The Message Passing Interface
MPI and MPI-2

Disclaimer: These notes DO NOT substitute the textbook for this class. The notes should be used IN CONJUNCTION with the textbook and the material presented in class. If a statement in these notes seems to be incorrect, report it to the instructor so that it be fixed immediately. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2004; distribution outside this group of students is NOT allowed.

BSP programming assumes that program conform to a bulk-synchronous paradigm, where processor occasionally synchronize; all or a subset of the processors can participate in these reocurring synchronizations. BSPlib in particular, does not fully implement the BSP model; it allows for global processor synchronization (no subset processor synchronization is available). In addition, bulk-synchronization is achieved in BSPlib through barrier-style synchronization.

An extreme case of parallel programming is the *fully-asynchronous* or the *loosely synchronous* paradigm. In the former all concurrent/parallel task are executed asynchronously, thus supporting MIMD programming, not just SPMD. However in asynchronous programming race conditions may occur and reasoning about program correctness migh be more difficult. In loosely synchronous systems task work independently as being asynchronous; however at certain points they synchronize to perform certain tasks.

The Message-Passing Interface (MPI) is an attempt to create a standard to allow tasks executing on multiple processors to communicate through some standardized communication primitives. It defines a standard library for message passing that one can use to develop message-passing program using C or Fortran. The MPI standard define both the syntax and the semantics of these functional interface to message passing. MPI comes intro a variety of flavors, freely available such as LAM-MPI and MPIch, and also commercial versions such as Critical Software's WMPI. It supporst message-passing on a variety of platforms from Linux-based or Windows-based PC to supercomputer and multiprocessor systems.

After the introduction of MPI whose functionality includes a set of 125 functions, a revision of the standard took place that added C++ support, external memory accessibility and also Remote Memory Access (similar to BSP's put and get capability) to the standard. The resulting standard is known as MPI-2 and has grown to almost 241 functions.

A minimum set

A minimum set of MPI functions is described below. MPI functions use the prefix MPI_ and after the prefix the remaining keyword start with a capital letter. One case observe some similarities between BSPlib and MPI.

A brief explation of the primitives can be found on the textbook (beginning page 242). A more elaborate presentation is available in the optional book.

Function Class	Standard	Function	Operation
Initialization	MPI	`MPI_Init`	Start of SPMD code
and Termination	MPI	`MPI_Finalize`	End of SPMD code
Abnormal Stop	MPI	`MPI_Abort`	One process halts all
Process Control	MPI	`MPI_Comm_size`	Number of processes
	MPI	`MPI_Comm_rank`	Identifier of Calling Process
	MPI	`MPI_Wtime`	Local (wall-clock) time
Synchronization	MPI	`MPI_Barrier`	Global Synchronization
DRMA	MPI-2	`MPI_Win_create`	Make memory info global
	MPI-2	`MPI_Win_free`	Undo global effect
	MPI-2	`MPI_Put`	Copy into remote memory
	MPI-2	`MPI_Get`	Copy from remote memory
Message Passing	MPI	`MPI_Send`	Send message to remote proc.
	MPI	`MPI_Recv`	Receive mess. from remot proc.

Comparison to same set of BSPlib

Function Class	Function	Operation
Initialization and Termination	`bsp_begin`	Start of SPMD code
	`bsp_end`	End of SPMD code
Abnormal Stop	`bsp_abort`	One process halts all
Process Control	`bsp_nprocs`	Number of processes
	`bsp_pid`	Identifier of Calling Process
	`bsp_time`	Local (wall-clock) time
Synchronization	`bsp_sync`	Global Synchronization
DRMA	`bsp_push_reg`	Make memory info global
	`bsp_pop_reg`	Undo global effect
	`bsp_put`	Copy into remote memory
	`bsp_get`	Copy from remote memory
	`bsp_hpput`	High performance put
	`bsp_hpget`	High performance get
Message Passing	`bsp_send`	Send message to remote proc.
	`bsp_move`	Receive from local receiving message queue

```
#include <mpi.h>
int MPI_Init (int *argc, char  **argv);   // similar to bsp_begin
int MPI_Finalize(void);                   // similar to bsp_end
int MPI_Abort(MPI_Comm comm, int errcode);// similar to bsp_abort()
```

Multiple processes from the same source are created by issuing the function `MPI_Init` and these processes are safely terminated by issuing a `MPI_Finalize`. The arguments of `MPI_Init` are the command line arguments minus the ones that were used/processed by the MPI implementation. Thus command line processing should only be performed in the program after the execution of this function call. Successful return returns a `MPI_SUCCESS`; otherwise an error-code that is implementation dependent is returned.

Note that `MPI_Abort` that aborts an MPI program cleanly has a different set of arguments than the BSPlib function. The first one is a communicator (see next page for details) and second argument is an integer error code. A default communicator is `MPI_COMM_WORLD`.

Definitions are available in `<mpi.h>`. **Note the angled brackets. They are not double quotes as in "bsp.h"!**

Under MPI, a communication domain is a set of processors that are allowed to communicate with each other. Information about such a domain is stored in a communicator that uniquely identify the processors that participate in a communication operation.

A default communication domain is all the processors of a parallel execution; it is called `MPI_COMM_WORLD`. By using multiple communicators between possibly overlapping groups of processors we make sure that messages are not interfering with each other.

```
#include <mpi.h>
int MPI_Comm_size ( MPI_Comm comm, int *size);
int MPI_Comm_rank ( MPI_Comm comm, int *rank);
```

Thus

```
MPI_Comm_size ( MPI_COMM_WORLD,  &nprocs);
MPI_Comm_rank ( MPI_COMM_WORLD,  &pid   );
```

return the number of processors `nprocs` and the processor id `pid` of the calling processor. A `hello world!` program in MPI is the following one.

```
#include <mpi.h>
int main(int argc, char **argv) {
  int nprocs, mypid;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&mypid );
  printf("Hello world from process %d of total  %d\n",mypid,nprocs);
  MPI_Finalize();
}
```


One can compile this MPI program as follows where instead of directly issuing the local C compiler, we call the LAMMPI frontend that handles all relevant work for us. Compilation is no more complicated than the compilation of the corresponding sequential code. LAM MPI is already preinstalled. There is nothing for you to install on our system. You compile code as follows.

```
% mpicc hello.c -o hello          //no optimizations chosen
```

If optimization options are required,

```
% mpicc -O3 hello.c  -o hello    // -O3 optimization -O is - Oh not - zero!
```

In LAM MPI we can define a cluster similarly to `.bsptcphosts` of BSPlib. Just define a file of any name eg `lamhosts`. The first name is assigned a processor id 0, the second 1, and so on. If one machine has two CPUs, you can enter it twice in this file.

```
pcc16
pcc13
pcc14
pcc15
```

A LAM is initiated similarly to `bsplibd`. One issues the command

```
% lamboot -v lamhosts
```

A LAM is terminated similarly to `bspshutd`. One issues the command

```
% wipe -v lamhosts
```


Following our BSPlib convetion in executing programs (i.e. creating a `run` directory on all machines and `ccp`'ing the executable to that directory of all machines, we can follow that convention with LAM MPI. A parallel program under LAM MPI can then be executed by

```
% mpirun -np 4 ./hello
```

The parameter to `np` is the number of processes that will be initiated and will be assigned one-to-one to the machines listed in `lamhosts`.

```
Hello World from process 3 of total 4
Hello World from process 1 of total 4
Hello World from process 0 of total 4
Hello World from process 2 of total 4
```

Note that printing wise, MPI is more disciplined and reliable that BSPlib. The order of printing however can be arbitrary on the standard output.


```
#include <mpi.h>
double MPI_Wtime(void);
```

It returns the time in seconds since an arbitrary point in the past on the calling processor. Gives wall clock time.

Note the difference between `MPI_Wtime` and `bsp_time()`. The latter gives the time since the beginning of the bsp program. There are no such guarantees for the former. If you want to maintain the `bsp_time()` functionality under MPI, you may need to do something like the fragment below.

```
double t1;
bsp_begin()


t1= bsp_time();
```

```
double t1, mpi_basetime;
MPI_Init(&argc,&argv), mpi_basetime = MPI_Wtime();


t1= (MPI_Wtime()-mpi_basetime);
```



```
#include <mpi.h>
int MPI_Barrier(MPI_Comm comm);
```

It 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.

Note that `bsp_sync()` does more that `MPI_Barrier`. The former also effects all communication performed in the current superstep. There is no such thing as a superstep or BSP in LAM MPI. `MPI_Barrier` is just a barrier synchronization. No side-effect other than the obvious is effected.

Remote Memory Access: MPI_Put

The RMA (Remote Memory Access) facility allowed by LAM MPI according the MPI-2 implementation is effected by `MPI_Put` and `MPI_Get`. There are multiple ways one can use such operations under LAM MPI. We mention only one.

```
#include <mpi.h>
       int MPI_Put(orgaddr, orgcnt, orgtype, rank, targdisp, targcnt, targtype, win)

       void   *orgaddr;
       int   orgcnt;
       MPI_Datatype  orgtype;
       int   rank;
       MPI_Aint  targdisp;
       int   targcnt;
       MPI_Datatype  targtype;
       MPI_Win   win;
```

Input Parameters are:

```
orgaddr
       - initial address of origin buffer (choice)
orgcnt - number of entries in origin buffer (nonnegative integer)
orgtype
       - datatype of each entry in origin buffer (handle)
rank   - rank of target (nonnegative integer)
targdisp
       - displacement from start of window to target buffer (nonnega-
       tive integer)
targcnt
       - number of entries in target buffer (nonnegative integer)
targtype
       - datatype of each entry in target buffer (handle)
win    - window object used for communication (handle)
```


A get has a similar syntax.

```
#include <mpi.h>
       int MPI_Get(orgaddr, orgcnt, orgtype, rank, targdisp, targcnt, targtype, win)

        void   *orgaddr;
        int   orgcnt;
        MPI_Datatype  orgtype;
        int   rank;
        MPI_Aint  targdisp;
        int   targcnt;
        MPI_Datatype  targtype;
        MPI_Win   win;

       orgaddr
              - initial address of origin buffer (choice)
       orgcnt - number of entries in origin buffer (nonnegative integer)
       orgtype
              - datatype of each entry in origin buffer (handle)
       rank   - rank of target (nonnegative integer)
       targdisp
              -  displacement from window start to the beginning of the tar-
              get buffer (nonnegative integer)
       targcnt
              - number of entries in target buffer (nonnegative integer)
       targtype
              - datatype of each entry in target buffer (handle)
       win    - window object used for communication (handle)
```

Memory is copied from target memory to the origin. The origin datatype may not specify overlapping entries in the origin bugger. The target buffer must be contained within the target window, and the copied data must fit, without truncation, in the origin buffer.

Similar to BSPlib registration of DRMA requests, MPI-2 requires that RMA requests be registered through window creation and termination operations.

```
#include <mpi.h>
  int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                MPI_Info info, MPI_Comm comm, MPI_Win *newwin)
  int MPI_Win_free(MPI_Win *newwin)
  int MPI_Win_fence(int assertion, MPI_Win newwin)
```

Operation `MPI_Win_create` creates a window for remote memory access operation.

```
base   - initial address of window (choice)
size   - size of window in bytes (nonnegative integer)
disp_unit
       -  local unit size for displacements, in bytes (positive inte-
       ger)
info   - info argument (handle)
comm   - communicator (handle)
newwin - window object returned by the call (handle)
assertion - A suggestion to the communication algorithm; use 0 if you don'w know
            what values are acceptable.
```

Operation `MPI_Win_free` frees the window object newwin and returns a null handle (equal to `MPI_WIN_NULL`). The MPI call `MPI_Win_fence(assertion, newwin)` synchronizes RMA calls on newwin. The call is collective on the group of newwin. All RMA operations on newwin originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on newwin started by a process after the fence call returns will access their target window only after MPI_Win_fence has been called by the target process. Calls to `MPI_Win_fence` should both precede and follow calls to put, get or accumulate that are synchronized with fence calls.

`MPI_Win_create` is similar to a `bsp_pushregister`, `MPI_Win_free` is similar to a `bsp_popregister`, and `MPI_Win_fence` statements must surround an MPI-2 RMA put or get operation.

BSPlib and MPI-2 correspondence

The syntax of RMA operations under MPI-2 is complicated and quite different from that of the corresponding BSPlib primitives. We how how to do MPI-2 communication using put and get operations using the BSPlib function calls as a point of reference.

Suppose that processor 0 wants to write his x value into processor 5's y value.

```
#include "bsp.h"                          #include <mpi.h>
 int x,y;                                 int x,y;
 int pid, nprocs;                         int pid, nprocs;
                                          MPI_Win win;

 ..... We assume we have  retrieved pid using an appropriate function call .....

 bsp_pushregister(&y,sizeof(int));        MPI_Win_create(&y,sizeof(int),1,
                                                  MPI_INFO_NULL,MPI_COMM_WORLD,&win);
 bsp_sync();                              MPI_Win_fence(0,win);
 if (pid ==0) {                           if (pid == 0) {
  bsp_put(5,&x,&y,0,sizeof(int));          MPI_Put(&x,sizeof(int),MPI_CHAR,5,0,
                                                  sizeof(int),MPI_CHAR,win);
 }                                        }
 bsp_sync();                              MPI_Win_fence(0,win);
 bsp_popregister(&y);                     MPI_Win_free(0,&win);
```


In the general case,

```
//bsp_put                     vs      MPI_Put ***************************************
char *des,*src;                       char *des,*src;
int  dp,off,size,K;                   int  dp,off,size,K;

bsp_pushregister(des,K);              MPI_Win_create(des,(K),1,MPI_INFO_NULL,
                                                     MPI_COMM_WORLD,&win);
bsp_sync();                           MPI_Win_fence(0,win);

if (pid==0) {                         if (pid == 0) {
  bsp_put(dp,src,des,off,size)         MPI_Put(src,size,MPI_CHAR,dp,off,size,
                                               MPI_CHAR,win);
}                                     }
bsp_sync();                           MPI_Win_fence(0,win);
bsp_popregister(des);                 MPI_Win_free(0,&win);
```

The correspondence between `bsp_get` and `MPI_Get` is also symmetric.

```
//bsp_get                     vs      MPI_Get ***************************************
bsp_pushregister(src,K);              MPI_Win_create((src),(K),1,MPI_INFO_NULL,
                                                     MPI_COMM_WORLD,&win)
bsp_sync();                           MPI_Win_fence(0,win);

if (pid==0) {                         if (pid == 0) {
  bsp_get(dp,src,off,des,size);        MPI_Get(des,size,MPI_CHAR,dp,off,size,
                                               MPI_CHAR,win);
}                                     }
bsp_sync();                           MPI_Win_fence(0,win);
bsp_popregister(src);                 MPI_Win_free(0,&win);
```



```
#include <mpi.h>
 /* Blocking send and receive */
 int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
 int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Status *stat);
 /* Non-Blocking send and receive */
 int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req);
 int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *req);
 int MPI_Wait(MPI_Request *preq, MPI_Status *stat);

       buf    - initial address of send/receive buffer
       count  - number of elements in send buffer (nonnegative integer)
                or maximum number of elements in receive buffer.
       dtyp   - datatype of each send/receive buffer element (handle)
   dest,src   - rank of destination/source (integer)
                Wild-card: MPI_ANY_SOURCE for recv only.
       tag    - message tag (integer). Range 0...32767.
                Wild-card: MPI_ANY_TAG for recv only; send must specify tag.
       comm   - communicator (handle)
       stat   -  status  object  (Status),  which  can  be  the MPI constant
              MPI_STATUS_IGNORE if the return status is not desired

 data type correspondence between MPI and C
   MPI_CHAR --> signed char      , MPI_SHORT --> signed short int , MPI_INT    --> signed int
   MPI_LONG --> signed long int  , MPI_FLOAT --> float            , MPI_DOUBLE --> double
```

The `MPI_Send` and `MPI_Recv` functions are blocking, that is they do not return unless it is safe to modify or use the contents of the send/receive buffer. MPI also provides for non-blocking send and receive primitives. These are `MPI_Isend` and `MPI_Irecv`, where the *I* stands for `Immediate`. These functions allow a process to post that it wants to send to or receive from another process, and then allow the process to call a function (eg. `MPI_Wait` to complete the send-receive pair. Non-blocking send-receives allow for the overlapping of computation/communication. Thus `MPI_Wait` plays the role of `bsp_sync()`: the send/receive are only advisories and communication is only effected at the `MPI_Wait`.

An example with the blocking operations

```
#include <stdio.h>
#include <mpi.h>
#define  N   10000000  // Choose N to be multiple of nprocs to avoid problems.
// Parallel sum of  1 , 2 , 3, ... , N
int main(int argc,char **argv){
 int pid,nprocs,i,j;
 int sum, start, end, total;
 MPI_Status  status;

 MPI_Init(argc,argv);
 MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&pid   );
 sum = 0;  total = 0;
 start =  (N/nprocs)*pid +1 ;  // Each processor
 end   =  (N/nprocs)*(pid+1);
 for(i=start;i<=end;i++) sum += i;
 if (pid != 0 ) {
   MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
 }
 else {
   for (j=1;j<nprocs;j++) {
    MPI_Recv(&total,1,MPI_INT,j,1,MPI_COMM_WORLD,&status);
    sum = sum + total;
   }
 }
 if (pid == 0 ) {
  printf(" The sum from 1 to %d is %d \n",N,sum);
 }
 MPI_Finalize();
}
// Note: Program neither compiled nor run!
```


An example with the non-blocking operations

```
#include <stdio.h>
#include <mpi.h>
#define  N   10000000  // Choose N to be multiple of nprocs to avoid problems.
// Parallel sum of  1 , 2 , 3, ... , N
int main(int argc,char **argv){
 int pid,nprocs,i,j;
 int sum, start, end, total;
 MPI_Status  status;
 MPI_Request request;
 MPI_Init(argc,argv);
 MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&pid   );
 sum = 0;  total = 0;
 start =  (N/nprocs)*pid +1 ;  // Each processor
 end   =  (N/nprocs)*(pid+1);
 for(i=start;i<=end;i++) sum += i;
 if (pid != 0 ) {
   // MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
   MPI_Isend(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD,&request);
   MPI_Wait(&request,&status);
 }
 else {
   for (j=1;j<nprocs;j++) {
    MPI_Recv(&total,1,MPI_INT,j,1,MPI_COMM_WORLD,&status);
    sum = sum + total;
   }
 }
 if (pid == 0 ) {
  printf(" The sum from 1 to %d is %d \n",N,sum);
 }
 MPI_Finalize();
} // Note: Program neither compiled nor run!
```