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.
- $S_p = T_1 / T_p$
- Ideally $S_p = p$
- The ideal is difficult to achieve.

Amdahl's Law
(Gene Amdahl, 1967)

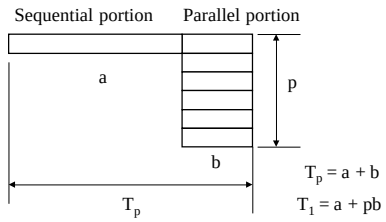
- Amdahl's law expresses a reason why ideal speedup may not be achieved.
- Make the idealized assumption that the code to be executed consists of
  - A perfectly-parallel portion (capable of using all p processors efficiently), and
  - A strictly-sequential portion (capable of using only 1 processor).
- 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 fraction = f = a / (a + p*b)

Amdahl's Law (4)

Sequential fraction = f = a / (a + p*b)
Therefore f*(a + p*b) = a
Therefore b = (a/f - a)/p = (a/p)(1/f - 1)
Therefore b/a = (1/f - 1)/p

Speedup = T_1 / T_p = (a+pb)/(a+b)
= (1+pb/a)/(1+b/a)
= (1 + (1/f - 1)/(1+(1/f-1)/p)
= (1/f)/(1+(1/f-1)/p)

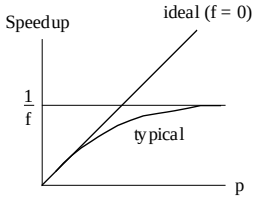
Speedup = 1/(f + (1-f)/p) ⟸ Amdahl's Law, where f is sequential

Amdahl's Law (5)

Speedup = 1/(f + (1-f)/p)

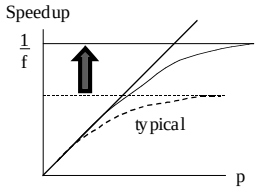
Limiting cases and examples:

- p →1: Speedup →1
- p →∞: Speedup →1/f
- f →0 : Speedup →p
- f →1: Speedup →1
- f = 0.1: Speedup < 10



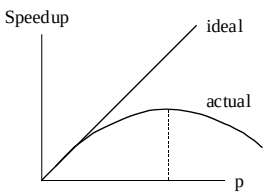
Amdahl's Law (6)

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



Slowdown

- Due to **overhead** of supporting more processors, speedup may actually *decrease* after a point:



Effort

- The effort used by a parallel processor in executing a program is the product of
  - the elapsed time, and
  - the number of processors used
- 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,

$$\text{Efficiency} = \frac{T_1}{p\ T_p}$$

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.

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.

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.

Gustafson's Law (2)

- 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)) (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 $\rightarrow\infty$ 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?

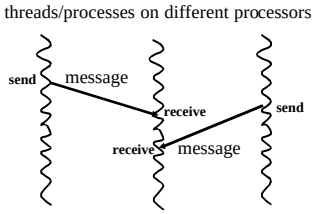
Granularity (3)

- 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.

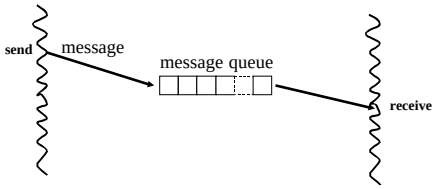
Message-Passing (2)



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



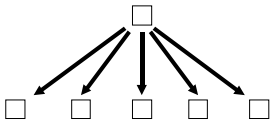
Message-Passing (4)

- 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.

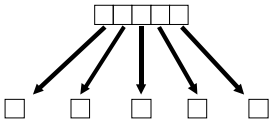
Multi-cast, Scatter, Gather, Reduce

- **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.

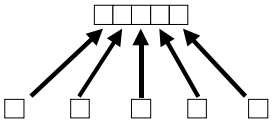
Multi-cast



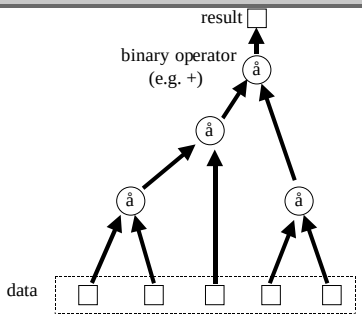
Scatter



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 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:

  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

```
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

```
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:

  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

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

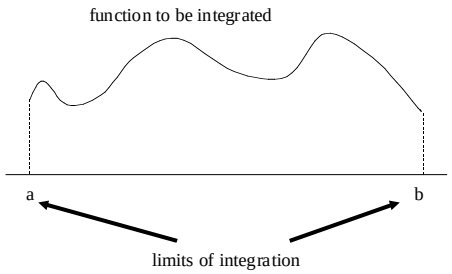
  indicating the corresponding information about the message received.
- It also contains the length of the message received, using a call of the form:

  MPI_Get_count(MPI_ Status, MPI_Datatype, int *count)

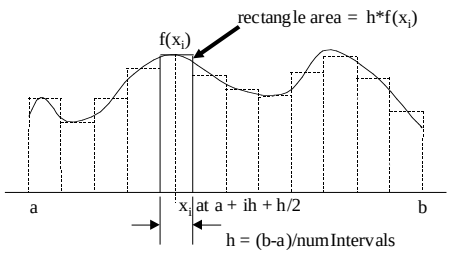
An Example

- 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



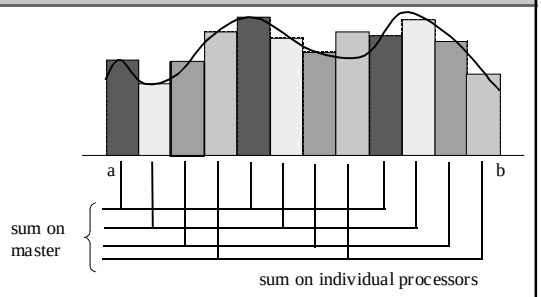
Rectangles approximate area under curve



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%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.

MPI Code (see handout for all)

```
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 buffsiz = 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 += 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, buffsiz, MPI_DOUBLE, MPI_ANY_SOURCE, tag, comm, &stat);
  integral += area;
  }
 }
else
 {                                         /* slave sends area to master     */
 MPI_Send(&area, buffsiz, 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.

MPI Code for reduce version

```
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));
 }

MPI_Reduce(&area, &integral, tag, MPI_DOUBLE, MPI_SUM, master, comm);
return h * integral;
```

message sent

message received

operation performed

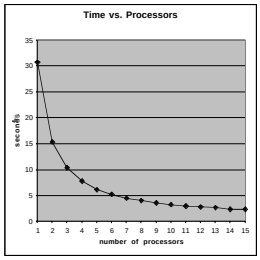
receiver

Note that the receiver sends also.

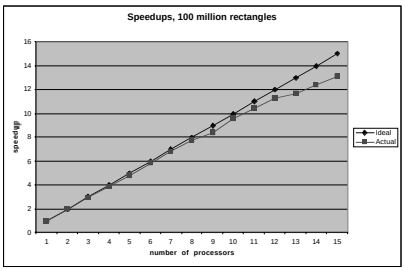
Results on HMC Math Beowulf
100 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265359023	4.41E-13	30.73	30.73	1.00
2	3.14159265358999	2.07E-13	15.48	30.96	1.99
3	3.14159265359014	3.49E-13	10.46	31.38	2.94
4	3.14159265359019	4.01E-13	7.90	31.60	3.89
5	3.14159265358964	-1.50E-13	6.38	31.89	4.82
6	3.14159265358965	-1.37E-13	5.24	31.43	5.86
7	3.14159265358964	-1.53E-13	4.51	31.60	6.81
8	3.14159265358960	-1.85E-13	3.97	31.77	7.74
9	3.14159265358974	-4.53E-14	3.67	33.01	8.37
10	3.14159265358974	-4.84E-14	3.22	32.16	9.54
11	3.14159265358973	-5.46E-14	2.95	32.41	10.42
12	3.14159265358973	-5.51E-14	2.72	32.64	11.30
13	3.14159265358973	-5.37E-14	2.64	34.32	11.64
14	3.14159265358974	-4.71E-14	2.48	34.67	12.39
15	3.14159265358974	-4.62E-14	2.34	35.05	13.13

Time vs. Processors



Speedup vs. Processors



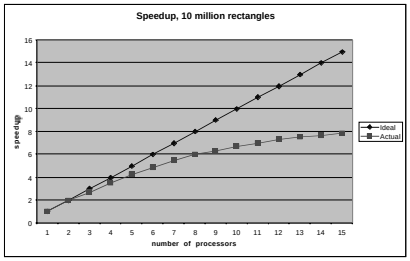
Perspective

- 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:

Results on HMC Math Beowulf
10 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265358972	-6.44E-14	3.07	3.07	1.00
2	3.14159265358998	1.89E-13	1.54	3.08	1.99
3	3.14159265358990	1.12E-13	1.14	3.42	2.69
4	3.14159265358968	-1.10E-13	0.88	3.53	3.49
5	3.14159265358970	-8.44E-14	0.73	3.65	4.21
6	3.14159265358976	-2.71E-14	0.63	3.77	4.87
7	3.14159265358979	-4.44E-16	0.56	3.92	5.48
8	3.14159265358980	1.11E-14	0.51	4.04	6.02
9	3.14159265358979	1.33E-15	0.49	4.42	6.27
10	3.14159265358979	-8.88E-16	0.46	4.60	6.67
11	3.14159265358977	-1.47E-14	0.44	4.87	6.98
12	3.14159265358976	-2.62E-14	0.42	5.02	7.31
13	3.14159265358977	-2.22E-14	0.41	5.31	7.49
14	3.14159265358978	-6.66E-15	0.40	5.55	7.68
15	3.14159265358978	-7.55E-15	0.39	5.80	7.87

Speedup vs. 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 Math Beowulf
1 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265358976	-2.98E-14	0.31	0.31	1.00
2	3.14159265358994	1.47E-13	0.16	0.33	1.94
3	3.14159265358990	1.16E-13	0.22	0.66	1.41
4	3.14159265358990	1.10E-13	0.19	0.78	1.63
5	3.14159265358992	1.28E-13	0.18	0.90	1.72
6	3.14159265358989	1.04E-13	0.17	1.03	1.82
7	3.14159265358990	1.12E-13	0.16	1.13	1.94
8	3.14159265358989	9.77E-14	0.16	1.27	1.94
9	3.14159265358987	8.62E-14	0.15	1.39	2.07
10	3.14159265358987	7.73E-14	0.19	1.87	1.63
11	3.14159265358986	7.37E-14	0.19	2.07	1.63
12	3.14159265358987	8.26E-14	0.08	0.90	3.88
13	3.14159265358987	8.22E-14	0.08	1.02	3.88
14	3.14159265358987	8.44E-14	0.08	1.14	3.88
15	3.14159265358987	8.53E-14	0.09	1.34	3.44