*

Synchronous Computations

*Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

Introduction

We now consider problems that require a group of individual computations that need to wait for one another at certain times, before continuing.

Waiting for each other causes them to become *synchronized.*

When all processes in an application are synchronized at regular points, we say the application is a *fully synchronous application.*

Normally, same operations applied in parallel to a set of data.

All operations start in lock-step like SIMD computations.

Barriers

Say we have a number of processes performing a computation.

The processes that finish first must wait until all processes reach a common reference point.

A common reason is when processes must exchange data, and then continue from a known state.

Need a mechanism that will stop all processes from proceeding past a particular point until all are ready.

Such a mechanism is called a *barrier.*

A barrier is placed at a point in all processes where the process must wait. All processes can only continue when every processes has reached it. See Figure 6.1.

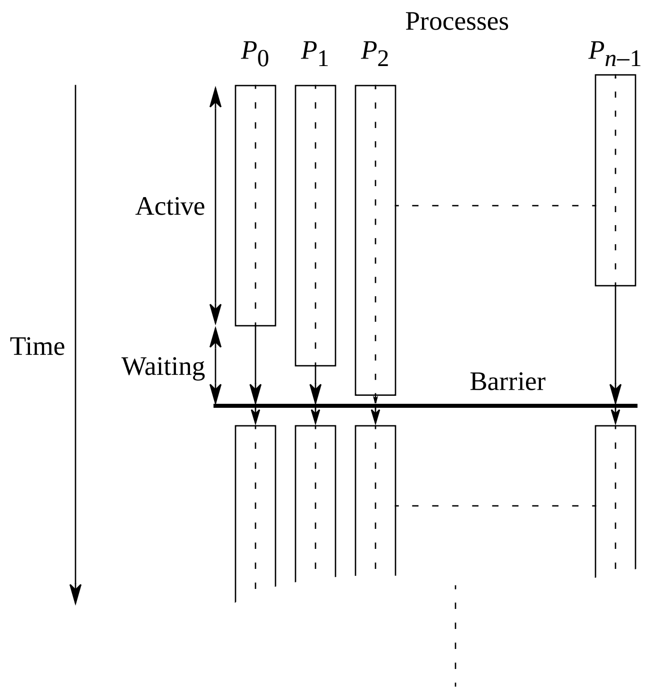


Figure 6.1 Processes reaching the barrier at different times.

Barriers Cont.

Barriers can be used in both shared memory and message-passing systems.

Usually provided by library routines.

For MPI, the routine is MPI_Barrier() which takes a communicator as parameter.

MPI_Barrier() must be called by all members of the communicator, and blocks until all members have arrived at the barrier call, and returning only then.

Figure 6.2 shows library call approach.

How the routine is actually implemented depends on the implementer of the libraries, who take into consideration the systems architecture.

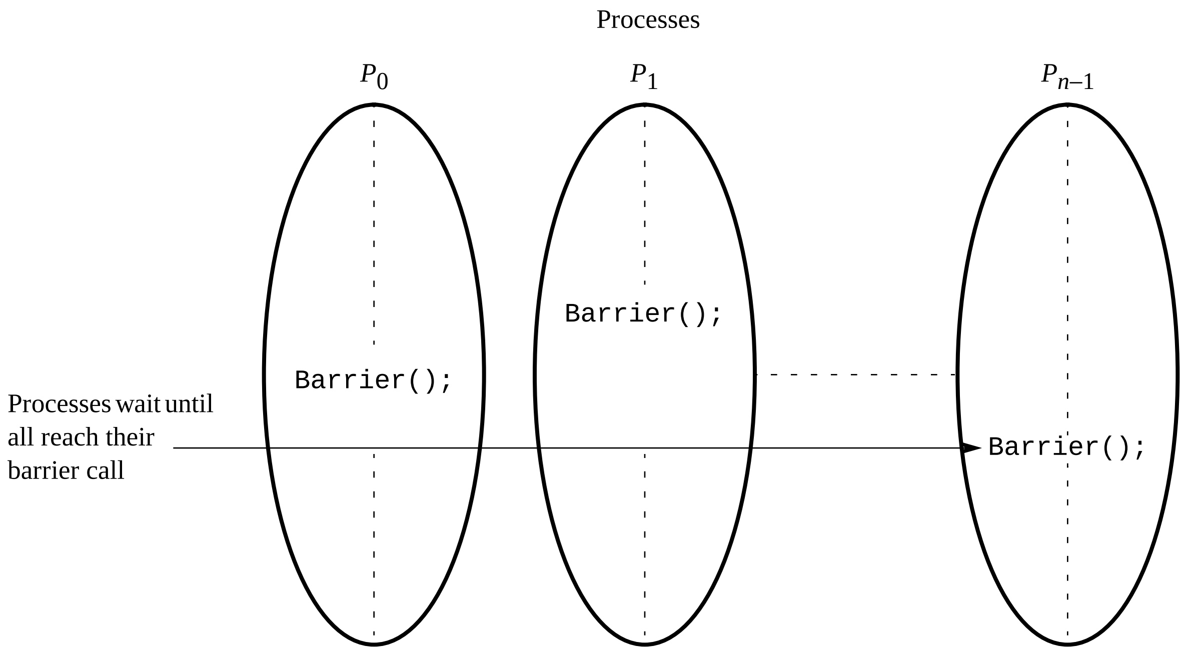


Figure 6.2 Library call barriers.

Counter Implementation

One way to implement a barrier is as a centralized counter (also called a *linear barrier*) as in Figure 6.3.

Single counter used to keep track of number of processes to arrive at barrier.

Initialize to zero then count to some value n.

If process has reached barrier and count $< n$, then process is stalled.

When counter reaches n, all processes waiting on it are released.

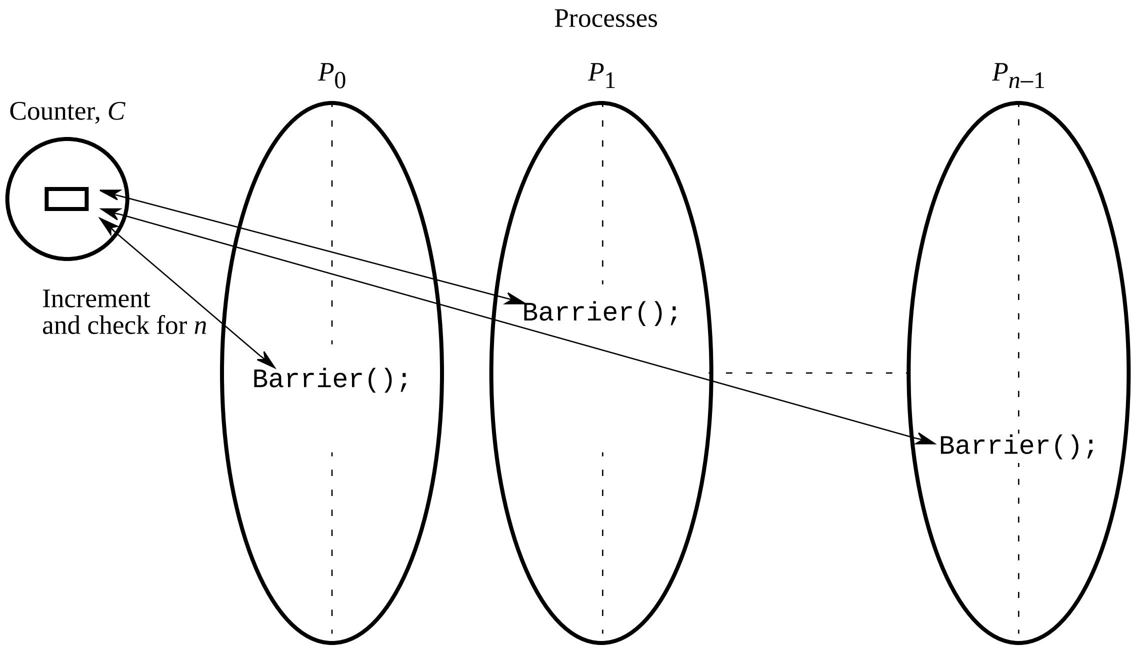


Figure 6.3 Barrier using a centralized counter.

Counter Implementation Phases

These type of barriers normally have two phases:

Arrival Phase: The arrival or trapping phase is the phase when the process reaches the barrier call. It stays here until all processes reach this phase.

Departure Phase: Once all processes reach the arrival phase, they move to the departure phase and are released from the barrier.

As barriers often used more than once in a program, must keep in mind that a process can enter a barrier for a second time before others have left from the first time.

Counter Implementation Phases Cont.

This problem can be handled as below.

Master counts messages received from slaves during the arrival phase, and then releases slaves in departure phase. See Figure 6.4.

The code would look like:*

```
/* count slaves as they reach barrier */
for (i = 0; i < n; i++)
   recv(Pany);

/* release slaves */
for (i = 0; i < n; i++)
   send(Pi);
```

Slave code would be:

```
send(Pmaster);
recv(Pmaster);
```

*This and other code from "Parallel Programming.. " by Wilkinson et al.

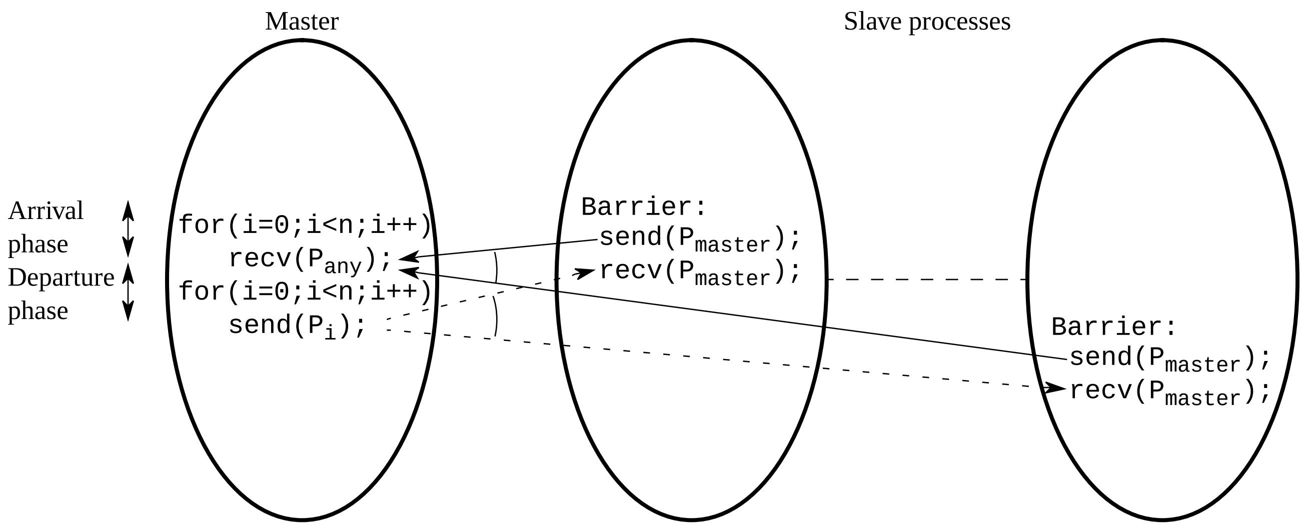


Figure 6.4 Barrier implementation in a message-passing system.

Tree Implementation

Counter implementation is $O(n)$. Can implement more efficiently as decentralized tree construction.

If we have eight processes, P0, P1, ... , P7, the algorithm would be as follows:

```
Stage 1: P1 sends message to P0; (when P1 hits barrier)
         P3 sends message to P2; (when P3 hits barrier)
         P5 sends message to P4; (when P5 hits barrier)
         P7 sends message to P6; (when P7 hits barrier)

Stage 2: P2 sends message to P0; (P2 and P3 at barrier)
         P6 sends message to P4; (P6 and P7 at barrier)

Stage 3: P4 sends message to P0; (P4-P7 at barrier)
         P0 ends arrival phase; (P0 hits barrier and
                 received message from P4)
```

To release processes, reverse tree construction. See Figure 6.5.

In general, requires $2\log n$ steps and is thus $O(\log n)$.

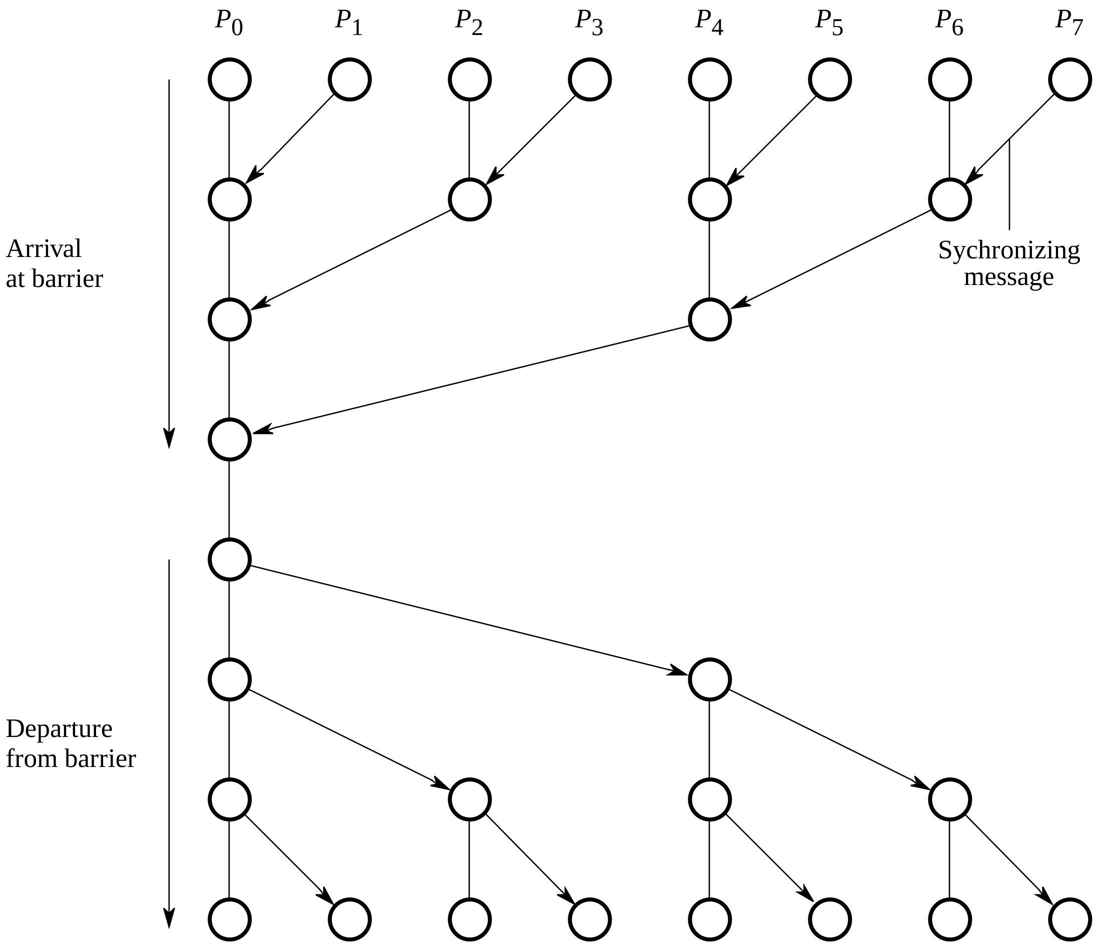


Figure 6.5 Tree barrier.

Butterfly Barrier

Tree construction can be extended into a *butterfly* construction, where pairs of processes synchronize at each step (See Figure 6.6):

```
Stage 1: P0 <-> P1, P2 <-> P3, P4 <-> P5, P6 <-> P7
Stage 2: P0 <-> P2, P1 <-> P3, P4 <-> P6, P5 <-> P7
Stage 3: P0 <-> P4, P1 <-> P5, P2 <-> P6, P3 <-> P7
```

After stage 3, all processes have been synchronized with all others, and all can continue.

If we have n processes (n a power of 2), then the butterfly construction takes $\log n$ steps and is thus $O(\log n)$.

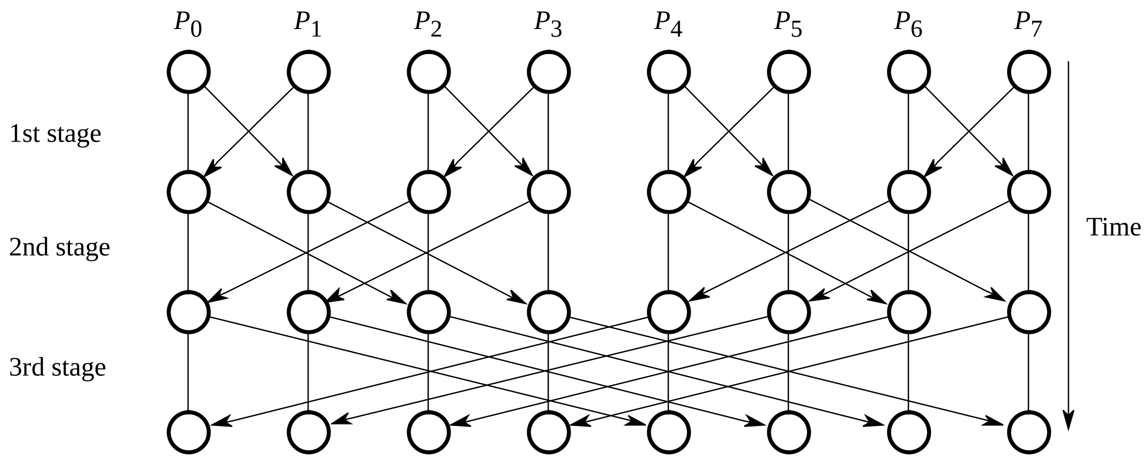


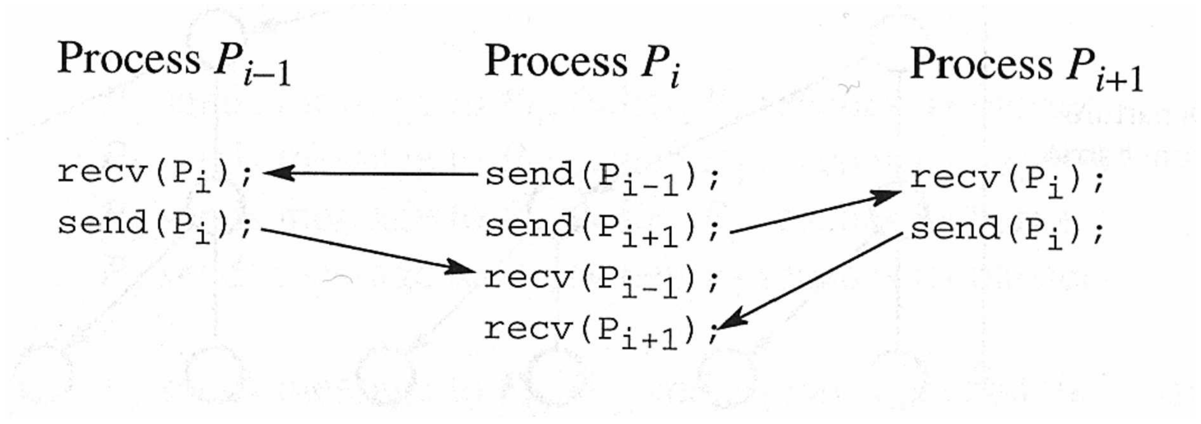
Figure 6.6 Butterfly construction.

Local Synchronization

Sometimes, for a given problem, a process only needs to be synchronized with a subset of processes.

For example, in algorithms when processes are structured as a mesh or pipeline.

Say process P_i needs to exchange data and be synchronized with processes P_{i-1} and P_{i+1}.



Not perfect three process barrier, but may be good enough for many problems.

Deadlock

Deadlock can occur when two processes both send and receive to each other.

For example, if both first perform synchronous sends or blocking send without sufficient buffering.

One solution is to have one first send and the other start with a receive, and then swap.

As bidirection swapping of data common, MPI provides the function *MPI_Sendrecv()* which handles this in such a way to prevent deadlock.

Data Parallel Computations

Data parallel computations have inherent synchronization requirements.

A data parallel computation is when the same operation is applied to separate data elements concurrently (ie. SIMD).

Data parallel programming is useful because:

- Easy to program. Just have to create one program.
- Scales easy to handle big problems.
- A large number of numeric (and some non-numeric) can be cast in this form.

SIMD Computers

SIMD (single instruction stream multiple data stream) computers are data parallel computers.

In a SIMD computer, each processor concurrently performs the same operation, but on different data.

The synchronism is a part of the hardware; each processor executes in lock-step.

Below is an example data parallel computation (k is a constant).

```
for (i = 0; i <n; i++)
   a[i] = a[i] + k;
```

SIMD Computers Cont.

Line $a[i] = a[i] + k$ can be evaluate in parallel by multiple processors, each taking a different value for i (ie. $i \in \{0, 1, \ldots, n-1\}$). See Figure 6.7.

On an SIMD computer, each processor would execute concurrently an instruction equivalent to $a[] = a[] + k$.

As such a parallel operation as above are common, many parallel languages have data parallel operations.

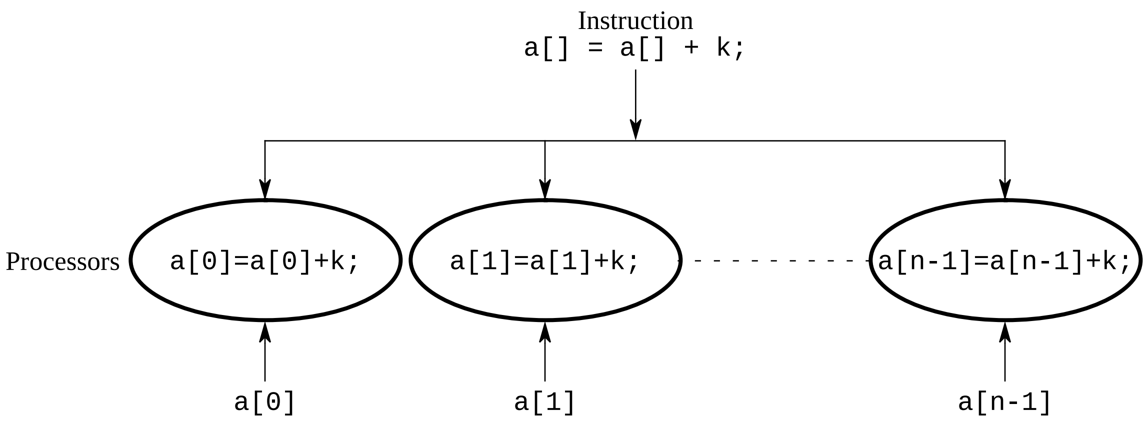


Figure 6.7 Data parallel computation.

Forall Statement

forall operation: A *forall* statement consists of a loop variable, a sequence of values it can take, and a loop body as shown below.

```
forall (i = 0; i <n; i++)
   body
```

The statement means the n instances of the body should be evaluated concurrently, with each instance using a different value for i in i's possible range.

Our example becomes:

```
forall (i = 0; i <n; i++)
   a[i] = a[i] + k;
```

Note: Each instance of the body must be independent of all others.

Forall Statement and Message-Passing

On parallel computer, each instance would run in a different process, but the overall construct will not be completed until all processes complete.

Construct has inherent barrier.

Pseudocode for the SPMD would be:

```
i = myrank;
a[i] = a[i] + k;  /* body */
barrier(mygroup);
```

Body to simple to be practical, but more complex applications such as image processing (Chapter 11) are applicable.

Synchronous Iteration

In sequential programming, iteration important technique.

Used in the *iterative method*, an effective method for solving numerical problems.

Often in numerical problems, we perform a calculation repeatedly.

We use the result from current calculation in the next one.

Can parallelize if there are multiple independent instances of each iteration.

Synchronous Iteration Cont.

Synchronous Iteration: When we solve a problem using iteration and in every iteration we have several processes that begin as a group at the start of the iteration, and the next iteration can not start until every process has completed the current iteration, we call this synchronous iteration.

Pseudocode would be:

```
i = myrank;

for (j = 0; j < n; j++) {
   body(i);
   barrier(mygroup);
}
```

Solving System of Linear Equations by Iteration

Given n equations of n unknowns, we want to solve for the n unknowns.

$$\begin{array}{ccccccc}
a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & \ldots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1} \\
 & & & \vdots & & & & \\
a_{1,0}x_0 & + & a_{1,1}x_1 & \ldots & + & a_{1,n-1}x_{n-1} & = & b_1 \\
a_{0,0}x_0 & + & a_{0,1}x_1 & \ldots & + & a_{0,n-1}x_{n-1} & = & b_0
\end{array}$$

Want to solve for the unknowns $x_0, x_1, \ldots, x_{n-1}$ by using iteration.

If we rearrange the i^{th} equation ($0 \leq i \leq n-1$) by solving for x_i, we get:

$$\begin{aligned}
x_i &= (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + \ldots)] \\
&= \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]
\end{aligned}$$

Jacobi Iteration

The iteration equation $x_i = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j \right]$ gives x_i as a function of the other unknowns.

We can use it iteratively for each of the n unkowns to provide improved approximations. This is called the *Jacobi iteration.*

1. Start with an initial estimate for each x_i, usually b_i.

2. Using the current estimate for each unknown, use the iteration equation to compute a new estimate for each x_i.

3. Once a new estimate for each variable has been found, repeat step 2 until the termination condition has been reached.

Convergence and Termination

One way to ensure that the *Jacobi iteration* converges is to make sure that the array of a constants is *diagonally dominant.* The iterations are guaranteed to converge if:

$$\sum_{j \neq i} \left| a_{i,j} \right| < \left| a_{i,i} \right|$$

A common termination condition would be as below. The notation x_i^t means the value of x_i after the t^{th} iteration.

$$\left| x_i^t - x_i^{t-1} \right| < \text{ error tolerance}$$

Unfortunately, this doesn't mean that our solution will be accurate to that error tolerance. See Figure 6.9.

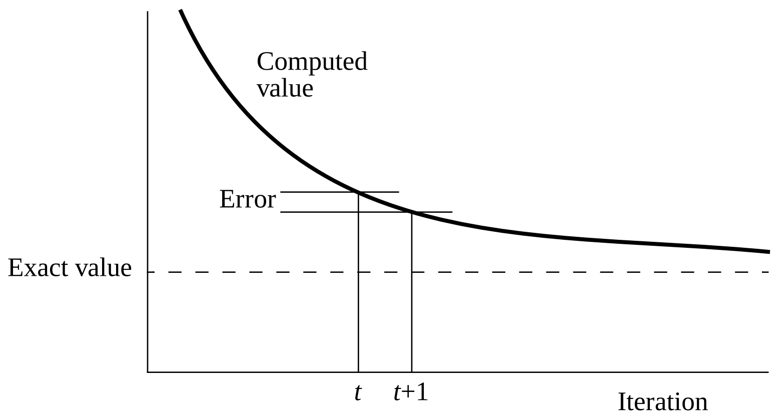


Figure 6.9 Convergence rate.

Convergence and Termination Cont.

Another point is that errors in one unknown will influence other x_i when they are used in the next iteration.

A better termination condition might be:

$$\sqrt{\sum_{i=0}^{n-1} (x_i^t - x_i^{t-1})^2} < \text{ error tolerance}$$

Whichever one chosen, one must also set a maximum number of iterations as the *Jacobi iteration* may not converge or converge too slowly.

Sequential Algorithm

Let arrays a[][] and b[] contain the constants, and x[] hold the unkowns:

```
for (i = 0; i < n; i++)
  x[i] = b[i];   /* initialize unknowns */

/* iterate for fixed number of times */
for (iteration  = 0; iteration < limit; iteration++) {
  /* for each unknown, do: */
  for (i = 0; i < n; i++) {
    sum = -a[i][i] * x[i];
    /* compute summations */
    for (j = 0; j < n; j++)
       sum = sum + a[i][j] * x[j];
    /* compute unknown */
    new_x[i] = (b[i] - sum)/a[i][i];
  }
  /* set new values to current */
  tmp_loc = new_x;
  new_x = x;
  x = tmp_loc;
}
```

Parallel Algorithm

Let each process compute the iteration equation for an unkown. Assume we have a group routine *broadcast_receive()* that send local value to all processes, and receives copy of local value from all other processes.

For Process *Pi:*

```
x[i] = b[i];   /* initialize unknowns */

/* iterate for fixed number of times */
for (iteration  = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    /* compute summations */
    for (j = 0; j < n; j++)
       sum = sum + a[i][j] * x[j];
    /* compute unknown */
    new_x = (b[i] - sum)/a[i][i];
    broadcast_receive(&new_x, x);
    barrier(my_group);
}
```

Parallel Algorithm Cont.

Routine *broadcast_receive()* could be coded as individual send/recv calls, but not efficient ($O(n)$).

Instead, can use the butterfly barrier ($O(\log n)$) not only as a barrier, but to exchange data. Means we can also get rid of the barrier call.

MPI provides library routine *MPI_Allgather()* to do this. See Figure 6.10.

Also, want to iterate until estimations are sufficiently close, not just to some fixed number of iterations.

Need some sort of global *tolerance()* routine, as need all processes to do same number of iterations.

Want each process to iterate until its unknown has converged. *Tolerance()* returns TRUE until all processes have converged.

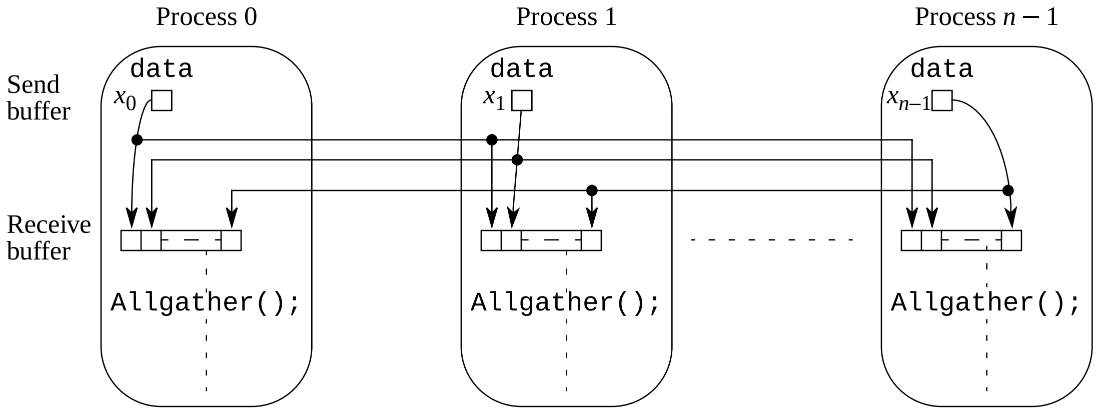


Figure 6.10 Allgather operation.

New Parallel Algorithm

For Process *Pi:*

```
x[i] = b[i];   /* initialize unknowns */
iteration  = 0;

do {
    iteration++;
    sum = -a[i][i] * x[i];
    /* compute summations */
    for (j = 0; j < n; j++)
       sum = sum + a[i][j] * x[j];
    /* compute unknown */
    new_x = (b[i] - sum)/a[i][i];
    /* broadcast value and wait */
    broadcast_receive(&new_x, x);
} while ((tolerance()) && (iteration < limit));
```

Partitioning

If there are more unknowns than processes, we must partition our problem further.

Assign a group of unknowns to be computed to each process.

Easiest solution is to assign unknowns in sequential order:

- Process P0 handles: unknowns x_0 to $x_{n/p-1}$.
- Process P1 handles: unknowns $x_{n/p}$ to $x_{2n/p-1}$ and so on.

Timing Analysis

Sequential time will be the time required for one iteration (evaluating all n unknowns once), times the total number of iterations.

The parallel time will be the time for a single process to perform all of its iterations.

Each process operates on n/p unknowns, and performs τ iterations.

Computation Phase: For each iteration, we have an outer loop executed n/p times, and an inner loop executed n times.

In each pass of the inner loop, we do one addition and one multiplication ($2n$ steps).

Timing Analysis Cont.

In each pass of the outer loop, we have a subtraction and a multiplication before the inner loop, and a subtraction and division after the inner loop (4 steps + $2n$ steps from inner loop).

$$t_{\text{comp}} = n/p(2n+4)\tau$$

If τ is constant, then the computation is $O(n^2/p)$.

Communication Phase: At the end of each iteration, we must transmit the new estimates to all processes. We have p processes that each must broadcast n/p data elements.

$$\begin{aligned} t_{\text{comm}} &= p(t_{\text{startup}} + (n/p)t_{\text{data}})\tau \\ &= (pt_{\text{startup}} + nt_{\text{data}})\tau \end{aligned}$$

More Timing Analysis

Putting it together, we get:

$$\begin{aligned} t_p &= (n/p(2n+4) + pt_{\text{startup}} + nt_{\text{data}})\tau \\ &= (1/p(2n^2+4n) + pt_{\text{startup}} + nt_{\text{data}})\tau \end{aligned}$$

From our computation phase discussion, we can see that:

$$t_s = n(2n+4)\tau = (2n^2+4n)\tau$$

Our speedup factor is:

$$S = \frac{(2n^2+4n)\tau}{(1/p(2n^2+4n) + pt_{\text{startup}} + nt_{\text{data}})\tau}$$

If we take $t_{\text{startup}} = 10,000$ and $t_{\text{data}} = 50$ (from example in Chapter 2), we can examine Figure 6.11 and we see that our minimum execution time occurs for $p = 16$.

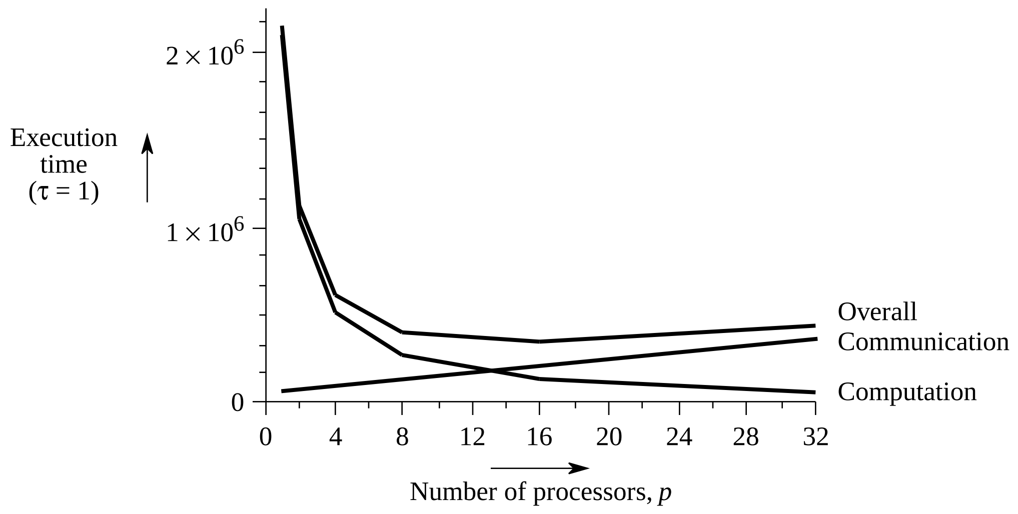


Figure 6.11 Effects of computation and communication in Jacobi iteration.

Heat Distribution Problem (HDP)

We now consider applying parallel techniques to simulating heat distribution.

We have a square piece of metal that we know the temperature values at each edge.

To find the temperature distribution, divide area into 2-dimensional mesh of points, $h_{i,j}$.

Temperature at interior point is average of four surounding points. See Figure 6.12.

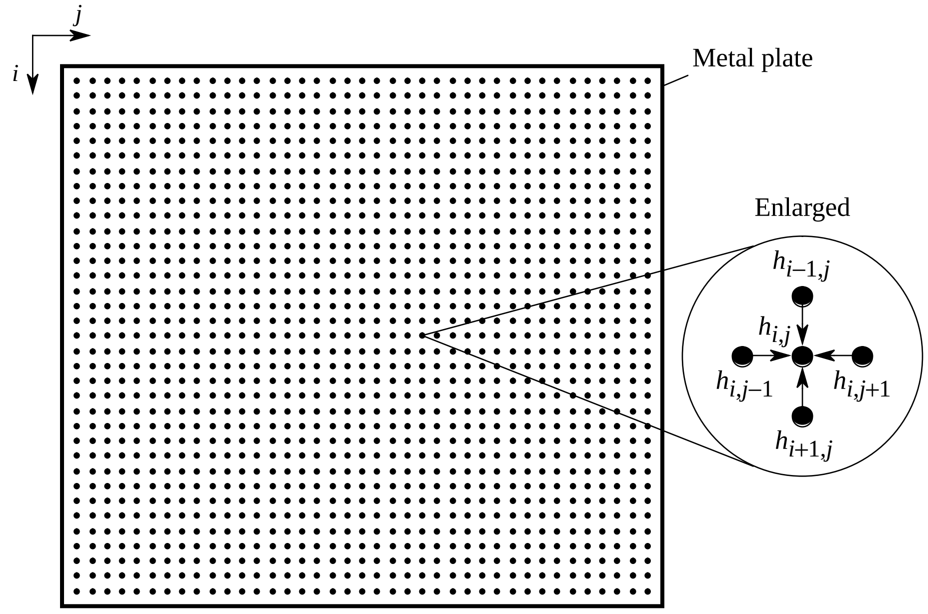


Figure 6.12 Heat distribution problem.

HDP Points Range

For point $h_{i,j}$, the range of values for indices is: $0 \leq i \leq k$, $0 \leq j \leq k$

The indices for the $n \times n$ interior points are: $1 \leq i \leq k-1$, $1 \leq j \leq k-1$

The edge points are when: $i = 0$, or $i = k$, or $j = 0$, or $j = k$

We compute an interior point's temperature by iterating:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

Stop after a set number of iterations or until difference between iterations is less than the desired error tolerance.

HDP Sequential Code

Assume temperature of each point held in array $h[i][j]$

Initialize boundary points $h[0][x]$, $h[x][0]$, $h[k][x]$, $h[x][k]$ ($0 \leq x \leq k$) to edge temperatures.

Initialize all other points to some initial default value such as room temperature.

```
for (iteration = 0;  iteration < limit; iteration++){
  for (i = 1;  i < k; i++)
    for (j = 1;  j < k; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                h[i][j-1] + h[i][j+1]);
  for (i = 1;  i < k; i++)
    for (j = 1;  j < k; j++)
       h[i][j] =  g[i][j]; }
```

Terminiation and Precision

If want to stop at a desired precision, need All points to have reached desired error tolerance.

```
do {
  for (i = 1;  i < k; i++)
    for (j = 1;  j < k; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                h[i][j-1] + h[i][j+1]);
  for (i = 1;  i < k; i++)
    for (j = 1;  j < k; j++)
       h[i][j] =  g[i][j];

  continue = FALSE;
  for (i = 1;  i < k; i++)
    for (j = 1;  j < k; j++)
      if (!converged(i,j)) {
        continue = TRUE;
        break;
      }

} while (continue == TRUE);
```

HDP Parallel Version

Want to evaluate multiple points at once, then synchronize with 4 nearest neighbours.

Initially, we assume one process per point. Processes arranged in a mesh (subscripts: row, column).

```
iteration = 0;    /* for process i,j */
do {
  iteration++;
  g = 0.25 * (w + x+ y + z);
  send(&g, P_i-1,j);   /* non-blocking send  */
  send(&g, P_i+1,j);
  send(&g, P_i,j-1);
  send(&g, P_i,j+1);
  recv(&w, P_i-1,j);   /* synchronous receives  */
  recv(&x, P_i+1,j);
  recv(&y, P_i,j-1);
  recv(&z, P_i,j+1);
} while ((!converged(i,j)) && (iteration < limit));
send(&g, &i, &j, &iteration,  P_Master);
```

See Figure 6.14.

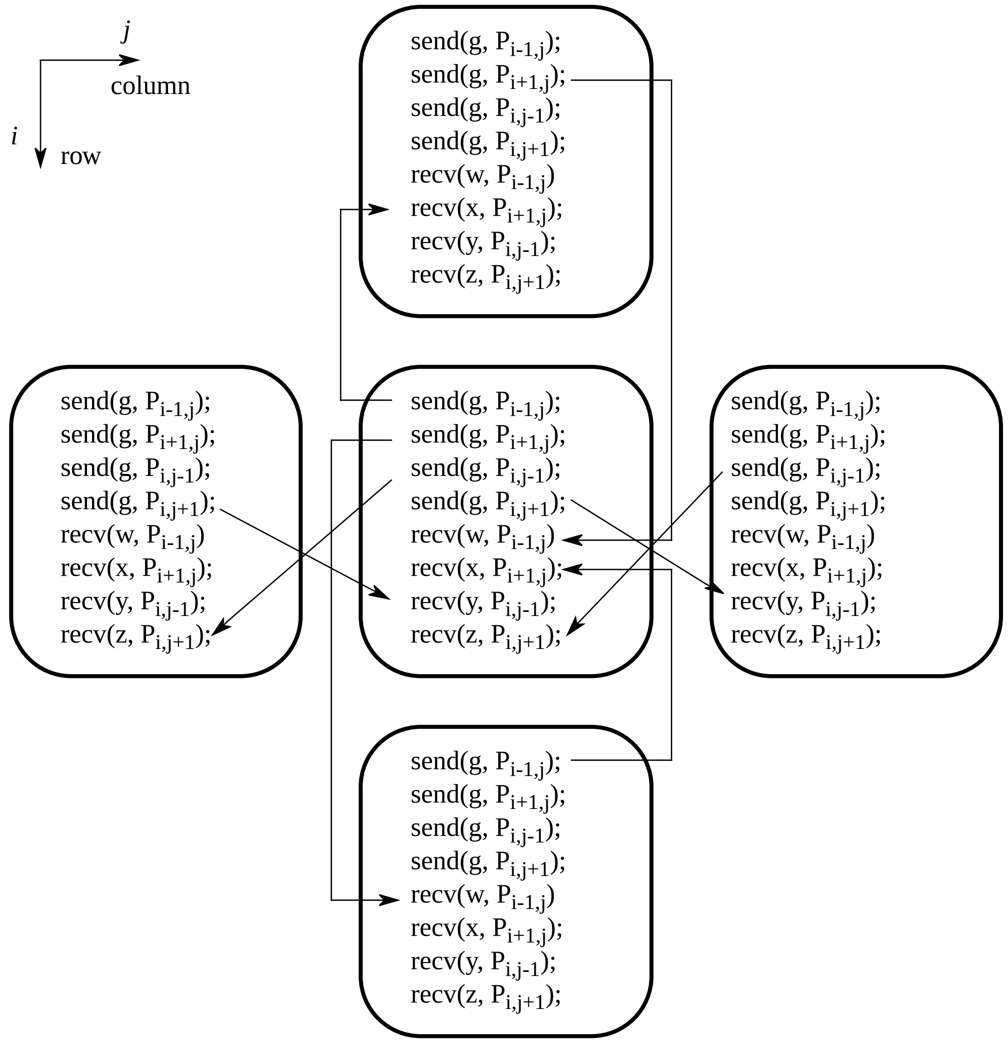


Figure 6.14 Message passing for heat distribution problem.

Interior Edge points.

Can use process IDs to determine which points are edge points.

We only calculate values for the interior points.

```
if (first_row) w = bottom_value;
if (last_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;    /* for process i,j */
do {
  iteration++;
  g = 0.25 * (w + x+ y + z);
  if !(first_row) send(&g, P_i-1,j);
  if !(last_row) send(&g, P_i+1,j);
  if !(first_column) send(&g, P_i,j-1);
  if !(last_column) send(&g, P_i,j+1);
  if !(first_row) recv(&w, P_i-1,j);
  if !(last_row) recv(&x, P_i+1,j);
  if !(first_column) recv(&y, P_i,j-1);
  if !(last_column) recv(&z, P_i,j+1);
} while ((!converged(i,j)) && (iteration < limit));
send(&g, &i, &j, &iteration,  P_Master);
```

Partitioning HDP

Easy to map the n^2 interior mesh points to process indices.

Process zero (P_0) is top left of mesh ($P_{1,1}$), then process numbered across rows in natural order.

Means process P_i exchanges data with P_{i-1} (left), P_{i+1} (right), P_{i-n} (above), P_{i+n} (below) $0 \leq i \leq n^2 - 1$.

However, likely that we have only $p < n^2$ processes and we have to partition our set of points.

Partitioning HDP Cont.

Must allocate blocks of points to each process.

Blocks are usually either square blocks or strips as in Fig 6.15.

Want to minimize communication between partitions.

With n^2 points, p processes, and assuming equal partitions, we have n^2/p points per partition.

Process must transmit at each edge $\frac{n}{\sqrt{p}}$ points for square block, and n points for strip. See Figure 6.16

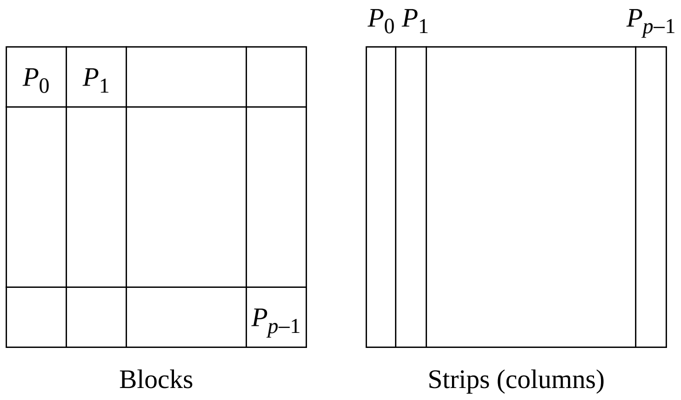


Figure 6.15 Partitioning heat distribution problem.

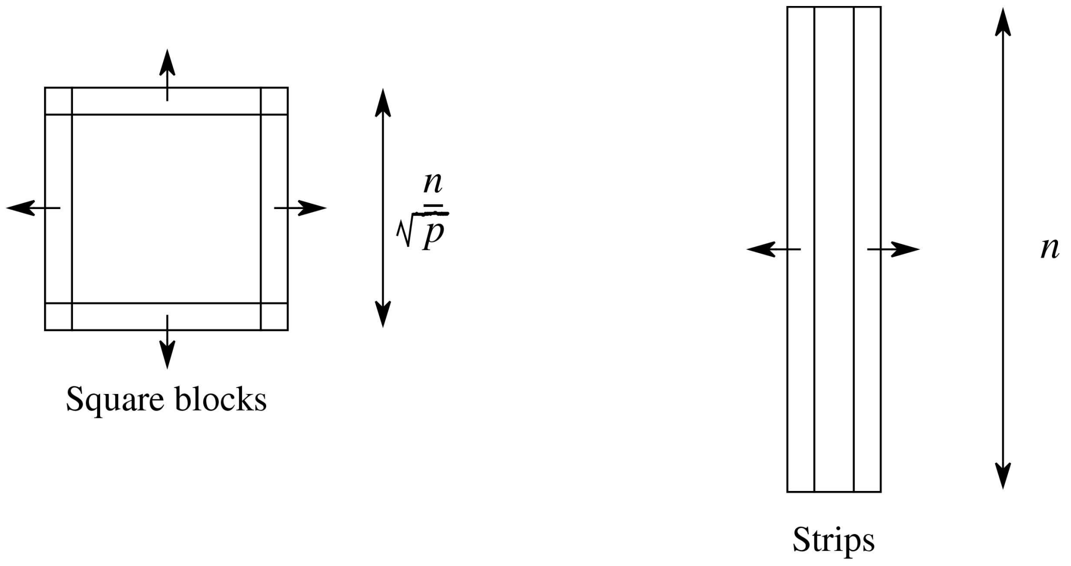


Figure 6.16 Communication consequences of partitioning.

Communication Time

Block partitions have 4 edges where data exchanged.

Each process sends 4 messages and receives 4 messages per iteration.

$$t_{\text{commsq}} = 8(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}})$$

Strip partitions have only two edges, thus only two messages are sent then two received per iteration.

$$t_{\text{commcol}} = 4(t_{\text{startup}} + n t_{\text{data}})$$

Example Times

Eg. 1: Let $t_{\text{startup}} = 10\,000$, $t_{\text{data}} = 50$, and $n^2 = 1024$

This gives: $t_{\text{commsq}} = 80\,000 + \frac{12\,800}{\sqrt{p}}$ time units and $t_{\text{commcol}} = 46,400$.

Block partitions will take more time for any value of p.

Eg. 2: As above but $t_{\text{startup}} = 100$.

This gives: $t_{\text{commsq}} = 800 + \frac{12\,800}{\sqrt{p}}$ time units and $t_{\text{commcol}} = 6800$.

Strip partitions will have larger time for $p > 4$.

Crossover Point

Strip partition has lower communication time as long as:

$$8(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}) > 4(t_{\text{startup}} + n t_{\text{data}})$$

which is equivalent to:

$$t_{\text{startup}} > n\left(1 - \frac{2}{\sqrt{p}}\right) t_{\text{data}}$$

With $t_{\text{startup}} = 1200$, $t_{\text{data}} = 50$, and $n^2 = 1024$, the crossover point is when $p = 64$ (see also Figure 6.17).

One can solve for the crossover point of the above equation by replacing "$>$" by "$=$" and solving for p.

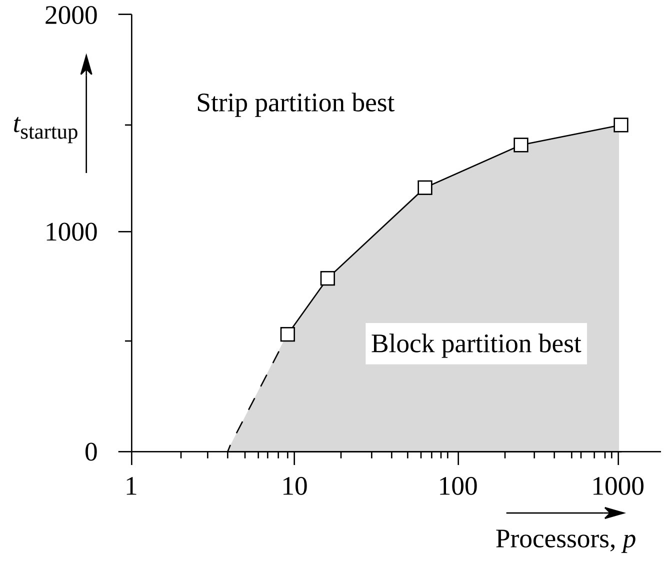


Figure 6.17 Startup times for block and strip partitions.

Implementation

Rather than partion into columns, we will partition by rows as we will be using the 'C' language.

Each process will have an array containing its $\frac{n}{p}$ rows of data points it must compute.

Process also has an additional row (1 per adjacent edge) of "ghost points" to hold the data row from its adjacent edge. See Figure 6.18.

```
for (i = 1;  i <= n/p; i++)
  for (j = 1;  j <= n; j++)
    g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
             h[i][j-1] + h[i][j+1]);
for (i = 1;  i <=n/p; i++)
  for (j = 1;  j <= n; j++)
     h[i][j] =  g[i][j];

send(&g[1][1], n, P_i-1);   /* send rows to */
send(&g[n/p][1], n, P_i+1); /* adjacent process */
recv(&h[0][1], n, P_i-1); /* recv rows from */
recv(&h[n/p +1][1], n, P_i+1); /*adjacent process */
```

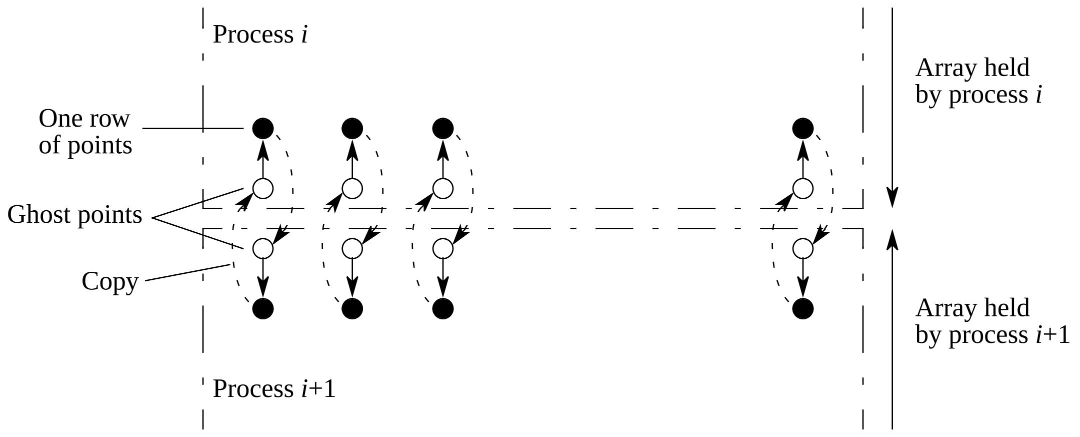


Figure 6.18 Configurating array into contiguous rows for each process, with ghost points.

Deadlock

The data exchange just described is considered unsafe in the MPI setting.

Method relies on the amount of buffering available in the *send()* routines.

If insufficent, process will block until enough data has been received by the other process to copy the remaining data to an internal buffer.

As the other process is also trying to send data before it can receive, they deadlock.

Deadlock Solutions

- Alternate send and receives of adjacent processes. ie. only one process does a send first, the other receives etc.
- Combined send/receive functions like: MPI_Sendrecv().
- Buffered sends such as: MPI_Bsend(). User provides explicit storage space for buffers.
- Globally nonblocking routines such as: MPI_ISend() and MPI_Irecv().

Deadlock Solutions II

Alternating send()s and rec()s:

```
if ((i % 2) == 0) {   /* even process */
  send(&g[1][1], n, P_i-1);
  recv(&h[0][1], n, P_i-1);
  send(&g[n/p][1], n, P_i+1);
  recv(&h[n/p +1][1], n, P_i+1);
} else {
  recv(&h[n/p +1][1], n, P_i+1);
  send(&g[n/p][1], n, P_i+1);
  recv(&h[0][1], n, P_i-1);
  send(&g[1][1], n, P_i-1);
}
```

Segment for using nonblocking routines:

```
isend(&g[1][1], n, P_i-1);
isend(&g[n/p][1], n, P_i+1);
irecv(&h[0][1], n, P_i-1);
irecv(&h[n/p +1][1], n, P_i+1);
waitall(4);
```

Prefix Sum Problem

The *prefix sum* problem is when we are given a set of n numbers, $x_0, x_1, \ldots, x_{n-1}$, and we want to compute all of the partial sums:

ie. $S_1 = x_0 + x_1; S_2 = x_0 + x_1 + x_2; \ldots$

Can replace addition with any other associative operator (multiplication, subtraction etc.)

Sequential code is:

```
for (i = 0; i <n; i++) {
  sum[i] = 0;
  for(j =0; j <= i; j++)
    sum[i] = sum[i] + x[j];
}
```

Algorithm is $O(n^2)$.

Prefix Sum Problem Cont.

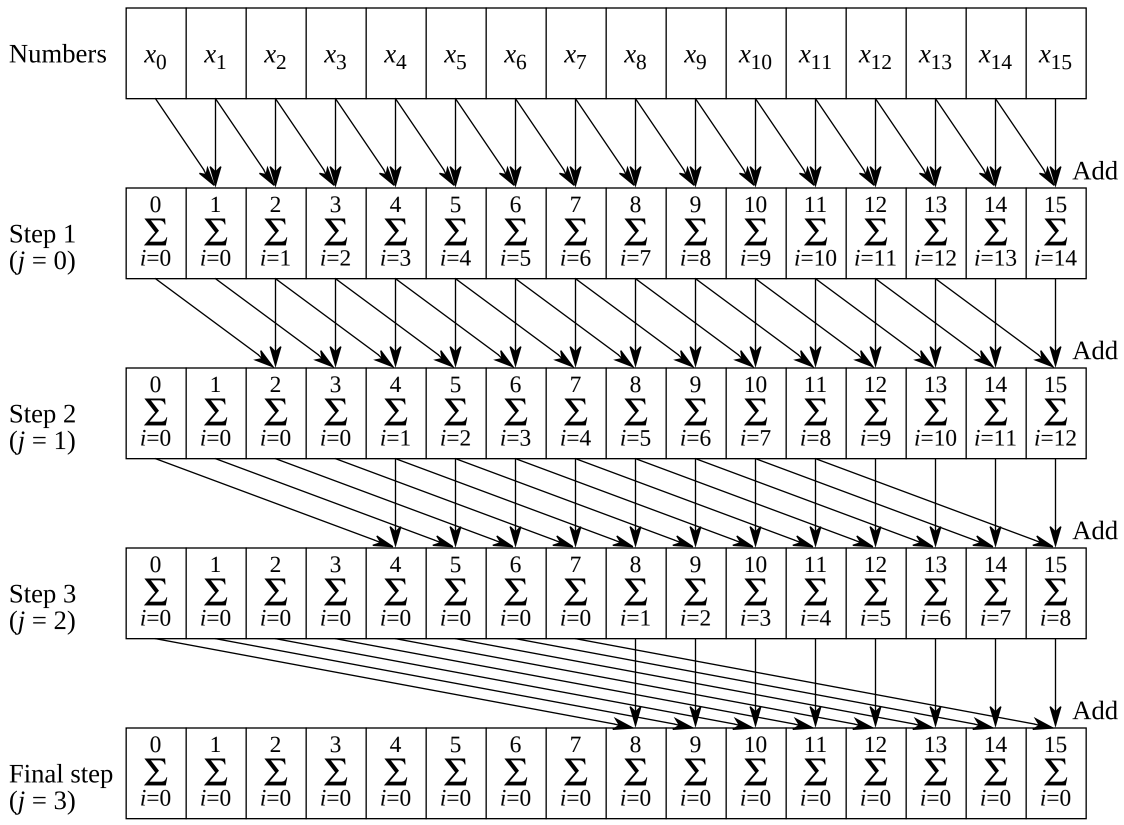
Figure 6.8 shows data parallel approach to calculate all partial sums for 16 numbers.

Uses multiple treelike constructions and computes partial sums storing values at $x[i]$ with $0 \leq i \leq 16$).

Note: Original values of array x lost.

```
Step 1: 15 (16-1) adds occur (x[i] + x[i-1]) for 1<=i<16
Step 2: 14 (16-2) adds occur (x[i] + x[i-2]) for 2<=i<16
Step 3: 12 (16-4) adds occur (x[i] + x[i-4]) for 4<=i<16
Step 4: 8 (16-8) adds occur (x[i] + x[i-8]) for 8<=i<16
```

For n numbers, need $\log n$ steps.



Numbers
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
Add
Step 1 (j = 0)
Add
Step 2 (j = 1)
Add
Step 3 (j = 2)
Add
Final step (j = 3)

Figure 6.8 Data parallel prefix sum operation.

Sequential & Parallel Algorithm

At step $j \in \{0, 1, \ldots, (\log n) - 1\}$, we do $n - 2^j$ adds with $x[i - 2^j]$ and $x[i]$ ($i \in \{2^j, 2^j + 1, \ldots, n - 1\}$)

Sequential algorithm:

```
for (j = 0; j < log(n); j++)
  for (i = 2^j; i <n ; i++)
    x[i] = x[i] + x[i - 2^j];
```

Parallel pseudocode:

```
i = myrank;

for (j = 0; j < log(n); j++) {
  if (i >= 2^j)
     xtmp = x[i] + x[i - 2^j];
  barrier(mygroup);
  x[i] = xtmp;
  barrier(mygroup);
}
```

Complexity is $O(\log n)$ if we ignore the complexity from the barriers (SIMD computer). Otherwise it's $O((\log n)^2)$. Efficiency is less than 100% as we use fewer processors as we go along.