Performance Metrics

- Speedup
- Efficiency
- Work
- Scaled speedup

Speedup

- A problem is ideally solvable faster with multiple processors.
- Speedup = \( \frac{\text{time to solve sequentially}}{\text{time to solve in parallel}} \)
- Some qualification is necessary:
  - Are the sequential and parallel times necessarily using the same algorithm?

Algorithm Dependence

- Not all algorithms parallelize equally well.
- Parallel execution may introduce overheads not present in sequential execution of a given algorithm.
- Rigorous definition of speedup demands that parallel execution of a given algorithm be compared against serial execution using the “best” sequential algorithm.
- Often this is not done: instead, the same algorithm is used for both sequential and parallel.

Speedup as a function of the number of processors

- Let \( T_p \) be the time required to solve a given problem using \( p \) processors.
- Let \( S_p \) be the speedup using \( p \) processors.
  \[ S_p = \frac{T_1}{T_p} \]
- Ideally \( S_p = p \)
- The ideal is difficult to achieve.

Amdahl’s Law

- Amdahl’s law expresses a reason why ideal speedup may not be achieved.
- Make the idealized assumption that the code to be executed consists of
  - A perfectly-parallel portion (capable of using all \( p \) processors efficiently), and
  - A strictly-sequential portion (capable of using only 1 processor).
- Let \( f \) be the fraction of instructions that fall into the strictly-sequential category.

Amdahl’s Law (2)

- Ideally, \( f \) is low. If \( f \) is 0, perfect speedup can be expected, while if \( f \) is high, speedup will be near 1.
- What is unexpected is how quickly speedup drops off as a function of \( f \).
Amdahl's Law (3)

Program Execution Profile

Sequential portion Parallel portion

\[ T_p = a + b \]

Sequential fraction = \( f = \frac{a}{a + pb} \)

Amdahl's Law (4)

Sequential fraction = \( f = \frac{a}{a + pb} \)

Therefore \( f(a + pb) = a \)

Therefore \( b = \frac{(a - a)}{p} = \frac{a/p}{1/f - 1} \)

Therefore \( b/a = \frac{1/f - 1}{p} \)

Speedup = \( \frac{T_1}{T_p} = \frac{(a + pb)/(a + b)}{(a + b)} = \frac{1 + pb/a}{1 + b/a} = \frac{(1 + (1/f - 1))(1 + (1/f - 1)/p)}{(1/f)(1 + (1/f - 1)/p)} \)

Speedup = \( \frac{1/f}{1 + (1/f - 1)/p} \)

Amdahl's Law, where \( f \) is sequential

Amdahl's Law (5)

Speedup = \( \frac{1}{f + (1-f)/p} \)

Limiting cases and examples:

\( p \to 1 : \text{Speedup} \to 1 \)

\( p \to \infty : \text{Speedup} \to 1/f \)

\( f \to 0 : \text{Speedup} \to p \)

\( f \to 1 : \text{Speedup} \to 1 \)

\( f = 0.1 : \text{Speedup} < 10 \)

Amdahl's Law (6)

To lift the ceiling on speedup, we need to decrease \( f \).

Effort

\( \text{The effort used by a parallel processor in executing a program is the product of} \)

\( \text{the elapsed time, and} \)

\( \text{the number of processors used} \)

\( \text{Included in effort are processors that are used for some portion of the computation, but which are idle for other portions.} \)
Effort (2)

- Ideally the effort is the same regardless of the number of processors.
- In practice, the effort tends to go up with more processors, due to:
  - Overhead in spawning parallel processes
  - Communication overhead
  - Some processors being idle part of the time
- Usually we are willing to sacrifice some effort to attain speedup.

Efficiency

- How well are the parallel processors being utilized?
  - If there are $p$ processors, with parallel execution time $T_p$, then the effort is $p T_p$.
  - The actual “work” done is $T_1$, the time it would take to do the work on one processor.
  - Therefore, Efficiency = $\frac{T_1}{p T_p}$, which happens to be equal to Speedup / $p$.

Ideal Efficiency

- The ideal efficiency is 1, with actual practice being somewhat worse, due to the additional effort mentioned earlier.
- We shouldn’t be lulled into thinking that efficiency is how busy the processors are, because they could be doing work that is parallel overhead.

Gustafson’s “Law” (John Gustafson, 1988)

- Gustafson tried to refute Amdahl’s law, which assumes that we are interested in applying ever larger numbers of processors to a fixed-sized problem.
- In practice, we are only interested in applying more processors as the size of the problem scales.
- Moreover, scaling the problem usually scales the parallel part disproportionately.
Let \( n \) be a measure of the problem size.

The execution of the program on a parallel computer is decomposed into

\[
a(n) + b(n) = 1
\]

where \( a \) is the sequential fraction and \( b \) the parallel fraction (ignoring overhead for now).

On a sequential computer, the relative time would be \( a(n) + pb(n) \) where \( p \) is the number of processors in the parallel case.

### Gustafson’s Law (3)

- Speedup is therefore
  \[
  (a(n) + pb(n)) \text{ (relative to } a(n)+b(n) = 1) \\
  = a(n)+p*(1-a(n))
  \]
  where \( a(n) \) is the serial fraction.

- Assuming the serial fraction \( a(n) \) diminishes with problem size \( n \), then speedup approaches \( p \) as \( n \to \infty \) as desired.

- Thus Gustafson’s law seems to rescue parallel processing from Amdahl’s law.

### Granularity Considerations

- Roughly speaking, granularity means the ratio of computation interval to communication time needed to achieve a reasonable speedup.

- 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.

### Granularity (2)

- 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 (with or without warts) = coarse grain

- Which is easier to distribute?

### Granularity (3)

- Fine-grain parallelism requires relatively-frequent communication compared to the computation interval.

- 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

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

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 Library
(Message-Passing Interface)

- 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 Library

*(Message-Passing Interface)*

- 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):
  ```
  MPI_Init(&argc, &argv);
  ```
  where `argc` and `argv` are from the command line.

### MPI (4)

- Always terminate execution with:
  ```
  MPI_Finalize();
  ```

### MPI (5)

- The program can find out the number of processes:
  ```
  MPI_Comm_size(Communicator, &nprocs);
  ```
  e.g.
  ```
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  ```

### MPI (6)

- A process can determine its own *rank*:
  ```
  MPI_Comm_size(Communicator, &id);
  ```
- and the name of its processor:
  ```
  MPI_Get_processor_name(name, &namelen);
  ```
MPI (7)

- A process can join a **barrier** within its group:
  
  ```c
  MPI_Barrier(Communicator);
  ```

Master/Slaves

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

Sending Messages

```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.

Receiving Messages

```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.

Send-Receive Matching

- 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:
  
  ```c
  MPI_ANY_SOURCE
  ```

- The tag value can also be a wild-card:
  
  ```c
  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
- MPI_BYTE

### Status indicator

- is a struct containing three fields:
  - MPI_SOURCE
  - MPI_TAG
  - MPI_ERROR

  indicating the corresponding information about the message received.

- 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

- Function to be integrated
- Limits of integration

### Rectangles approximate area under curve

\[
h = \frac{b-a}{\text{numIntervals}}
\]

\[
\text{rectangle area} = h \cdot f(x_i)
\]

\[
ap + ih + h/2
\]

### 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

```
Example with 4 processes

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum on master</td>
<td>sum on individual processors</td>
</tr>
</tbody>
</table>
```

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.

### MPI Code (see handout for all)

```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 buffSize = 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 area;                        /* area of this process' rectangles */
    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 += h * ((double)i + 0.5);    
    }

    if (id == master)                                  /* master adds up all areas         */
    {
        integral = area;
        for( i = 1; i < numProcs; i++ )
        {
            MPI_Recv(&area, buffSize, MPI_DOUBLE, MPI_ANY_SOURCE, tag, comm, &status);
            integral += area;
        }
    }
    else                                               /* slave sends area to master       */
    {
        MPI_Send(&area, buffSize, 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 += numProcs )
{
    area += h * ((double)i + 0.5);    
}

MPI_Reduce(&area, &integral, tag, MPI_DOUBLE, MPI_SUM, master, comm);
return h * integral;
```
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>3.14159265358972</td>
<td>-6.44E-14</td>
<td>3.07</td>
<td>3.07</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.14159265358998</td>
<td>1.89E-13</td>
<td>1.54</td>
<td>3.08</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>3.14159265358990</td>
<td>1.12E-13</td>
<td>1.14</td>
<td>3.42</td>
<td>2.69</td>
</tr>
<tr>
<td>4</td>
<td>3.14159265358968</td>
<td>-1.10E-13</td>
<td>0.88</td>
<td>3.52</td>
<td>3.49</td>
</tr>
<tr>
<td>5</td>
<td>3.14159265358970</td>
<td>-8.44E-14</td>
<td>0.73</td>
<td>3.65</td>
<td>4.21</td>
</tr>
<tr>
<td>6</td>
<td>3.14159265358976</td>
<td>-2.71E-14</td>
<td>0.63</td>
<td>3.77</td>
<td>4.87</td>
</tr>
<tr>
<td>7</td>
<td>3.14159265358979</td>
<td>-4.44E-16</td>
<td>0.56</td>
<td>3.92</td>
<td>5.48</td>
</tr>
<tr>
<td>8</td>
<td>3.14159265358980</td>
<td>1.11E-14</td>
<td>0.51</td>
<td>4.04</td>
<td>6.02</td>
</tr>
<tr>
<td>9</td>
<td>3.14159265358979</td>
<td>1.33E-15</td>
<td>0.49</td>
<td>4.42</td>
<td>6.27</td>
</tr>
<tr>
<td>10</td>
<td>3.14159265358975</td>
<td>-6.66E-15</td>
<td>0.46</td>
<td>4.60</td>
<td>6.67</td>
</tr>
<tr>
<td>11</td>
<td>3.14159265358977</td>
<td>-7.14E-15</td>
<td>0.44</td>
<td>4.87</td>
<td>6.98</td>
</tr>
<tr>
<td>12</td>
<td>3.14159265358975</td>
<td>-6.26E-14</td>
<td>0.42</td>
<td>5.00</td>
<td>7.31</td>
</tr>
<tr>
<td>13</td>
<td>3.14159265358977</td>
<td>-2.26E-14</td>
<td>0.41</td>
<td>5.31</td>
<td>7.49</td>
</tr>
<tr>
<td>14</td>
<td>3.14159265358975</td>
<td>-6.66E-15</td>
<td>0.40</td>
<td>5.50</td>
<td>7.68</td>
</tr>
<tr>
<td>15</td>
<td>3.14159265358975</td>
<td>-7.55E-15</td>
<td>0.39</td>
<td>5.80</td>
<td>7.87</td>
</tr>
</tbody>
</table>

Time vs. Processors

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:
What about 1 million rectangles?

- Describe what is going on using the vocabulary presented thus far.

- What do you predict for 1 million?

<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>3.141592653589796</td>
<td>2.38E-14</td>
<td>0.31</td>
<td>0.31</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.141592653589794</td>
<td>1.47E-13</td>
<td>0.16</td>
<td>0.33</td>
<td>1.94</td>
</tr>
<tr>
<td>3</td>
<td>3.141592653589900</td>
<td>1.16E-13</td>
<td>0.22</td>
<td>0.66</td>
<td>1.41</td>
</tr>
<tr>
<td>4</td>
<td>3.141592653589900</td>
<td>1.10E-13</td>
<td>0.19</td>
<td>0.78</td>
<td>1.63</td>
</tr>
<tr>
<td>5</td>
<td>3.141592653589900</td>
<td>1.28E-13</td>
<td>0.18</td>
<td>0.90</td>
<td>1.72</td>
</tr>
<tr>
<td>6</td>
<td>3.141592653589900</td>
<td>1.04E-13</td>
<td>0.17</td>
<td>1.03</td>
<td>1.92</td>
</tr>
<tr>
<td>7</td>
<td>3.141592653589900</td>
<td>1.12E-13</td>
<td>0.16</td>
<td>1.13</td>
<td>1.94</td>
</tr>
<tr>
<td>8</td>
<td>3.141592653589900</td>
<td>9.77E-14</td>
<td>0.15</td>
<td>1.27</td>
<td>1.94</td>
</tr>
<tr>
<td>9</td>
<td>3.141592653589900</td>
<td>6.02E-14</td>
<td>0.15</td>
<td>1.39</td>
<td>2.07</td>
</tr>
<tr>
<td>10</td>
<td>3.141592653589900</td>
<td>7.73E-14</td>
<td>0.19</td>
<td>1.67</td>
<td>1.63</td>
</tr>
<tr>
<td>11</td>
<td>3.141592653589900</td>
<td>7.37E-14</td>
<td>0.19</td>
<td>2.07</td>
<td>1.63</td>
</tr>
<tr>
<td>12</td>
<td>3.141592653589900</td>
<td>8.20E-14</td>
<td>0.08</td>
<td>0.90</td>
<td>3.56</td>
</tr>
<tr>
<td>13</td>
<td>3.141592653589900</td>
<td>8.20E-14</td>
<td>0.08</td>
<td>1.02</td>
<td>3.88</td>
</tr>
<tr>
<td>14</td>
<td>3.141592653589900</td>
<td>8.44E-14</td>
<td>0.08</td>
<td>1.14</td>
<td>3.88</td>
</tr>
<tr>
<td>15</td>
<td>3.141592653589900</td>
<td>8.33E-14</td>
<td>0.09</td>
<td>1.34</td>
<td>3.44</td>
</tr>
</tbody>
</table>