The handle http://hdl.handle.net/1887/18622 holds various files of this Leiden University dissertation.

**Author:** Vu, Van Thieu  
**Title:** Opportunities for performance optimization of applications through code generation  
**Issue Date:** 2012-03-22
Chapter 4

Automatic code generation of SPMD parallel programs

Previous studies have shown that CTADEL can generate highly optimized serial code for several schemes in a weather forecast model. In this chapter we show how to extend CTADEL so that it can automatically generate SPMD parallel programs. We applied this technique to generate parallel code for the Shallow-Water equations which have been discussed in Chapter 2 and performed experiments on the DAS-3 system [67].

4.1 Parallel programming models

Many parallel programming models have been developed. In general, popular models which are used in many parallel programs include Master-Slave, Data Pipelining, Divide and Conquer, and Single Program Multiple Data (SPMD). These models are presented in detail in [11]. In the following paragraphs, we give a brief description of each.

Master-Slave

The master-slave model consists of a master process and multiple slave processes. The master decomposes the problem into small tasks, distributes these tasks to slave processes, and gathers and combines the results from slave processes to a final result. The slave receives a task, executes it, and sends the result to the master. The required communication is only between the master and the slaves. This model is suitable for applications where a sequential algorithm can be executed simultaneously and independently on different processes, each with different input data. This model is also suitable if there are several different algorithms, perhaps even all operating on the same input data.
Data Pipelining

In the data pipelining model, a number of processes form a virtual pipeline. A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion. This model is usually applied in data reduction or image processing applications.

Divide and Conquer

In the divide and conquer approach, a problem is split into several subproblems, where each is solved independently and their results are combined to give the final result. Different from the master-slave approach, where the communication is needed between the master and the slaves, the divide and conquer method does not require any communication between the processes because the subproblems in this approach are independent.

Single Program Multiple Data

The Single Program Multiple Data model (SPMD) is the most widely used method [4]. Within a SPMD program, each process executes the same code but on a different part of data. This involves the splitting of application data among the available processes. This type of parallelism is also referred to as geometric parallelism, domain decomposition, or data parallelism. SPMD applications can be very efficient if data are well distributed over processes and the problem is homogeneous.

The choice of a parallel model depends on the type of parallelism inherent in the problem, which reflects the structure of either the application or its data. We chose the SPMD parallel programming model as base for our parallel code generation, because of the following reasons:

- SPMD is widely used for the parallelization of scientific models [4].
- SPMD works particularly well for problems which can be partitioned into static chunks which interact only with their nearest neighbors. Atmospheric models, for example, tend to be suitable for this type of parallelization [5].
- Because data is split among the processes, the SPMD model is well-suited for platform that has a distributed memory. Hence, a grid platform is also suitable for the SPMD model.
4.2. Implementation

A parallel program using the SPMD model consists of four steps: (1) domain decomposition; (2) data distribution; (3) calculations and communications; and (4) results collection. In this section, we will show how to adapt CTADEL to generate codes for these steps. We specify each step by templates.

Currently, CTADEL generates Fortran code. As communication primitives, we use MPI [76]. In the next subsections we will use MPI primitives to define templates that generate the communication code. We can easily change to use other communication methods by adapting these templates to use target communication primitives.

4.2.1 Domain decomposition

Domain decomposition refers to splitting a global computational domain into local processor domains or subdomains. Currently we only decompose in the two horizontal directions, not in the vertical one. We chose a two-dimensional decomposition because in the most atmospheric models, the horizontal grid is larger compared to the vertical grid component. This allows a higher degree of parallelization in the horizontal than in the vertical direction. Another important consideration is that in Numerical Weather models, most data dependency is column-wise. As a result, the number of horizontal communications is far smaller than that in the vertical [12]. Hence, the decomposition in the horizontal is more efficient than in the vertical direction.

In the rest of this chapter, all codes are assumed for the two-dimensional problem. For the three-dimensional problem, we need to add the loop for the
third dimension to the code.

Figure 4.1 shows the two-dimensional domain decomposition. Each subdomain is characterized by 4 parameters: the subdomain sizes and its relative position in the global domain which is reflected by the offset information. Let $N_x$ and $N_y$ be the two horizontal sizes of the global domain which is decomposed to a $P_x \times P_y$ processor configuration. Then the sizes of the subdomain, denoted by $npoint_x$ and $npoint_y$, and the offset informations, denoted by $offset_x$ and $offset_y$, are calculated as

$$npoint_x = \frac{N_x}{P_x},$$

$$npoint_y = \frac{N_y}{P_y},$$

$$offset_x = npoint_x \times p_x,$$

$$offset_y = npoint_y \times p_y,$$

where $(p_x, p_y)$ is the index of the processor. In Equation 4.1, we assume that $N_x$ and $N_y$ is exact divisible by $P_x$ and $P_y$, respectively. If this is not the case, we assign the remainders to the first processors. For example, if $N_x = 13$ and $P_x = 5$, then the processors 1, 2, and 3 will have $npoint_x = 3$, and the processors 4 and 5 will have $npoint_x = 2$. Mathematically, the subdomain sizes and the offset informations in case that the global domain is not exact divisible by the processor configuration are calculated as

$$npoint_x = \begin{cases} 
\frac{N_x}{P_x} + 1 & \text{if } p_x < \text{MOD}(N_x, P_x), \\
\frac{N_x}{P_x} & \text{otherwise},
\end{cases}$$

$$npoint_y = \begin{cases} 
\frac{N_y}{P_y} + 1 & \text{if } p_y < \text{MOD}(N_y, P_y), \\
\frac{N_y}{P_y} & \text{otherwise},
\end{cases}$$

$$offset_x = \begin{cases} 
npoint_x \times p_x & \text{if } p_x < \text{MOD}(N_x, P_x), \\
npoint_x \times p_x + \text{MOD}(N_x, P_x) & \text{otherwise},
\end{cases}$$

$$offset_y = \begin{cases} 
npoint_y \times p_y & \text{if } p_y < \text{MOD}(N_y, P_y), \\
npoint_y \times p_y + \text{MOD}(N_y, P_y) & \text{otherwise},
\end{cases}$$

where MOD denotes the remainder of a division. We note that Equation 4.2 is also true if the global domain is exact divisible by the processor configuration.
4.2. Implementation

Because if this is the case then \( \text{MOD}(N_x, P_x) = 0 \) and \( \text{MOD}(N_y, P_y) = 0 \), and Equation 4.2 becomes Equation 4.1.

To generate the decomposition code, we define the template \textit{decomposition} which has as input the global domain size \((N_x, N_y)\) and the number of processors \((P_x, P_y)\), and as output the size \((\text{npoint}_x, \text{npoint}_y)\) and the offset information \((\text{oset}_x, \text{oset}_y)\) of the subdomain. The code of \textit{decomposition} reads

\[
\text{decomposition}(N_x, N_y, P_x, P_y) \to \begin{cases} 
\% \text{Print Fortran code} \\
\% \text{Calculate subdomain size} \\
\text{if} (px<\text{MOD}(N_x, P_x)) \text{ then} \\
\text{npoint}_x = \frac{N_x}{P_x} + 1 \\
\text{else} \\
\text{npoint}_x = \frac{N_x}{P_x} \\
\text{endif} \\
\text{if} (py<\text{MOD}(N_y, P_y)) \text{ then} \\
\text{npoint}_y = \frac{N_y}{P_y} + 1 \\
\text{else} \\
\text{npoint}_y = \frac{N_y}{P_y} \\
\text{endif} \\
\% \text{Calculate offset information} \\
\text{if} (px<\text{MOD}(N_x, P_x)) \text{ then} \\
\text{oset}_x = \text{npoint}_x \times px \\
\text{else} \\
\text{oset}_x = \text{npoint}_x \times px + \text{MOD}(N_x, P_x) \\
\text{endif} \\
\text{if} (py<\text{MOD}(N_y, P_y)) \text{ then} \\
\text{oset}_y = \text{npoint}_y \times py \\
\text{else} \\
\text{oset}_y = \text{npoint}_y \times py + \text{MOD}(N_y, P_y) \\
\text{endif} 
\end{cases}
\]

The code generated by the template \textit{decomposition} is exactly the script inside this template.

4.2.2 Data distribution

After decomposing the global domain, we obtain subdomains with size and offset parameters. With this information we distribute the input data to all processors.

Let \( G \) be the input array on the global domain which has a size of \((N_x, N_y)\) and the data that should be distributed to the local subdomain \( S_g \) has as size and offset information \((\text{npoint}_x, \text{npoint}_y)\) and \((\text{oset}_x, \text{oset}_y)\), respectively. Then a grid point \( G(I, J) \) in the global domain is mapped to grid point \( S_g(i, j) \).
in the local subdomain where $i$ and $j$ are determined by

$$
i = I - offset_x, \\
j = J - offset_y.
$$

(4.3)

In CTADEL Equation (4.3) is specified in the template $split\_data$ as

$split\_data(G,Sg) \rightarrow (forall (j = 1 .. npointy) \forall (i = 1 .. npointx) (Sg:i:j := G:(i+offsetx):(j+offsety)))$

In the $split\_data$ template, $forall$ is the template that generates a Fortran DO loop. The code of the $forall$ template reads as

$$\$stmt(forall (I = L .. U, Sg)) \rightarrow (fortran % Print Fortran code \\
"DO ", I, ", ", L, ", ", U, \\
\$stmt(Sg), \\
"ENDDO"
),$$

where $I$ is the index variable, $L$ and $U$ are the lower and upper loop boundaries, respectively, and $Sg$ is the statement inside the loop.

After determining the subdomain $Sg$, we distribute these data to all processors. The template that generates the distribution data code reads as

$$distribution(G,Sg,S) \rightarrow (fortran IF (rank.EQ.0) THEN \\
forall (p = 1 .. P) (split\_data(G,Sg), \\
SEND\_DATA(Sg,npointx*npointy,p)) ENDIF \\
RECV\_DATA(S,npointx*npointy,0) ).$$

In the distribution template rank is the index of processor, $P$ denotes the total number of processors, and $S$ is the data that has been distributed to the processor $p$. We assume that the input value is initiated at the processor with the index of 0. Therefore, we only need to distribute the input data to the processors which have the indexes varying from 1 to $P$. The $SEND\_DATA$ and $RECV\_DATA$ templates are specified as

$$SEND\_DATA(send\_buffer,buffer\_length,destination) \rightarrow (fortran % Print Fortran code \\
CALL MPI\_SEND(send\_buffer,buffer\_length,MPI\_REAL, \\
destination,tag,MPI\_COMM\_WORLD,ierr) ) .$$
4.2. Implementation

Figure 4.2: Two-dimensional data distribution. Grid points in the global domain are indexed from 1 to 8 in both x and y direction, each grid point is denoted by a square area. Processors are numbered from 0 to 15.

\[
\begin{array}{cccc}
  & 12 & 13 & 14 & 15 \\
 8 &  &  &  & \\
 7 & 8 & 9 & 10 & 11 \\
 6 & 5 & 6 & 7 & \\
 5 & 4 & 5 & 6 & 7 \\
 4 & 3 & 4 & 5 & 6 \\
 3 & 2 & 3 & 4 & 5 \\
 2 & 1 & 2 & 3 & 4 \\
 1 & 0 & 1 & 2 & 3 \\
\end{array}
\]

Figure 4.2 is an example of a distribution of an \(8 \times 8\) matrix to \(4 \times 4\) processors.

4.2.3 Communication

The calculations in a subdomain may require information from adjoining subdomains. This information is obtained by communication with the neighboring processors. The required data received from the neighboring processors is stored in a “ring” of virtual grid points around the subdomain, the so-called “halo zone.”
The halo zone is described in detail in Figure 1.1 in Chapter 3. In this figure, the Wo, Eo, No, and So halo zones are filled by data received from the Ei, Wi, Si, and Ni areas of the East, West, South, and North adjacent subdomains, respectively. Therefore, we call Wo, Eo, No, and So the data areas to be filled, and Ei, Wi, Si, and Ni the data areas to be sent.

The data communication between processors consists of the following steps:

1. Determine the size of the halo zones; Allocate the Wi, Ei, Ni, Si, Wo, Eo, No, and So buffers;
2. Copy the data to be sent into the Wi and Ei buffers; Exchange with the west and east neighboring processors;
3. Fill the received data in the Wo and Eo halo zones;
4. Copy the data to be sent into the Ni and Si buffers; Exchange with the north and south neighboring processors;
5. Fill the received data in the No and So halo zones.

In the next paragraphs we will present how these steps are specified in CTADEL.

Determine the size of the halo zones

Because we use the two-dimensional domain decomposition, the halo zones are therefore considered as two-dimensional areas. Let the size of the halo zones be represented by the width and the length. The width of the halo zones depends on the discretization method used to solve the problem. Let \( hw \) and \( he \) denote the number of left and right extra points that are needed to discretize the problem at one grid point in the x-direction. Then a calculation on a grid point at the west and east boundaries of the subdomain requires information from \( hw \) and \( he \) grid points of the west and east neighboring subdomains, respectively. It means that the Wo and Eo areas have the width of \( hw \) and \( he \), respectively. Similarly, the No and So areas have a width of \( hn \) and \( hs \) grid points, respectively. Figure 4.3 is an example of a halo zone. In this example, the calculation on a grid point (gray color) requires information from eight grid points (light gray color): four grid points in the west, east, south, and north directions, and four grid points at the four corners north-west, north-east, south-west, and south-east. Therefore, the Wo, Eo, No, and So halo zones have a width of one grid point.

The length of the halo zones is derived from the subdomain size. If the subdomain has the size of \( (npoint_x, npoint_y) \), then the length of the Wo and Eo halo zones is \( npoint_y \). The four corners of the halo zones come from the four non-adjacent processors, indirectly: first the data is communicated with the west and east adjacent processors, and then to the north and south adjacent
4.2. Implementation

(a) Grid point stencil

Figure 4.3: (a) Grid point stencil, each grid point is denoted by a square area. The calculation on a grid point (gray) requires information from eight grid points around it (light gray). (b) Halo zone, the gray area denotes the subdomain with the size of $(\text{npoint}_x, \text{npoint}_y)$ and the light gray area denotes the halo zone.

processors. Hence, the No and So halo zones include these corner areas. As a result, the No and So halo zones has length of $(\text{npoint}_x + \text{hw} + \text{he})$ (see Figure 4.3 b). Summarizing, the halo zones Wo, Eo, No, and So have a size of \( \text{hw} \times \text{npoint}_y, \text{he} \times \text{npoint}_y, \text{hn} \times (\text{npoint}_x + \text{hw} + \text{he}), \) and \( \text{hs} \times (\text{npoint}_x + \text{hw} + \text{he}) \) grid points, respectively.

Since the buffers receive the data that have been sent, the data areas to be sent, Ei, Wi, Si, and Ni have the same size as the data areas to be filled, Wo, Eo, No, and So, respectively. Specifically, the halo zones Ei, Wi, Si, and Ni have a size of \( \text{hw} \times \text{npoint}_y, \text{he} \times \text{npoint}_y, \text{hn} \times (\text{npoint}_x + \text{hw} + \text{he}), \) and \( \text{hs} \times (\text{npoint}_x + \text{hw} + \text{he}) \) grid points, respectively.

The code generator automatically derives the sizes of the halo zones from an analysis of the discretized equations, so the user does not have to worry about them.

**Copy the data to be sent into the buffers**

Based on the size of the halo zones, the data to be sent can be extracted, merged and copied into the Wi, Ei, Ni, and Si buffers. Let \( S \) denotes a variable that contains data on a subdomain, then the data to be sent Wi, Ei, Ni, and Si, are copied as

```matlab
% Copy Wi zone
    Do j = 1,npointy
    Do i = 1,he
        Wi(i,j) = S(i,j)
    Enddo
Enddo
```
% Copy Ei zone
Do j = 1, npointy
   Do i = 1, hw
      Ei(i,j) = S(i+npointx-hw,j)
   Enddo
Enddo

% Copy Ni zone
Do j = 1, hs
   Do i = 1-hw, npointx+he
      Ni(i,j) = S(i,j+npointy-hs)
   Enddo
Enddo

% Copy Si zone
Do j = 1, hn
   Do i = 1-hw, npointx+he
      Si(i,j) = S(i,j)
   Enddo
Enddo

To generate code for the copying of the data to be sent, we define the following templates in CTADEL.

west_east_sending_data(S) -> (
   % Copy Wi zone
   forall (j = 1..npointy)
   forall (i = 1..he)
   (Wi:i:j := S:i:j);
   % Copy Ei zone
   forall (j = 1..npointy)
   forall (i = 1..hw)
   (Ei:i:j := S:(i+npointx-hw):j);
).

north_south_sending_data(S) -> (
   % Copy Ni zone
   forall (j = 1..hs)
   forall (i = 1-hw..npointx+he)
   (Ni:i:j := S:i:(j+npointy-hs));
   % Copy Si zone
   forall (j = 1..hn)
   forall (i = 1-hw..npointx+he)
   (Si:i:j := S:i:j);
).

The forall template, which has been defined in Subsection 4.2.2, is used to generate the Fortran DO loops.
4.2. Implementation

Exchange data

After copying the data into the send buffers, we exchange these buffers with the neighboring processors. The four neighboring processors, denoted by west, east, north, and south, are determined as in Figure 4.4. Let \((p_x, p_y)\) denotes the index of a processor, the indexes of its four neighboring processors are west \(=(p_x - 1, p_y)\), east \(=(p_x + 1, p_y)\), north \(=(p_x, p_y + 1)\), and south \(=(p_x, p_y - 1)\). The four corners of the halo zones come from the four non-adjacent processors north_west \(=(p_x - 1, p_y + 1)\), south_west \(=(p_x - 1, p_y - 1)\), north_east \(=(p_x + 1, p_y + 1)\), and south_east \(=(p_x + 1, p_y - 1)\).

The receive buffer of a processor located at a boundary of the processor configuration is filled by the boundary condition instead of by the data received from the neighboring processor. A processor is at the west or east boundary if its index in the x-direction is 0 or \(P_x\), respectively, where \(P_x\) is the size in the x-direction of the processor configuration. Similarly, a processor is at the south or north boundary if its index in the y-direction is 0 or \(P_y\), respectively, where \(P_y\) is the size in the y-direction of the processor configuration.

The templates for generating communication statements have the following form:

```
west_east_exchange_data:=(
    % Exchange with east neighbor
    IF (.not.EAST_BOUND) THEN % If not at EAST boundary,
        SEND_DATA(Ei,Ei_length,east); % send to EAST processor
    ENDIF
    IF (.not.WEST_BOUND) THEN % If not at WEST boundary,
        RECV_DATA(Wo,Wo_length,west); % receive from WEST processor
    ENDIF
```

![Figure 4.4: Neighboring processors](image)
% Exchange with west neighbor
    IF (.not.WEST_BOUND) THEN % If not at WEST boundary,
        SEND_DATA(Wi,Wi_length,west); % send to WEST processor
    ENDIF
    IF (.not.EAST_BOUND) THEN % If not at EAST boundary,
        RECV_DATA(Eo,Eo_length,east); % receive from EAST processor
    ENDIF

north_south_exchange_data:=(
% Exchange with north neighbor
    IF (.not.NORTH_BOUND) THEN % If not at NORTH boundary,
        SEND_DATA(Ni,Ni_length,north); % send to NORTH processor
    ENDIF
    IF (.not.SOUTH_BOUND) THEN % If not at SOUTH boundary,
        RECV_DATA(So,So_length,south); % receive from SOUTH processor
    ENDIF

% Exchange with south neighbor
    IF (.not.SOUTH_BOUND) THEN % If not at SOUTH boundary,
        SEND_DATA(Si,Si_length,south); % send to SOUTH processor
    ENDIF
    IF (.not.NORTH_BOUND) THEN % If not at NORTH boundary,
        RECV_DATA(No,No_length,north); % receive from NORTH processor
    ENDIF
).%

In the above templates, the SEND_DATA and RECV_DATA templates have been defined in Subsection 4.2.2. Wi, EI, Ni, and Si represent the send buffers. Eo, Wo, No, and So are the receive buffers. To avoid deadlock, we arrange communication so that the communication in one direction starts when the communication in other direction has been finished. Therefore, we firstly exchange with the east neighboring (send to east and receive from west). Once this exchange is complete, we start to exchange with the west and then with north and south neighboring processors.

**Fill-in halo zone**

The halo zones are filled by the data received from the neighboring processors as follows

% Fill-in Wo zone
    Do j = 1,npointy
        Do i = 1,hw
            S(i-hw,j) = Wo(i,j)
        Enddo
    Enddo
% Fill-in Eo zone
    Do j = 1, npointy
    Do i = 1, he
        S(i + npointx, j) = Eo(i, j)
    Enddo
    Enddo

% Fill-in No zone
    Do j = 1, hn
    Do i = 1-hw, npointx+he
        S(i, j+npointy) = No(i, j)
    Enddo
    Enddo

% Fill-in So zone
    Do j = 1, hs
    Do i = 1-hw, npointx+he
        S(i, j-hs) = So(i, j)
    Enddo
    Enddo

The templates which generate code for fill-in data, read as

fillin_west_east_zone(S) -> (  
% Fill in Wo zone
    forall (j = 1..npointy)
    forall (i = 1..hw)
        (S:(i-hw):j:=Wo:i:j);
% Fill in Eo zone
    forall (j = 1..npointy)
    forall (i = 1..he)
        (S:(i+npointx):j:=Eo:i:j);
).

fillin_north_south_zone(S) -> (  
% Fill in No zone
    forall (j = 1-hw..npointx+he)
    forall (i = 1..hn)
        (S:i:(j+npointy):=No:i:j);
% Fill in So zone
    forall (j = 1-hw..npointx+he)
    forall (i = 1..hs)
        (S:i:(j-hs):=So:i:j);
).

The fill-in data templates are similar to the templates for copy data to the send buffer.
4.2.4 Results collection

After the calculations, the results are still distributed over the processors and need to be gathered to a final result. An important issue in gathering the data is to copy the local distributed data to the appropriate locations in the global domain. Let $G$ denote the gathered data on the global domain. Then a grid point $Sg(i,j)$ in the local subdomain is mapped to grid point $G(I,J)$ in the global domain where $I$ and $J$ are determined by

$$I = i + offset_x,$$
$$J = j + offset_y.$$  \hfill (4.4)

Equation 4.4 is specified in CTADEL as

```plaintext
reorder_data(G,Sg) -> (forall (j = 1 .. npointy)
forall (i = 1 .. npointx)
(G:(i+offsetx):(j+offsety) := Sg:i:j)).
```

The gathering result is basically the opposite of the input data distribution. The template that generates code for gathering result is specified as

```plaintext
gather(G,Sg,S) -> (fortran %Print Fortran code
SEND_DATA(S,npointx*npointy,0)
IF (rank.EQ.0) THEN
forall (p = 1 .. P)
(RECV_DATA(Sg,npointx*npointy,p),
reorder_data(G,Sg))
ENDIF
).
```

Similar to the data distribution, the deadlock does not occur at the gathering result because all processors either send or receive the result data.

4.3 Generating parallel code for the Shallow-Water equations

We have presented our extensions to CTADEL to generate parallel programs based on the SPMD model. In this section, we show, as a test case, how to apply this technique to generate parallel code for the Shallow-Water equations [63].

The Shallow-Water equations have been described in Chapter 2. In that chapter we also presented the Galerkin finite element method applied to the Shallow-Water equations, and how a code based on this method is generated by CTADEL. In this chapter we show how a parallel code for the Shallow-Water equations is generated by CTADEL.
4.3.1 Specification

In CTADEL, a parallel program is generated in two steps: firstly, generate the sequential code and then generate the parallel program. The sequential code determines the outputs of the Shallow-Water equations. The parallel code provides domain decomposition, data distribution, communication, and results collection. The specification and code generation of the sequential code was described in Chapter 2. Below we show the specification that is used to generate the parallel code.

% GENERATE PARALLEL CODE
% Step 1: Domain decomposition
  decomposition(Nx,Ny,Px,Py);

% Step 2: Data distribution
  distribution(U,Usg,Us);
  distribution(V,Vsg,Vs);
  distribution(H,Hsg,Hs);

% Step 3: Communications & calculations
% Copy the data to be sent to west-east buffers,
% exchange with west-east neighboring processors,
% and fillin west-east halo zones
  west_east_sending_data(Us,Vs,Hs);
  west_east_exchange_data;
  fillin_west_east_zone(Us,Vs,Hs);

% Copy the data to be sent to north-south buffers,
% exchange with north-south neighboring processors,
% and fillin south-north halo zones
  north_south_sending_data(Us,Vs,Hs);
  north_south_exchange_data;
  fillin_north_south_zone(Us,Vs,Hs);

% Do calculation in all processors
% by calling the generated sequential code
  sequential_code(Us,Vs,Hs,Ust,Vst,Hst);

% Step 4: Results collection
  gather(Ut,Ustg,Ut);
  gather(Vt,Vstg,Vt);
  gather(Ht,Hstg,Ht);

In the above specification, \((U,V,H)\) and \((U_t,V_t,H_t)\) denote respectively the inputs and outputs of the program, \((U_s,V_s,H_s)\) are the distributed inputs and \((U_{st},V_{st},H_{st})\) are the corresponding outputs on the subdomain, \(NX\times NY\) is the domain size of the problem, and \(PX\times PY\) is the processor configuration. The
sequential_code is obtained from the sequential code generated by CTADEL for the Shallow-Water equations applying the Galerkin finite element method (see Chapter 2) by modifying the input/output names from (U, V, H, Ut, Vt, Ht) to (Us, Vs, Hs, Ust, Vst, Hst) and the computational domain from global domain to subdomain.

4.3.2 Code generation

From the given specification, CTADEL generates parallel code for the Shallow-Water equations. The generated parallel code is listed in Section Appendix.

4.4 Experiments

This section describes the experiments with the generated parallel program for the Shallow-Water equations. We performed the experiments on the Leiden University (LU) cluster of the DAS-3 system [67]. The hardware specification of the LU cluster has been given in Section 4 in Chapter 3. For compilation we used the mpif77 compiler version 1.2.7. The MPI communication routines were from MPICH [49], version 1.2.7.

We checked the correctness of the generated parallel code by verifying that the parallel program reproduces the output of the serial code exactly.

Firstly, to determine the speedup of the parallel program for a domain of 1000 \times 1000 grid points, we ran it on 2, 4, 8, 16, and 32 processors, which are configured as 2 \times 1, 2 \times 2, 2 \times 4, 4 \times 4, and 4 \times 8 configurations. The result is shown in Figure 4.5. The speedup is defined by the ratio of the time needed to execute the sequential code and the execution time of the parallel program on a processor. We did not include the time for distributing the input data and collecting the final result to the execution time of the parallel program because distribution and collection are performed one time in the program, whereas communication and calculation are executed at every time step of the program. From Figure 4.5 we see that the speedup is almost linear.

In order to assess the scalability of the parallel code, we ran the program on a 2 \times 2 processor configuration with different domain sizes. Figure 4.6 shows the execution times for different domains of 1000 \times 1000, 2000 \times 1000, 2000 \times 2000, and 4000 \times 2000 grid points. We observe that the generated parallel code scales very well with respect to increased domain sizes.

4.5 Related work

The automatic generation of parallel codes has been studied in several projects. Ferner [20] presented a method of automatically deriving communication code
4.5. Related work

Figure 4.5: Speedup of the generated parallel code as function of the number of processors. The “Linear line” shows the linear speedup from one processor.

Figure 4.6: Execution time as function of the computational domain

for distributed memory architectures. The method was based on a parallelization technique of Lim and Lam [38] that generates so-called partitions of the index space from an input program. These partitions can be executed in parallel on logical processors. Further proposed a mapping algorithm from logical processors (partitions) to real processors and illustrated how the corresponding communication code can be obtained.

Several tools have been developed for automatic generation of parallel code such as PARADIGM [6], CAPTools [31], Uintah [24], and HPC-MW [32].

PARADIGM (Parallelizing compiler for distributed memory general-purpose multicomputers) provides an automated means of parallelizing and optimizing serial programs for efficient use on a distributed memory system. PARADIGM
allows automatic data distribution, communication optimization, and the exploitation of both functional and data parallelism.

CAPTools (Computer aided parallelization tools) is targeted at facilitating the generation of parallel Fortran 77 code with standard communication calls. The final parallel code generated by CAPTools adheres to the SPMD program. The nucleus of CAPTools is its powerful, symbolic, and interprocedural value based dependence analysis. The need for user interaction is imperative to ensure that the dependence analysis obtained is as accurate as possible. The user interaction is extensive so that the user may at any stage of the parallelization examine the information provided by the system and is also provided with additional information.

The Uintah PSE framework provides an environment that allows scientific programmers to more easily create coupled, parallel simulation components while at the same time allowing them to easily explore the effects of dynamically changing a large number of parameters during a simulation run. Because of Uintah's component architecture which automates the work of communication and distribution, scientists can also more easily explore the use of different methods to solve the same problem. The Uintah PSE is a powerful simulation tool that provides a number of advantages. These include support of both distributed and shared memory computations, increasing the number and types of components that are interoperable within a single framework, adding additional data interfaces between components (Dataflow and Uses/Provides ports for parallel communication), and allowing a detachable user interface that supports a number of implementations (such as a TCL, Java, or Web Based GUI). Uintah combines the interaction capabilities of SCIRun [66], a PSE for modeling, simulation, and visualization of scientific problems, with the parallel-communication capabilities of the Common Component Architecture [14]. This allows Uintah to support a large number of interoperable, highly parallel components.

HPC-MW was designed as a PSE for FEM (Finite element method) application. Various procedures common to FEM are provided by the HPC-MW PSE, and parallelization is automatically achieved by the PSE. An incompressible fluid analysis code was developed using this PSE. A parallel efficiency of approximately 94% was obtained on a cluster which has 24 Xeon 2.8 GHz CPUs, each with 2 GB RAM, and Myrinet-MX connection.

All the tools described above take a sequential code as input to generate a parallel program. Different from those, CTADEL generates parallel code from the problem specification, through two steps: generate the sequential code and generate the parallel program. In addition, our code generator automatically derives the size of halo zone, and hence reduces the intervention from the user.
4.6 Conclusion

Previous studies have shown that CTADEL can generate efficient serial code. We showed that, after extension, CTADEL can generate SPMD parallel code. We extended CTADEL by defining templates that generate code for the steps involved in a SPMD parallel program such as decomposition, distribution, communication, and collection. Then we applied this technique for the Shallow-Water equations. The experimental results showed that the parallel code generated by CTADEL has a good scalability and speedup.

4.7 Appendix

This appendix lists the generated parallel code for the Shallow-Water equations solved by the Galerkin finite element method.

Decomposition

From the computational domain \((N_x \times N_y)\) and the processor configuration \((P_x \times P_y)\), determine the information of the subdomain including the subdomain sizes \((npoint_x \times npoint_y)\) and the offsets \((offset_x \times offset_y)\).

* Index of processor
  \[
  px = \text{MOD}(\text{rank}, P_x) \\
  py = (\text{rank} - px) / P_x
  \]

* Neighboring processors
  \[
  \text{west} = (px + 1) + py \times P_x \\
  \text{east} = (px + 1) + py \times P_x \\
  \text{north} = px + (py + 1) \times P_x \\
  \text{south} = px + (py - 1) \times P_x
  \]

* Subdomain sizes
  \[
  \text{if } (px < \text{MOD}(N_x, P_x)) \text{ then} \\
  npointx = \frac{N_x}{P_x} + 1 \\
  \text{else} \\
  npointx = \frac{N_x}{P_x}
  \]
  \[
  \text{if } (py < \text{MOD}(N_y, P_y)) \text{ then} \\
  npointy = \frac{N_y}{P_y} + 1 \\
  \text{else} \\
  npointy = \frac{N_y}{P_y}
  \]

* Offset information
  \[
  \text{if } (px < \text{MOD}(N_x, P_x)) \text{ then} \\
  offsetx = npointx \times px \\
  \text{else} \\
  offsetx = npointx \times px + \text{MOD}(N_x, P_x)
  \]
Chapter 4. Automatic code generation of SPMD parallel programs

Data distribution

Next, we determine the parts of data that are processed by each processor $U_{sg}$, $V_{sg}$, and $H_{sg}$ and distribute them to the processors. $U$, $V$, and $H$ denote the inputs of the program.

```fortran
IF (rank.EQ.0) THEN
  DO 116 p=0,P
    DO 117 j=1,npointy
      DO 118 i=1,npointx
        Usg(i,j)=U(i+offsetx,j+offsety)
      118 CONTINUE
    117 CONTINUE
    CALL MPI_SEND(Usg,npointx*npointy,MPI_REAL,
                  p,tag(p),MPI_COMM_WORLD,ierr)
  116 CONTINUE
ENDIF
CALL MPI_RECV(Us,npointx*npointy,MPI_REAL,
               0,tag(rank),MPI_COMM_WORLD,ierr)
IF (rank.EQ.0) THEN
  DO 119 p=0,P
    DO 120 j=1,npointy
      DO 121 i=1,npointx
        Vsg(i,j)=V(i+offsetx,j+offsety)
      121 CONTINUE
    120 CONTINUE
    CALL MPI_SEND(Vsg,npointx*npointy,MPI_REAL,
                  p,tag(p),MPI_COMM_WORLD,ierr)
  119 CONTINUE
ENDIF
CALL MPI_RECV(Vs,npointx*npointy,MPI_REAL,
               0,tag(rank),MPI_COMM_WORLD,ierr)
IF (rank.EQ.0) THEN
  DO 122 p=0,P
    DO 123 j=1,npointy
      DO 124 i=1,npointx
        Hsg(i,j)=H(i+offsetx,j+offsety)
      124 CONTINUE
    123 CONTINUE
ENDIF
CALL MPI_RECV(Hs,npointx*npointy,MPI_REAL,
               0,tag(rank),MPI_COMM_WORLD,ierr)
```

endif
if (py<MOD(Ny,Py)) then
  offsety = npointy*py
else
  offsety = npointy*py + MOD(Ny,Py)
endif
CALL MPI_SEND(Hs,npointx*npointy,MPI_REAL,
   p,tag(p),MPI_COMM_WORLD,ierr)
122 CONTINUE
ENDIF
CALL MPI_RECV(Hs,npointx*npointy,MPI_REAL,
   0,tag(rank),MPI_COMM_WORLD,ierr)

Communication

Before the calculations, the processor needs to communicate with its neighboring processors. The communication code consists of copying the data into the send buffers, exchanging with the neighboring processors, and filling the received data in the halo zones.

* Copy the data to the west-east send buffers
   DO 125 j=1,npointy
      DO 126 i=1,1
         Wi(i,j,1)=Us(i,j)
      126 CONTINUE
      125 CONTINUE
   DO 127 j=1,npointy
      DO 128 i=1,1
         Wi(i,j,2)=Vs(i,j)
      128 CONTINUE
      127 CONTINUE
   DO 129 j=1,npointy
      DO 130 i=1,1
         Wi(i,j,3)=Hs(i,j)
      130 CONTINUE
      129 CONTINUE
   DO 131 j=1,npointy
      DO 132 i=1,1
         Ei(i,j,1)=Us(npointx-1+i,j)
      132 CONTINUE
      131 CONTINUE
   DO 133 j=1,npointy
      DO 134 i=1,1
         Ei(i,j,2)=Vs(npointx-1+i,j)
      134 CONTINUE
      133 CONTINUE
   DO 135 j=1,npointy
      DO 136 i=1,1
         Ei(i,j,3)=Hs(npointx-1+i,j)
      136 CONTINUE
      135 CONTINUE
* Exchange with the west-east neighboring processors
  IF (.not.ISEAST) THEN
    CALL MPI_SEND(Ei,3*npointy,MPI_REAL,east,
      tag1,MPI_COMM_WORLD,ierr)
  ENDIF
  IF (.not.ISWEST) THEN
    CALL MPI_RECV(Wo,3*npointy,MPI_REAL,west,
      tag1,MPI_COMM_WORLD,status,ierr)
  ENDIF
  IF (.not.ISWEST) THEN
    CALL MPI_SEND(Wi,3*npointy,MPI_REAL,west,
      tag2,MPI_COMM_WORLD,ierr)
  ENDIF
  IF (.not.ISEAST) THEN
    CALL MPI_RECV(Eo,3*npointy,MPI_REAL,east,
      tag2,MPI_COMM_WORLD,status,ierr)
  ENDIF

* Fill in the west-east halo zones
  DO 137 j=1,npointy
    DO 138 i=1,1
      IF (.not.ISWEST) THEN
        Us(i-1,j)=Wo(i,j,1)
      ELSE
        Us(i-1,j)=U(Nx-1+i,j+offsety)
      ENDIF
    CONTINUE
  CONTINUE
  DO 139 j=1,npointy
    DO 140 i=1,1
      IF (.not.ISWEST) THEN
        Vs(i-1,j)=Wo(i,j,2)
      ELSE
        Vs(i-1,j)=V(Nx-1+i,j+offsety)
      ENDIF
    CONTINUE
  CONTINUE
  DO 141 j=1,npointy
    DO 142 i=1,1
      IF (.not.ISWEST) THEN
        Hs(i-1,j)=Wo(i,j,3)
      ELSE
        Hs(i-1,j)=H(Nx-1+i,j+offsety)
      ENDIF
    CONTINUE
  CONTINUE
DO 143 j=1,npointy
DO 144 i=1,1
   IF (.not.ISEAST) THEN
      Us(npointx+i,j)=Eo(i,j,1)
   ELSE
      Us(npointx+i,j)=U(i+0,j+offsety)
   ENDIF
CONTINUE
DO 145 j=1,npointy
DO 146 i=1,1
   IF (.not.ISEAST) THEN
      Vs(npointx+i,j)=Eo(i,j,2)
   ELSE
      Vs(npointx+i,j)=V(i+0,j+offsety)
   ENDIF
CONTINUE
DO 147 j=1,npointy
DO 148 i=1,1
   IF (.not.ISEAST) THEN
      Hs(npointx+i,j)=Eo(i,j,3)
   ELSE
      Hs(npointx+i,j)=H(i+0,j+offsety)
   ENDIF
CONTINUE
* Copy the data to the north-south send buffers
DO 149 j=1,1
DO 150 i=0,npointx+1
   Ni(i,j,1)=Us(i,npointy-1+j)
CONTINUE
DO 151 j=1,1
DO 152 i=0,npointx+1
   Ni(i,j,2)=Vs(i,npointy-1+j)
CONTINUE
DO 153 j=1,1
DO 154 i=0,npointx+1
   Ni(i,j,3)=Hs(i,npointy-1+j)
CONTINUE
DO 155 j=1,1
DO 156 i=0,npointx+1
   Si(i,j,1)=Us(i,j)
Chapter 4. Automatic code generation of SPMD parallel programs

156     CONTINUE
155     CONTINUE
     DO 157 j=1,1
     DO 158 i=0,npointx+1
       Si(i,j,2)=Vs(i,j)
     158 CONTINUE
157     CONTINUE
     DO 159 j=1,1
     DO 160 i=0,npointx+1
       Si(i,j,3)=Hs(i,j)
     160 CONTINUE
159     CONTINUE

* Exchange with the north-south neighboring processors
IF (.not.ISTOP) THEN
   CALL MPI_SEND(Ni,3*(npointx+2),MPI_REAL,north,
                 tag3,MPI_COMM_WORLD,ierr)
ENDIF
IF (.not.ISDOWN) THEN
   CALL MPI_RECV(So,3*(npointx+2),MPI_REAL,south,
                 tag3,MPI_COMM_WORLD,status,ierr)
ENDIF
IF (.not.ISDOWN) THEN
   CALL MPI_SEND(Si,3*(npointx+2),MPI_REAL,south,
                 tag4,MPI_COMM_WORLD,ierr)
ENDIF
IF (.not.ISTOP) THEN
   CALL MPI_RECV(No,3*(npointx+2),MPI_REAL,north,
                 tag4,MPI_COMM_WORLD,status,ierr)
ENDIF

* Fill in the north-south halo zones
     DO 161 j=1,1
     DO 162 i=0,npointx+1
       IF (.not.ISTOP) THEN
         Us(i,npointy+j)=No(i,j,1)
       ELSE
         Us(i,npointy+j)=U(i+offsetx,j+0)
       ENDIF
     162 CONTINUE
161     CONTINUE
     DO 163 j=1,1
     DO 164 i=0,npointx+1
       IF (.not.ISTOP) THEN
         Vs(i,npointy+j)=No(i,j,2)
       ELSE
         Vs(i,npointy+j)=V(i+offsetx,j+0)
       ENDIF
     164 CONTINUE
163     CONTINUE
ENDIF
164     CONTINUE
163     CONTINUE
DO 165 j=1,1
DO 166 i=0,npointx+1
  IF (.not.ISTOP) THEN
  Hs(i,npointy+j)=No(i,j,3)
  ELSE
  Hs(i,npointy+j)=H(i+offsetx,j+0)
  ENDIF
166 CONTINUE
165 CONTINUE
DO 167 j=1,1
DO 168 i=0,npointx+1
  IF (.not.ISDOWN) THEN
  Us(i,j-1)=So(i,j,1)
  ELSE
  Us(i,j-1)=0.0
  ENDIF
168 CONTINUE
167 CONTINUE
DO 169 j=1,1
DO 170 i=0,npointx+1
  IF (.not.ISDOWN) THEN
  Vs(i,j-1)=So(i,j,2)
  ELSE
  Vs(i,j-1)=0.0
  ENDIF
170 CONTINUE
169 CONTINUE
DO 171 j=1,1
DO 172 i=0,npointx+1
  IF (.not.ISDOWN) THEN
  Hs(i,j-1)=So(i,j,3)
  ELSE
  Hs(i,j-1)=0.0
  ENDIF
172 CONTINUE
171 CONTINUE

Calculation

Perform calculations on the processors by invoking the serial code.

CALL sequential_code(Us,Vs,Hs,Ust,Vst,Hst)
Results collection

After the calculations, the results, which are still distributed over the processors, have to be gathered to the final result.

```
CALL MPI_SEND(Ust,npointx*npointy,MPI_REAL,
              0,tag(rank),MPI_COMM_WORLD,ierr)
IF (rank.EQ.0) THEN
  DO 173 p=0,P
    CALL MPI_RECV(Ustg,npointx*npointy,MPI_REAL,
                  p,tag(p),MPI_COMM_WORLD,ierr)
    DO 174 j=1,npointy
      DO 175 i=1,npointx
        Ut(i+offsetx,j+offsety)=Ustg(i,j)
      175 CONTINUE
    174 CONTINUE
  173 CONTINUE
ENDIF
CALL MPI_SEND(Vst,npointx*npointy,MPI_REAL,
              0,tag(rank),MPI_COMM_WORLD,ierr)
IF (rank.EQ.0) THEN
  DO 176 p=0,P
    CALL MPI_RECV(Vstg,npointx*npointy,MPI_REAL,
                  p,tag(p),MPI_COMM_WORLD,ierr)
    DO 177 j=1,npointy
      DO 178 i=1,npointx
        Vt(i+offsetx,j+offsety)=Vstg(i,j)
      178 CONTINUE
    177 CONTINUE
  176 CONTINUE
ENDIF
CALL MPI_SEND(Hst,npointx*npointy,MPI_REAL,
              0,tag(rank),MPI_COMM_WORLD,ierr)
IF (rank.EQ.0) THEN
  DO 179 p=0,P
    CALL MPI_RECV(Hstg,npointx*npointy,MPI_REAL,
                  p,tag(p),MPI_COMM_WORLD,ierr)
    DO 180 j=1,npointy
      DO 181 i=1,npointx
        Ht(i+offsetx,j+offsety)=Hstg(i,j)
      181 CONTINUE
    180 CONTINUE
  179 CONTINUE
ENDIF
```