Partitioning Strategies

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
Ó Prentice Hall, 1999

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

\[
\sum_{x=0}^{\left(\frac{n}{m}\right)-1} x \cdot \sum_{x=\left(\frac{2n}{m}\right)-1}^{\left(\frac{m-1}{m}\right)n} x
\]

Partitioning Strategies

We might consider dividing the sequence into \( m \) parts of \( \frac{n}{m} \) numbers each, where \( m \) processors can each add one sequence independently to create partial sums.

```
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 */
sum = 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 numbers from master */
part_sum = 0;
for (i = 0; i < s; i++) /* add numbers */
    part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster); /* send sum to master */
```
Using Broadcast/multicast Routine

Master

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

Slave

```
// Slave
bcast(numbers, s, Pmaster); /* 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, Pmaster); /* send sum to master */
```

Using scatter and reduce routines

Master

```
// Master
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

```
// Slave
scatter(numbers, &s, Pgroup, root=master); /* receive s numbers */
part_sum = 0;
for (i = start; i < end; i++) /* add numbers */
    part_sum = part_sum + numbers[i];
reduce_add(&part_sum, &s, Pgroup, root=master); /* send sum to master */
```

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
Prentice Hall, 1999
Using individual send and receive routines

Phase 1 — Communication

t_{comm1} = m(t_{startup} + t_{data})

Phase 2 — Computation

t_{comp1} = \frac{n}{m} - 1

Phase 3 — Communication

Returning partial results using individual send and receive routines

t_{comm2} = m(t_{startup} + t_{data})

Phase 4 — Computation

Final accumulation

t_{comp2} = m - 1

Overall

\[ t_p = (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2}) \]

or

\[ t_p = O(n/m) \]

We see that the parallel time complexity is worse than the sequential time complexity.

Divide and Conquer

Characterized by dividing a problem into subproblems that are of the same form as the original problem. Further problem divisions are used to solve each subproblem until the subproblems are small enough to be solved directly.

A sequential recursive definition for adding a list of numbers is:

```c
int add(int *s) /* add list of numbers, s */
{
    if (number(s) <= 2) return (n1 + n2); /* see explanation */
    else {
        Divide (s, s1, s2); /* divide s into two parts, s1 and s2 */
        part_sum1 = add(s1); /*recursive calls to add sub lists */
        part_sum2 = add(s2);
        return (part_sum1 + part_sum2);
    }
}
```
Parallel Implementation
Figure 4.4
Partial summation.

$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$

Final sum $x_{n-1}$

Parallel Code

Suppose we need to find the sum of a list of numbers.

Parallel Code

```c
Process P0
/* division phase */
divide(s1, s1, s2); send(s2, P4);
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;
```

The code for process $P_4$ might take the form

```c
Process P4
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 sequences are required for the other processes.
Assume that $n$ is a power of 2. The communication setup time, $t_{\text{startup}}$, is not included in the analysis.

### Communication Division Phase

\[
\frac{d}{n} \log_2 n + \frac{d}{(1-d)u} = \frac{d}{u} \quad \text{Time}
\]

### Total Communication Time

\[
d \log_2 n + \frac{d}{(1-d)u} = \sum_{i=1}^{\log_2 n} \frac{d}{u} = T_{\text{comm}}
\]

### Computation

\[
d \log_2 n + \frac{d}{u} = d_i
\]

### Total Parallel Execution Time

\[
T_{\text{total}} = T_{\text{comm}} + T_{\text{comp}} = \sum_{i=1}^{n} \frac{d}{u} + \sum_{i=1}^{\log_2 n} \frac{d}{u} = \sum_{i=1}^{n} \frac{d}{u} = \frac{T_{\text{comm}} + T_{\text{comp}}}{u}
\]
Divide and conquer can also be applied where a task is broken into more than two parts at each stage. For example, if the task is broken into four parts, the sequential recursive definition would be:

```c
int add(int *s) /* add list of numbers, s */
{
    if (number(s) <= 4) return(n1 + n2 + n3 + n4);
    else {
        Divide (s,s1,s2,s3,s4); /* divide s into s1,s2,s3,s4*/
        part_sum1 = add(s1); /*recursive calls to add sublists*/
        part_sum2 = add(s2);
        part_sum3 = add(s3);
        part_sum4 = add(s4);
        return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
    }
}
```
Parallel Algorithm

**Sorting Using Bucket Sort**

Works well if the original numbers are uniformly distributed across a known interval.

Divide-and-Conquer Examples

Sequential time:

$$t_s = n + n \log \left( \frac{n}{m} \right) = n + O(n \log \left( \frac{n}{m} \right))$$
Further Parallelization

By partitioning the sequence into $m$ regions, one region for each processor.

1. Partition numbers.
2. Sort into small buckets.
3. Send to large buckets.
4. Sort large buckets.

Analysis

Overall

$$\text{time} = \text{time}_{\text{comp}} + \text{time}_{\text{comm}}$$

Phase 4 — Computation

$$\text{time}_{\text{comp}} = \frac{n}{m}$$

Phase 3 — Communication

If all the communications could overlap:

$$\text{time}_{\text{comm}} = \left( p - 1 \right) \left( \text{time}_{\text{startup}} + \left( \frac{n}{p^2} \right) \text{time}_{\text{data}} \right)$$

Phase 2 — Computation

$$\text{time}_{\text{comp}} = \frac{n}{p} \log \left( \frac{n}{p} \right)$$

Phase 1 — Computation and Communication

1. Partition numbers.
2. Sort into small buckets.
3. Send to large buckets.
4. Sort large buckets.
5. Partition numbers.

The following phases are needed:
The "all-to-all" routine will actually transfer the rows of an array to columns:

```
For Phase 3 - sends data from each process to every other process
```

```
| A00 | A10 | A20 | A30 |
| A01 | A11 | A21 | A31 |
| A02 | A12 | A22 | A32 |
| A03 | A13 | A23 | A33 |
```

"all-to-all" routine
Numerical Integration

A Better Approximation

\[ \int_{a}^{b} f(x) \, dx = \int_{a}^{b} \left[ \sum_{n} f(x_n) \Delta x_n \right] \]

Example: Numerical Integration

A general divide-and-conquer technique divides the region continuously into parts and less

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
Prentice Hall, 1999
A reduce operation is used to add the areas computed by the individual processes.

```c
// SPMD pseudocode: Static Assignment

SPMD pseudocode:
```

```c
/* Process 0: */

// Function to compute f(x)

// MPI function calls

// MPI broadcast

// MPI reduce

// MPI finalize
```

Can simplify the calculation somewhat by algebraic manipulation (see text).

\[ \int_a^b f(x) \, dx = \sum_{i=1}^{n} \left( \frac{1}{2} \cdot \text{area} \right) \]

A reduce operation is used to add the areas computed by the individual processes.
Adaptive Quadrature

Method whereby the solution adapts to the shape of the curve.
The gravitational N-body problem is to find the positions and movements of the bodies in space (say planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics.

The gravitational force between two bodies of masses \(m_a\) and \(m_b\) is given by:

\[
F = \frac{G m_a m_b}{r^2}
\]

where \(G\) is the gravitational constant and \(r\) is the distance between the bodies.

Subject to forces, a body will accelerate according to Newton's second law:

\[
F = ma
\]

where \(m\) is the mass of the body, \(F\) is the force it experiences, and \(a\) is the resultant acceleration.

Let the time interval be \(D\)t. Then, for a particular body of mass \(m\), the force is given by:

\[
F = \frac{G m_a m_b}{r^2} \cdot \frac{1}{D\text{t}}
\]

The acceleration is:

\[
a = \frac{F}{m}
\]

and a new velocity is:

\[
v_{t+1} = v_t + a\cdot D\text{t}
\]

where \(v_t\) is the velocity of the body at time \(t\). The velocity of the body at time \(t + 1\) is:

\[
v_{t+1} = v_t + a\cdot D\text{t}
\]

where \(v_{t+1}\) is the velocity of the body at time \(t + 1\) and \(v_t\) is the velocity of the body at time \(t\).

The equation for three-dimensional space having a coordinate system \((x, y, z)\) is:

\[
\sum F_x = 0, \quad \sum F_y = 0, \quad \sum F_z = 0
\]

where \(F_x\), \(F_y\), and \(F_z\) are the components of the force in the x, y, and z directions, respectively.

In a three-dimensional space having a coordinate system \((x, y, z)\), the distance between the bodies at \((x_a, y_a, z_a)\) and \((x_b, y_b, z_b)\) is given by:

\[
r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}
\]

where \(r\) is the distance between the bodies.

The gravitational force between two bodies of masses \(m_a\) and \(m_b\) is given by:

\[
F = \frac{G m_a m_b}{r^2}
\]

where \(G\) is the gravitational constant and \(r\) is the distance between the bodies.

The objective is to find the positions and movements of the bodies in space (say planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics.
Sequential Code

Parallel Code
Barnes-Hut Algorithm

Starts with the whole space in which one cube contains the bodies (or particles).

First, this cube is divided into eight subcubes. If a subcube contains no particles, the subcube is deleted from further consideration. If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

This process creates an octtree; that is, a tree with up to eight edges from each node. The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and center of mass of the subcube is stored at each node.

The force on each body can then be obtained by traversing the tree, stopping at each node when the clustering approximation can be used, e.g., when

\[ \frac{\theta}{\tilde{\rho}} \leq \frac{1}{2} \]

where \( \theta \) is a constant typically 1.0 or less (it is called the opening angle).

Constructing the tree requires a time of \( O(n \log n) \), and so does computing all the forces, so that the overall time complexity of the method is \( O(n \log n) \).
Orthogonal Recursive Bisection

Example of a two-dimensional square area.