Application Granularity Considerations

- Two kinds of granularity:
  - Load-balancing granularity: ratio of size of parallel work units to overall work
  - Communication granularity: ratio of communication intervals to computation intervals

Load-Balancing Granularity

- Finer granularity is better, since it provides more ways to distribute the work.
- Imagine that the computation work load is a 10 kg. of material:
  - Sand = fine-grain
  - Cinder blocks = coarse grain
- Which is easier to distribute?

Communication Granularity

- Parallelism with fine-grain requires relatively-frequent communication compared to the computation interval.
- If a process needs to communicate frequently with other processes, then the communication must be very fast or the process’ waiting time will absorb the speedup from parallel execution.
- Consequently, fine-grain is more suited to shared memory than to distributed memory. Conversely, distributed memory requires relatively coarse grain to be effective.
- Because SIMD has less synchronization overhead, very-fine grain is more suited to SIMD than to MIMD.

Message-Passing Paradigm

- Message-passing is the programming paradigm most closely associated with distributed memory.
- However, it can also be used in a shared memory system if the problem permits.
- It is more effective for coarser granularity, since there is overhead in passing messages.

Message-Passing (2)

threads/processes on different processors

send message

receive message

send

send

Message-Passing (3)

- Two varieties of send:
  - **Blocking send**: The sending process waits for the message to be received before proceeding.
  - **Non-blocking send**: The sending process can proceed immediately. (The message may be buffered pending receipt.)
Message Buffering

Message-Passing (4)

- Two varieties of receive:
  - **Blocking receive** (most common): The receiving process waits until there is a message.
  - **Non-blocking receive**: The receiving process can check whether there is a message to be received.

Multi-cast, Scatter, Gather, Reduce

- **Multi-cast** is the equivalent of a `send` of a single message to each of a set of processes (broadcast means to all processes).
- **Scatter** means to send different elements of an array to different processes.
- **Gather** means to collect elements from different processes into a single array.
- **Reduce** means to form a single element using a specified binary operation.

Multi-cast

Scatter

Gather
Reduce

binary operator
(e.g. +)

result

+ + +
data

MPI Library
(Message-Passing Interface, Lusk et al.)

Based on the SPMD (Single Program, Multiple Data Stream) idea.

All processes run the same program, but processes can differentiate themselves using assigned ID's (called the rank of the process), so the code actually executed can be different in different processes.

Processes are divided into groups and the rank (0, 1, 2, …) applies within the group.

MPI (2)

Communication between or within a group is defined by an abstraction called a Communicator (type MPI_Comm).

A common pre-defined communicator is

MPI_COMM_WORLD

MPI (3)

The number of processes is defined on the command line:

`mpirun -np Number-of-processes Executable Args`

* The program initializes using (C syntax):

```c
MPI_Init(&argc, &argv);
```

where argc and argv are from the command line.

MPI (4)

Always terminate execution with:

```c
MPI_Finalize();
```

MPI (5)

* The program can find out the number of processes:

```c
MPI_Comm_size(Communicator, &nprocs);
```

* e.g.

```c
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```
A process can determine its own rank:

```c
MPI_Comm_size(Communicator, &id);
```

and the name of its processor:

```c
MPI_Get_processor_name(name, &namelen);
```

A process can join a barrier within its group:

```c
MPI_Barrier(Communicator);
```

It is common to declare one process (usually the one with id 0) as the master and others as slaves.

A process can then execute code conditioned upon whether it is master or slave (by checking its own id).

The master is in charge of initial setup and later direction.

The slaves do the main work in parallel.

```c
int MPI_Send(
    void* buf, // address of buffer
    int count, // number of items
    MPI_Datatype datatype, // type of each item
    int dest, // rank of destination
    int tag, // tag value of message
    MPI_Comm comm) // communicator
```

The return value indicates a success code, which will be MPI_SUCCESS if the operation is successful.

```c
int MPI_Recv(
    void* buf, // address of buffer
    int count, // maximum number of items
    MPI_Datatype datatype, // type of each item
    int source, // rank of source
    int tag, // tag value of message
    MPI_Comm comm, // communicator
    MPI_Status *status) // status indicator
```

The status indicator gives information about what was received.

```
int MPI_Recv(
    void* buf, // address of buffer
    int count, // maximum number of items
    MPI_Datatype datatype, // type of each item
    int source, // rank of source
    int tag, // tag value of message
    MPI_Comm comm, // communicator
    MPI_Status *status) // status indicator
```

The status indicator gives information about what was received.

The purpose of the tag argument is to allow a single receive operation to discriminate among different tags of messages that might be sent.

For a message to be received from a sender, both the tag and the source must match the sender values in the receive statement.
Wild Cards

- Wild cards can also be used to designate receiving from any source:
  - MPI_ANY_SOURCE
- The tag value can also be a wild-card:
  - MPI_ANY_TAG

MPI Datatypes

- Most correspond to C datatypes of a similar name:
  - MPI_CHAR
  - MPI_SHORT
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE
  - MPI_UNSIGNED
  - MPI_UNSIGNED_SHORT
  - MPI_UNSIGNED_LONG
  - MPI_UNSIGNED_CHAR
  - MPI_PACKED
  - MPI_BYTE

These do not correspond to any C datatype:
- MPI_Packed

Status indicator

- Is a struct containing three fields:
  - MPI_SOURCE
  - MPI_TAG
  - MPI_ERROR
- It also contains the length of the message received, using a call of the form:
  - MPI_Get_count(MPI_Status, MPI_Datatype, int *count)

An Example

- Integrate a function of one real variable numerically.
- The function will be passed as an argument to the integrate function.
- Other arguments to the integrate function include:
  - The limits of integration
  - The number of sub-divisions
  - The MPI communicator to be used

Integration Example

- Rectangles approximate area under curve
  - h = (b-a)/numIntervals
  - rectangle area = hf(xi)
  - xi at a + ih + h/2
  - b = (b-a)/numIntervals
Point-to-Point Version

- The number of processes is given on the command line.
- Process 0 will be the master.
- Each process $j$, including the master, computes the sum the rectangles (implicitly) numbered $i$ such that $i\%\text{numProcs} = j$.
- All of the slave processes send their sum to the master, which sums them together with its own.

Example with 4 processes

Reading/Writing MPI Code

- can be a little tricky.
- Must keep in mind that MPI is an SPMD (single-program, multiple-data stream) model.
- All processes execute the same program.
- Some processes execute one branch or another based on value of the processes' id.

```c
double integrate(
    double f(double), /* function to integrate */
    double low,      /* lower limit of integration */
    double high,     /* upper limit of integration */
    int numIntervals, /* number of intervals to be used */
    MPI_Comm comm    /* MPI communicator to use */
) {
    MPI_Status stat;     /* status indicator */
    int numProcs;        /* number of processes in comm */
    int buff siz = 1;    /* buffer size for messages */
    int tag = 1;         /* tag for messages */
    int id;              /* id of this process */
    int master = 0;      /* id of master process */
    double h;           /* width of rectangle */
    double integral;    /* approximation to integral */
    int i;
    MPI_Comm _size( comm, &numProcs );  /* get number of processes */
    MPI_Comm _rank( comm, &id );        /* get this process' id */

    h = (high - low) / numIntervals;    /* compute rectangle width */
    area = 0;                           /* compute area of rectangles */
    for (i = id; i < numIntervals; i += numProcs) {
        area += f( h * ((double) i + 0.5) );
    }
    if (id == master) {                 /* master adds up all areas */
        integral = area;
        for (i = 1; i < numProcs; i++) {
            MPI_Recv(&area, buff siz, MPI_DOUBLE, MPI_ANY_SOURCE, tag, comm, &stat);
            integral += area;
        }
    } else {                            /* slave sends area to master */
        MPI_Send(&area, buff siz, MPI_DOUBLE, master, tag, comm);
    }
    return h * integral;
}
```

Reduce Version

- The same basic idea as the point-to-point version, except that rather than explicitly sending and receiving messages, the reduce operation of MPI is used.

```c
h = (high - low) / numIntervals;    /* compute rectangle width */
area = 0;                           /* compute area of rectangles */
for (i = id; i < numIntervals; i++) {
    area += f( h * ((double) i + 0.5) );
}
if (id == master) {                 /* master adds up all areas */
    integral = area;
    for (i = 1; i < numProcs; i++) {
        MPI_Recv(&area, buff siz, MPI_DOUBLE, MPI_ANY_SOURCE, tag, comm, &stat);
        integral += area;
    }
} else {                            /* slave sends area to master */
    MPI_Send(&area, buff siz, MPI_DOUBLE, master, tag, comm);
}
return h * integral;
```
MPI Code for reduce version

\[ h = (\text{high} - \text{low}) / \text{numIntervals}; \quad /* \text{compute rectangle width} */ \]
\[ \text{area} = 0; \quad /* \text{compute area of rectangles} */ \]
\[ \text{for}(i = \text{id}; i < \text{numIntervals}; i += \text{numProcs}) \]
\[ \text{area} += f(h \times ((\text{double})i + 0.5)); \]
\[ \text{MPI\_Reduce}(&\text{area}, &\text{integral}, \text{tag}, \text{MPI\_DOUBLE}, \text{MPI\_SUM}, \text{master}, \text{comm}); \]
\[ \text{return } h \times \text{integral}; \]

Note that the receiver sends also.

Results on HMC Math Beowulf
100 million rectangles

<table>
<thead>
<tr>
<th>processors</th>
<th>result</th>
<th>error</th>
<th>time (sec)</th>
<th>effort</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14159265359023</td>
<td>4.41E-13</td>
<td>30.73</td>
<td>30.73</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.1415926535909999</td>
<td>2.67E-13</td>
<td>15.48</td>
<td>30.73</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>3.1415926535909689</td>
<td>1.35E-13</td>
<td>7.91</td>
<td>30.73</td>
<td>2.99</td>
</tr>
<tr>
<td>4</td>
<td>3.1415926535909191</td>
<td>6.98E-14</td>
<td>3.98</td>
<td>30.73</td>
<td>4.99</td>
</tr>
<tr>
<td>5</td>
<td>3.1415926535909699</td>
<td>3.81E-14</td>
<td>2.03</td>
<td>30.73</td>
<td>5.99</td>
</tr>
<tr>
<td>6</td>
<td>3.1415926535909695</td>
<td>1.96E-14</td>
<td>1.01</td>
<td>30.73</td>
<td>6.99</td>
</tr>
<tr>
<td>7</td>
<td>3.1415926535909685</td>
<td>9.82E-15</td>
<td>0.52</td>
<td>30.73</td>
<td>7.99</td>
</tr>
<tr>
<td>8</td>
<td>3.1415926535909689</td>
<td>4.91E-14</td>
<td>0.51</td>
<td>30.73</td>
<td>8.99</td>
</tr>
<tr>
<td>9</td>
<td>3.1415926535909685</td>
<td>2.46E-14</td>
<td>0.49</td>
<td>30.73</td>
<td>9.99</td>
</tr>
<tr>
<td>10</td>
<td>3.1415926535909689</td>
<td>1.23E-14</td>
<td>0.48</td>
<td>30.73</td>
<td>10.99</td>
</tr>
<tr>
<td>11</td>
<td>3.1415926535909685</td>
<td>6.16E-15</td>
<td>0.47</td>
<td>30.73</td>
<td>11.99</td>
</tr>
<tr>
<td>12</td>
<td>3.1415926535909689</td>
<td>3.08E-15</td>
<td>0.46</td>
<td>30.73</td>
<td>12.99</td>
</tr>
</tbody>
</table>

10 million rectangles

<table>
<thead>
<tr>
<th>processors</th>
<th>result</th>
<th>error</th>
<th>time (sec)</th>
<th>effort</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14159265359072</td>
<td>6.64E-14</td>
<td>3.07</td>
<td>3.07</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.14159265359072</td>
<td>1.95E-13</td>
<td>1.04</td>
<td>3.08</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>3.14159265359072</td>
<td>1.02E-13</td>
<td>5.11</td>
<td>3.09</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>3.14159265359072</td>
<td>5.09E-14</td>
<td>2.54</td>
<td>3.10</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>3.14159265359072</td>
<td>2.54E-14</td>
<td>1.27</td>
<td>3.11</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>3.14159265359072</td>
<td>1.27E-14</td>
<td>0.63</td>
<td>3.12</td>
<td>1.00</td>
</tr>
<tr>
<td>7</td>
<td>3.14159265359072</td>
<td>6.36E-15</td>
<td>0.32</td>
<td>3.13</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>3.14159265359072</td>
<td>3.18E-15</td>
<td>0.16</td>
<td>3.14</td>
<td>1.00</td>
</tr>
<tr>
<td>9</td>
<td>3.14159265359072</td>
<td>1.59E-15</td>
<td>0.08</td>
<td>3.15</td>
<td>1.00</td>
</tr>
<tr>
<td>10</td>
<td>3.14159265359072</td>
<td>7.95E-16</td>
<td>0.04</td>
<td>3.16</td>
<td>1.00</td>
</tr>
<tr>
<td>11</td>
<td>3.14159265359072</td>
<td>3.97E-16</td>
<td>0.04</td>
<td>3.17</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>3.14159265359072</td>
<td>1.98E-16</td>
<td>0.03</td>
<td>3.18</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Perspective

- The application seems to have good speedup.
- However, we can get the same or better accuracy with only 10 million points.
- In the latter case, the speedup is not so dramatic:
Speedup vs. Processors

![Graph showing speedup vs. number of processors]

What about 1 million rectangles?

- Describe what is going on using the vocabulary presented thus far.
- What do you predict for 1 million?

Results on HMC Beowulf

<table>
<thead>
<tr>
<th>Processors</th>
<th>Result</th>
<th>Error</th>
<th>Time (sec)</th>
<th>Effort</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14159265358976</td>
<td>2.98E-14</td>
<td>0.31</td>
<td>0.31</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.14159265358994</td>
<td>1.47E-13</td>
<td>0.16</td>
<td>0.33</td>
<td>1.94</td>
</tr>
<tr>
<td>3</td>
<td>3.14159265358990</td>
<td>1.16E-13</td>
<td>0.22</td>
<td>0.66</td>
<td>1.41</td>
</tr>
<tr>
<td>4</td>
<td>3.14159265358990</td>
<td>1.10E-13</td>
<td>0.19</td>
<td>0.78</td>
<td>1.63</td>
</tr>
<tr>
<td>5</td>
<td>3.14159265358990</td>
<td>1.28E-13</td>
<td>0.18</td>
<td>0.90</td>
<td>1.72</td>
</tr>
<tr>
<td>6</td>
<td>3.14159265358990</td>
<td>1.04E-13</td>
<td>0.17</td>
<td>1.03</td>
<td>1.82</td>
</tr>
<tr>
<td>7</td>
<td>3.14159265358990</td>
<td>1.12E-13</td>
<td>0.16</td>
<td>1.13</td>
<td>1.94</td>
</tr>
<tr>
<td>8</td>
<td>3.14159265358990</td>
<td>9.77E-14</td>
<td>0.16</td>
<td>1.27</td>
<td>1.94</td>
</tr>
<tr>
<td>9</td>
<td>3.14159265358990</td>
<td>8.60E-14</td>
<td>0.15</td>
<td>1.39</td>
<td>2.07</td>
</tr>
<tr>
<td>10</td>
<td>3.14159265358990</td>
<td>7.73E-14</td>
<td>0.19</td>
<td>1.87</td>
<td>1.63</td>
</tr>
<tr>
<td>11</td>
<td>3.14159265358990</td>
<td>7.37E-14</td>
<td>0.19</td>
<td>2.37</td>
<td>1.63</td>
</tr>
<tr>
<td>12</td>
<td>3.14159265358990</td>
<td>8.26E-14</td>
<td>0.08</td>
<td>0.90</td>
<td>3.88</td>
</tr>
<tr>
<td>13</td>
<td>3.14159265358990</td>
<td>8.22E-14</td>
<td>0.08</td>
<td>1.02</td>
<td>3.88</td>
</tr>
<tr>
<td>14</td>
<td>3.14159265358990</td>
<td>8.44E-14</td>
<td>0.08</td>
<td>1.14</td>
<td>3.88</td>
</tr>
<tr>
<td>15</td>
<td>3.14159265358990</td>
<td>8.53E-14</td>
<td>0.09</td>
<td>1.34</td>
<td>3.44</td>
</tr>
</tbody>
</table>