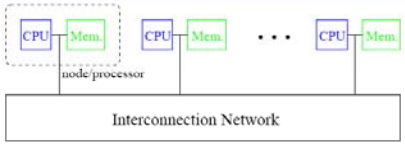
第十四章
基于消息传递的并行编程

Message Passing Programs

- Separate processors.
- Separate address spaces.
- Processors execute independently and concurrently.
- Processors transfer data cooperatively.
- Single Program Multiple Data (SPMD)
  - ▷ All processors are executing the same program, but act on different data.
- Multiple Program Multiple Data (MPMD)
  - ▷ Each processors may be executing a different program.
- Common software tools: PVM, MPI.

Message Passing Programming



- Each processor has its own private memory and address space.
- The processors communicate with one another through network.
- Ideally, each node is directly connected to every other node → too expensive to build.
- A compromise is to use crossbar switches connecting the processors.
- Use simple topology: e.g. linear array, ring, mesh, hypercube.
- Comm. time is the bottleneck of message passing programming.

Topics for Today

- **Principles of message passing**
- **Building blocks**
  - **—send**
  - **—receive**
- **MPI: Message Passing Interface**
- **Topologies and embedding**
- **Overlapping communication with computation**
- **Collective communication and computation**
- **Groups and communicators**

Message Passing Overview

- The logical view of a message-passing platform
  - —p processes
  - —each with its own exclusive address space
- All data must be explicitly partitioned and placed
- All interactions (read-only or read/write) are two-sided
  - —process that has the data
  - —process that wants to access the data
- Typically use single program multiple data (SPMD) model
- The bottom line …
  - —awkward to program
  - —simpler performance model: underlying costs are explicit

3

Send and Receive

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
P0                  P1
a = 100;            receive(&a, 1, 0)
send(&a, 1, 1);     printf("%d\n", a);
a = 0;
```

- The semantics of send
  - —value received by process P1 must be 100, not 0
  - —motivates the design of send and receive protocols

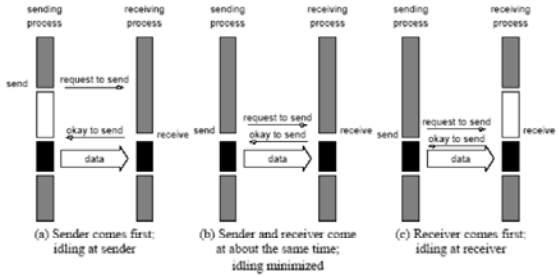
4

Blocking Message Passing

- Non-buffered blocking sends
  - —send does not return until the matching receive executes
  - —issues
    - – idling
    - – deadlocks
- Buffered blocking sends
  - —sender copies the data into the designated buffer
  - —returns after the copy completes
  - —data is copied into buffer at the receiver as well
  - —buffering avoids idling at the expense of copying

5

Non-Buffered, Blocking Message Passing



Handshaking for blocking non-buffered send/receive

Idling occurs when operations are not simultaneous

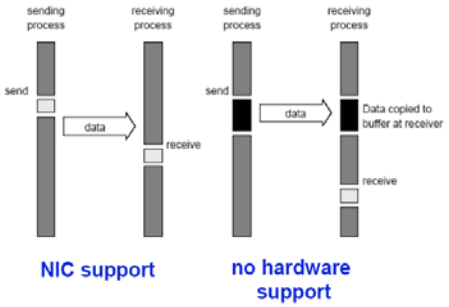
6

Buffered, Blocking Message Passing

- Buffers at the sending and receiving ends
  —avoid idling and deadlock
- Sender copies the data into the her buffer and returns
- Data is buffered at the receiver as well
- Assessment
  —buffering trades idling overhead for buffer copying overhead

7

Buffered, Blocking Message Passing



8

Buffered Blocking Message Passing

Bounded buffer sizes can have significant impact on performance

```
P0                                P1
for (i = 0; i < 1000; i++){       for (i = 0; i < 1000; i++){
    produce_data(&a);                 receive(&a, 1, 0);
    send(&a, 1, 1);                   consume_data(&a);
  }                               }
```

What if consumer is much slower than producer?

9

Buffered Blocking Message Passing

Deadlocks are possible with buffering
since receive operations block

```
P0                        P1
receive(&a, 1, 1);        receive(&a, 1, 0);
send(&b, 1, 1);           send(&b, 1, 0);
```

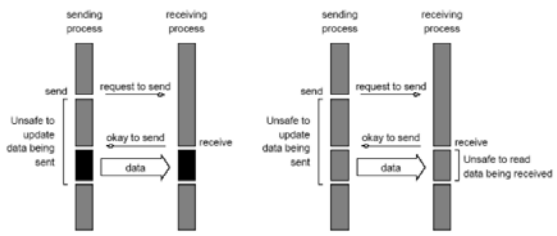
10

Non-Blocking Message Passing

- Non-blocking protocols
  - —send and receive return before it is safe
    - – sender: data can be overwritten before it is sent
    - – receiver: can read data out of buffer before it is received
    - – guaranteeing semantics is programmer responsibility
  - —status check operation to ascertain completion
- Benefit
  - —capable of overlapping communication with useful computation
- Typically both blocking and non-blocking primitives available

11

Non-Blocking Message Passing



no hardware support

NIC support

12

Send and Receive Protocols

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

Space of possible protocols for send and receive

13

MPI: the Message Passing Interface

- Standard library for message-passing
  - —portable
  - —almost ubiquitously available
  - —high performance
  - —C and Fortran APIs
- MPI standard defines
  - —syntax of library routines
  - —semantics of library routines
- Details
  - —MPI routines, data-types, and constants are prefixed by "MPI_"
- Simple to get started
  - —fully-functional programs using only six library routines

14

MPI: the Message Passing Interface

Minimal set of MPI routines

MPI_Init	initialize MPI
MPI_Finalize	terminate MPI
MPI_Comm_size	determine number of processes in group
MPI_Comm_rank	determine id of calling process in group
MPI_Send	send message
MPI_Recv	receive message

15

Starting and Terminating the MPI Programs

- `int MPI_Init(int *argc, char ***argv)`
  - —must call prior to other MPI routines
  - —effects
    - – strips off and processes any MPI command-line arguments
    - – initializes MPI environment
- `int MPI_Finalize()`
  - —must call at the end of the computation
  - —effect
    - – performs various clean-up tasks to terminate MPI environment
- Return codes
  - —MPI_SUCCESS
  - —MPI_ERROR

16

Communicators

- `MPI_Comm`: communicator = communication domain
  - —group of processes that can communicate with one another
- Supplied as arguments to all MPI message transfer routines
- Process can belong to many different communication domains
  - —domains may overlap
- `MPI_COMM_WORLD`: root communicator
  - — includes all the processes

17

Communicator Inquiry Functions

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - —determine the number of processes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - —index of the calling process
  - —0 ≤ rank < communicator size

18

"Hello World" Using MPI

```
#include <mpi.h>

main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello World!\n",
                myrank, npes);
        MPI_Finalize();
}
```

19

Sending and Receiving Messages

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Message source or destination
  - —index of process in the communicator
  - —receiver wildcard: `MPI_ANY_SOURCE`
    - – any process in the communicator can be source
- Message-tag: integer values, 0 ≤ tag < MPI_TAG_UB
  - —receiver tag wildcard: `MPI_ANY_TAG`
    - – messages with any tag are accepted
- Receiver constraint
  - — message size ≤ buffer length specified

20

- $ mpicc -o mpi_hello mpi_hello.c

在并行环境中的多台计算机上同时执行该程序，要将编译好的程序复制到不同的机器上

- $ mpirun -np 6 mpi_hello

```
From process 0 out of 6, Hello World!
From process 1 out of 6, Hello World!
From process 3 out of 6, Hello World!
From process 2 out of 6, Hello World!
From process 4 out of 6, Hello World!
From process 5 out of 6, Hello World!
```

MPI Data Types

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 bits
MPI_PACKED	packed sequence of bytes

21

Receiver Status Inquiry

- Mpi_Status
  - —stores information about an MPI_Recv operation
  - —data structure

```
typedef struct MPI_Status {
   int MPI_SOURCE;
   int MPI_TAG;
   int MPI_ERROR; };
```

- int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
  - —returns the count of data items received

22

Deadlock Pitfalls

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

destination

tag

Deadlock if MPI_Send is blocking

23

More Deadlock Pitfalls

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
         MPI_COMM_WORLD);

MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
         MPI_COMM_WORLD);
...
```

Deadlock if MPI_Send is blocking

24

Avoiding Deadlock

Break the circular wait

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank%2 == 1) {    // odd processes send first, receive second
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                 MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                 MPI_COMM_WORLD);
}
else {                  // even processes receive first, send second
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                 MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                 MPI_COMM_WORLD);
}
...
```

25

Message Exchange

To exchange messages (both send and receive)

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
   MPI_Datatype senddatatype, int dest, int
   sendtag, void *recvbuf, int recvcount,
   MPI_Datatype recvdatatype, int source, int recvtag,
   MPI_Comm comm, MPI_Status *status)
```

Requires both send and receive arguments

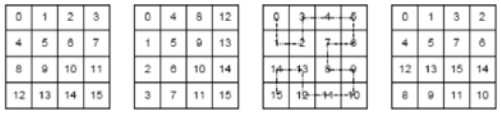
To use same buffer for both send and receive

```
int MPI_Sendrecv_replace(void *buf, int count,
   MPI_Datatype datatype, int dest, int sendtag,
   int source, int recvtag, MPI_Comm comm,
   MPI_Status *status)
```

26

Topologies and Embeddings

- Processor ids in `MPI_COMM_WORLD` can be remapped
  - —higher dimensional meshes
  - —space-filling curves



(a) Row-major mapping (b) Column-major mapping (c) Space-filling curve mapping (d) Hypercube mapping

- Goodness of any mapping
  - — determined by the interaction pattern
    - – program
    - – topology of the machine
  - —MPI does not provide any explicit control over these mappings

27

```
int a[10],b[10],npes,myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD,&npes);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_SendRecv(a,10,MPI_INT,(myrank+1)%npes, 1,
   b,10,MPI_INT,(myrank-1+npes)%npes, 1,
   MPI_COMM_WORLD, &status);
...
```

Cartesian Topologies

- `int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)`
- Map processes into a mesh
  - —dimensions = `ndims`
  - —length of each dimension in `dims`
- Processor identity in cartesian topology
  - —a vector of dimension `ndims`

28

Using Cartesian Topologies

- Sending and receiving still requires 1-D ranks
- Map Cartesian coordinates ⇔ rank

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
                   int *coords)

int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

- Most common operation on cartesian topologies is a shift
- Determine the rank of source and destination of a shift

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
                   int *rank_source, int *rank_dest)
```

29

```
MatrixMatrixMultiply(int n, double *a, double *b,
                     double *c, MPI_Comm comm) {
  int i, nlocal, npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsrc, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;
  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);
  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);
  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;
  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Cannon's Matrix-Matrix Multiplication with MPI's Topologies

- n The dimension of the matrices
- The parameters a, b, and c point to the locally stored portions of the matrices *A*, *B*, and *C*, respectively.
- The size of these arrays is $n/\sqrt{p} \times n/\sqrt{p}$, where *p* is the number of processes.
- This routine assumes that *p* is a perfect square and that *n* is a multiple of $\sqrt{p}$.
- The parameter comm stores the communicator describing the processes that call the MatrixMatrixMultiply function.

```
MatrixMatrixMultiply(int n, double *a,
     double *b, double *c, MPI_Comm comm)
```

```
  /* Get the rank and coordinates */
  MPI_Comm_rank(comm_2d, &my2drank);
  MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

  /* Determine the dimension of the local matrix block */
  nlocal = n/dims[0];

  /* Perform the initial matrix alignment. */
  MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsrc
   &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
   shiftdest, 1, shiftsrc, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsrc,
          &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
          shiftdest, 1, shiftsrc, 1, comm_2d, &status);
```

```
  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

  /* Compute ranks of the up and left shifts */
  MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
  MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
        leftrank, 1, rightrank, 1, comm_2d, &status);
    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
        uprank, 1, downrank, 1, comm_2d, &status);
  }
  /* Restore the original distribution of a and b */
  MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsrc,
   &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsrc, 1, comm_2d, &status);
```

Overlapping Communication & Computation

- Non-blocking send and receive

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm,
          MPI_Request *request)
```

- Operations return before completion
- `MPI_Test`: has non-blocking request finished?

```
int MPI_Test(MPI_Request *request, int *flag,
    MPI_Status *status)
```

- `MPI_Wait`: block until request completes

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

30

```
  MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsrc, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
      shiftdest, 1, shiftsrc, 1, comm_2d, &status);

  MPI_Comm_free(&comm_2d); /* Free up communicator */
}

/* This function performs a serial matrix-matrix multiplication c = a*b */
MatrixMultiply(int n, double *a, double *b, double *c)
{
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      for (k=0; k<n; k++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

Avoiding Deadlocks

Using non-blocking operations avoids most deadlocks

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

tag

NB send or the receive avoids this deadlock

31

Non-Blocking Cannon's Matrix-Matrix Multiplication

- n The dimension of the matrices
- The use of the additional arrays *a_buffers* and *b_buffers*, that are used as the buffer of the blocks of *A* and *B* that are being received while the computation involving the previous blocks if performed.
-

```
MatrixMatrixMultiply_NonBlocking(int n, double *a,
                    double *b, double *c, MPI_Comm comm)
{
   int i, j, nlocal, npes, dims[2], periods[2];
   int myrank, my2drank, mycoords[2];
   int uprank, downrank, leftrank, rightrank, coords[2];
   int shiftsrc, shiftdest;
    double *a_buffers[2], *b_buffers[2];
    MPI_Status status; MPI_Comm comm_2d;
    MPI_Request reqs[4];
/* Get the communicator related information */
   MPI_Comm_size(comm, &npes);
   MPI_Comm_rank(comm, &myrank);
   /* Set up the Cartesian topology */
   dims[0] = dims[1] = sqrt(npes);
   /* Set the periods for wraparound connections */
   periods[0] = periods[1] = 1;
   /* Create the Cartesian topology, with rank reordering */
   MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

```
   /* Get the rank and coordinates */
   MPI_Comm_rank(comm_2d, &my2drank);
   MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
   /* Determine the dimension of the local matrix block */
   nlocal = n/dims[0];
    /* Setup the a_buffers and b_buffers array. */
    a_buffers[0] = a;
a_buffers[1]=(double *)malloc(nlocal*nlocal*sizeof(double));
    b_buffers[0] = b;
b_buffers[1]=(double *)malloc(nlocal*nlocal*sizeof(double));
    /* Perform the initial matrix alignment. */
    MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsrc
    &shiftdest);
MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal,MPI_DOUBLE,
    shiftdest, 1, shiftsrc, 1, comm_2d, &status);
   MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsrc,
    &shiftdest);
MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal,MPI_DOUBLE,
            shiftdest, 1, shiftsrc, 1, comm_2d, &status);
```

```
  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
   MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
     MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
     MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
            leftrank, 1, comm_2d, &reqs[0]);
       MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
            uprank, 1, comm_2d, &reqs[1]);
MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
            rightrank, 1, comm_2d, &reqs[2]);
MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
            downrank, 1, comm_2d, &reqs[3]);
MatrixMultiply(nlocal, a_buffer[i%2], b_buffer[i%2], c);
/*c=c+a*b*/
      for (j=0;j<4;j++) MPI_Wait(&reqs[j],&status);
     }
   /* Restore the original distribution of a and b */
   MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsrc,
    &shiftdest);
   MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
shiftdest, 1, shiftsrc, 1, comm_2d, &status);
```

```
MPI_Cart_shift(comm_2d,1,+mycoords[1],&shiftsrc,&shiftdest);
  MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
shiftdest, 1, shiftsrc, 1, comm_2d, &status);

  MPI_Comm_free(&comm_2d); /* Free up communicator */
    free(a_buffers[1]); free(b_buffers[1]);
}

/* This function performs a serial matrix-matrix multiplication c = a*b */
MatrixMultiply(int n, double *a, double *b, double *c)
{
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      for (k=0; k<n; k++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

Other Collective Comm.

- Scatter (MPI_Scatter)
  - ▷ Split the data on proc *root* into p segments.
  - ▷ The 1st segement is sent to proc 0, the 2nd to proc 1, etc.
  - ▷ Similar to but more general than MPI_Bcast.
- Gather (MPI_Gather)
  - ▷ Collect the data from each proc and store the data on proc *root*.
  - ▷ Similar to but more general than MPI_Reduce.
- Can collect and store the data on *all* procs using MPI_Allgather.

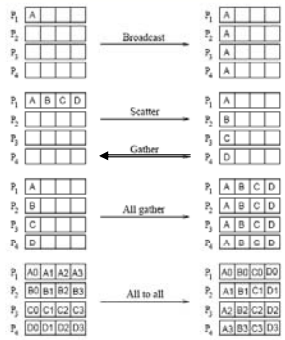
CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis 18

Collective Communication

- Comm. pattern involving a group of procs; usually more than 2.
- MPI_Barrier: synchronize all procs.
- Broadcast (MPI_Bcast)
  - ▷ A single proc sends the same data to every proc.
- Reduction (MPI_Reduce)
  - ▷ All the procs contribute data that is combined using a binary operation.
  - ▷ Example: max, min, sum, etc.
  - ▷ One proc obtains the final answer.
- Allreduce (MPI_Allreduce)
  - ▷ Same as MPI_Reduce but every proc contains the final answer.
  - ▷ Effectively as MPI_Reduce + MPI_Bcast, but more efficient.

CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis 17

Comparisons of Collective Comms.



CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis 19

Barrier Synchronization

```
MPI_BARRIER( comm )
[ IN comm] communicator (handle)

int MPI_Barrier(MPI_Comm comm )

MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

Broadcast

```
 MPI_BCAST( buffer, count, datatype, root, comm )
[ INOUT buffer] starting address of buffer (choice)
[ IN count] number of entries in buffer (integer)
[ IN datatype] data type of buffer (handle)
[ IN root] rank of broadcast root (integer)
[ IN comm] communicator (handle)

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

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of

group using the same arguments for comm, root. On return, the contents of root's communication buffer has been copied to all processes.

For example: Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
    int array[100];
    int root=0;
    ...
    MPI_Bcast( array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as comm in the above) have been assigned appropriate values.

Gather

```
 MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice, significant only at root)
[ IN recvcount] number of elements for any single receive
                (integer, significant only at root)
[ IN recvtype] data type of recv buffer elements
               (significant only at root) (handle)
[ IN root] rank of receiving process (integer)
[ IN comm] communicator (handle)

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
           RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to MPI_SEND and the root had executed n calls to MPI_RECV.

Gather data at all processes

```
int MPI_Allgather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          MPI_Comm comm)
```


Scatter

```
MPI_SCATTER( sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, root, comm)
[ IN sendbuf] address of send buffer (choice, significant only at root)
[ IN sendcount] number of elements sent to each process
                (integer, significant only at root)
[ IN sendtype] data type of send buffer elements
               (significant only at root) (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcount] number of elements in receive buffer (integer)
[ IN recvtype] data type of receive buffer elements (handle)
[ IN root] rank of sending process (integer)
[ IN comm] communicator (handle)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```



```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
            RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

The outcome is as if the root executed n send operations, and each process executed a receive.

All-to-All

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, com
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements sent to each process (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcount] number of elements received from any process (integer)
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

Reduce

```
 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 (integer)
[ IN datatype] data type of elements of send buffer (handle)
[ IN op] reduce operation (handle)
[ IN root] rank of root process (integer)
[ IN comm] communicator (handle)

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

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```


MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence.

There are a series of pre-defined operations

```
[ 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 xor
[ MPI_BXOR] bit-wise xor
[ MPI_MAXLOC] max value and location
[ MPI_MINLOC] min value and location
```

The user can also define global operations to perform on distributed data with MPI_OP_CREATE.

Limitations of Point-To-Point Communication

So far, all point-to-point communication has involved contiguous buffers containing elements of the same fundamental types provided by MPI such as MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, etc. This is rather limiting since one often wants to:

- Send messages with different kinds of datatypes (say a few integers followed by a few reals.)
- Send non-contiguous data (such as a sub-block of a matrix or an unstructured subset of a one-dimensional array.)

These two goals could easily be accomplished by either sending a number of messages with all data of one type and by copying the non-contiguous data to a contiguous array that is later sent in a single message.

CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis 32

There are several disadvantages

- A larger number of messages than necessary may need to be sent.
- Additional inefficient memory-to-memory copying may be required to fill contiguous buffers.
- Resulting code is needlessly complicated.

MPI *derived datatypes* allow us to carry out these operations: we can send/receive messages with non-contiguous data of different types without the need for multiple messages or additional memory-to-memory copying.

CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis 33

```
int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm) {
  int i, j, nlocal, npes, myrank;
  int *sorted_elmnts, *splitters, *allpicks;
  int *scounts, *sdispls, *rcounts, *rdispls;
  /* Get communicator-related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);
  nlocal = n/npes;
  /* Allocate memory for the arrays that will store the splitters */
  splitters = (int *)malloc(npes*sizeof(int));
  allpicks = (int *)malloc(npes*(npes-1)*sizeof(int));
  /* Sort local array */
  qsort(elmnts, nlocal, sizeof(int), IncOrder);
  /* Select local npes-1 equally spaced elements */
  for (i=1; i<npes; i++)
        splitters[i-1] = elmnts[i*nlocal/npes];
```

```
  /* Gather the samples in the processors */
  MPI_Allgather(splitters, npes-1, MPI_INT, allpicks,
        npes-1, MPI_INT, comm);
  /* sort these samples */
  qsort(allpicks,npes*(npes-1),sizeof(int),IncOrder);
  /* Select splitters */
  for (i=1;i<npes;i++)
        splitters[i-1]=allpicks[i*npes];
  splitters[npes-1] = MAXINT;
  /* Compute the number of elements that belong to each bucket */
  scounts = (int *)malloc(npes*sizeof(int));
  for (i=0; i<npes; i++) scounts[i] = 0;
  for (j=i=0; i<nlocal; i++) {
    if (elmnts[i] < splitters[j])
      scounts[j]++;
    else
      scounts[++j]++;
  }
```

```
/* Determine the starting location of each bucket's elements in the elmnts */
/* array */
    sdispls = (int *)malloc(npes*sizeof(int));
    sdispls[0] = 0;
    for (i=1; i<npes; i++)
      sdispls[i] = sdispls[i-1]+scounts[i-1];
/* Perform an all-to-all to inform the corresponding processes of the number of */
/* elements they are going to receive. This information is stored in rcounts array */
    rcounts = (int *)malloc(npes*sizeof(int));
    MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1,
    MPI_INT, comm);
/* Based on rcounts determine where in the local array the data from each */
/* processor will be stored. This array will store the received elements */
/* as well as the final sorted sequence */
    rdispls = (int *)malloc(npes*sizeof(int));
    rdispls[0] = 0;
    for (i=1; i<npes; i++)
      rdispls[i] = rdispls[i-1]+rcounts[i-1];
    *nsorted = rdispls[npes-1]+rcounts[npes-1];
  sorted_elmnts = (int *)malloc((*nsorted)*sizeof(int));
```

PVM = Parallel Virtual Machine

- PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine.
- PVM transparently handles all
  - » Message routing
  - » Data conversion
  - » Process control
- across a network of incompatible computer architectures
- PVM developed at Oak Ridge National Lab by Geist, Dongarra, Beguelin, Manchek, Sunderam, and Jiang
- Source code free for download from the Net
- PVM came before MPI

CSE 160 Chien, Spring 2005 Lecture #13, Slide 8

```
/* Each process sends and receives the corresponding elements, using the */
/* MPI_Alltoallv operation. The arrays scounts and sdispls are used to specify */
/* the number of elements to be sent and where these elements are stored, */
/* respectively. The arrays rcounts and rdispls are used to specify the number of */
/* elements to be received, and where these elements will be stored, respectively. */
 MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT,
     sorted_elmnts, rcounts, rdispls, MPI_INT, comm);
 /* Perform the final local sort *没有采用归并/
 qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);
 free(splitters); free(allpicks); free(scounts);

 free(sdispls); free(rcounts); free(rdispls);
 return sorted_elmnts;
}
```

PVM Principles

- User-configured host pool
  - » Host pool may be modified dynamically during execution
- Transparent access to hardware: heterogeneity
  - » XDR data conversion used to address heterogeneous data formats
- Process-based communication
  - » Unit of parallelism in PVM is a task (single thread of control which alternates between computation and communication)
- Explicit message-passing model
  - » Tasks communicate by explicitly passing messages

CSE 160 Chien, Spring 2005 Lecture #13, Slide 9

PVM Communication

- Communication model provides point-to-point and collective communication constructs
- Point-to-point communications include
  - » Blocking receive
  - » Time-out blocking receive (receive blocks until specified time)
  - » Non-blocking receive
  - » Non-blocking send
  - » PVM "blocking send" returns as soon as send buffer is free for reuse
- Collective communications include
  - » Multicast
  - » Broadcast
  - » Reduce

CSE 160 Chien, Spring 2005

"Hello" program

```
main()
{
  int cc, tid, msgtag;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());
  cc = pvm_spawn("hello_other", NULL, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Prints task ID

Spawns another PVM program

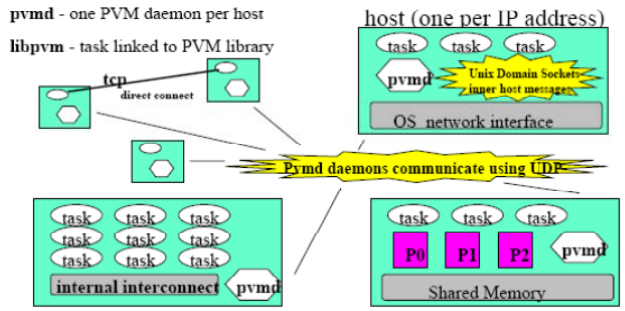
Blocking receive from spawned process

Unpacks a null terminated character string

Prints message stored in buf (from spawned process) as well as task ID

CSE 160 Chien, Spring 2005 Lecture #13, Slide 12

PVM Architecture



distributed memory MPP

Exploits native communication within Parallel Machines

Shared memory multiprocessor

Pvmd's can perform message passing thru shared memory segments

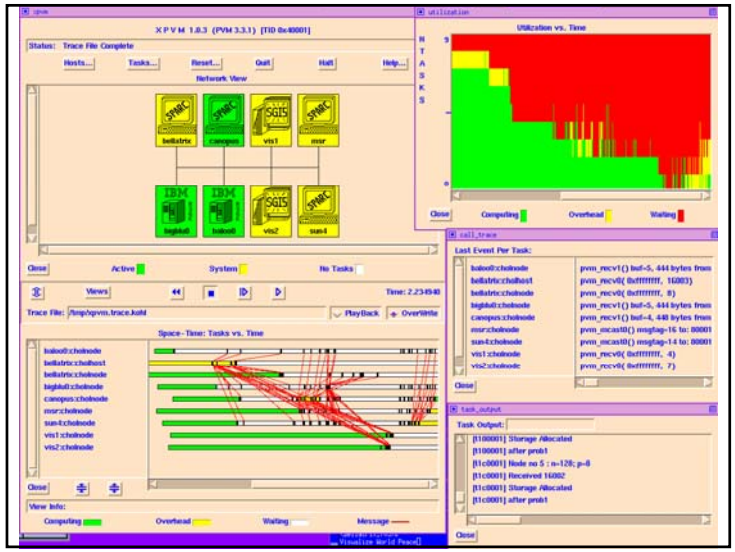
CSE 160 Chien, Spring 2005

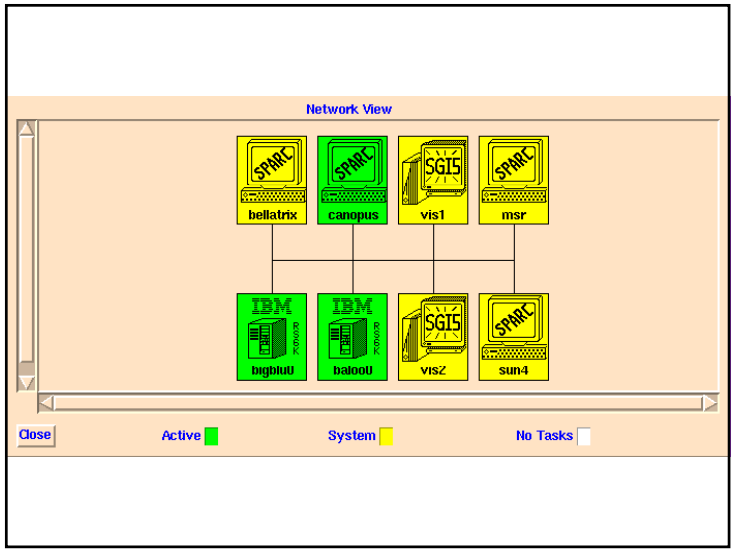
A simple program

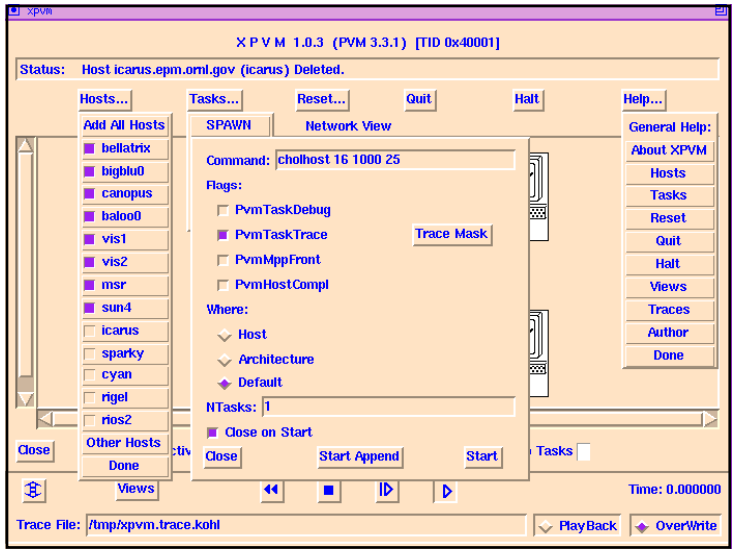
Process 0	Process 1
int x=1, y=0	int x=0, y=1
Recv (x) Send(x,1) Print x, y	Recv(y) Send(y,0) Print x, y

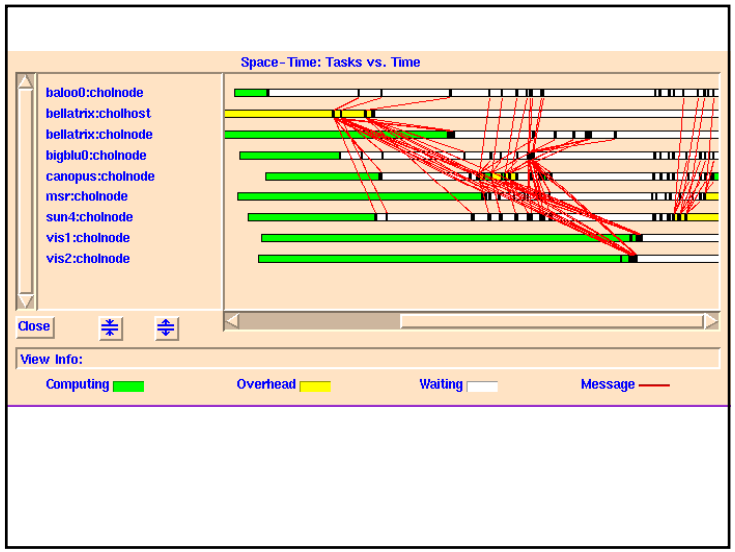
What is the outcome of this program?

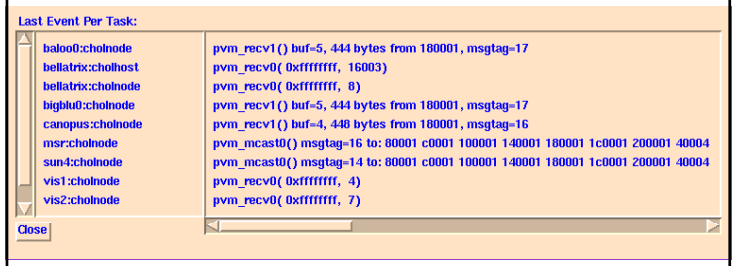
CSE 160 Chien, Spring 2005 Lecture #13, Slide 14




Network View
bellatrix
canopus
vis1
msr
bigblu0
baloo0
vis2
sun4
Close
Active
System
No Tasks


X P V M 1.0.3 (PVM 3.3.1) [TID 0x40001]
Status: Host icarus.epm.ornl.gov (icarus) Deleted.
Hosts...
Tasks...
Reset...
Quit
Halt
Help...
Add All Hosts
SPAWN
Network View
General Help:
About XPVM
Hosts
Tasks
Reset
Quit
Halt
Views
Traces
Author
Done
bellatrix
bigblu0
canopus
baloo0
vis1
vis2
msr
sun4
icarus
sparky
cyan
rigel
rios2
Other Hosts
Done
Command: cholhost 16 1000 25
Flags:
PvmTaskDebug
PvmTaskTrace
PvmMppFront
PvmHostCompl
Trace Mask
Where:
Host
Architecture
Default
NTasks: 1
Close on Start
Close
Start Append
Start
Close
Views
Time: 0.000000
Trace File: /tmp/xpvm.trace.kohl
PlayBack
OverWrite


Space-Time: Tasks vs. Time
baloo0:cholnode
bellatrix:cholhost
bellatrix:cholnode
bigblu0:cholnode
canopus:cholnode
msr:cholnode
sun4:cholnode
vis1:cholnode
vis2:cholnode
Close
View Info:
Computing
Overhead
Waiting
Message



提交方式

- 同时提交给：
  - zyl9910@mail.xjtu.edu.cn
  - Yinliang_zhao@163.com
- 标题：姓名 学号 并行计算作业
- 提交文档一律用附件，请打包成1个文件，这个文件在一个目录下面，目录名称为姓名学号
  - 例如，目录名为：张三1234567
- 作业提交截至日期，本学期第19周周五

后两周安排

- 每人10-15分钟，共12-20人
- 第15周两次安排为Presentation
- 第16周周二复习，周四考试

- Cilk编程
  - 编写NQueens问题的程序，打印出1个解。
  - 编写FFT程序
  - TSP问题。http://www.tsp.gatech.edu/
- MPI编程
  - 非方阵矩阵乘法
  - FFT
  - TSP
  - 高斯消去法解线性方程组
  - 3D-Mesh上的排序
- OpenMP
  - 高斯消去法
  - 矩阵乘法

- 算法综述
  - 典型并行数据挖掘算法
  - 并行BLAST
  - Linpack及其应用
- 用PCAM方法学设计算法
  - 选择数值计算中的计算量大的问题，如有限元、流体力学、。。。