Chapter 4 – Partitioning and Divide-and-Conquer Strategies

Two fundamental techniques in parallel programming:

1. Partitioning
   – divide problem into separate parts and compute each part separately.

2. Divide and conquer
   – applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts before solving the smaller parts and combining the results

Partitioning Strategies

- Partitioning divides the problem into parts.
  – it is the basis of all parallel programming.
  – most partitioning strategies require the results of the parts to be combined later on to obtain the desired result.

  – Data Partitioning or Domain Decomposition:
    - dividing the data and operating upon the divided data concurrently.

    This is the main strategy for parallel programming.
  – Functional Decomposition:
    - dividing the program into independent functions and executing the functions concurrently.

Example: Adding Numbers

We are to add a sequence of numbers $x_0, \ldots, x_{n-1}$

1. divide sequence into *m* parts of *n/m* numbers each (distribute sequences to corresponding processors)

2. *m* processors can each add one sequence independently to create partial sums

3. partial sums are added together on master to form the final sum.

Master-slave approach.

- How to broadcast data?
  – broadcast whole list of numbers to every slave ?
  – send the specific numbers to each slave ?

Broadcast operation will have a single startup time rather than separate startup times when using multiple send routines and may be preferable.

The code using separate send() s and recv () s:

Master:

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

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

Slave:

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

Remarks:

- Slaves are identified by a process ID in PVM, that can be usually obtained by calling a library routine, e.g., **`pvm_spawn(...)`**

- Slave number is the rank within the group in MPI
  – an integer from ***0*** to ***m-1***, where ***m*** is the # processes in the group.

The code using a broadcast or multicast routine:

Master:

```
s = n / m;
bcast(numbers, s, P_slave_group);

result = 0;
for (i=0; i<m; i++) {
   recv(&part_sum, P_ANY);
   sum += part_sum;
}
```

Slave:

```
bcast(numbers, s, P_slave_group);
start = slave_number * s;
end = start + s;
sum = 0;
for (i=start; i<end; i++)
   part_sum += numbers[i];
send(&part_sum, P_master);
```

The code using scatter and reduce routines, if available:

Master:

```
s = n / m;
scatter(numbers, &s, P_group, root=master);
reduce_add(&sum, &s, P_group, root=master);
```

Slave:

```
scatter(numbers, &s, P_group, root=master);

sum = 0;
for (i=0; i<s; i++)
   part_sum += numbers[i];
reduce_add(&part_sum, &s, P_group, root=master);
```

Similar to adding numbers, we could do other operations as well:
- find maximum number
- find number of occurrences of a number

Analysis.

Sequential Implementation.

requires n-1 additions or O(n).

Parallel Implementation.

- Phase 1 - Communication. *m* slave processes reads their *n/m* numbers.
  - $t_{comm1} = m(t_{startup} + (n/m)t_{data})$ (Using send/recv)
  - $t_{comm1} = t_{startup} + n\, t_{data}$ (Using scatter)
- Phase 2 - Computation. Slaves concurrently add n/m number together.
  - $t_{comp1} = n/m - 1$
- Phase 3 - Communication. Slaves return partial sums to master
  - $t_{comm2} = m(t_{startup} + t_{data})$ (Using send/recv)
  - $t_{comm2} = t_{startup} + m\, t_{data}$ (Using gather and reduce)
- Phase 4 - Computation. Final accumulation.
  - $t_{comp1} = m - 1$

Analysis (continued)

- Overall $t_p = O(n+m)$ and worse than t_s
- What about speedup?

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

The speedup tends to *m* for large *n*.

- The speedup will be quite low for increasing # slaves, as *m* slaves are idle in the 2nd phase forming the final result.
- Ideally we want all the processes to be active all of the time.

Divide and Conquer

1. Divide the problem into smaller subproblems
2. Divide subproblems into still smaller subproblems recursively until the tasks cannot be broken down into smaller parts.
3. Combine the results of elementary tasks, continue combining results of larger and larger tasks recursively.

Example: Sequential Recursive definition for adding a list of numbers:
- *What is the termination condition below?*

```
int add(int *s)
{
   if (number(s) <= 2) return(n1 + n2);
   else {
      Divide(s, s1, s2);   /* divide s into two parts: s1&s2*/
      part_sum1 = add(s1);
      part_sum2 = add(s2);
      return(part_sum1 + part_sum2);
   }
}
```

- **The same divide-and-conquer method can be used for**
  - **sorting a list, for finding the maximum number in a list, etc.**

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree.

- The tree is traversed
  - downward as calls are made(preorder traversal)
  - and upward when the calls return

Consider the following case:

- The tree is not a complete binary tree (not perfectly balanced with all bottom nodes at the same level)
  - happens if number of parts is not a power of 2

Parallel Implementation.

- In a serial implementation, only one node of the tree can be visited at a time.
- In parallel solution, several parts of the tree can be traversed simultaneously.

- Let's do it without using recursion.
  - The key point is that the construction is a tree.
  1. Inefficient Solution: Assign one processor to each node in the tree.
     - 2^{m+1} - 1 processors to divide the task into 2^m parts.
     - Each processor would only be active at one level in the tree (Very inefficient solution!)

2. A More Efficient Solution: reuse processors at each level of the tree (Figure 4.3)
   - division stops when the total # processors is committed
   - until then, at each stage each processor keeps the half of the list and passes on the other half.
   - Each list at final stage has *n/p* numbers
   - There are *log p* levels in the tree.

- The combining act of summation of the partial sums can be done similarly but in reverse order.
  - The constructions are the same as the binary hypercube broadcast and gather algorithms.
  - We use the communicating neighbors from their binary addresses.

With 8 processors:

Process P0:

```
divide(s1, s1, s2); /*division*/
send(s2, P4);
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);

part_sum = *s1; /*combining*/

recv(&part_sum1, P1);
part_sum += part_sum1;
recv(&part_sum1, P2);
part_sum += part_sum1;
recv(&part_sum1, P4);
part_sum += part_sum1;
```

With 8 processors:

Process P4:

```
recv(s1, P0); /*division*/
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);

part_sum = *s1; /*combining*/

recv(&part_sum1, P5);
part_sum += part_sum1;
recv(&part_sum1, P6);
part_sum += part_sum1;
send(&part_sum, P0);
```

Analysis.

Assume n is a power of 2 and ignore the $t_{startup}$ for simplicity.

The division phase only contains communication, required computation is minimal.

The combining phase requires both communication and computation to add the partial sums received and pass on the result.

Communication. *log p* steps with *p* processes.

$t_{comm1} = (n/2)\, t_{data} + (n/4)\, t_{data} + (n/8)\, t_{data} + \ldots (n/p)\, t_{data} = (n(p-1)/p)\, t_{data}$

$\Rightarrow t_{comm1}$ is marginally better than a simple broadcast.

Combining phase , only one data item (partial sum) is sent in each message

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

Computation.

$t_{comp} = (n/p) + \log p$, with time complexity of O(n) for constant p.

For large n and variable p, we get O(n/p).

M-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into more than two parts at each stage.

For example, let's divide a task into four parts:

```
int add(int *s)
{
   if (number(s) <= 4) return(n1 + n2 + n3 + n4);
   else {
      Divide(s, s1, s2, s3, s4);
      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);
   }
}
```

Tree Types

Quadtree: a tree in which each node has four children
- has particular applications in decomposing 2-D regions into 4 subregions.

Octtree: a tree in which each node has 8 children.
- has application for dividing a 3-D space recursively.

m-ary tree: a tree in which each node has *m* parts.
- suggests greater parallelism.

Sorting with Bucket Sort

- Most sequential algorithms are based upon the compare and exchange of pairs of numbers.
- In contrast, bucket sort is naturally a partitioning method:
  - works well if the original numbers are uniformly distributed across a known interval, say *0* to *a-1*.
  - This interval is divided into *m* equal regions and one "bucket" is assigned to hold numbers that fall within each region: *m* buckets.
  - We can use *m=n* buckets, i.e., one bucket for each number
  - we can develop this into a divide-and-conquer method by continually dividing the buckets into smaller buckets:
    - here we will use a limited number of buckets.

Sequential Algorithm

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

assuming sorting *n* numbers requires
$O(n \log n)$ comparisons.

Parallel Algorithm

- Four phases:
  - Partition numbers.
  - Sort into small buckets.
  - Send to large buckets.
  - Sort large buckets.

- Phase 1. Computation and Communication.
  - tcomp1 = n
  - tcomm1 = tstartup + tdata n
- Phase 2. Computation.
  - tcomp2 = n/p
- Phase 3. Communication.
  - tcomm3 = p(p-1)(tstartup + (n/p2)tdata)
  - tcomm3 = (p-1)(tstartup + (n/p2)tdata)
- Phase 4. Computation.
  - tcomp4 = (n/p)log(n/p)
- Overall.

Numerical Integration

- Sometimes simple partitioning will not give the optimal solution, especially if the amount of work in each part is difficult to estimate.
  - Bucket sort, for example, is only effective when each region has approximately the same # of numbers.

A general divide-and-conquer technique divides the region continually into parts and lets some optimization function decide when certain regions are sufficiently divided.
I = ∫ a b f(x) dx

To integrate this function (i.e., to compute the area under the curve,) we can divide the area into separate parts, each of which can be calculated by a separate process.

Quadrature Methods:

approximate numerical methods for computing a definite integral using a linear combination of values

- Rectangular regions
- Aligned rectangular regions whose middle points intersect with the function
- Trapezoidal regions

- Static Assignment.
  - Let's consider trapezoidal method.

Process Pi:

```
if (i == master) {
   printf("Enter number of intervals: ");
   scanf("%d", &n);
}
bcast(&n, P_group);
region = (b-a)/p;
start = a + region * i;
end = start + region;
d = (b-a)/n;
area = 0.0;
for (x=start; x<end; x += d)
   area += 0.5 * ( f(x) + f(x+d)) * d;
reduce_add(&integral, &area, P_group);
```

N-Body Problem

Barnes-Hut Algorithm

Whole space in which one cube contains the bodies (particles)

- divide the cube into 8 subcubes
- if a subcube contains no particles, the subcube is deleted from further consideration
- if a subcube contains > 1 body, it is recursively divided until every subcube contains one body.
- This process creates an octtree.

Orthogonal Recursive Bisection. (Salmon, 1990)