SILICON GRAPHICS | The Source of Innovation and Discovery™

SGI Altix
MPI - Message Passing Interface
Programming and Tuning

Reiner Vogelsang

SGI GmbH

reiner@sgi.com

January 19, 2005

sgi®

Module Objectives

After completing this module, you will be able to

- **Define MPI**
- **Describe why message passing is a viable parallel programming paradigm**
- **Explain why MPI is a popular message passing library**
- **Identify common MPI components**
- **Write simple parallel programs using MPI calls**
- **Tune MPI applications**

SGI Altix : MPI - Message Passing Interface

Programming

Message Passing

- **Explicit parallel programming**
  - Programmer inserts communication calls into the program ``manually''
  - All processors execute all the code
- **Based on ``message'' transmittal**
  - Message consists of status and, usually, data
- **Offers point-to-point (process-to-process) or global (broadcast) messages**
- **Normally requires a sender and a receiver**
  - However, MPI-2 allows one-sided communication
  - Processes within the cache coherence domains on Altix do have direct access to each other's memory

Why Message Passing?

- Only way to program parallel applications for non-shared memory systems
- Gives programmer 100% control about how to divide the problem
- Can perform better than implicit methods
- Portable -- does not require a shared memory machine

What Is MPI?

- The de-facto standard message passing library
  - Similar functionality to PVM and other libraries
- Goals
  - Provide source-code portability
  - Allow efficient implementation
  - Functionality
- Callable from Fortran, C, C++
  - MPI 1.2 has 129 routines plus 13 ``deprecated'' ones (big!)
  - MPI-2 adds 157 routines (bigger!)
  - Subset of 6 is enough to do basic communications (small!)

Which MPI ?

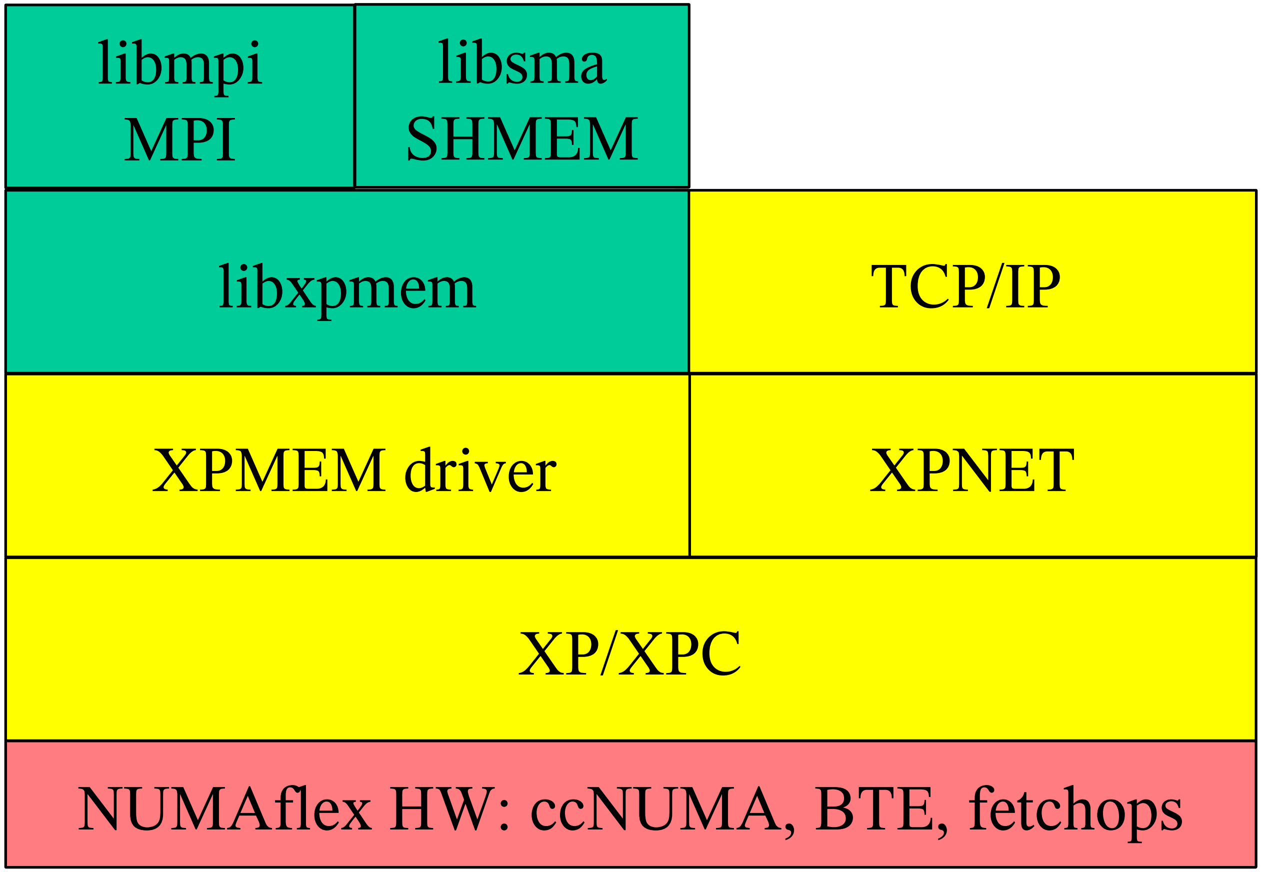
Commonly used MPI libraries:

- **MPI-lam**
- **mpich**
- **SGI-MPI**

SGI-MPI is compatible to MPI 1.2 with most MPI 2 extensions …

….”Let us know if you miss anything”

ccNUMA Software Layers



MPI Header Files and Functions

- **Header file**
  - **Fortran**

```
INCLUDE 'mpif.h'
```

- **C/C++**

```
#include <mpi.h>
```

- **Function format**
  - **Fortran**

```
CALL MPI_xxx (...., ISTAT)
```

  - **C/C++**

```
int stat = MPI_Xxx (....);
```

MPI Startup and Shutdown

- **MPI initialization**
  - **Fortran**

```
CALL MPI_INIT (istat)
```

  - **C/C++**

```
MPI_Init (int *argc, char ***argv);
```

  - **Must be called *before* any other MPI calls**
- **MPI termination**
  - **Fortran**

```
CALL MPI_FINALIZE (istat)
```

  - **C/C++**

```
int MPI_Finalize (void);
```

  - **Must be called *after* all other MPI calls**

Communicator and Rank

- **Communicator**
  - **Group of processes, either system or user defined**
  - **Default communicator is** `MPI_COMM_WORLD`
  - **Use the function** `MPI_COMM_SIZE` **to determine how many processes are in the communicator**
- **Rank**
  - **Process number (zero based) within the communicator**
  - **Use the function** `MPI_COMM_RANK` **to determine which process is currently executing**

Compiling MPI Programs

icc `prog.c -lmpi`

```
icc prog.C -lmpi
```

```
ifort prog.f -lmpi
```

Launching MPI Programs

- **On most machines, the** `mpirun` **command launches MPI applications:**

  `mpirun -np` *num_Procs user_executable* [*user_args*]

- **Launching a program to run with 5 proccesses on one computer**

  `% mpirun -np 5 ./a.out`

- **Example: Launching a program to run with 64 processes on each of two systems**

  `% mpirun host1,host2 64 ./a.out`

Example: simple1_mpi.c

```
#include <mpi.h>
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
int num_procs;
  int my_proc;

/* Initialize MPI */
   MPI_Init(&argc, &argv);
/* Determine the size of the communicator */
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
/* Determine processor number */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);

     if (my_proc == 0)
     printf("I am process %d. Total number of \
           processes: %d\n", my_proc,num_procs);

  /* Terminate MPI */
   MPI_Finalize();
}
```

Example: simple1_mpi.c (continued)

```
% icc simple1_mpi.c -lmpi
% mpirun -np 5 ./a.out
I am process 0. Total number of processes: 5
```

Example: simple1_mpi.f

```
  program simple1
  include 'mpif.h'
C Initialize MPI
      call mpi_init(istat)
C Determine the size of the communicator
      call mpi_comm_size(mpi_comm_world, num_procs,
    &                 ierr)
C Determine processor number
      call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
      if (my_proc .eq. 0)
     &   write(6,1) 'I am process ',myproc,
     &   '. Total number of processes: ',num_procs
1        format(a,i1,a,i1)
C Terminate MPI
  call mpi_finalize(ierr)
     end
% ifort simple1_mpi.f -lmpi
% mpirun -np 5 ./a.out
I am process 0. Total number of processes: 5
```

MPI Basic (Blocking) Send Format

- **C/C++ synopsis**

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

- **Fortran synopsis**

```
CALL MPI_SEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, ISTAT)
 <type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, ISTAT
```

- **Standard allows for implementation to choose buffering scheme**
- `buf` **contains the array of data to be sent**
- `MPI_Datatype` **is one of several predefined types or a derived (user-defined) type**

MPI Basic (Blocking) Receive Format

- **C/C++ synopsis**

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

- **Fortran synopsis**

```
CALL MPI_RECV (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
              STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS (MPI_STATU
 S_SIZE)
```

- `MPI_ANY_SOURCE` **and** `MPI_ANY_TAG` **can be put in as wildcards when exact source/tag is not known or is not critical to the application**
- **`buf` contains the array of data to be received**
- **`MPI_Datatype` is one of several pre-defined types or a derived (user-defined) type**

Elementary Data Types

MPI	Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER (1)
MPI_BYTE	
MPI_PACKED	

Elementary Data Types

MPI	C/C++
MPI_CHAR	signed char
MPI_SHORT	singed 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	
MP_PACKED	

Example: simple2_mpi.c

```
#include <mpi.h> #include <stdio.h>
#define N 1000
main(argc, argv)
int argc;
char *argv[];
{
   int num_procs;
   int my_proc;
   int init, size, rank, send, recv, final;
   int i, j, other_proc, flag = 1;
   double sbuf[N], rbuf[N];
   MPI_Status recv_status;

  /* Initialize MPI */
   if ((init = MPI_Init(&argc, &argv)) != MPI_SUCCESS) {
      printf("bad init\n");
      exit(-1);
   }
```

Example: simple2_mpi.c (continued)

```
/* Determine the size of the communicator */
   if ((size = MPI_Comm_size(MPI_COMM_WORLD, &num_procs))
             != MPI_SUCCESS) {
      printf("bad size\n");
      exit(-1);
   }
/* Make sure we run with only 2 processes */
   if (num_procs != 2) {
      printf("must run with 2 processes\n");
      exit(-1);
   }
/* Determine process number */
   if ((rank = MPI_Comm_rank(MPI_COMM_WORLD, &my_proc))
             != MPI_SUCCESS) {
      printf("bad rank\n");
      exit(-1);
   }
```

Example: simple2_mpi.c (continued)

```
  if (my_proc == 0) other_proc = 1;
  if (my_proc == 1) other_proc = 0;

  for (i = 0; i < N; i++)
      sbuf[i] = i;
/*Both processes send and receive data */
  if (my_proc == 0) {
      if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 99,
                  MPI_COMM_WORLD)) != MPI_SUCCESS) {
          printf("bad send on %d\n",my_proc);
          exit(-1);
      }
      if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 98,
                  MPI_COMM_WORLD, &recv_status))
              != MPI_SUCCESS) {
          printf("bad recv on %d\n", my_proc);
          exit(-1);
      }
```

Example: simple2_mpi.c (continued)

```
} else if (my_proc == 1) {
     if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 99,
                     MPI_COMM_WORLD, &recv_status))
                != MPI_SUCCESS) {
         printf("bad recv on %d\n", my_proc);
         exit(-1);
      }
      if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 98,
                     MPI_COMM_WORLD)) != MPI_SUCCESS) {
         printf("bad send on %d\n",my_proc);
         exit(-1);
      }
   }
/* Terminate MPI */
   if ((final = MPI_Finalize()) != MPI_SUCCESS) {
      printf("bad finalize \n");
      exit(-1);
   }
```

Example: simple2_mpi.c (continued)

```
/* Making sure clean data has been transferred */
   for(j = 0; j < N; j++) {
      if (rbuf[j] != sbuf[j]) {
         flag = 0;
         printf("processor %d: rbuf[%d]=%f. Should be %f\n",
                my_proc, j, rbuf[j], sbuf[j]);
      }
   }
   if (flag == 1) printf("Test passed on processor %d\n",
                         my_proc);
   else printf("Test failed on processor %d\n", my_proc);
}
% icc -w simple2_mpi.c -lmpi
% mpirun -np 2 ./a.out
Test passed on process 1
Test passed on process 0
```

Example: simple2_mpi.f

```
 program two_procs
include 'mpif.h'
parameter (n=1000)
integer other_proc
integer send, recv
integer status(mpi_status_size)
dimension sbuf(n), rbuf(n)

  call mpi_init(init)
if (init .ne. mpi_success) stop 'bad init'
call mpi_comm_size(mpi_comm_world, num_procs, ierr)
if (num_procs .ne. 2) stop 'npes not 2'
if (ierr .ne. mpi_success) stop 'bad size'
call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
if (jerr .ne. mpi_success) stop 'bad rank'
if (my_proc .eq. 0) other_proc = 1
if (my_proc .eq. 1) other_proc = 0

  do i = 1, n
sbuf(i) = i
enddo
```

Example: simple2_mpi.f (continue)

```
       if (my_proc .eq. 0) then
        call mpi_send(sbuf, n, mpi_real, other_proc, 99,
     &              mpi_comm_world, send)
        if (send .ne. mpi_success) stop 'bad 0 send'
        call mpi_recv(rbuf, n, mpi_real, other_proc, 98,
     &              mpi_comm_world, status, recv)
        if (recv .ne. mpi_success) stop 'bad 0 recv'
      else if (my_proc .eq. 1) then
        call mpi_recv(rbuf, n, mpi_real, other_proc, 99,
     &              mpi_comm_world, status, recv)
        if (recv .ne. mpi_success) stop 'bad 1 recv'
        call mpi_send(sbuf, n, mpi_real, other_proc, 98,
     &              mpi_comm_world, send)
        if (send .ne. mpi_success) stop 'bad 1 send'
      endif

        call mpi_finalize(ierr)
      if (ierr .ne. mpi_success)stop 'bad final'
      iflag = 1
```

Example: simple2_mpi.f (continue)

```
      do j = 1, n
        if (rbuf(j) .ne. sbuf(j)) then
          iflag = 0
          print*,'process ', my_proc, ':rbuf(', j, ')=',
     &         rbuf(j),'.Should be ',sbuf(j)
        endif
      enddo

       if (iflag .eq. 1) then
        print*,'Test passed on process ',my_proc
      else
        print*,'Test failed on process ',my_proc
      endif

        end
```

```
% ifort -w simple2_mpi.f -lmpi
% mpirun -np 2 ./a.out
Test passed on process 0
Test passed on process 1
```

Additional MPI Messaging Routines

- **Buffered messages**

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

- **Asynchronous messages**

```
MPI_Isend (buf, length, data_type, destination,
           message_tag, communicator, &request)
MPI_Ibsend(buf, count, datatype, dest, tag, comm, &request)
```

- **Return receipt messages**

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
 MPI_Issend(buf, count, datatype, dest, tag, comm, &request)
```

- **Notes**

When using buffered sends, the user must provide a usable buffer for MPI using an `MPI_Buffer_attach` command. Additional MPI calls are available to manage these buffers.

When using the asynchronous MPI calls, a handle is returned to the user. The user cannot modify/delete ``data'' until the message is completed or freed. See the next slide for checking the status of requests.

MPI Asynchronous Messaging Completion

`MPI_Wait(request, status)`

- Wait until request is completed

`MPI_Test(request, flag, status)`

- Logical flag indicates whether request has completed

`MPI_Request_free(request)`

- Removes request

Asynchronous Message Receipt

`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

`MPI_Iprobe(source, tag, comm, flag, status)`

- Checks for messages without blocking
- Probe will check for messages without receiving them

sgi®

Commonly Used MPI Features

- Point-to-point messages
- Collective operations
  - Broadcast
    - For example, one task reads in a data item and wants to send to all other tasks
  - Global reductions
    - Sums, products, minimums, maximums
- Derived data types
  - Necessary for noncontiguous patterns of data
- Functions to assist with topology grids
  - Convenience--no performance advantage

Collective Routines

- Called by all processes in the group
- Examples
  - Broadcast
  - Gather
  - Scatter
  - All-to-all broadcast
  - Global reduction operations (such as sums, products, max, and min)
  - Scan (such as partial sums)
  - Barrier synchronization

Synchronization

- **Format**
  - **C/C++ synopsis**

```
int MPI_Barrier(MPI_Comm comm)
```

  - **Fortran synopsis**

```
CALL MPI_BARRIER (COMM, ISTAT)
INTEGER COMM, ISTAT
```

- **Blocks the calling process until all processes have made the call**
  - **Ensures synchronization for time-dependent computations**
  - **Most commonly used synchronization routine**

Broadcast

- Format
- C/C++ synopsis

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

- Fortran synopsis

```
CALL MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT, COMM,
ISTAT) <TYPE> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, ISTAT
```

- Broadcasts a message from root to all processes in the group, including itself

Commonly Used MPI Features

- Point-to-point messages
- Collective operations
  - Broadcast
    - For example, one task reads in a data item and wants to send to all other tasks
  - Global reductions
    - Sums, products, minimums, maximums
- Derived data types
  - Necessary for noncontiguous patterns of data
- Functions to assist with topology grids
  - Convenience--no performance advantage

Example: bcast.c

```
#include <mpi.h> #include <stdio.h>
#define N 5
main(argc, argv)
int argc;
char *argv[];
{
   int num_procs, my_proc;
   int a[N];
   int i, j, root=0;
   for(i = 0; i < N; i++)
      a[i] = -11;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);
   if (my_proc == root) {
      for(i=0;i<N;i++)
         a[i] = -20;
   }
```

Example: bcast.c (continue)

```
MPI_Bcast((void *)a, N, MPI_INT, root, MPI_COMM_WORLD);
   for (j = 0; j < N; j++)
      printf("%d ",a[j]);
   printf("\n");
   MPI_Finalize();
}
% icc -w bcast.c -lmpi
% mpirun -np 7 ./a.out
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
```

Example: bcast.f

```
 program cast
include 'mpif.h'
parameter (n=5)
dimension buf(n)
integer root
parameter (root = 0)

 do i = 1, n
  buf(i) = -11.0
enddo
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, num_procs, ierr)
call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
if (my_proc .eq. root) then
  do i=1, n
    buf(i) = -20.0
  enddo
```

Example: bcast.f (continue)

```
      endif
     call mpi_bcast(buf,n,mpi_real,root,mpi_comm_world,
     &             istat)
      write(6,1) (buf(j), j=1,n)
    1 format(5(f5.1,1x))
      call mpi_finalize(ierr)
      end
```

% ifort -w bcast.f -lmpi

% mpirun -np 7 ./a.out

```
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
```

Reduction

- **Format**
  - **C/C++ synopsis**

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

  - **Fortran synopsis**

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

- **Combines the elements of `sendbuf` in each process in the group, using the operation op, and returns the results in `root` process's `recvbuf`**
- **MPI provides predefined operations for op:**

```
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND
MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR
MPI_MAXLOC, MPI_MINLOC
```

Example: reduce.c

```
#include <mpi.h>
#include <stdio.h>
#define N 5
main(int argc, char *argv[])
{
   int num_procs;
   int my_proc;
   int myarr[N];
   int global_res[N];
   int i;

    MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);
   for (i = 0; i< N; i++)
      myarr[i] = my_proc+i+1;
   MPI_Reduce((void *) myarr, (void *) global_res, N, MPI_INT,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

Example: reduce.c (continue)

```
  if (my_proc == 0) {
      for(i = 0; i< N ; i++)
         printf("%d\n",global_res[i]);
         printf("\n");
   }
   MPI_Finalize();
}
% icc -w reduce.c -lmpi
% mpirun -np 5 ./a.out
15
20
25
30
35
```

Example: reduce.f

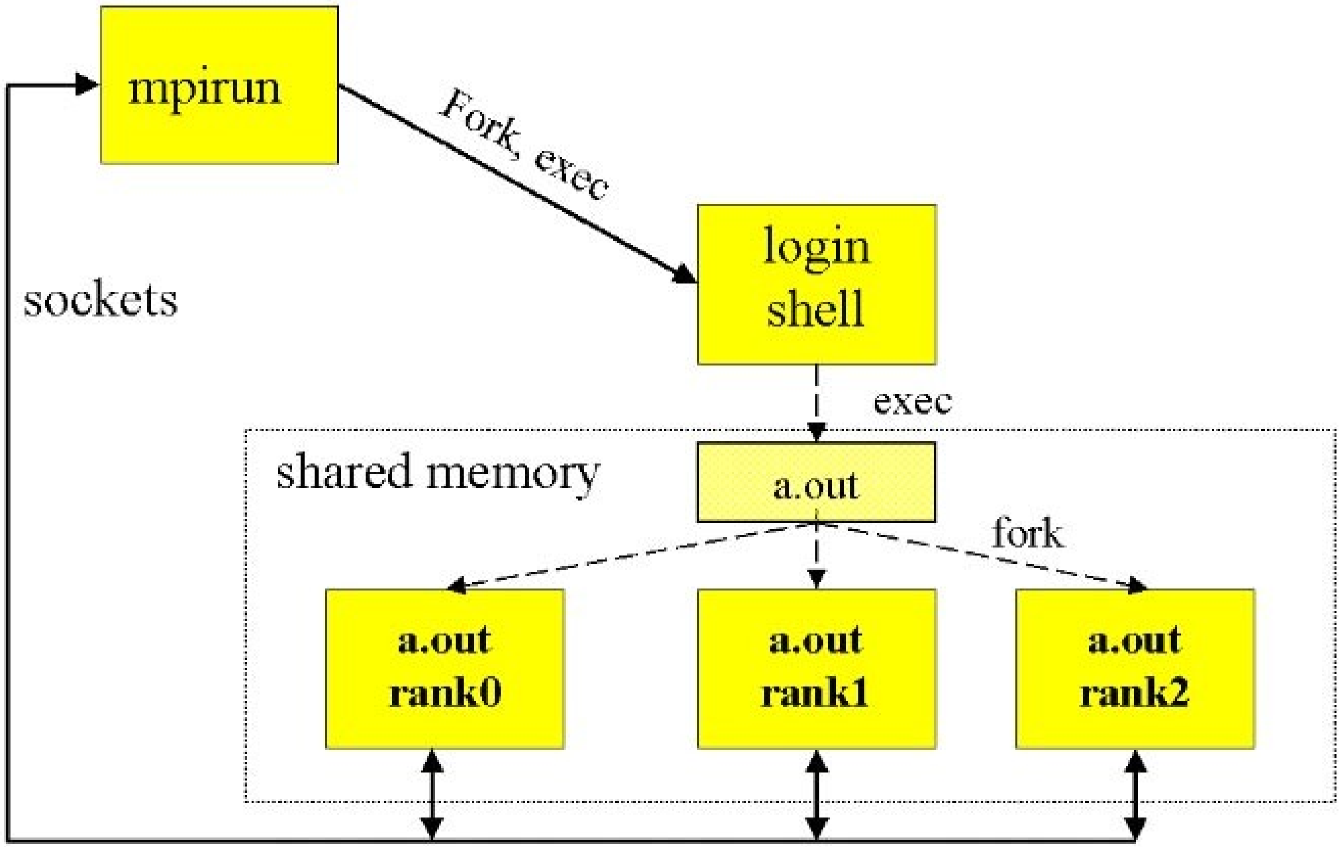
```
       program reduce
      include 'mpif.h'
      parameter (n = 5)
      integer myarr(n)
      integer global(n)

        call MPI_Init(ierr)
      call MPI_Comm_size(MPI_COMM_WORLD, num_procs, ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, my_proc, jerr)
      do i = 1, n
        myarr(i) = my_proc + i
      enddo
      call MPI_Reduce(myarr, global, n, MPI_INTEGER, MPI_SUM, 0,
     & MPI_COMM_WORLD, ierr)
      if (my_proc .eq. 0)
     &  write(6,1) (global(i), i=1, n)
    1 format(5(i2,1x))
      call MPI_Finalize(ierr)
      end
```

```
% ifort -w reduce.f -lmpi
% mpirun -np 5 ./a.out
15 20 25 30 35
```

SGI Altix: MPI - Message Passing Interface

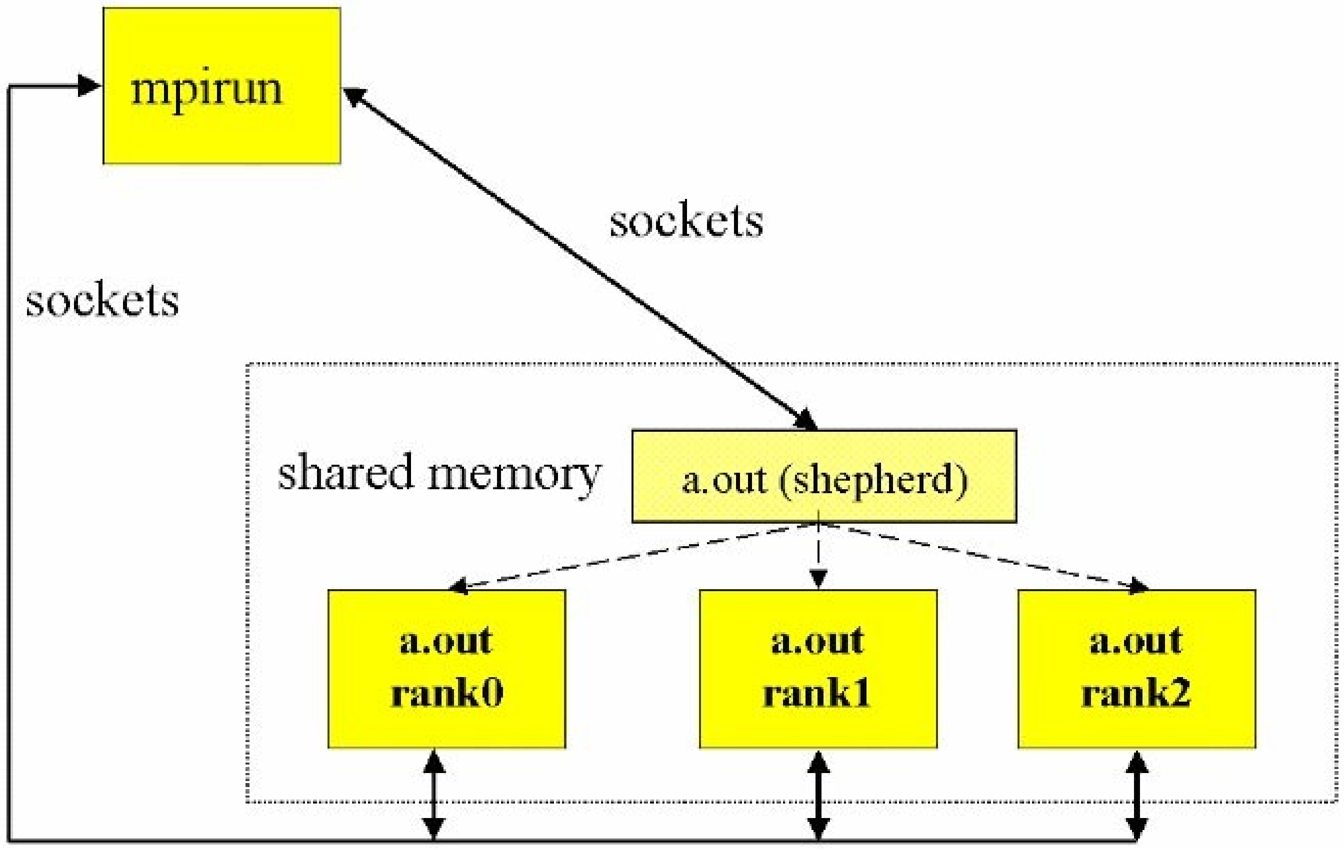
Tuning

MPI Application on SGI Altix Systems

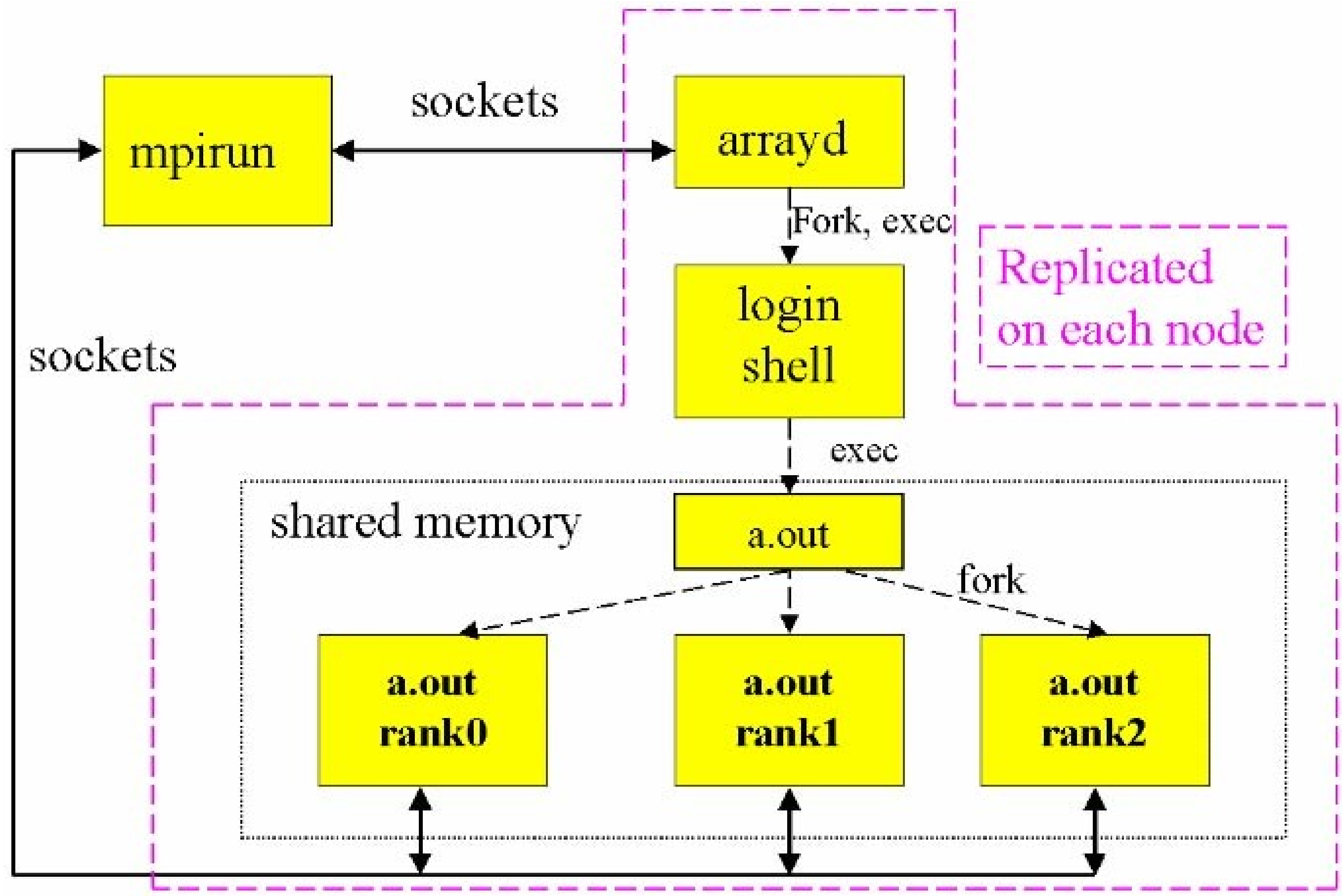


sgi

MPI Application After Startup



MPI Application on SGI Altix Cluster



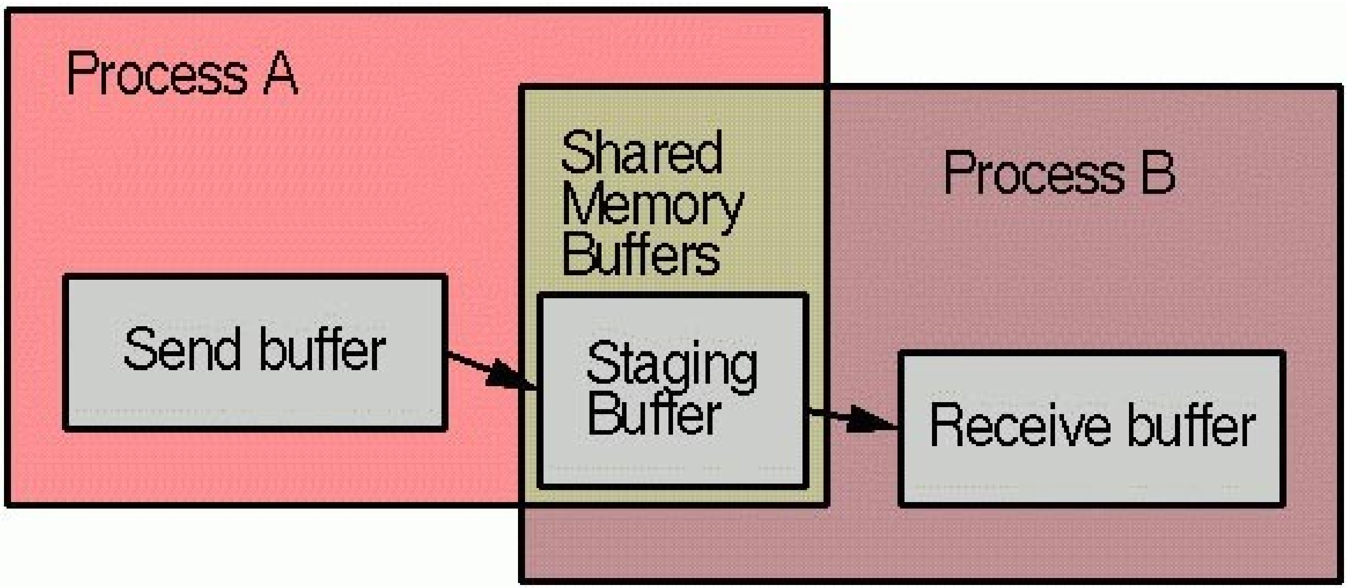
MPI Process Initiation

- `mpirun` requests the array services daemon, `arrayd`, to create processes
- Login shell is created--creates a single copy of program
- This copy becomes the ``shepherd'' or MPI daemon
  - Forks the MPI processes, does bookkeeping
  - One daemon per machine in a cluster

MPI Process Relationships

- Ancestor/descendent relationship is lost (true only on clusters)
- Extra shepherd process is introduced
- Most, but not all, job control functions are retained
  - Cntl C/interrupt works
  - Cntl Z/suspend works
  - `fg` works to resume
  - bg is not supported

MPI Messaging Implementation



sgi

MPI on SGI Altix Clusters

- Can communicate across machines in a supercluster of SGI Altix Systems
- Three interhost communication modes:
  - TCP/IP
  - Infiniband (IB)
  - NUMAflex (XPMEM)

NUMA Memory Layout

- **Explicit NUMA placement used for static memory and symmetric heap**
- **``First-touch'' is used for heap and stack**
- **`dplace` may be used, but remember to skip the shepherd process**

```
mpirun -np 4 dplace -s1 -c0-3 a.out
```

- **If cpusets are used, the numbers in the `-c` option refers to logical CPUs within the cpuset**
- **Alternatively, use the** `MPI_DSM_DISTRIBUTE` **or** `MPI_DSM_CPULIST` **environment variables**

Cluster Example

- **See the */usr/lib/array/arrayd.conf* file**

```
array goodarray
        machine fast.sgi.com
        machine faster.sgi.com
        machine another.sgi.com
```

- **Sample command to run 128 processes across two of the clustered machines:**

```
mpirun -a goodarray fast 64 a.out : faster 64 a.out
```

  - **``64'' is number of processors**

Standard in/out/err Behavior

- **All** `stdout/stderr` **from MPI tasks directed to mpirun**
- `stdout` **is line buffered**
- **Sent to** `mpirun` **as a message**
- `stdin` **is limited to MPI rank 0**
- `stdin` **is line buffered**
- **New line is needed for** `mpirun` **to process input from** `stdin`

Debugging

- **Etnus Totalview**

```
totalview mpirun -a -np 4 a.out
```

Using Performance Tools

- **profile.pl**

```
mpirun -np 4 profile.pl [options] a.out
```

Scheduling With cpusets

- **In a time-shared environment, use** `cpusets` **to ensure that message passing processes are scheduled together:**

```
cpuset -q myqueue -A mpirun -np 100 a.out
```

- **Dynamic cpuset creation is supported in Platform's LSF and Altair Engineering's PBSpro**

Instrumenting MPI

- **MPI has `PMPI*` names**

```
int MPI_Send(args)
 {
   sendcount++;
   return PMPI_Send(args);
}
```

  - ``` ``MPI_Send'' ``` **is user defined send function;** ``` ``PMPI_Send'' ``` **is the actual** `MPI_Send` **function in the library**

Perfcatcher profiling library

- *Source code that instruments many of the common MPI calls*
- *Only need to modify an RLD variable to link it in*
- *Does call counts and timings and writes a summary to a file upon completion*
- *Eval version available around MPT 1.6 time frame*
- *Use as a base and enhance with your own instrumentation*

Perfcatcher profiling library

```
Total job time 2.203333e+02 sec
Total MPI processes 128
Wtime resolution is 8.000000e-07 sec

activity on process rank 0
 comm_rank calls 1      time 8.800002e-06
 get_count calls 0      time 0.000000e+00
    ibsend calls 0      time 0.000000e+00
     probe calls 0      time 0.000000e+00
      recv calls 0      time 0.00000e+00   avg datacnt 0   waits 0  wait time
    0.00000e+00
     irecv calls 22039  time 9.76185e-01   datacnt 23474032 avg datacnt 1065
      send calls 0      time 0.000000e+00
     ssend calls 0      time 0.000000e+00
     isend calls 22039  time 2.950286e+00
      wait calls 0      time 0.00000e+00   avg datacnt 0
   waitall calls 11045  time 7.73805e+01   # of Reqs 44078  avg datacnt 137944
   barrier calls 680    time 5.133110e+00
  alltoall calls 0      time 0.0e+00   avg datacnt 0
 alltoallv calls 0      time 0.000000e+00
    reduce calls 0      time 0.000000e+00
 allreduce calls 4658   time 2.072872e+01
     bcast calls 680    time 6.915840e-02
    gather calls 0      time 0.000000e+00
   gatherv calls 0      time 0.000000e+00
   scatter calls 0      time 0.000000e+00
  scatterv calls 0      time 0.000000e+00

activity on process rank 1
```

MPI Optimization Hints

- **Do not use wildcards, except when necessary**
- **Do not oversubscribe number of processors**
- **Collective operations are not all optimized**
  - **Use `SHMEM` to optimize bottlenecks**
- **Minimize use of** `MPI_barrier` **calls**
- **Optimized paths**
  - `MPI_Send() / MPI_Recv()`
  - `MPI_Isend() / MPI_Irecv()`
- **Less optimized:**
  - `ssend, rsend, bsend, send_init`
- **When using** `MPI_Isend()/MPI_Irecv()`**, be sure to free your request by either calling** `MPI_Wait()` **or** `MPI_Request_free()`

Environment Variables

- `MPI_DSM_CPULIST`
  - **Allows specification of which CPUs to use**
  - **If running within an *n*-processor cpuset, use 0-<*n*-1 > rather than physical CPU numbers**
  - **Works like an implicit** `dplace -s1`
- `MPI_DSM_DISTRIBUTE`
  - **Equivalent to** `MPI_DSM_CPULIST 0-<n-1>`
- `MPI_BUFS_PER_PROC`
  - **Number of 16-kB buffers for each processor (default 32)**
  - **For use within a host; they are assigned locally so copy into buffer is efficient and has no contention**

Environment Variables (continued)

- `MPI_BUFS_PER_HOST`
  - **Single pool of buffers for interhost communication, 16 kB each (default 32)**
  - **Less memory usage but more contention**
- **Many others, see** `man mpi`

Tunable Optimizations

Eliminate Retries (Use MPI statistics)

```
setenv MPI_STATS
or
mpirun -stats -prefix "%g:" -np 8 a.out
```

```
3: *** Dumping MPI internal resource statistics...
3:
3:  0 retries allocating mpi PER_PROC headers for collective calls
3:  0 retries allocating mpi PER_HOST headers for collective calls
3:  0 retries allocating mpi PER_PROC headers for point-to-point calls
3:  0 retries allocating mpi PER_HOST headers for point-to-point calls
3:  0 retries allocating mpi PER_PROC buffers for collective calls
3:  0 retries allocating mpi PER_HOST buffers for collective calls
3:  0 retries allocating mpi PER_PROC buffers for point-to-point calls
3:  0 retries allocating mpi PER_HOST buffers for point-to-point calls
3:  0 send requests using shared memory for collective calls
3:  6357 send requests using shared memory for point-to-point calls
3:  0 data buffers sent via shared memory for collective calls
3:  2304 data buffers sent via shared memory for point-to-point calls
3:  0 bytes sent using single copy for collective calls
3:  0 bytes sent using single copy for point-to-point calls
3:  0 message headers sent via shared memory for collective calls
3:  6357 message headers sent via shared memory for point-to-point calls
3:  0 bytes sent via shared memory for collective calls
3:  15756000 bytes sent via shared memory for point-to-point calls
```

Using direct copy send/recv

- Set MPI_BUFFER_MAX to N
  - any message with size > N bytes will be transferred by direct copy if
    - MPI semantics allow it
    - the memory region it is allocated in is a globally accessible location
  - N=2000 seems to work well
    - shorter messages don’t benefit from direct copy transfer method
  - Look at stats to verify that direct copy was used.

Making memory globally accessible for direct copy send/recv

- User's send buffer must reside in one of the following regions:
  - static memory (common blocks, f90-modules)
  - symmetric heap (allocated with SHPALLOC or shmalloc)
  - On Altix even stack allocated arrays globally accessible.

Typical MPI-Env Variables Set

- Always:
  - MPI_BUFFER_MAX = 2000
  - MPI_DSM_DISTRIBUTE=1
- occasional:
  - MPI_BUFS_PER_PROC=32 or larger
  - MPI_DSM_CPULIST=0-xx
  - MPI_STATS = 1
  - MPI_OPENMP_INTEROP=1

Message Passing References

- **Man pages**
  - **`mpi`**
  - **`mpirun`**
  - **`shmem`**
- **Release notes**
  - **`rpm -ql sgi-mpt | grep relnotes`**
- **Message Passing Toolkit: MPI Programmer's Manual**
  - http://techpubs.sgi.com
  - **`rpm -ql sgi-mpt | grep MPT_MPI_PM.pdf`**
- **MPI Standard**
  - http://www.mpi-forum.org/docs/docs.html

General MPI Issues

- **Most programs use blocking calls**
- **Use of nonblocking and synchronization calls can lead to faster codes**
  - **SHMEM library (man intro_shmem)**
  - **MPI-2 one-sided calls, `MPI_Put` and `MPI_Get`**
- **Synchronization (barrier, wait, and so on) calls are often overused; algorithm rethinking often eliminates the unnecessary calls**
- **Instead of writing your own procedures, investigate whether MPI has one and use it (for example, reductions, synchronization, broadcast)**
- **URL** http://www.mpi-forum.org/docs/docs.html **and** http://www-unix.mcs.anl.gov/mpi/index.htm **are good starting points to learn more about the standard**