DTS202TC Foundation Of Parallel Computing

Lecture 2

Hong-Bin Liu

Xi'an Jiaotong Liverpool University

September 13, 2021

Recall from last time

- Understand what are serial and parallel programs
- Share-memory and distributed-memory system
- Basic C programming

Common Problems that Students have

Goals for this week

- How to design parallel algorithms
- Distributed-memory programming using MPI
- Hands on demo on OpenMPI

- Assessment groups will be randomly generated
- Lab 2 is weight 30% of the total marks of A3

Warning

This week's content is extensive.

Outline

- Foster's methodology
- Steps
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping

- Domain decomposition
  - Decompose the data associated with a problem
  - We divide these data into small pieces of approximately equal size if possible.
- Functional decomposition
  - Divide the computation into disjoint tasks

- Work out messages that are to be sent and received.
- In domain decomposition problems, communication requirements can be difficult to determine.
- In contrast, communication requirements in parallel algorithms obtained by functional decomposition are often straightforward.

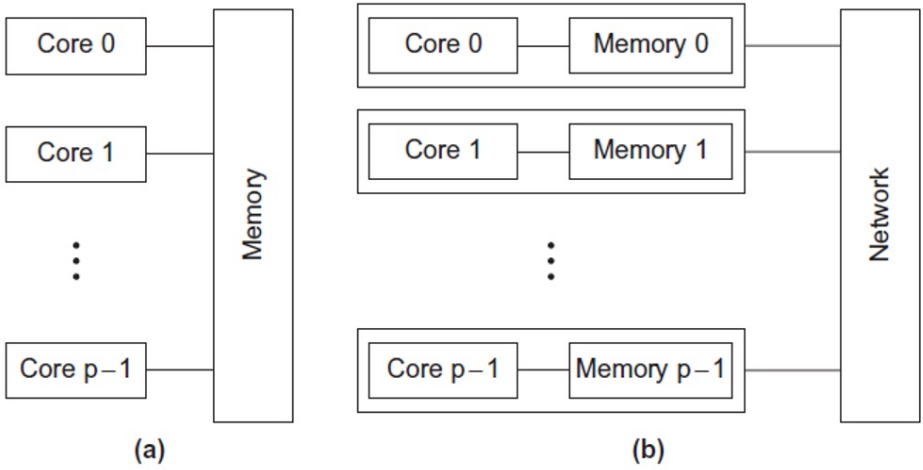
- Combine tasks and communications identified in the first step into larger tasks.

- Assign the composite tasks identified in the previous step to processes/threads.
- This should be done so that the communication is minimised, and each processes/threads gets roughly the same amount of work.
- Many algorithms developed using domain decomposition techniques feature a fixed number of equal-sized tasks and structured local and global communication. In such cases, an efficient mapping is straightforward.

More of Foster's Methodology

https://www.mcs.anl.gov/~itf/dbpp/text/node14.html

Outline



Shared-memory Distributed-memory

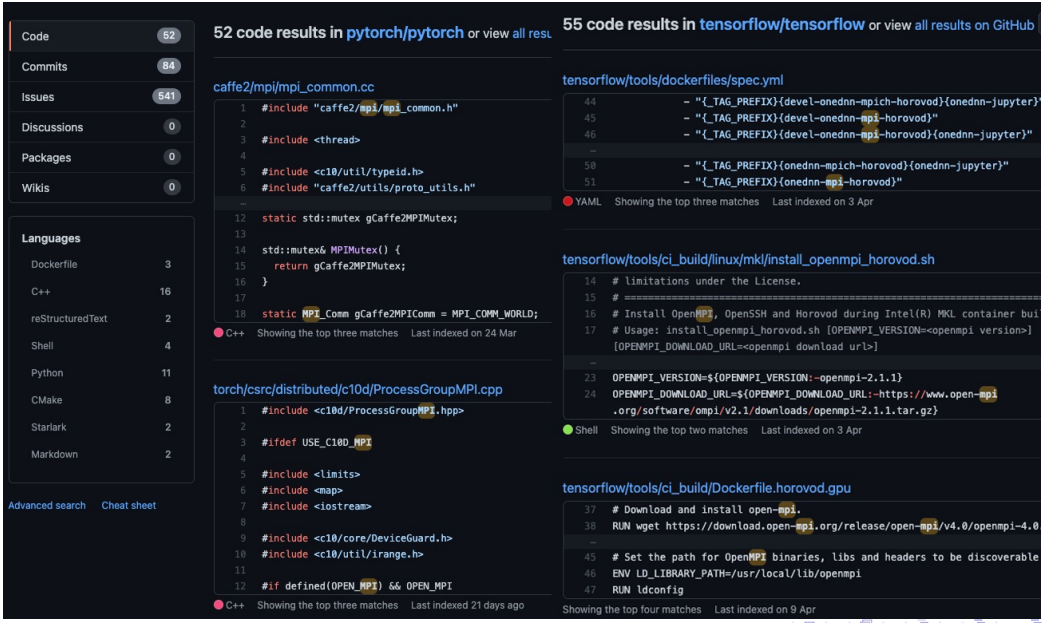
- Each process can use its local memory for computation
- When it needs data from remote process, it has to send messages
- MPI forum was formed in 1992 to standardise message passing models and MPI 1.0 was release around 1994, current version v4.0.

- It is an interface standard – defines the operations / routines needed for message passing
- Implemented by the community for different platforms – meant to be able to run the same code on different platforms without modifications.
- Some popular implementations are MPICH, MVAPICH, **OpenMPI**

MPI Resources

- MPI Standard and documentation:
  - http://www.mpi-forum.org
- Other information:
  - http://www.mcs.anl.gov/research/projects/mpi/index.htm
  - Tutorial https://mpitutorial.com/tutorials/

MPI Applications



```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank,
             comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

All MPI use must be between init and finalise

Compilation

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

```
mpiexec -n <number of processes> <executable>
```

- Each process runs a "copy" of main
- Compile one program
- Can branch independently
- Can also run on clusters on local network

- Two things we want to know
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI_Comm_size** reports the number of processes
  - **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program started, called MPI_COMM_WORLD.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf page 32.

MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

C binding

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Recv_c(void *buf, MPI_Count count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf page 37.

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t

Message matching

MPI_SEND(buf, count, datatype		MPI_RECV(buf, count, datatype	
IN	buf	OUT	buf
IN	count	IN	count
IN	datatype	IN	datatype
IN	dest	IN	source
IN	tag	IN	tag
IN	comm	IN	comm
		OUT	status

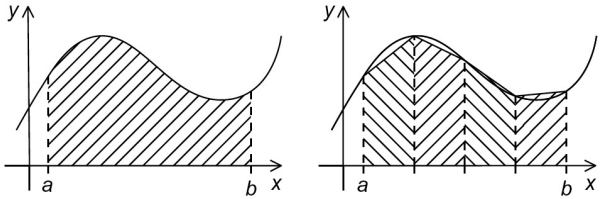


FIGURE 3.3

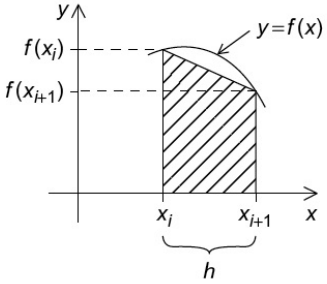
The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

$$\text{Area of one trapezoid} = \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right]. \tag{1}$$

$$\text{Sum of trapezoid areas} = h\left[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2\right] \tag{2}$$

Pseudo-code for a Serial Program


```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
  x_i = a + i*h;
  approx += f(x_i);
}
approx += h * approx;
```

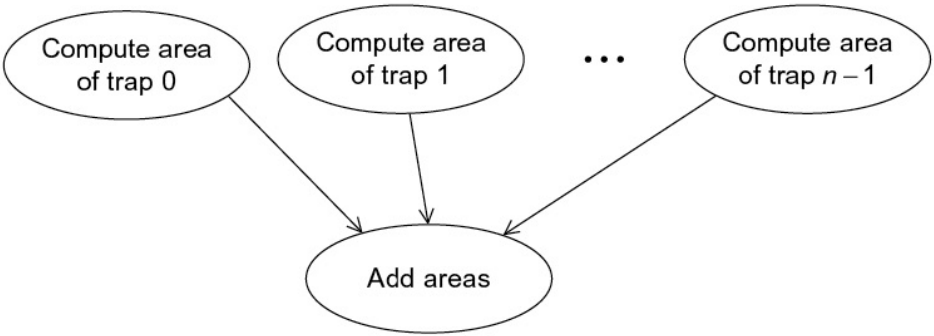


- Partition problem into small tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores.

Parallel pseudo-code

```
Get a, b, n;
h = (b-a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
  Send local_integral to process 0;
else * my_rank == 0 */
  total_integral = local_integral;
  for (proc = 1; proc < comm_sz; proc++) {
    Receive local_integral from proc;
    total_integral += local_integral;
  }
}
if (my_rank == 0)
  print result;
```

Tasks and communications for Trapezoidal Rule



```
int main(void) {
  int my_rank, comm_sz, n = 1024, local_n;
  double a = 0.0, b = 3.0, h, local_a, local_b;
  double local_int, total_int;
  int source;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

  h = (b-a)/n; /* h is the same for all processes*/
  local_n = n/comm_sz; /* So is the number of trapezoids*/

  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  local_int = Trap(local_a, local_b, local_n, h);
```

```
  if (my_rank != 0) {
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
        MPI_COMM_WORLD);
  } else {
    total_int = local_int;
    for (source = 1; source < comm_sz; source++) {
      MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      total_int += local_int;
    }
  }

  if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15e\n",
      a, b, total int);
  }
  MPI_Finalize();
  return 0;
} /* main */
```

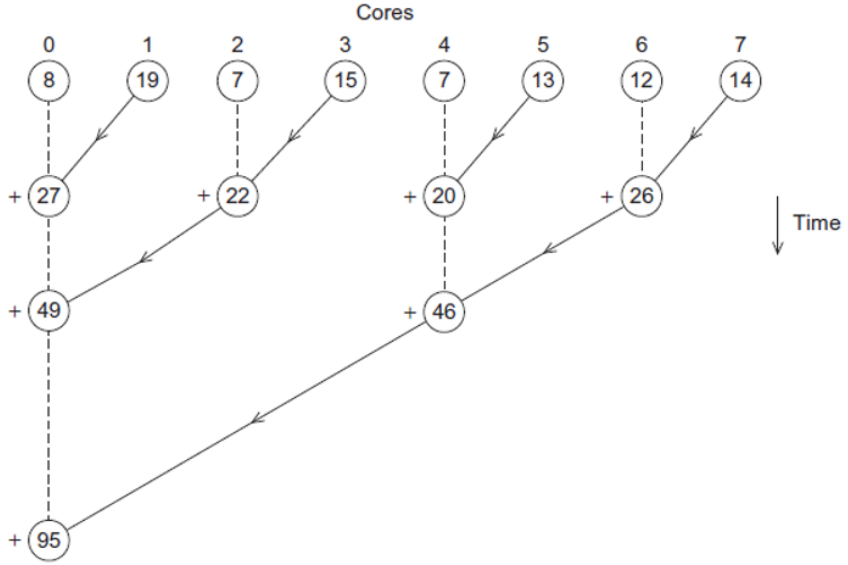
Outline

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.
- Process 0 must read the data (scanf) and send to the other processes.

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

Function for reading user input


```
void Get_input(
  int my_rank /* in */,
  int comm_sz /* in */,
  double* a_p /* out */,
  double* b_p /* out */,
  int* n_p /* out */) {
    int dest;

    if (my_rank == 0) {
      printf("Enter a, b, and nnn");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
        MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
    }else { /* my rank != 0 */
       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
} /* Get input */
```


Cores
0
1
2
3
4
5
6
7
8
19
7
15
7
13
12
14
+ 27
+ 22
+ 20
+ 26
Time
+ 49
+ 46
+ 95

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

C binding

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
            MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf page 224.

Predefined Reduction Operations in MPI

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf page 226.

```
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_int = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
     MPI_COMM_WORLD);
```

- All the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to MPI Reduce on one process with a call to MPI Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
- Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags, so they're matched solely on the basis of the communicator and the order in which they're called.

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /* in  */);
```

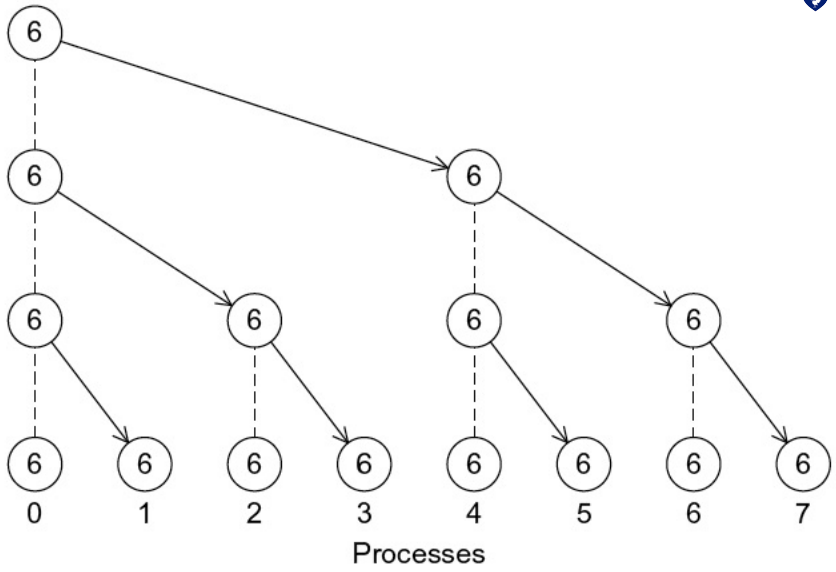
- Data belonging to a single process is sent to all of the processes in the communicator.

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	datatype of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

C binding

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm)
```



A tree-structured broadcast

A Version of Get_inpt that Uses MPI_Bcast


```
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */) {

      if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
      }
      MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } /* Get input */
```

Elapsed Serial Time

```
double start, finish;
...
GET_TIME(start);

/* Code to be timed */

...
GET_TIME(finish);

printf("Elapsed time = %e seconds \n", finish - start);
```

Elapsed Parallel Time

```
double start, finish;
...
start = MPI_Wtime();

/* Code to be timed */

...
finish = MPI_Wtime();

printf("Proc %d > Elapsed time = %e seconds \n", my_rank, finish - start);
```

- Speedup: Ratio of execution time on one process to that on p processes

$$Speedup = \frac{t_1}{t_p} \tag{3}$$

- Efficiency: Speedup per process

$$Efficiency = \frac{t_1}{t_p \times p} \tag{4}$$

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of a Parallel Application

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

- A program is scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable.
- Programs that can maintain a constant efficiency if the problem size increase at the same rate as the number of processes are sometimes said to be weakly scalable.

- The MPI standard allows MPI_Send to behave in two different ways:
  - it can simply copy the message into an MPI managed buffer and return,
  - or it can block until the matching call to MPI_Recv starts.

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
- Relatively small messages will be buffered by MPI_Send.
- Larger messages, will cause it to block.

- If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.
- Each process is blocked waiting for an event that will never happen.

- A program that relies on MPI provided buffering is said to be unsafe.
- Such a program may run without problems for various sets of input, but it may hang or crash with other sets.

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won't hang or crash.

Other Collective Operations

- MPI_Scatter
  - Send data from root to all processes
- MPI_Gather
  - Gather data from all processes to the root
- And many more...

Conclusions

- Design parallel algorithm
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping
- Basic MPI programming
  - MPI_Send / MPI_Recv ...
- Advance MPI
  - MPI_Reduce, MPI_Bcast ...

- Shared-memory Programming with Pthreads