*

Partitioning and Divide-and-Conquer Strategies

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

Strategies

We now discuss two of the most fundamental approaches in parallel programming.

Partitioning: Problem divided into distinct parts and then each part is separately computed.

Divide and Conquer: Uses partitioning , but in a recursive manner.

The problem is continually divided into smaller and then smaller parts. The smallest parts are then solved and the results combined.

Partitioning Strategies

Partitioning simply divides the problem into parts.

Basis of all parallel programs in one form or another.

Unlike the partitioning done for the embarrassingly parallel case, the general case requires that the results of the individual parts have to be combined to obtain the final result.

Two common approaches to partition the problem:

Data Partitioning: This is when the data is partitioned. The program data is divided and operated upon in parallel.

This is also called *domain decomposition.*

Partitioning Strategies Cont.

Functional Decomposition: This is when the function of the program is partitioned. The program is divided into separate functions which then run in parallel.

Not common to have concurrent functions in a problem, but data partitioning is an often used strategy.

Data Partitioning

Simple example: suppose want to add a sequence of numbers $x_0, \ldots, x_{n-1}$.

Recurring problem in text to demonstrate concept. Not worthwhile to parallelize unless n is large.

Approach can be used on large databases to do complex calculations.

Could divide sequence into m parts with n/m numbers in each part.

ie. $(x_0 \ldots x_{(n/m)-1}), (x_{n/m} \ldots x_{(2n/m)-1}), \ldots (x_{(m-1)n/m} \ldots x_{n-1})$

Next, m processes would each add a sequence creating m partial sums.

The m partial sums are then added together. See Figure 4.1.

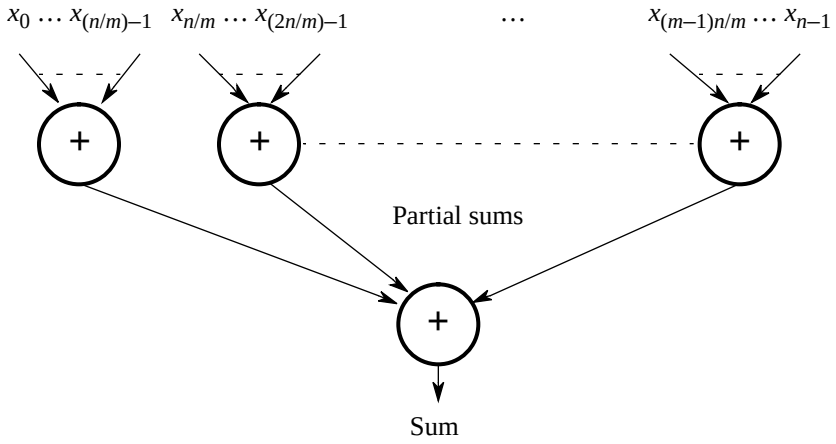

Figure 4.1 Partitioning a sequence of numbers into parts and adding the parts.

Data Partitioning Cont.

Need to distribute the required portion of numbers to each processor.

With MPI, would require data passed individually to processes. A shared-memory processor could simply access the data from a commonly accesible location.

Parallel code for master/slave simple:

- Master sends numbers to slaves.
- Slaves add their numbers in parallel.
- Partial sums sent to master.
- Master adds partial sums to obtain results.

Individual Send or Broadcast

Whether broadcast entire list or send individually to processes, send the same amount of data.

Which is better, depends on broadcast method.

Broadcast has single startup time, and thus may be better than multiple sends.

Using broadcast, increases complexity of every slave as it must determine itself which portion of the problem to handle.

Also, each slave must have enough memory to hold all of the data.

Individual Send

Pseudocode for addition that uses separate send/recvs.

Master

```
s = n/m;                              /* number of numbers for slaves*/
for (i = 0, x = 0; i < m; i++, x = x + s)
   send(&numbers[x], s, P_i);         /* send s numbers to slave */

sum = 0;
for (i = 0; i < m; i++) {             /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;              /* accumulate partial sums */
}
```

Slave

```
recv(numbers, s, P_master);           /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)               /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);            /* send sum to master */
```

*

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

Using Broadcast Cont.

Pseudocode using broadcast.

Master

```
s = n/m;                                  /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);         /* send all numbers to slaves */
sum = 0;
for (i = 0; i < m; i++){                  /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
bcast(numbers, s, P_master);              /* receive all numbers from master*/
start = slave_number * s;                 /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)             /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

Scatter and Reduce

If using scatter and reduce, pseudocode would be:

Master

```
s = n/m;                                          /* number of numbers */
scatter(numbers,&s,P_group,root=master);          /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);          /* results from slaves */
```

Slave

```
scatter(numbers,&s,P_group,root=master);          /* receive s numbers */
.                                                 /* add numbers */
reduce_add(&part_sum, 1 ,P_group,root=master);/* send sum to master */
```

Instead of addition, could have done many other operations such as:

- Found maximum number of each group, then master finds maximum of results.
- Number of occurrences of a number (character, string etc.) in a group can be found and given to master.

Timing Analysis

Sequential computation requires $n - 1$ additions, thus $O(n)$.

For parallel we have $m + 1 processes$.

Phase 1: Communication. Evaluate time for the m slaves to receive their n/m numbers.

For separate send/recs, we'd get:

$$\begin{aligned} t_{\text{comm1}} &= m(t_{\text{startup}} + (n/m)t_{\text{data}}) \\ &= m\,t_{\text{startup}} + n\,t_{\text{data}} \end{aligned}$$

If we used scatter, we might get:

$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

The results dependent on implementation of scatter. Either case, time complexity is $O(n)$.

Timing Analysis Cont.

Phase 2: Computation. Each slave must add n/m numbers.

$$t_{\text{comp1}} = (n/m) - 1$$

Phase 3: Communication. Sending partial sums to master.

Using separate send/recs:

$$\begin{aligned} t_{\text{comm2}} &= m(t_{\text{startup}} + t_{\text{data}}) \\ &= m\,t_{\text{startup}} + m\,t_{\text{data}} \end{aligned}$$

Using gather or reduce might give:

$$t_{\text{comm2}} = t_{\text{startup}} + mt_{\text{data}}$$

Phase 4: Computation. The master now adds the m partial sums:

$$t_{\text{comp2}} = m - 1$$

More Timing Analysis

The overall runtime with send and receive is:

$$\begin{aligned} t_p &= (t_{\text{comm1}} + t_{\text{comm2}}) + (t_{\text{comp1}} + t_{\text{comp2}}) \\ &= (m\,t_{\text{startup}} + n\,t_{\text{data}} + m\,t_{\text{startup}} + m\,t_{\text{data}}) \\ &\quad +(n/m - 1 + m - 1) \\ &= 2m\,t_{\text{startup}} + (n+m)t_{\text{data}} + m + n/m - 2 \end{aligned}$$

Thus t_p is $O(n+m)$.

The parallel algorithm actually has worse time complexity than sequential.

Ignoring communication, the speedup factor, S, is:

$$S = \frac{t_s}{t_p} = \frac{n-1}{n/m + m - 2}$$

OK for large n, but very low for smaller n.

Divide and Conquer

Characterized by breaking a problem down into subproblems of the same form.

These subproblems are in turn broken down, and so on, until the tasks are simple enough to perform.

Normally done by recursion.

The simple tasks are performed and the results are combined to obtain larger results, and these results are combined, and so on.

One way to determine if a method is partitioning or divide and conquer is as follows:

Partitioning: A method is partitioning when the majority of work is done in dividing the problem.

Divide and Conquer Cont.

Divide and Conquer: When the majority of work is performed in combining the results.

We will instead categorize a method as divide and conquer when the initial partitioning is continued to make smaller and smaller problems.

Sequential Recursive Approach

Pseudocode for a sequential recursive method for adding group of numbers is:

```
int add(int *s) {  /* add #s in list s */

                /* see defn of n1, n2 below */
  if ( number(s) =< 2) return (n1 + n2);
  else {
    /* divide s into lists s1 and s2 */
    Divide(s, s1, s2);
    /* recursive call to add sublist */
    part_sum1 = add(s1);
    part_sum2 = add(s2);
    return (part_sum1 + part_sum2);
  }
}
```

If 2 #, we name them $n1$ and $n2$. If one, it's $n1$ and we set $n2 = 0$. None, then $n1 = n2 = 0$.

Can extend to find maximum value or to sort list (ie. mergesort and quicksort sorting algorithms) etc.

Wouldn't use recursion when could use iterative, but our method can be applied to anything that can be formulated as above.

Tree Construction

When problem is divide into two parts, divide and conquer forms binary tree.

Tree traversed downwards when function calls made, and upwards when the functions return. See Figure 4.2.

The tree construction can be used to divide the list into first two, and then four parts and so on until every process has an equal part.

Once pairs at bottom are added, the numbers are added in a reverse tree traversal.

Figure 4.2 shows a complete tree. Only happens if tree can be divided into a number of parts that is power of 2.

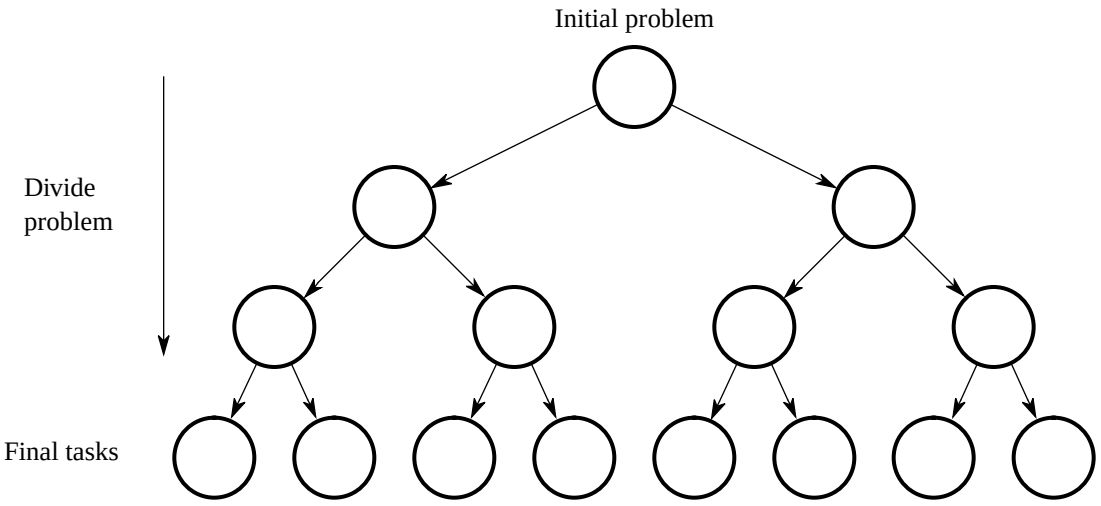

Figure 4.2 Tree construction.

Parallel Implementation

For sequential case, can only visit one node of tree at a time.

In parallel implementation, can traverse several parts simultaneously.

Once problem divided, both parts can be processed concurrently.

If visualize problem as a tree, don't need recursion.

One solution is to assign a process per node. For a tree of height m, this would require $2^{m+1}-1$ nodes to break the problem down into 2^m parts. Inefficient.

Parallel Implementation Cont.

Better solution is to reuse processes at each stage in tree.

Division ends when all processes are committed.

At each stage, a process keeps half of list, and passes other half on.

Consider Figure 4.3 which uses 8 processes.

Each list at leaves will have n/p numbers, when we have p processes.

We thus have a tree of height $\log p$.

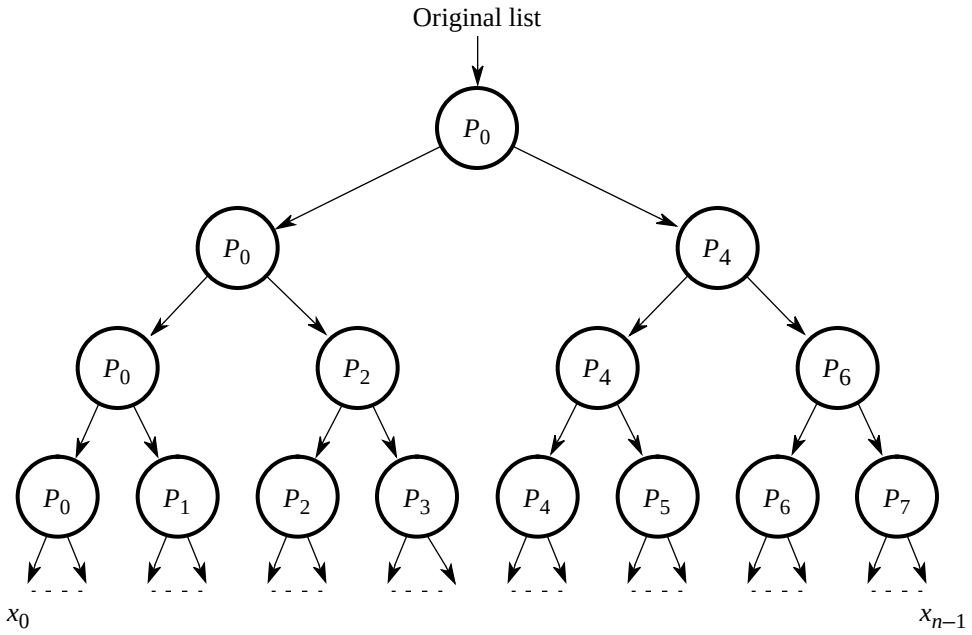

Figure 4.3 Dividing a list into parts.

Combining the Results

Once partial sums at the leafs have been computed, every odd number process sends it's sum to the adjacent even numbered process.

The even numbered process then adds the partial sum with its own and passes the results on.

Stops when Process 0 has final result.

Figure 4.4 can show how the partial sums can be combined back to produce the final results.

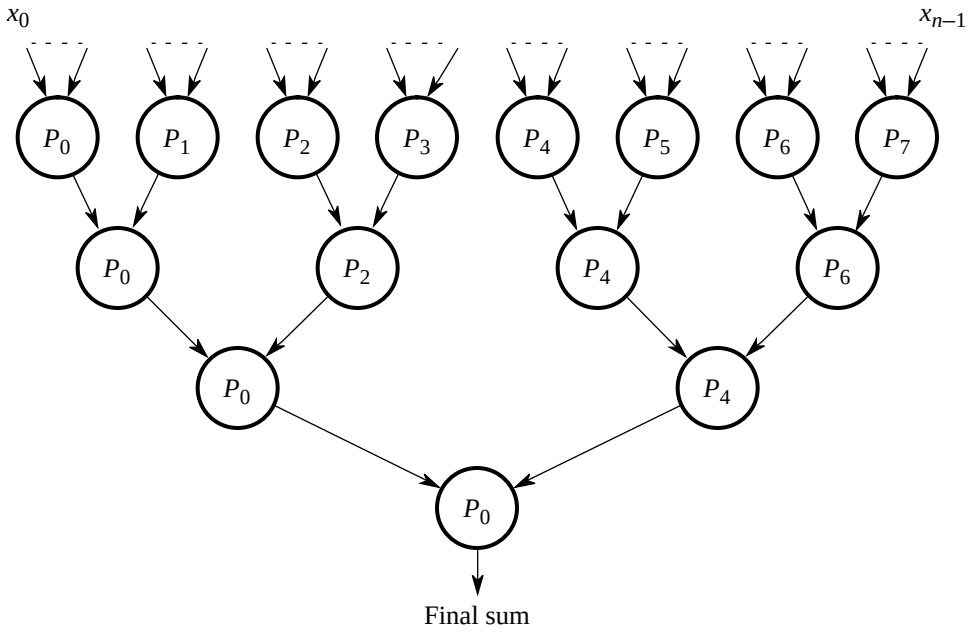

Figure 4.4 Partial summation.

Relationship to Hypercube

Same communication pattern as used in a binary hypercube broadcast and gather algorithm from slides5.

Processes that are suppose to communicate with other processes can be found from their binary address ($000 - 111$ for $p = 8$).

Processes that communicate with each other differ by one bit.

In division of data phase, this starts with the most significant bit (left-most bit). Next, processes that differ in the bit to the right of the most significant bit communicate, and so on.

In combining stage, starts with least significant bit.

Parallel Pseudocode

Pseudo code for process 0 would look like:

Process P_0

```
                          /* division phase */
divide(s1, s1, s2);       /* divide s1 into two, s1 and s2 */
send(s2, P4);             /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1};
part_sum = *s1;           /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

Parallel Pseudocode Cont.

Code for process 4 would be as below.

Process P_4

```
recv(s1, P0);              /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;            /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

Similar for other processes.

Timing Analysis

Assume that n (number of elements to add) is a power of two.

Leave as exercise to include t_{startup} (communication setup time).

Division phase only communication. Combining phase both.

Communication: For division phase, we have $logp$ steps, p the number of processes.

$$\begin{aligned} t_{\text{comm1}} &= \frac{n}{2}t_{\text{data}} + \frac{n}{4}t_{\text{data}} + \frac{n}{8}t_{\text{data}} + \cdots + \frac{n}{p}t_{\text{data}} \\ &= \frac{n(p-1)}{p}t_{\text{data}} \end{aligned}$$

For the combining phase, only send one piece of data:

$$t_{\text{comm2}} = t_{\text{data}} \log p$$

Timing Analysis Cont.

Computation: Once divide phase over, the n/p numbers added together. Next, one addition occurs at each stage of combining phase.

$$t_{\text{comp}} = \frac{n}{p} - 1 + \log p$$

For constant p, we get time complexity $O(n)$. For variable p, and large n, we get $O(n/p)$.

Total parallel execution time is:

$$t_p = \frac{n(p-1)}{p} t_{\text{data}} + t_{\text{data}} \log p + \frac{n}{p} - 1 + \log p$$

More Timing Analysis

Ignoring communication, and we get a scaling factor of p for constant p and large n.

$$S = \frac{t_s}{t_p} = \frac{n-1}{\frac{n}{p} - 1 + \log p}$$

Comparing this to partitioning (m = p-1), and we see a definite improvement:

$$S = \frac{n-1}{n/m + m - 2} = \frac{n-1}{n/(p-1) + p - 3}$$

Other Operators

Clearly, other associative operators could replace the sum (ie. subtraction, multiplication, logical OR etc).

Tree construction can also be used for operations such as a search.

Information passed up is boolean flag asserting if condition or a specific item has been found.

Operation at each node would then be an OR function as in Figure 4.5.

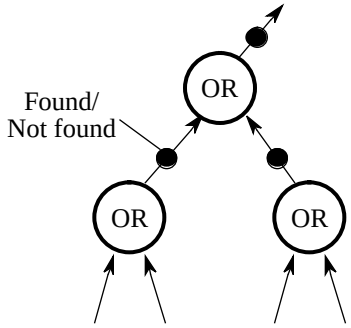

Figure 4.5 Part of a search tree.

M-ary Divide and Conquer

Can also use when tasks divided at each stage into > 2 parts.

ie. the task can be divided into four parts at each stage.

The recursive sequential formulation would be:

```
int add(int *s) {  /* add #s in list s */

  if ( number(s) =< 4) return (n1 + n2 + n3 + n4);
  else {
    /* divide s into lists s1, s2, s3, s4 */
    Divide(s, s1, s2, s3, s4);
    /* recursive call to add sublist */
    part_sum1 = add(s1);
    part_sum2 = add(s2);
    part_sum3 = add(s3);
    part_sum4 = add(s4);
    return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
  }
}
```

M-ary Divide and Conquer Cont.

This leads to a tree where each non-leaf node has four children. Called a *quadtree.* See Figure 4.6.

A quadtree is used in decomposing a 2-dimensional area into four subregions.

This would be done with a digital image. The image would be divided into four quadrants.

The quadrants would then each be divided into four subquadrants. See Figure 4.7.

An *octtree* is a tree when each non-leaf node has 8 children. Useful to divide a 3-dimensional space.

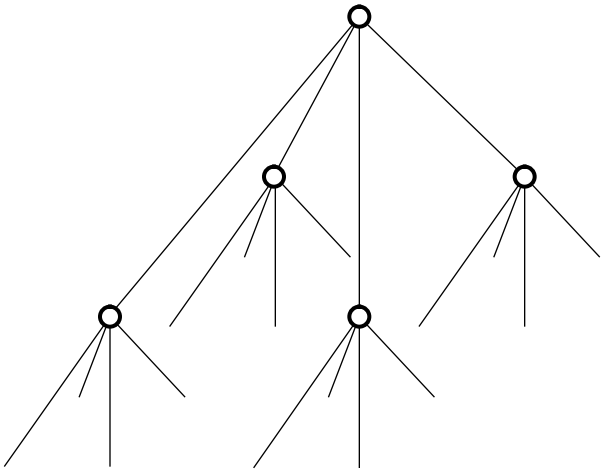

Figure 4.6 Quadtree.

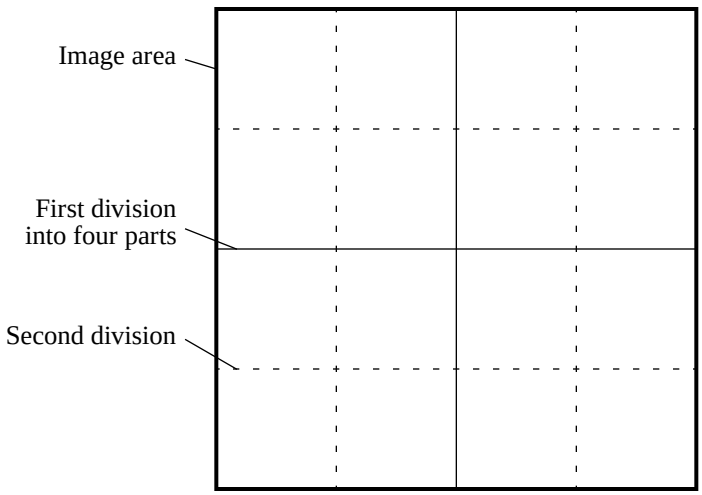

Figure 4.7 Dividing an image.

M-ary Tree

Generalizes to a *m-ary tree*, a tree where each non-leaf node has m children.

The fact that we divide a problem into m parts at each stage, suggests that we are exposing more parallelism at each stage.

Cons: For $m > 2$, the algorithm is more complicated.

Particularly if the architecture isn't well suited, this may not give an increase in parallelism.

Pros: Using $m > 2$ may be a more natural fit with the problem.

If the architecture is well suited, then can expose more parallelism sooner.

Sorting Using Bucket Sort

Instead of just adding group of numbers, we could sort list into numerical order.

Most sequential sorting algorithms based on compare and swapping of a pair of numbers.

Will examine a method called *bucket sort* which doesn't use this method.

It is actually a partitioning method.

Sorting Using Bucket Sort Cont.

Bucket sort only works well if numbers are uniformally distributed across a specified interval. ie. 0 to $a-1$

We then break this interval into m equal regions: 0 to $a/m-1$, a/m to $2a/m-1$, $2a/m$ to $3a/m-1, \ldots$

We assign one "bucket" to hold the numbers that belong to each region, thus m buckets.

We place each number in the appropriate bucket, and then sort the bucket.

We will use a limited number of buckets, and then sort the buckets using a sequential algorithm, and then merge the lists. See Figure 4.8.

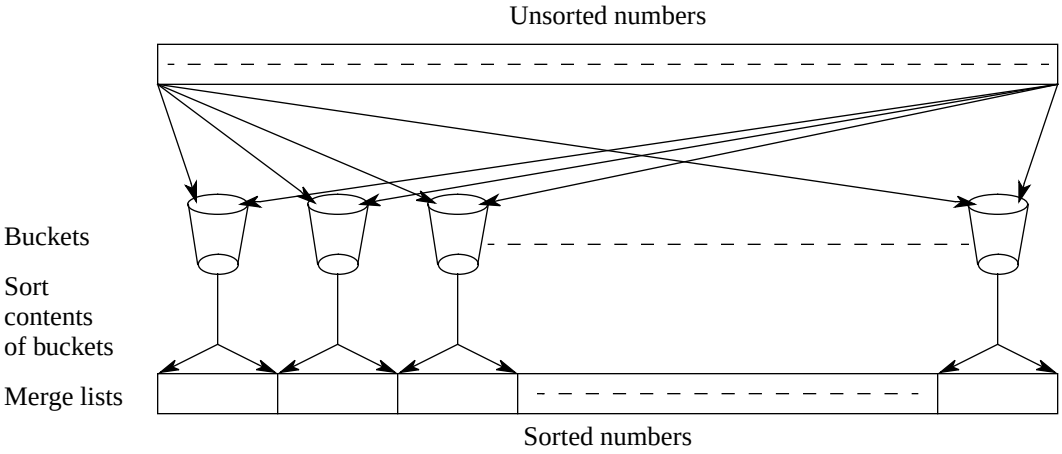

Figure 4.8 Bucket sort.

Sequential Algorithm

Fist step: place each number in their bucket. Requires determining region number falls in.

- Could compare number to start value of each region $(a/m, 2a/m, \ldots)$.

  Would require $m-1$ steps for every number.

- Better way is to divide each number by n/m, and quotient gives bucket.

  Gives 1 step per number, but division may be slow.

- If m a power of two, can derive bucket by looking at upper digits in binary.

  If $m = 8 = 2^3$, then number 1100101 falls into region $110 = (6)_{10}$. Simply take three most significant bits.

Sequential Algorithm Cont.

We will thus assume that placing a number in a bucket takes one step, thus n steps to place all numbers. Will have n/m numbers per bucket.

Step two: Each bucket must be sorted.

Sequential algorithms such as quicksort and mergesort are $O(n \log n)$ to sort n numbers.

We will assume that sequential algorithm needs $n \log n$ comparisons, and we'll take one comparison to be one computation step.

To sort n/m numbers in each bucket will thus take $(n/m) \log(n/m)$ steps.

Sequential Algorithm Analysis

After numbers in buckets are sorted, they must be contenated into one list. We will assume this takes no additional steps.

Combining all steps together, our serial time is:

$$t_s = n + m((n/m)\log(n/m)) = n + n\log(n/m)$$

This is $O(n\log(n/m))$. If $n = km$, k being some constant, then we have $O(n)$.

Only applies if numbers are properly distributed.

Parallel Algorithm

Easiest way to parallelize is to use one process per bucket.

This changes the $n \log(n/m)$ term in the t_s equation to $n/p \log(n/p)$, where $p = m$ is the number of processes. See Figure 4.9.

Lots of wasted effort as each process must examine all n numbers to determine which ones belong in its bucket.

Better approach is to divide the n unsorted numbers in the list into p regions, one per process.

Parallel Algorithm Cont.

Each process has p small buckets into which it puts the numbers from its region. Each Process also has one large bucket to contain all the numbers from the list that fall into its sorting interval.

The contents of the small buckets are then sent to the appropriate process. ie. small bucket i is sent to process i. See Figure 4.10.

This requires the following phases:

1. Partition numbers.

2. Sort numbers into small buckets.

3. Empty small buckets into large buckets.

4. Sort large buckets.

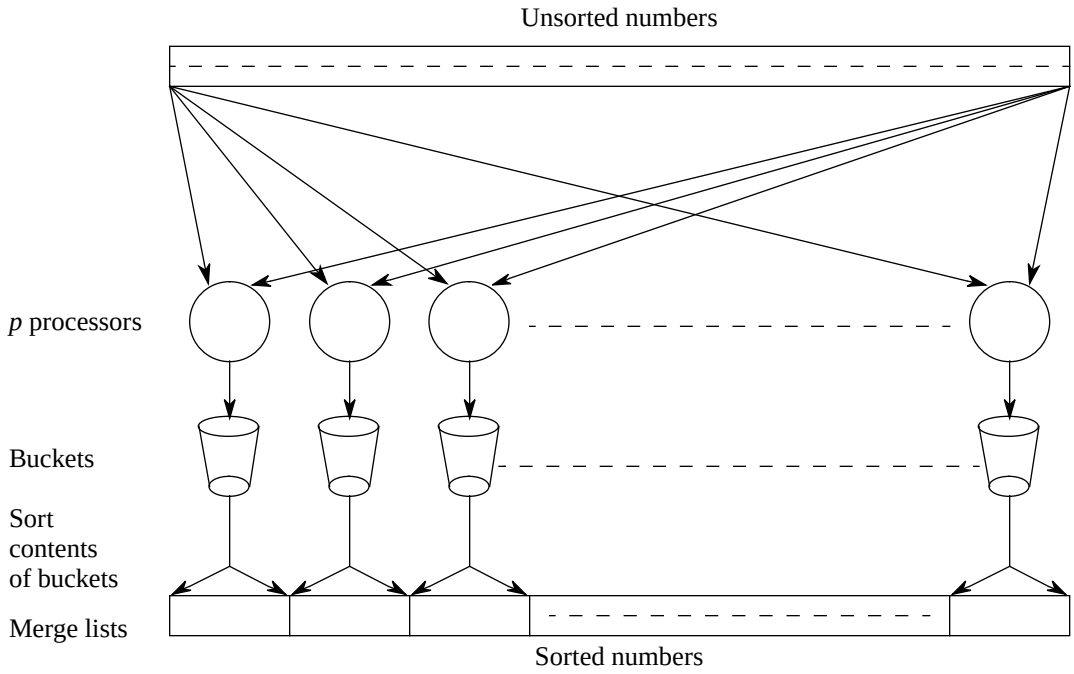

Figure 4.9 One parallel version of bucket sort.

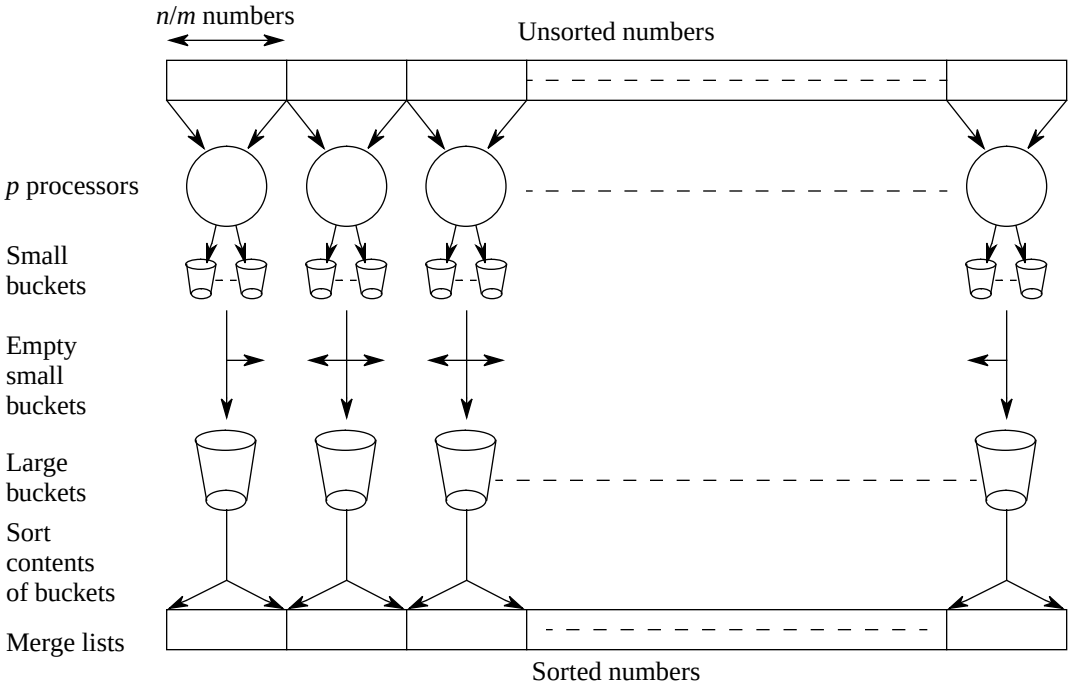

Figure 4.10 Parallel version of bucket sort.

Timing Analysis

Phase 1: Computation and Communication. We assume that it takes n steps to partition numbers into p regions.

$$t_{\text{comp1}} = n$$

We now need to send n/p numbers to each process. With broadcast or scatter, the communication time is:

$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

Phase 2: Computation. To put the n/p numbers into small buckets takes:

$$t_{\text{comp2}} = n/p$$

Timing Analysis Cont.

Phase 3: Communication. We now send the small buckets to their repective process.

If we assume numbers evenly distributed across the p small buckets, each will have $n/p \times 1/p = n/p^2$ numbers.

Thus, p processes must each send $p-1$ buckets. Upperbound would be when communication can not be overlapped and separate *send()* calls used:

$$t_{\text{comm3}} = p(p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

Lowerbound is when all $p-1$ transmissions can overlap:

$$t_{\text{comm3}} = (p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

Can use MPI's *MPI_Alltoall()* which should be more efficient than individual send and receive routines. See Figure 4.12.

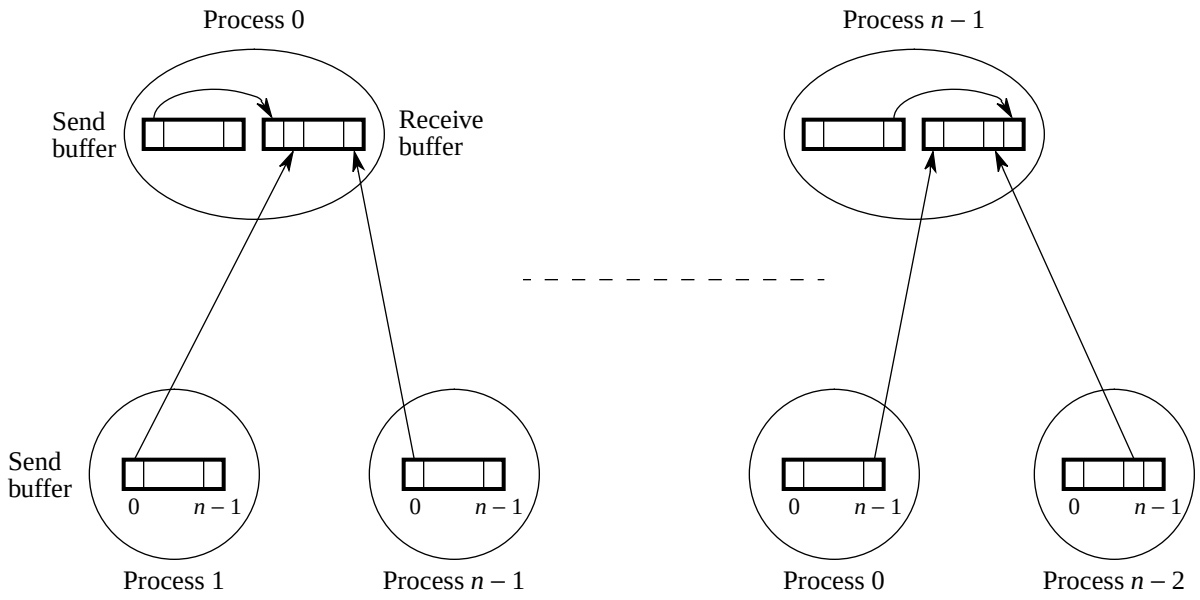

Figure 4.11 "All-to-all" broadcast.

More Timing Analysis

Phase 4: Computation. Each large bucket, containing n/p numbers, is now sorted in parallel.

$$t_{\text{comp4}} = n/p \log(n/p)$$

Total Runtime: The total parallel runtime is:

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

Ignoring communication and t_{comp1}, we get the following speedup factor:

$$S = \frac{n + n \log(n/p)}{n/p + n/p \log(n/p)} = p$$