Partitioning, Revisited

- Partitioning
  - Embarrassingly parallel examples last time used partitioning
  - Most partitioning formulations require results form the partitions to be combined to yield the final result
  - Data/domain decomposition versus functional decomposition
- Example: sum a sequence of numbers
  - Divide sequence of n numbers into n/m parts, for m processors
  - Approach (details in figure):
    - Master sends numbers to slaves, slaves add their numbers (concurrently) and then send partial sums to master, master adds partial sums to produce final sum
    - Perhaps use broadcast to send entire list to every slave (is this better? depends on implementation of broadcast)

Partitioning, Revisited (cont.)

Master:

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

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

Slave:

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

Master with broadcast:

```
s = n/m;                                    /* number of numbers for slaves */
broadcast(numbers, s, Pslave_group);        /*send all numbers to slaves */
…
```

Slave with broadcast:

```
broadcast(numbers, s, Pmaster);             /*receive all nums from master */
start = slave_number * s;                   /*slave number obtained earlier */
end = start + s;
sum = 0;
for (i = start ; i < end ; i++)             /* add numbers */
…
```


Partitioning, Revisited (cont.)

Master with broadcast:

```
s = n/m;                                    /* number of numbers for slaves */
broadcast(numbers, s, Pslave_group);        /*send all numbers to slaves */

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

Slave with broadcast:

```
broadcast(numbers, s, Pmaster);             /*receive all nums from master */
start = slave_number * s;                   /*slave number obtained earlier */
end = start + s;

sum = 0;
for (i = start ; i < end ; i++)             /* add numbers */
       part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);                   /* send sum to master */
```

Master with scatter:

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

Slave with scatter and reduce:

```
gather(my_nums, s, Pgroup, root=master);        /* receive s numbers */
reduce_add(&part_sum, &s, Pgroup, root=master); /* partial sum to mast */
```


Partitioning, Revisited (cont.)

- Analysis of example:
  - Sequential computation requires n–1 additions, complexity of O(n)
  - Communication
    - $t_{comm1} = m(t_{startup} + (n/m)t_{data})$
    - m slave processors, master sending n/m numbers to each
    - $t_{comm1} = t_{startup} + nt_{data}$ with scatter/gather
  - Computation
    - $t_{comp1} = n/m - 1$
    - Each slave adds n/m numbers, in parallel
  - Communication
    - $t_{comm2} = m(t_{startup} + t_{data})$
    - Each slave sends partial result to master
    - $t_{comm2} = t_{startup} + mt_{data}$ with reduce
  - Computation
    - $t_{comp2} = m - 1$
    - Master adds the m partial sums

Partitioning, Revisited (cont.)

- Analysis of example (cont.)
  - Overall execution time
    - $(t_{startup} + nt_{data}) + (t_{startup} + mt_{data}) + (n/m - 1) + (m - 1) = 2t_{startup} + (n+m)t_{data} + n/m + m - 2$
    - $O(n+m)$
      - Worse than sequential version!!
  - What if communication is ignored?
    - Speedup = $(n-1) / (n/m+m-2)$
    - For large n, speedup tends toward m
    - For small n, speedup is low and worsens for an increasing number of slaves

Divide and Conquer

- Divide and conquer
  - Divide a problem into subproblems that are of the same form as the larger problem, then keep doing this recursively
  - Continue until it's not possible to divide further, then solve the small subproblems
  - Combine all the results, then continue this combining with larger subproblems
- Example: sum a sequence of numbers
  - Sequential approach (details in figure):
    - Need a method for termination
      - If 2 numbers, they are n1 and n2
      - If 1 number, it is n1 and n2 is zero
      - if 0 numbers, n1 and n2 are both zero
    - Can also use this method for other operations, e.g. finding maximum value
    - Can sort a list by dividing it into smaller and smaller lists (mergesort, quicksort)

Divide and Conquer (cont.)

Sequential:

```
int add(int *s)                              /* add a list of numbers */
{
      if (number(s) <= 2) return (n1+n2);    /* see explanation */
      else {
            Divide(s, s1, s2);               /* divide s into s1 and s2 */
            part_sum = add(s1);              /* recursive calls  to add */
            part_sum = add(s2);              /*      the sub lists */
            return(part_sum1+part_sum2);
      }
}
```


Divide and Conquer (cont.)

- Example: sum a sequence of numbers
  - Parallel approach (details omitted):
    - Think of processing a tree, where a division of the problem into to parts produces two subtrees,and assign one processor to each node in the tree
      - Requires $2^{m+1}-1$ processors for a task divided into 2^m parts
      - Inefficient because each processor is active only at one level in the tree
    - Reuse processors at each level
      - Stop the division when the total number of processors has been committed
      - Until then, at each stage each processor keeps half the list and passes on the other half (P0 to P4, then P0 to P2 and P4 to P6, then those to P1, P3, P5, and P7)
      - At final stage each list has n/8 numbers, n/p in general for p processors
      - Combining partial sums works in reverse
      - Particularly appropriate on a hypercube: processors communicate with processors that differ by the most significant bit

Divide and Conquer (cont.)

- Analysis of example:
  - Assume n is a power of 2
  - Startup time is not included in the analysis, but left as an exercise
  - The division phase consists mostly of communication, since the division is easy
  - The combining phase requires both computation and communication
  - Communication (division phase)
    - $t_{comm1} = (n/2)t_{data} + (n/4)t_{data} + (n/8)t_{data} \ldots = (n(p-1)/p)t_{data}$
    - Slightly better than simple broadcast
  - Computation (end of division phase)
    - $t_{comp} = n/p + \log p$
    - n/p numbers are added together, then one addition at each stage during combination
    - O(n) for constant p, O(n/p) for large n and variable p

Divide and Conquer (cont.)

- Analysis of example (cont.)
  - Communication (combining phase)
    - $t_{comm2} = \log p \; t_{data}$
    - Only one data item (the partial sum) sent each time
    - Total communication time is $(n(p-1)/p)t_{data} + \log p \; t_{data}$
    - O(n) for constant p
  - Overall execution time
    - $(n(p-1)/p)t_{data} + \log p \; t_{data} + n/p + \log p$
    - O(n) for constant p
      - Speedup will be less than p due to division and combining phases
  - Additional comments
    - Can break the task into more than 2 parts at each stage, resulting in a quadtree (4), and octtree (8), or in general an m-ary tree