} (or $-d_{i_j}$) and $d_{i_{j+1}}$ (or $-d_{i_{j+1}}$) in vectors always represent contiguous 0s. According to the parallel code in Fig. 8, $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots)}$ is sent to processor $\vec{p} + \vec{d_{i_1}}$ and stored in $\mathcal{R}^{\vec{p}+\vec{d_{i_1}}}_{(\cdots,-d_{i_1},\cdots)}$. Thus, we have

$$\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots)} \stackrel{\vec{d_{i_1}}}{\equiv} \mathcal{R}^{\vec{p}+\vec{d_{i_1}}}_{(\cdots,-d_{i_1},\cdots)}$$

Since $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_2},\cdots)}$ is a subset of $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots)}$ according to Lemma 5, it is already in processor $\vec{p} + \vec{d_{i_1}}$. According to Lemma 6, it must have been stored in $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$ and we have

$$\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_2},\cdots)} \stackrel{\vec{d_{i_1}}}{\equiv} \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$$

According to Lemma 7, $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$ is the same as $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$ because the $(i_1 + 1)$-th element of $(\cdots, -d_{i_1}, \cdots, d_{i_2}, \cdots)$ is 0. Thus, we have

$$\begin{aligned} & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)} \\ \equiv\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)} \\ \equiv\ & \cdots \\ \equiv\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)} \end{aligned}$$

because $d_j = 0$ for all $i_1 < j < i_2$. According to the parallel code in Fig. 8, $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$ is then piggy-backed onto $\mathcal{W}^{\vec{p}+\vec{d_{i_1}}}_{(\cdots,d_{i_2},\cdots)}$ and sent to processor $\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}$, because it is one of the forward sets in the union due to $d_{i_1} \neq 0$. $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)}$ is stored in $\mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_2},\cdots)}$ after it is received and we have

$$\mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_2,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots)} \stackrel{\vec{d_{i_2}}}{\equiv} \mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\vec{d_{i_2}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_2},\cdots)}$$

Now consider the general cases of $k > 2$. According to the inductive assumption, we already have

$$\begin{aligned} & \mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_{k-1}},\cdots)} \\ \stackrel{\vec{d_{i_1}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots,d_{i_{k-1}},\cdots)} \\ \equiv\ & \cdots \\ \stackrel{\vec{d_{i_{k-2}}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-2}}}}_{i_{k-2}+1,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,d_{i_{k-1}},\cdots)} \\ \stackrel{\vec{d_{i_{k-1}}}}{\equiv}\ & \mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots)} \end{aligned}$$

after the message passing in the first $k - 1$ elementary directions, $\vec{d_{i_1}}, \cdots, \vec{d_{i_{k-1}}}$. According to Lemma 5, $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_{k-1}},\cdots,d_{i_k},\cdots)}$ is a subset of $\mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_{k-1}},\cdots)}$. Therefore, it is already in each of the sets shown above. That is, we have

$$\begin{aligned} & \mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \stackrel{\vec{d_{i_1}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots,d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \equiv\ & \cdots \\ \stackrel{\vec{d_{i_{k-1}}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{i_{k-1}+1,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \end{aligned}$$

Using the same analysis in the base case, we have

$$\begin{aligned} & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{i_{k-1}+1,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \equiv\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{i_k,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \end{aligned}$$

In the last message passing in direction $\vec{d_{i_k}}$, $\mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{i_k,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,d_{i_k},\cdots)}$ is piggy-backed to $\mathcal{W}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{(\cdots,d_{i_k},\cdots)}$, sent to processor $\vec{p} + \vec{d_{i_1}} + \cdots + \vec{d_{i_{k-1}}} + \vec{d_{i_k}}$ and stored in $\mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}+\vec{d_{i_k}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,-d_{i_k},\cdots)}$. Now we have

$$\begin{aligned} & \mathcal{W}^{\vec{p}}_{(\cdots,d_{i_1},\cdots,d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \stackrel{\vec{d_{i_1}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}}_{i_1+1,(\cdots,-d_{i_1},\cdots,d_{i_2},\cdots,d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \equiv\ & \cdots \\ \stackrel{\vec{d_{i_{k-1}}}}{\equiv}\ & \mathcal{F}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}}_{i_{k-1}+1,(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,d_{i_k},\cdots)} \\ \stackrel{\vec{d_{i_k}}}{\equiv}\ & \mathcal{R}^{\vec{p}+\vec{d_{i_1}}+\cdots+\vec{d_{i_{k-1}}}+\vec{d_{i_k}}}_{(\cdots,-d_{i_1},\cdots,-d_{i_{k-2}},\cdots,-d_{i_{k-1}},\cdots,-d_{i_k},\cdots)} \end{aligned}$$

We have completed the induction step from $k - 1$ to k. That proves the theorem.

0-7695-2284-X/05 $20.00 © 2005 **IEEE**