BDMPI — Big Data Message Passing Interface

User/Reference Manual

Karypis Lab

Department of Computer Science & Engineering

University of Minnesota

Release 0.1, April 2014

Contents

1 Overview of BDMPI 1
1.1 Motivation of the approach . . . 1
1.2 Execution & memory model . . . 2
1.2.1 Master & slave processes . . . 2
1.2.2 Efficient loading & saving of a process's address space . . . 2

2 Setting up BDMPI 4
2.1 System requirements . . . 4
2.2 System configuration . . . 4
2.3 Building and installing BDMPI . . . 5

3 Compiling BDMPI Programs 7

4 Running BDMPI Programs 8
4.1 Running on a single node . . . 8
4.2 Running on multiple nodes . . . 9
4.3 Options of bdmprun . . . 10
4.3.1 Options related to the execution environment . . . 10
4.3.2 Options related to memory resources . . . 11
4.3.3 The remaining options . . . 11

5 API Documentation 12
5.1 MPI functions supported by BDMPI . . . 12
5.2 BDMPI-specific functions . . . 12
5.2.1 Communicator-related functions . . . 12
5.2.2 Intra-slave synchronization . . . 13
5.2.3 Storage-backed memory allocations . . . 13
5.3 Predefined communicators . . . 13
5.4 Supported datatypes . . . 14
5.5 Supported reduction operations . . . 15

6 Troubleshooting BDMPI Programs 17
6.1 Developing and debugging BDMPI programs . . . 17
6.2 Issues related to sbmalloc . . . 17
6.3 Cleaning up after a failed execution . . . 17

7 Guidelines for Developing Efficient BDMPI Programs 18
7.1 Use collective communication operations . . . 18
7.2 Group blocking communication operations together . . . 18
7.3 Freeing scratch memory . . . 18
7.4 Structure memory allocations in terms of active blocks . . . 18

8 Credits & Contact Information **19**

9 Copyright & License Notice **20**

10 Module Documentation **21**

10.1 List of implemented MPI functions 21

10.1.1 Detailed Description 21

10.2 Functions for further communicator interrogation 24

10.2.1 Detailed Description 24

10.2.2 Function Documentation 24

10.3 Functions for critical sections 26

10.3.1 Detailed Description 26

10.3.2 Function Documentation 26

10.4 Functions for storage-backed memory allocation 27

10.4.1 Detailed Description 27

10.4.2 Function Documentation 27

Bibliographic References **30**

Index **31**

1 Overview of BDMPI

BDMPI is a message passing library and associated runtime system for developing out-of-core distributed computing applications for problems whose aggregate memory requirements exceed the amount of memory that is available on the underlying computing cluster. BDMPI is based on the `Message Passing Interface` (MPI) and provides a subset of MPI's API along with some extensions that are designed for BDMPI's memory and execution model.

A BDMPI-based application is a standard memory-scalable parallel MPI program that was developed assuming that the underlying system has enough computational nodes to allow for the in-memory execution of the computations. This program is then executed using a sufficiently large number of processes so that the per-process memory fits within the physical memory available on the underlying computational node(s). BDMPI maps one or more of these processes to the computational nodes by relying on the OS's virtual memory management to accommodate the aggregate amount of memory required by them. BDMPI prevents memory thrashing by coordinating the execution of these processes using node-level co-operative multi-tasking that limits the number of processes that can be running at any given time. This ensures that the currently running process(es) can establish and retain memory residency and thus achieve efficient execution. BDMPI exploits the natural blocking points that exist in MPI programs to transparently schedule the co-operative execution of the different processes. In addition, BDMPI's implementation of MPI's communication operations is done so that to maximize the time over which a process can execute between successive blocking points. This allows it to amortize the cost of loading data from disk over the maximal amount of computations that can be performed.

Since BDMPI is based on the standard MPI library, it also provides a framework that allows the automated out-of-core execution of existing MPI applications. BDMPI is implemented in such a way so that to be a drop-in replacement of existing MPI implementations and allow existing codes that utilize the subset of MPI functions implemented by BDMPI to compile unchanged.

A detailed description of BDMPI's design along with an experimental evaluation of the performance that it can achieve can be found in [1], [2] (also included in BDMPI's source distribution). In the rest of this section we provide a brief overview of some of its elements as well as the key changes that have been included since that work was published.

1.1 Motivation of the approach

The general approach used by out-of-core algorithms is to structure their computations into a sequence of *steps* such that the working set of each step can fit within the available physical memory and the data associated with each step can be loaded/stored from/to the disk in a disk-friendly fashion (e.g., via sequential accesses or via a small number of bulk accesses).

Scalable distributed memory parallel algorithms decompose the computations into different tasks and each task along with its associated data is mapped on the available compute nodes. This decomposition is optimized so that it maximizes the computations that can be done with the local data (i.e., maximize locality) and reduce the frequency as well as the volume of the data that needs to be communicated across the nodes (i.e., minimize communication overheads). In addition, most of these algorithms structure their computations into a sequence of *phases* involving a local computation step followed by inter-process communication step.

BDMPI relies on the observation that a scalable distributed memory parallel algorithm can be transformed into an algorithm whose structure is similar to that used by out-of-core algorithms. In particular, if p is the number of processes required to ensure that the per-process memory fits within the compute node's available physical memory, then the computations performed by each process in a single phase will correspond to a distinct step of the out-of-core algorithm. That is, one parallel phase will be executed as p sequential steps. Since the working set of each of these steps fits within the physical memory of a node, the computations can be performed efficiently. Moreover, if the underlying computational infrastructure has n available nodes, each node will perform p/n of these steps in sequence, leading to a distributed out-of-core execution.

BDMPI performs this transformation in a way that is entirely transparent to the programmer. It uses the OS's virtual memory management (VMM) mechanisms to provide the programmer with the illusion that the parallel program is operating as if all the data fits in memory and, when appropriate, uses disk-based message buffering to ensure the correct and efficient execution of the communication operations.

1.2 Execution & memory model

BDMPI uses two key elements in order to enable efficient out-of-core execution. The first relates to how the MPI processes are executed on each node and the second relates to the memory requirements of the different MPI processes. The first is called the *execution model* and the second is called the *memory model*.

BDMPI's execution model is based on *node-level co-operative multi-tasking*. BDMPI allows only up to a fixed number of processes to be executing concurrently with the rest of the processes blocking. When a running process reaches an MPI blocking operation (e.g., point-to-point communication, collective operation, barrier, etc.), BDMPI blocks it and selects a previously blocked and runnable process (i.e., whose blocking condition has been satisfied) to resume execution.

BDMPI's memory model is based on *constrained memory over-subscription*. It allows the aggregate amount of memory required by all the MPI processes spawned on a node to be greater than the amount of physical memory on that node. However, it requires that the sum of the memory required by the processes that are allowed to run concurrently to be smaller than the amount of physical memory on that node. Within this model, an MPI program will rely on BDMPI's and OS's VMM mechanisms to load in memory the data needed by each process in a way that is transparent to the running program.

The coupling of constrained memory over-subscription with node-level co-operative multi-tasking is the key that allows BDMPI to efficiently execute an MPI program whose aggregate memory requirements far exceed the aggregate amount of physical memory in the system. First, it allows the MPI processes to amortize the cost of loading their data from the disk over the longest possible uninterrupted execution that they can perform until they need to block due to MPI's semantics. Second, it prevents memory thrashing (i.e., repeated and frequent page faults), because each node has a sufficient amount of physical memory to accommodate all the processes that are allowed to run.

1.2.1 Master & slave processes

The execution of a BDMPI program creates two sets of processes. The first are the MPI processes associated with the program being executed, which within BDMPI, they are referred to as the *slave* processes. The second is a set of processes, one on each node, that are referred to as the *master* processes. The master processes are at the heart of BDMPI's execution as they spawn the slaves, coordinate their execution, service communication requests, perform synchronization, and manage communicators.

The master processes are implemented by a program called `bdmprun`, which is a parallel program written in MPI (not BDMPI). When a user program is invoked, `bdmprun` is first loaded on the nodes of the cluster and then proceeds to spawn the slave processes. Section Running BDMPI Programs provides detailed information on how to use `bdmprun` to start a BDMPI program.

1.2.2 Efficient loading & saving of a process's address space

The initial implementation of BDMPI ([1] and [2]), explored three different approaches for saving and/or restoring the address space of a slave process. The first approach relied entirely on the OS's VMM system to save/load the unused/used pages from the system's swap file. The second approach incorporated application-directed prefetching by relying on the `mlock()` system calls to lock in physical memory parts of the address space. Finally, the third bypassed the swap file entirely by requiring the application to explicitly save/load the various data structures that it needs to/from a file. The first approach is entirely transparent to the programmer, whereas the last two approaches require that the programmer modifies his/her program in order to either insert the appropriate `mlock()/munlock`() calls or explicitly perform file I/O. However, the gains achieved by the last two approaches were often considerable, with the third approach performing the best.

The current implementation of BDMPI uses its own memory allocation subsystem that is designed to achieve the performance of explicit file I/O with no or minimal modifications of the underlying program. This memory management subsystem, which is referred to as *storage-backed memory allocation* and will be abbreviated as *sbmalloc*, is implemented as a wrapper around `libc's` standard `malloc()` library. That is, a call to `malloc()` in a BDMPI program will be performed by sbmalloc. BDMPI provides wrappers for the following malloc-related functions: `malloc()`, `realloc()`, and `free()`. Note that `calloc()` is not in that list due to an issue with GNU libc's implementation of `dlsym()`, which prevents `calloc()` to be wrapped.

The key ideas behind sbmalloc are the following:

- It uses `mmap()` to satisfy an allocation request and creates a file that will be used to persist the data of the associated memory pages when the slave is blocked.
- It relies on memory protection and signal handling to determine if the program is accessing any of the allocated pages and if so, the access mode (read or write).
- It saves any pages that have been modified to its associated file when the slave process blocks due to a communication operation and
  - informs the OS that the associated address space does not need to be saved in the swap file, and
  - modifies the memory protection of the associated pages to remove read/write permissions.
- When a process first reads anywhere in the allocation, it reads the previous data from the disk for the entire allocation (if they exist) and gives the process read permissions to the allocation.

As a result of the above, if the aggregate amount of memory that the running slaves need to access between successive blocking operations fits within the physical memory of the node, these allocations will not use the system's swap file. The advantage of this approach is that as long as the application tends to read and/or write all the data within an allocation request, this data will be brought into memory and saved with fast sequential read/write operations.

Section Options related to memory resources provides additional information on how to control which memory allocations will be handled by sbmalloc and which ones will be handled by the standard malloc library. Section Storage-backed memory allocations provides information on how to explicitly use the sbmalloc subsystem in a program. Finally, Issues related to sbmalloc provides information on some issues that may arise with sbmalloc and how to resolve them.

2 Setting up BDMPI

2.1 System requirements

BDMPI has been developed to run on Linux systems. It can potentially run on non-Linux systems as long as they support POSIX inter-processor communication constructs. However, it has not been tested on anything else other than Linux. Besides the OS, the following software packages are required in order to build and use BDMPI:

1. GCC 4.x or higher (`http://gnu.gcc.org`).
2. CMake 2.8 or higher (`http://www.cmake.org`).
3. MPICH 3.0.4 or higher (`http://www.mpich.org`).
4. Perl 5 or higher (`http://www.perl.org`).

All of the above packages are available for most Linux distributions as installable packages. Note that BDMPI has not been tested with OpenMPI, though in principle, it should work with it.

In terms of hardware, the systems on which BDMPI is running should have enough local storage. Note that even though BDMPI can use a network attached file system for temporary storage, its performance may degrade.

2.2 System configuration

In order to execute BDMPI programs on a single workstation or a cluster of workstations, the underlying system must be configured to execute MPI jobs. Before trying to run any BDMPI programs follow the instructions in MPICH's documentation on how to setup the system for running MPI jobs. This usually involves enabling password-less ssh remote process execution and setting up a shared file system.

In addition, the following system configuration parameters need to be modified (the names of the files discussed are valid for at least the Ubuntu 12.04.4 LTS distribution):

1. Increase the `nofile`, `msgqueue`, and `memlock` process limits in the `/etc/security/limits.conf` file. Specifically, you should add the following limits:

   ```
   *       soft    nofile          1000000
   *       hard    nofile          1000000
   *       soft    msgqueue        unlimited
   *       hard    msgqueue        unlimited
   *       soft    memlock         1048576
   *       hard    memlock         1048576
   ```

   Note that the limit for the `memlock` parameter can be adjusted up or down based on the available memory in your system. However, you should probably leave at least 1GB of non-lockable memory.

2. Increase the number of default POSIX message queues. This is done by modifying the `/etc/sysctl.conf` file to add/modify the following lines:

   ```
   msg_default     512
   msg_max         1024
   msgsize_default 256
   msgsize_max     512
   queues_max      1024
   ```

   Besides directly editing the `/etc/sysctl.conf` file, the above changes can also be done by executing the following commands:

   ```
   sudo sysctl fs.mqueue.msg_default=512
   sudo sysctl fs.mqueue.msg_max=1024
   sudo sysctl fs.mqueue.msgsize_default=256
   sudo sysctl fs.mqueue.msgsize_max=512
   sudo sysctl fs.mqueue.queues_max=1024
   ```

Note that if your system is already configured with higher values for any of the above parameters, you should not change them.

3. Increase the size of the swap file as it will be used for storing the data of the slave processes that are blocked. The size of the swap file depends on the size of the jobs that will be run, the extent to which you allow BDMPI to use its own storage backed memory allocation, and the extent to which your program explicitly manages the size of its memory resident data (e.g., by relying on explicit out-of-core execution). Further details about these three cases are provided in Execution & memory model.

2.3 Building and installing BDMPI

BDMPI is distributed as a source package, which needs to be compiled and installed on the systems that it will run. BDMPI's uses `CMake` to generate the various system-specific `Makefiles` for building it. Instructions on how to use `cmake` are provided in the BUILD.txt file, which is included verbatim here:

```
------------------------------------------------------------------------------
Building BDMPI requires CMake 2.8, found at http://www.cmake.org/, as
well as GNU make. Assumming CMake and GNU make are installed, two
commands should suffice to build BDMPI:

     $ make config
     $ make


Configuration
-------------
BDMPI is primarily configured by passing options to make config. For
example:

     $ make config cc=gcc-4.2

would configure BDMPI to be built using GCC 4.2.

Common configuration options are:
  cc=[compiler]    - The C compiler to use [default is determined by CMake]
  prefix=[PATH]    - Set the installation prefix [/usr/local/ by default]

Advanced debugging related options:
  gdb=1        - Build with support for GDB [off by default]
  debug=1      - Enable debugging support [off by default]
  assert=1     - Enable asserts [off by default]



Installation
------------
To install BDMPI, run

    $ make install

The default installation prefix is /usr/local. To pick an installation
prefix for BDMPI pass prefix=[path] to make config. For example,

    $ make config prefix=~/local

will cause BDMPI to be installed in ~/local when make install is run.


Other make commands
-------------------
   $ make uninstall
          Removes all files installed by 'make install'.

   $ make clean
          Removes all object files but retains the configuration options.
   $ make distclean
          Performs clean and completely removes the build directory.

-------------------------------------------------------------------------------
```

3 Compiling BDMPI Programs

BDMPI provides two helper programs `bdmpicc` and `bdmpic++` for compiling and linking BDMPI programs written in `C` and `C++`, respectively. These programs are simple Perl scripts that setup the appropriate include paths for header files and libraries based on the installation path that you specified when building BDMPI (i.e., specified via the `prefix` option in CMake) and then call either `gcc` or `g++`. All source files of a BDMPI program should be compiled with these programs.

If for some reason you cannot use them, make sure that your include paths contain

```
-IPREFIX/include/bdmpi
```

and for linking you include

```
-LPREFIX/lib/bdmpi -lbdmpi -lGKlib -lrt -lm -ldl
```

In both cases, `PREFIX` is the directory that you specified in the `prefix` option when you built BDMPI (Setting up BDMPI).

4 Running BDMPI Programs

For the purpose of the discussion in this section, we will use the simple "Hello world" BDMPI program that is located in the file `test/helloworld.c` of BDMPI's source distribution. This program is listed bellow:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <bdmpi.h>

int main(int argc, char *argv[])
{
  int i, myrank, npes;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  for (i=0; i<npes; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (myrank == i) {
      printf("[%5d:%5d] Hello from rank: %2d out of %2d\n",
          (int)getppid(), (int)getpid(), myrank, npes);
      fflush(stdout);
    }
  }
  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Finalize();

  /* It is important to exit with a EXIT_SUCCESS status */
  return EXIT_SUCCESS;
}
```

This program has its process print its parent and own process ID along with its rank in the `MPI_WORLD_COMM` and the size of that communicator. Note that the calls to MPI_Barrier() are used to ensure that the output from the different processes is displayed in an orderly fashion.
This program can be compiled by simply executing

```
bdmpicc -o helloworld test/helloworld.c
```

and is also been automatically built during BDMPI's build process and resides in the `build/Linux-x86_64/test` directory.

4.1 Running on a single node

BDMPI provides the `bdmprun` command to run BDMPI program on a single node. For example, the following command

```
bdmprun -ns 4 build/Linux-x86_64/test/helloworld
```

will create a single master process that will spawn four slave processes (i.e., due to `-ns 4` option), each executing the `helloworld` program. Here is a sample output of the above execution:

```
[28933:28936] Hello from rank:  0 out of  4
[28933:28937] Hello from rank:  1 out of  4
[28933:28938] Hello from rank:  2 out of  4
[28933:28939] Hello from rank:  3 out of  4
```

```
[ bd4-umh: 28933]------------------------------------------------
[ bd4-umh: 28933]Master 0 is done.
[ bd4-umh: 28933]------------------------------------------------
[ bd4-umh: 28933]Master timings
[ bd4-umh: 28933]    totalTmr:      0.022s
[ bd4-umh: 28933]    routeTmr:      0.001s
[ bd4-umh: 28933]------------------------------------------------
```

The first four lines came from the `helloworld` program itself, whereas the remaining output lines came from `bdmprun`. Looking at this output we can see that all four processes share the same parent process, which corresponds to the master process, `bdmprun`. The master process reports the timing statistics at the end of the execution of `helloworld`.

4.2 Running on multiple nodes

In order for a program to run at multiple nodes, `bdmprun` needs to be combined with the `mpiexec` command that is provided by MPI. For example, the following command

```
mpiexec -hostfile ~/machines -np 3 bdmprun -ns 4 build/Linux-x86_64/test/helloworld
```

will start three master processes (i.e., due to `-np 3`) and each master process will spawn four slaves, resulting at a total of 12 processes executing the `helloworld` program. Here is a sample output of the above execution:

```
[  331:  334] Hello from rank:  0 out of 12
[  331:  335] Hello from rank:  1 out of 12
[  331:  336] Hello from rank:  2 out of 12
[  331:  337] Hello from rank:  3 out of 12
[27162:27165] Hello from rank:  4 out of 12
[27162:27166] Hello from rank:  5 out of 12
[27162:27167] Hello from rank:  6 out of 12
[27162:27168] Hello from rank:  7 out of 12
[23530:23533] Hello from rank:  8 out of 12
[23530:23534] Hello from rank:  9 out of 12
[23530:23535] Hello from rank: 10 out of 12
[23530:23536] Hello from rank: 11 out of 12
[ bd1-umh:   331]------------------------------------------------
[ bd1-umh:   331]Master 0 is done.
[ bd2-umh: 27162]------------------------------------------------
[ bd2-umh: 27162]Master 1 is done.
[ bd3-umh: 23530]------------------------------------------------
[ bd3-umh: 23530]Master 2 is done.
[ bd1-umh:   331]------------------------------------------------
[ bd1-umh:   331]Master timings
[ bd1-umh:   331]    totalTmr:      0.058s
[ bd1-umh:   331]    routeTmr:      0.001s
[ bd1-umh:   331]------------------------------------------------
```

Notice that the parent process IDs of the different BDMPI processes reveal the master-slave relation that exists (e.-g., ranks 0–3, 4–7, and 8–11 have the same parent process). Also, the output lines generated by `bdmprun` provide information as to the machine on which the particular master process was executed (e.g., `bd1-umh`, `bd2-umh`, and `bd3-umh`) and overall timing information. The names of these machines are specified in the machines file that was provided as the `-hostfile` of `mpiexec`, following `mpiexec's` host machine specification guidelines (see MPICH documentation).

Do not configure your hostfile to have `mpiexec` start multiple processes on the same node. If you do, then each node will have multiple master processes running on it (i.e., multiple instances of `bdmprun`), each of which will have its own set of slave processes. If you wish to have multiple slave processes run at once on the same node, use the `-nr` option instead.

4.3 Options of bdmprun

There are a number of optional parameters that control `bdmprun`'s execution. You can get a list of these options by simply executing `bdmprun -h`, which will display the following:

```
Usage: bdmprun [options] exefile [options for the exe-file]

 Required parameters
    exefile     The program to be executed.

 Optional parameters
  -ns=int [Default: 1]
     Specifies the number of slave processes on each node.

  -nr=int [Default: 1]
     Specifies the maximum number of concurrently running slaves.

  -nc=int [Default: 1]
     Specifies the maximum number of slaves in a critical section.

  -sm=int [Default: 20]
     Specifies the number of shared memory pages allocated for each slave.

  -im=int [Default: 4]
     Specifies the maximum size of a message that will be buffered
     in the memory of the master. Messages longer than that are buffered
     on disk. The size is in terms of memory pages.

  -mm=int [Default: 32]
     Specifies the maximum size of the buffer to be used by MPI for
     inter-node communication. The size is in terms of memory pages.

  -sb=int [Default: 32]
     Specifies the size of allocations for which the explicit storage backed
     subsystem should be used. The size is in terms of number of pages and a
     value of 0 turns it off.

  -wd=string [Default: /tmp/bdmpi]
     Specifies where working files will be stored.

  -dl=int [Default: 0]
     Selects the dbglvl.

  -h
     Prints this message.
```

4.3.1 Options related to the execution environment

The `-ns`, `-nr`, `-nc`, and `-wd` options are used to control the execution environment of a BDMPI job.

The `-ns` option specifies the number of slaves that `bdmprun` will spawn on each node. These slaves then proceed to execute the provided BDMPI program (i.e., `exefile`). If multiple instances of `bdmprun` are started via `mpiexec`, then the total number of processes that are involved in the parallel execution of `exefile` is `np*ns`, where `np` is the number of processes specified in `mpiexec` and `ns` is the number of slaves supplied via the `-ns` option. In other words, the size of the `MPI_COMM_WORLD` communicator is `np*ns`.

The `-nr` options specifies the number of slave processes spawned by a single `bdmprun` process that are allowed to be executing concurrently. This option enables BDMPI's node-level cooperative multitasking execution model, which is designed to ensure that the aggregate memory requirements of the concurrently executing processes do not overwhelm the amount of available physical memory on the system. The default value for `-nr` is one, allowing only one slave to be executing per master process at a time. However, for multi-core systems, `-nr` can be increased as long as the aggregate amount of memory required by the concurrently running processes can comfortably fit within the available memory of the system. For example,

```
mpiexec -hostfile ~/machines -np 3 bdmprun -ns 4 -nr 2 build/Linux-x86_64/test/helloworld
```

will allow a maximum of two slave processes on each of the three nodes to be executing concurrently. The value specified for `-nr` should be less than or equal to `-ns`.

Besides the aggregate memory requirements, another issue that needs to be considered when the number of running processes is increased is the capability of the underlying I/O subsystem to handle concurrent I/O operations. On some systems, a faster execution can be obtained by allowing multiple slave processes to run concurrently but reduce the concurrency allowed during I/O operations. To facilitate this, BDMPI provides a pair of API functions that implement critical sections, which limit the number of processes that can be at a critical section at any give time. These are described in Intra-slave synchronization. The number of slave processes that are allowed to be running at a critical section is controlled by `bdmprun`'s `-nc` option, which should be less than or equal to the value specified for `-nr`.

Finally, the `-wd` option specifies the name of the directory that BDMPI will use to store the various intermediate file that it generates. Since BDMPI's execution is primarily out-of-core, this directory should be on a drive/volume that has a sufficient amount of available storage and preferably the underlying hardware should support fast I/O.

4.3.2 Options related to memory resources

The `-sm`, `-im`, `-mm`, and `-sb` options are used to control various aspects of the execution of a BDMPI program as it relates to how it uses the memory. Among them, `-im` and `-sb` are the most important, whereas `-sm` and `-mm` are provided for fine tuning and most users will not need to modify them. Note that for all memory-related options, the size is specified in terms of memory pages and not bytes. On Linux, a memory page is typically 4096 bytes.

Since processes in BDMPI can be suspended and as such are not available to receive data sent to them, the master processes buffer data at the message's destination node. The `-im` parameter controls if the message will be buffered in DRAM or if it will buffered on disk. Messages whose size is smaller than the value specified by `-im` are buffered in memory, otherwise they are buffered on disk. The only exception to the above rule are the broadcast and the reduction operations, in which the buffering is always done in memory. Also note that in the case of collective communication operations, the size of the message is defined to be the amount of data that any processor will need to send/receive to/from any other processor.

As discussed in Efficient loading & saving of a process's address space, BDMPI uses its own dynamic memory allocation library that bypasses the system's swap file and preserves the data and virtual memory mappings throughout the execution of the program. The `-sb` parameter controls the size of the allocations that will be handled by the sbmalloc subsystem. Any allocation that is smaller than the value specified by `-sb`, is handled by the standard `malloc()` library, whereas the rest of the allocations are handled by sbmalloc. If you want to entirely disable sbmalloc, you can specify 0 as the value for this option.

The `-sm` option specifies the amount of shared memory to allocate for each slave process. This memory is used to facilitate fast communication between the master and the slave processes via `memcpy()`. The `-mm` option specifies the size of the buffer that a master process will allocate for inter-node communication. Note that if the data that needs to be communicated between the master and the slaves or other masters exceeds is greater than what is specified by these options, the transfer is done in multiple steps. Thus, the values of these options do not impact the correctness of the execution, but if they are relatively small, may lead to (somewhat) higher communication cost.

4.3.3 The remaining options

The last two options, `-dl` and `-h`, are used to turn on various reporting messages and display `bdmprun`'s help page. Setting `-dl` to anything else than 0 can potentially generate a lot of reporting messages and is not encouraged. It is there primarily for debugging BDMPI during its development.

5 API Documentation

BDMPI programs are message-passing distributed memory parallel programs written using a subset of MPI's API. As such, BDMPI program development is nearly identical to developing MPI-based parallel programs. There are many online resources providing guidelines and documentation for developing MPI-based programs, all of which apply for BDMPI as well.

Though it is beyond the scope of this documentation to provide a tutorial on how to develop MPI programs, the following is a list of items that anybody using BDMPI should be aware off:

- BDMPI supports programs written in C and C++.
  - Our own testing was done using C-based BDMPI programs, and as such the C++ support has not been tested.
- All BDMPI programs must include the `mpi.h` or `bdmpi.h` header file and must call `MPI_Init()` and `MPI_Finalize()` before and after finishing with their work.
  - Ideally, `MPI_Init()` and `MPI_Finalize()` should be the first and last functions called in the `main()`, respectively.
- All BDMPI programs should exit from their `main()` function by returning `EXIT_SUCCESS`.

5.1 MPI functions supported by BDMPI

BDMPI implements a subset of the MPI specification that includes functions for querying and creating communicators and for performing point-to-point and collective communications. For each of these functions, BDMPI provides a variant that starts with the `MPI_` prefix and a variant that starts with the `BDMPI_` prefix. The calling sequence of the first variant is identical to the MPI specification whereas the calling sequence of the second variant has been modified to make it 64-bit compliant (e.g., replaced most of the sizes that MPI assumed that were `int` to either `size_t` or `ssize_t`).

Since the calling sequence of these functions is the same as that specified in MPI's specification (available at `http://www.mpi-forum.org`) it is not included here.

- List of implemented MPI functions.

Differences with the MPI specification

- BDMPI's error checking and error reporting are significantly less robust than what the standard requires.
- In all collective operations, the source and destination buffers of the operations are allowed to overlap (or be identical). In such cases, the operations will complete correctly and the new data will overwrite the old data.

5.2 BDMPI-specific functions

BDMPI provides a small set of additional functions that an application can use to improve its out-of-core execution performance and get information about how the different MPI slave processes are distributed among the nodes. Since these functions are not part of the MPI specification, their names only start with the `BDMPI_` prefix.

These functions are organized in three main groups that are described in the following subsections.

5.2.1 Communicator-related functions

Since BDMPI organizes the MPI processes into groups of slave processes, each running on a different node, it is sometimes beneficial for the application to know how many slaves within a communicator are running on the same node and the total number of nodes that is involved. To achieve this, BDMPI provides a set of functions that can be used to *interrogate* each communicator in order to obtain information that relates to the number of slaves per node, the rank of a process among the other processes in its own node, the number of nodes, and their ranks.

- Functions for further communicator interrogation.

In addition, BDMPI defines some additional predefined communicators that are used to describe the processes of each node. These are described in Predefined communicators.

5.2.2 Intra-slave synchronization

On a system with a quad core processor it may be reasonable to allow 3-4 slaves to run concurrently. However, if that system's I/O subsystem consisted of only a single disk, then in order to prevent I/O contention, it may be beneficial to limit the number of slaves that perform I/O. To achieve such a synchronization, BDMPI provides a set of functions that can be used to implement mutex-like synchronization among the processes running on the same slave node. The number of slave processes that can be at a critical section concurrently is controlled by the `-nc` option of `bdmprun` (Options of bdmprun).

- Functions for critical sections.

5.2.3 Storage-backed memory allocations

BDMPI exposes various functions that relate to its sbmalloc subsystem (Efficient loading & saving of a process's address space). An application can use these functions to explicitly allocate sbmalloc-handled memory and to also force loading/saving of memory regions that were previously allocated by sbmalloc (see the discussion in Issues related to sbmalloc).

- Functions for storage-backed memory allocation

5.3 Predefined communicators

The following are the communicators that are predefined in BDMPI. Note that both the ones with the `MPI_` and the `BDMPI_` prefix are defined.

MPI Name	BDMPI Name	Description
`MPI_COMM_WORLD`	`BDMPI_COMM_WORLD`	Contains all processes
`MPI_COMM_SELF`	`BDMPI_COMM_SELF`	Contains only the calling process
`MPI_COMM_NULL`	`BDMPI_COMM_NULL`	A null group
N/A	`BDMPI_COMM_NODE`	Contains all processes in a node
N/A	`BDMPI_COMM_CWORLD`	Contains all processes in cyclic order

The last two communicators are BDMPI specific. `BDMPI_COMM_NODE` is used to describe the group of slave processes that were spawned by the same master process (i.e., `bdmprun`). Unless multiple instances of `bdmprun` was started on a single node, these processes will be the ones running on a node for the program.

The ranks of the processes in the `MPI_COMM/WORLD/BDMPI_COMM_WORLD` communicator are ordered in increasing order based on the rank of the hosts specified in `mpiexec's hostfile` and the number of slaves spawned by `bdmprun`. For example, the execution of the `helloworld` program (see Running BDMPI Programs):

```
mpiexec -hostfile ~/machines -np 3 bdmprun -ns 4 build/Linux-x86_64/test/helloworld
```

with the hostfile:

```
bd1-umh:1
bd2-umh:1
bd3-umh:1
bd4-umh:1
```

will create an `MPI_COMM_WORLD/BDMPI_COMM_WORLD` communicator in which the processes with ranks 0–3 are the slave processes on `bd1-umh`, ranks 4–7 are the slave processes on `bd2-umh`, and ranks 8–11 are the slave processes on `bd3-umh`. The `BDMPI_COMM_CWORLD` communicator also includes all processes but assignes ranks in a cyclic fashion based on the hosts specified in `mpiexec's` hostfile. For example, the ranks of the four slaves on `bd1-umh` for the above `helloworld` execution will be 0, 3, 6, and 9; on `bd2-umh` will be 1, 4, 7, and 10; and on `bd3-umh` will be 2, 5, 8, and 11.

5.4 Supported datatypes

The following are the datatypes that are predefined in BDMPI. Note that both the ones with the `MPI_` and the `BDMPI_` prefix are defined.

MPI Name	BDMPI Name	C equivalent
MPI_CHAR	BDMPI_CHAR	char
MPI_SIGNED_CHAR	BDMPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	BDMPI_UNSIGNED_CHAR	unsigned char
MPI_BYTE	BDMPI_BYTE	unsigned char
MPI_WCHAR	BDMPI_WCHAR	wchar_t
MPI_SHORT	BDMPI_SHORT	short
MPI_UNSIGNED_SHORT	BDMPI_UNSIGNED_SHORT	unsigned short
MPI_INT	BDMPI_INT	int
MPI_UNSIGNED	BDMPI_UNSIGNED	unsigned int
MPI_LONG	BDMPI_LONG	long
MPI_UNSIGNED_LONG	BDMPI_UNSIGNED_LONG	unsigned long
MPI_LONG_LONG_INT	BDMPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_LONG_LONG	BDMPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_INT8_T	BDMPI_INT8_T	int8_t
MPI_UINT8_T	BDMPI_UINT8_T	uint8_t
MPI_INT16_T	BDMPI_INT16_T	int16_t
MPI_UINT16_T	BDMPI_UINT16_T	uint16_t
MPI_INT32_T	BDMPI_INT32_T	int32_t
MPI_UINT32_T	BDMPI_UINT32_T	uint32_t
MPI_INT64_T	BDMPI_INT64_T	int64_t
MPI_UINT64_T	BDMPI_UINT64_T	uint64_t
MPI_SIZE_T	BDMPI_SIZE_T	size_t
MPI_SSIZE_T	BDMPI_SSIZE_T	ssize_t
MPI_FLOAT	BDMPI_FLOAT	float
MPI_DOUBLE	BDMPI_DOUBLE	double
MPI_FLOAT_INT	BDMPI_FLOAT_INT	struct {float, int}
MPI_DOUBLE_INT	BDMPI_DOUBLE_INT	struct {double, int}
MPI_LONG_INT	BDMPI_LONG_INT	struct {long, int}
MPI_SHORT_INT	BDMPI_SHORT_INT	struct {short, int}
MPI_2INT	BDMPI_2INT	struct {int, int}

5.5 Supported reduction operations

The following are the reduction operations that are supported by BDMPI. Note that both the ones with the `MPI_` and the `BDMPI_` prefix are defined.

MPI Name	BDMPI Name	Description
MPI_OP_NULL	BDMPI_OP_NULL	null
MPI_MAX	BDMPI_MAX	max reduction
MPI_MIN	BDMPI_MIN	min reduction
MPI_SUM	BDMPI_SUM	sum reduction
MPI_PROD	BDMPI_PROD	prod reduction
MPI_LAND	BDMPI_LAND	logical and reduction
MPI_BAND	BDMPI_BAND	boolean and reduction
MPI_LOR	BDMPI_LOR	logical or reduction
MPI_BOR	BDMPI_BOR	boolean or reduction
MPI_LXOR	BDMPI_LXOR	logical xor reduction
MPI_BXOR	BDMPI_BXOR	boolean xor reduction
MPI_MAXLOC	BDMPI_MAXLOC	max value and location reduction

MPI_MINLOC	BDMPI_MINLOC	min value and location reduction

6 Troubleshooting BDMPI Programs

6.1 Developing and debugging BDMPI programs

BDMPI is still at early development stages and as such its implementation does not contain robust parameter checking and/or error reporting. For this reason, while developing a BDMPI program it may be easier to start with focusing on the MPI aspect of the program and relying on MPICH's robust error checking and reporting capabilities. Once you have an MPI program that runs correctly, it can then be converted to a BDMPI program by simply compiling it using `bdmpicc` or `bdmpic++` and potentially optimized using any of the additional API's provided by BDMPI.

6.2 Issues related to sbmalloc

We are aware of two cases in which sbmalloc's memory subsystem will lead to incorrect program execution.

The first has to do with MPI/BDMPI programs that are multi-threaded (e.g., they rely on OpenMP or Pthreads to parallelize the single node computations). In such programs, the memory that was allocated by sbmalloc and is accessed concurrently by multiple threads (e.g., within an OpenMP parallel region) needs to be pre-loaded prior to entering the parallel region. This is something that needs to be done by the application. See the API in Storage-backed memory allocations on how to do that and specifically the `BDMPI_load()` and `BDMPI_loadall()` functions.

The second has to do with the functions from the standard library that block signals. Examples of such functions are the file I/O functions, such as `read()/write()` and `fread()/fwrite()`. If these functions are used to read/write data to/from memory that has been allocated by sbmalloc, the memory needs to have the appropriate access permissions (read or write). BDMPI provides wrappers for the above four functions that perform such permission changes automatically. However, there may be other functions in the standard library that block signals for which BDMPI does not provide wrappers. If you encounter such functions do the following:

- Send us a note so that we can provide wrappers for them.
- Use `BDMPI_load()` and `BDMPI_loadall()` to obtain read permissions.
- Use `memset()` to zero-fill the associated memory to obtain write permissions.

6.3 Cleaning up after a failed execution

When a BDMPI program exits unsuccessfully (either due to a program error or an issue with BDMPI itself), there may be a number of files that needs to be removed manually. These files include the following:

- Temporary files that BDMPI uses and are located in the working directory specified by the `-wdir` option of `bdmprun` (Options of bdmprun).
- POSIX message queues that are located at `/dev/mqueue/`.
- POSIX shared memory regions that are located at `/dev/shm/`.

Accessing the message queues will require to create/mount the directory. The commands for that are:

```
sudo mkdir /dev/mqueue
sudo mount -t mqueue none /dev/mqueue
```

Information related to that can be obtained by looking at the manpage of mq_overview (i.e., "`man mq_overview`").

7 Guidelines for Developing Efficient BDMPI Programs

In this section we provide some guidelines for developing parallel MPI programs that leverage BDMPI's execution and memory model in order to achieve good out-of-core execution.

7.1 Use collective communication operations

BDMPI's implementation of MPI's collective communication operations has been optimized so that a process becomes runnable only when all the data that it requires is locally available. As a result, when it is scheduled for execution it will proceed to get any data that it may require and resume execution without needing to block for the same collective operation. This approach minimizes the time spent in saving/restoring to/from disk the active parts of the processes' address space, resulting in fast out-of-core execution. For this reason, the application should try to structure its communication patterns so that it uses the collective communication operations.

7.2 Group blocking communication operations together

When a running processes is blocked due to a blocking MPI operation, the part of the address space that it accessed since the last time it was scheduled will most likely be unmapped from physical memory. When the process' blocking condition is lifted (e.g., received the data that it was waiting for) and is scheduled for execution, it will load the parts of the address space associated with the computations that it will perform until the next time it blocks. Note that even if a process requires to access the same parts of the address space as in the previous step, because of the earlier unmapping, they still need to be remapped from disk to physical memory.

The cost of these successive unmapping/remapping operations from/to the physical memory can potentially be reduced by restructuring the computations so that if an application needs to perform multiple blocking communication operations, it performs them one-after-the-other with little computations between them. Of course, such restructuring may not always be possible, but if it can be done, it will lead to considerable performance improvements.

7.3 Freeing scratch memory

If a process, after returning from a blocking communication operation, proceeds to overwrite some of the memory that it allocated previously without first reading from it, then the cost associated with saving them to disk and restoring them from disk is entirely wasted. In such cases, it is better to free that memory prior to performing a blocking communication operation and re-allocating when returning from it. Alternatively, if the allocation is handled by the sbmalloc subsystem (Efficient loading & saving of a process's address space), an application can use the `BDMPI_sbdiscard()` function (Storage-backed memory allocations) to inform the sbmalloc subsystem that the memory associated with the provided allocation does not need to be saved and restored during the next block/resume cycle.

7.4 Structure memory allocations in terms of active blocks

As discussed in Efficient loading & saving of a process's address space, when the application accesses an address from the memory area that was allocated by the sbmalloc subsystem, BDMPI loads in physical memory the entire allocation that contains that address (i.e., all the memory that was allocated as part of the `malloc()` call that allocated the memory containing that address). Given this, the application should structure computations and memory allocations so that it is accessing most of the loaded data prior to performing a blocking operation. This will often involve breaking the memory allocations into smaller segments that include just the elements that will be accesses and potentially restructuring the computations so that they exhibit a segment-based spatial locality (i.e., if they access some data in an allocated segment, then they will most likely access all/most of the data in that segment).

8 Credits & Contact Information

BDMPI was written by George Karypis with contributions by Dominique LaSalle and Shaden Smith.

This work was supported in part by NSF (IOS-0820730, IIS-0905220, OCI-1048018, CNS-1162405, and IIS-1247632) and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

If you encounter any problems or have any suggestions, please contact George Karypis at `karypis@cs.umn.edu`.

9 Copyright & License Notice

10 Module Documentation

10.1 List of implemented MPI functions

10.1.1 Detailed Description

This is the set of MPI functions that are currently implemented in BDMPI.

For each of these functions, BDMPI provides a variant that starts with the `MPI_` prefix and a variant that starts with the `BDMPI_` prefix. The calling sequence of the first variant is identical to the MPI specification whereas the calling sequence of the second variant has been modified to make it 64-bit compliant (e.g., replaced most of the sizes that MPI assumed that were `int` to either `size_t` or `ssize_t`).

Functions

- int **BDMPI_Init** (int ∗argc, char ∗∗argv[])
- int **BDMPI_Finalize** ()
- int **BDMPI_Comm_size** (BDMPI_Comm comm, int ∗size)
- int **BDMPI_Comm_rank** (BDMPI_Comm comm, int ∗rank)
- int **BDMPI_Comm_dup** (BDMPI_Comm comm, BDMPI_Comm ∗newcomm)
- int **BDMPI_Comm_free** (BDMPI_Comm ∗comm)
- int **BDMPI_Comm_split** (BDMPI_Comm comm, int color, int key, BDMPI_Comm ∗newcomm)
- int **BDMPI_Send** (void ∗buf, size_t count, BDMPI_Datatype datatype, int dest, int tag, BDMPI_Comm comm)
- int **BDMPI_Isend** (void ∗buf, size_t count, BDMPI_Datatype datatype, int dest, int tag, BDMPI_Comm comm, BDMPI_Request ∗request)
- int **BDMPI_Recv** (void ∗buf, size_t count, BDMPI_Datatype datatype, int source, int tag, BDMPI_Comm comm, BDMPI_Status ∗status)
- int **BDMPI_Irecv** (void ∗buf, size_t count, BDMPI_Datatype datatype, int source, int tag, BDMPI_Comm comm, BDMPI_Request ∗request)
- int **BDMPI_Probe** (int source, int tag, BDMPI_Comm comm, BDMPI_Status ∗status)
- int **BDMPI_Iprobe** (int source, int tag, BDMPI_Comm comm, int ∗flag, BDMPI_Status ∗status)
- int **BDMPI_Sendrecv** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, int dest, int sendtag, void ∗recvbuf, size_t recvcount, BDMPI_Datatype recvtype, int source, int recvtag, BDMPI_Comm comm, BDMPI_-Status ∗status)
- int **BDMPI_Get_count** (BDMPI_Status ∗status, BDMPI_Datatype datatype, size_t ∗count)
- int **BDMPI_Test** (BDMPI_Request ∗request, int ∗flag, BDMPI_Status ∗status)
- int **BDMPI_Wait** (BDMPI_Request ∗request, BDMPI_Status ∗status)
- int **BDMPI_Waitall** (int count, BDMPI_Request ∗requests, BDMPI_Status ∗statuses)
- int **BDMPI_Barrier** (BDMPI_Comm comm)
- int **BDMPI_Bcast** (void ∗buf, size_t count, BDMPI_Datatype datatype, int root, BDMPI_Comm comm)
- int **BDMPI_Allgather** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_t recv-count, BDMPI_Datatype recvtype, BDMPI_Comm comm)
- int **BDMPI_Allgatherv** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_-t ∗recvcounts, size_t ∗displs, BDMPI_Datatype recvtype, BDMPI_Comm comm)
- int **BDMPI_Reduce** (void ∗sendbuf, void ∗recvbuf, size_t count, BDMPI_Datatype datatype, BDMPI_Op op, int root, BDMPI_Comm comm)
- int **BDMPI_Allreduce** (void ∗sendbuf, void ∗recvbuf, size_t count, BDMPI_Datatype datatype, BDMPI_Op op, BDMPI_Comm comm)
- int **BDMPI_Gather** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_t recvcount, BDMPI_Datatype recvtype, int root, BDMPI_Comm comm)
- int **BDMPI_Gatherv** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_t ∗recvcounts, size_t ∗rdispls, BDMPI_Datatype recvtype, int root, BDMPI_Comm comm)
- int **BDMPI_Scatter** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_t recv-count, BDMPI_Datatype recvtype, int root, BDMPI_Comm comm)

- int **BDMPI_Scatterv** (void ∗sendbuf, size_t ∗sendcounts, size_t ∗sdispls, BDMPI_Datatype sendtype, void ∗recvbuf, size_t recvcount, BDMPI_Datatype recvtype, int root, BDMPI_Comm comm)
- int **BDMPI_Alltoall** (void ∗sendbuf, size_t sendcount, BDMPI_Datatype sendtype, void ∗recvbuf, size_t recvcount, BDMPI_Datatype recvtype, BDMPI_Comm comm)
- int **BDMPI_Alltoallv** (void ∗sendbuf, size_t ∗sendcounts, size_t ∗sdispls, BDMPI_Datatype sendtype, void ∗recvbuf, size_t ∗recvcounts, size_t ∗rdispls, BDMPI_Datatype recvtype, BDMPI_Comm comm)
- int **BDMPI_Scan** (void ∗sendbuf, void ∗recvbuf, size_t count, BDMPI_Datatype datatype, BDMPI_Op op, BDMP-I_Comm comm)
- int **BDMPI_Exscan** (void ∗sendbuf, void ∗recvbuf, size_t count, BDMPI_Datatype datatype, BDMPI_Op op, BD-MPI_Comm comm)
- double **BDMPI_Wtime** (void)
- int **MPI_Init** (int ∗argc, char ∗∗argv[ ])
- int **MPI_Finalize** ()
- int **MPI_Comm_size** (MPI_Comm comm, int ∗size)
- int **MPI_Comm_rank** (MPI_Comm comm, int ∗rank)
- int **MPI_Comm_dup** (MPI_Comm comm, MPI_Comm ∗newcomm)
- int **MPI_Comm_free** (MPI_Comm ∗comm)
- int **MPI_Comm_split** (MPI_Comm comm, int color, int key, MPI_Comm ∗newcomm)
- int **MPI_Send** (void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int **MPI_Isend** (void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request ∗request)
- int **MPI_Recv** (void ∗buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status ∗status)
- int **MPI_Irecv** (void ∗buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request ∗request)
- int **MPI_Probe** (int source, int tag, MPI_Comm comm, MPI_Status ∗status)
- int **MPI_Iprobe** (int source, int tag, MPI_Comm comm, int ∗flag, MPI_Status ∗status)
- int **MPI_Sendrecv** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status ∗status)
- int **MPI_Get_count** (MPI_Status ∗status, MPI_Datatype datatype, int ∗count)
- int **MPI_Test** (MPI_Request ∗request, int ∗flag, MPI_Status ∗status)
- int **MPI_Wait** (MPI_Request ∗request, MPI_Status ∗status)
- int **MPI_Waitall** (int count, MPI_Request ∗requests, MPI_Status ∗statuses)
- int **MPI_Barrier** (MPI_Comm comm)
- int **MPI_Bcast** (void ∗buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- int **MPI_Allgather** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_-Datatype recvtype, MPI_Comm comm)
- int **MPI_Allgatherv** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int ∗recvcounts, int ∗displs, MPI_Datatype recvtype, MPI_Comm comm)
- int **MPI_Reduce** (void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_-Comm comm)
- int **MPI_Allreduce** (void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- int **MPI_Gather** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_-Datatype recvtype, int root, MPI_Comm comm)
- int **MPI_Gatherv** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int ∗recvcounts, int ∗rdispls, MPI_Datatype recvtype, int root, MPI_Comm comm)
- int **MPI_Scatter** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_-Datatype recvtype, int root, MPI_Comm comm)
- int **MPI_Scatterv** (void ∗sendbuf, int ∗sendcounts, int ∗sdispls, MPI_Datatype sendtype, void ∗recvbuf, int recv-count, MPI_Datatype recvtype, int root, MPI_Comm comm)

- int **MPI_Alltoall** (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_-Datatype recvtype, MPI_Comm comm)
- int **MPI_Alltoallv** (void ∗sendbuf, int ∗sendcounts, int ∗sdispls, MPI_Datatype sendtype, void ∗recvbuf, int ∗recvcounts, int ∗rdispls, MPI_Datatype recvtype, MPI_Comm comm)
- int **MPI_Scan** (void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- int **MPI_Exscan** (void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- double **MPI_Wtime** (void)

10.2 Functions for further communicator interrogation

10.2.1 Detailed Description

The description of the functions that follow assume that each compute node runs a single instance of the `bdmprun` master process. In that case, there is a one-to-one mapping between the set of slaves spawn by the `bdmprun` process and the set of ranks of the BDMPI program that execute on the node. However, when `mpiexec` assigns multiple instances of `bdmprun` on the same compute node, then the above mapping does not hold. In such cases, the term *node* does not correspond to a compute node but to a `bdmprun` process and the set of ranks that execute on that node correspond to the slave processes spawned by that `bdmprun` instance.

Functions

- int BDMPI_Comm_lsize (BDMPI_Comm comm, int ∗lsize)
- int BDMPI_Comm_lrank (BDMPI_Comm comm, int ∗lrank)
- int BDMPI_Comm_nsize (BDMPI_Comm comm, int ∗nsize)
- int BDMPI_Comm_nrank (BDMPI_Comm comm, int ∗nrank)
- int BDMPI_Comm_rrank (BDMPI_Comm comm, int ∗rrank)

10.2.2 Function Documentation

10.2.2.1 int BDMPI_Comm_lsize (BDMPI_Comm *comm*, int ∗ *lsize*)

It is used to get the local size of a communicator. The *local size* is the number of ranks of the supplied communicator that are assigned to the same node as the calling process. For example, if a communicator has 5, 3, and 8 processes assigned to three different nodes, then the local size of all the processes in the first node will be 5, for the second node will be 3, and for the third node will be 8.

Parameters

`in`	*comm*	is the communicator being interrogated.
`out`	*lsize*	returns the local size of the calling process in `comm`.

10.2.2.2 int BDMPI_Comm_lrank (BDMPI_Comm *comm*, int ∗ *lrank*)

It is used to get the local rank of the calling process within the supplied communicator. The *local rank* of process is the number of lower-ranked processes that are assigned to the same node in the communicator. For example, if `comm` has 20 ranks and ranks 4, 5, 8, and 15 are assigned to the same node, then the intra-node ranks of these four ranks are 0, 1, 2, and 3, respectively.

Parameters

`in`	*comm*	is the communicator being interrogated.
`out`	*lrank*	returns the intra-node rank of the calling process in the `comm`.

10.2.2.3 int BDMPI_Comm_nsize (BDMPI_Comm *comm*, int ∗ *nsize*)

It is used to get the number of nodes used by the processes of the supplied communicator.

Parameters

`in`	*comm*	is the communicator being interrogated.
`out`	*nsize*	returns the number of nodes used by the union of the ranks in `comm`.

10.2.2.4 int BDMPI_Comm_nrank (BDMPI_Comm *comm*, int ∗ *nrank*)

It is used to get the rank of the node on which the calling process is assigned.

Parameters

`in`	*comm*	is the communicator being interrogated.
`out`	*nrank*	returns the rank of the node in `comm` of the calling process. Note that all the slaves that are assigned to the same node in `comm` will return the same node rank.

10.2.2.5 int BDMPI_Comm_rrank (BDMPI_Comm *comm,* int ∗ *rrank*)

It is used to get the rank of the process on the same node that has a local rank of 0. This is referred to as *root rank*. For example, if a communicator has 10 processes that are distributed among three nodes as follows: {0, 1, 2, 3}, {4, 5}, and {6, 7, 8, 9}; then, the root rank for all processes in the first node will be 0, the root rank for all processes in the second node will be 4, whereas for the third node will be 6.

Parameters

`in`	*comm*	is the communicator being interrogated.
`out`	*rrank*	returns the root rank of calling process in `comm`.

10.3 Functions for critical sections

10.3.1 Detailed Description

These functions and the `-nr` option of `bdmprun` are used to implement a generalized mutex-like synchronization protocol involving the slave processes that were spawned by the same master process. For the rest of this discussion we will refer to the set of slaves spawned by the same master process as *related slaves*.

In the context of BDMPI, a *critical section* is a part of the code that can be executed by fewer slaves than the number of related slaves that can be concurrently running (i.e., as controlled by the `-nr` option of `bdmprun`).

BDMPI uses POSIX semaphores to implement critical sections.

Functions

- int BDMPI_Entercritical (void)
- int BDMPI_Exitcritical (void)

10.3.2 Function Documentation

10.3.2.1 int BDMPI_Entercritical (void)

BDMPI_Entercritical() is used to indicate the start of the critical section. If the number of related slaves that are already executing in their critical sections is equal to `-nr`, then the calling process blocks until one of the other processes exits its critical section (by calling BDMPI_Exitcritical()). At that point, if no other process wants to enter its critical section, the function returns control to the calling process. If there is another process that wants to enter its critical section, the one that called BDMPI_Entercritical() will be allowed to proceed. Note that the blocking of a slave process performed by BDMPI_Entercritical() does not change its execution state w.r.t. BDMPI. That is, the slave process is still considered to be running.

10.3.2.2 int BDMPI_Exitcritical (void)

BDMPI_Entercritical() is used to indicate that the calling process is exiting its critical section.

10.4 Functions for storage-backed memory allocation

10.4.1 Detailed Description

This set of functions provides access to BDMPI's *storage-backed memory allocation* subsystem, which is designed to bypass the system's swap file and achieve faster I/O performance during loading/saving of a process' address space.

Functions

- void ∗ BDMPI_sbmalloc (size_t size)
- void ∗ BDMPI_sbrealloc (void ∗oldptr, size_t size)
- void BDMPI_sbfree (void ∗ptr)
- void BDMPI_sbload (void ∗ptr)
- void BDMPI_sbloadall (void)
- void BDMPI_sbunload (void ∗ptr)
- void BDMPI_sbunloadall (void)
- void BDMPI_sbdiscard (void ∗ptr, size_t size)

10.4.2 Function Documentation

10.4.2.1 void∗ BDMPI_sbmalloc (size_t *size*)

Provides the same functionality as `malloc()`.

10.4.2.2 void∗ BDMPI_sbrealloc (void ∗ *oldptr,* size_t *size*)

Provides the same functionality as `realloc()`. The `oldptr` must be a pointer previously returned by either BDMP-I_sbmalloc() or BDMPI_sbrealloc().

10.4.2.3 void BDMPI_sbfree (void ∗ *ptr*)

Implementation of `free()`. The `ptr` should be a pointer previously returned by either BDMPI_sbmalloc() or BDMPI-_sbrealloc().

10.4.2.4 void BDMPI_sbload (void ∗ *ptr*)

It is used to make the memory associated with a previous BDMPI_sbmalloc() or BDMPI_sbrealloc() allocation available for accessing. If the allocation has been previously written to disk, then it will be restored from the disk.

Parameters

`in`	*ptr*	is a pointer to a previous allocation. Note that this does not have to be a pointer to the start of the region; a pointer anywhere in the allocated region will work.

Note

In order to ensure correct execution of BDMPI multi-threaded programs (e.g., programs whose processes rely on pthreads or OpenMP), any memory that has been allocated with either BDMPI_sbmalloc() or BDMPI_sbrealloc() and is accessed concurrently by multiple threads needs to be loaded prior to entering the multi-threaded region (e.g., parallel region in OpenMP).

10.4.2.5 void BDMPI_sbloadall (void)

It is used to make all the memory associated with any previous BDMPI_sbmalloc() or BDMPI_sbrealloc() allocations available for accessing. If these allocations have been previously written to disk, they will be restored from the disk.

10.4.2.6 void BDMPI_sbunload (void ∗ *ptr*)

It is used to remove from active memory the memory pages associated with a BDMPI_sbmalloc() or BDMPI_sbrealloc() allocation. If any of the pages have been modified since the last time there were unloaded, they are first written to disk prior to removing them from the active memory. The unloading process does not change any of the virtual memory mappings and any subsequent access of the unloaded memory will automatically load it back in active memory.

Parameters

in	*ptr*	is a pointer to a previous allocation. Note that this does not have to be a pointer to the start of the region; a pointer anywhere in the allocated region will work.

10.4.2.7 void BDMPI_sbunloadall (void)

It is used to remove from active memory the memory pages that were previously allocated by any BDMPI_sbmalloc() or BDMPI_sbrealloc() calls.

10.4.2.8 void BDMPI_sbdiscard (void ∗ *ptr*, size_t *size*)

It is used to remove from active memory the memory pages associated with a BDMPI_sbmalloc() or BDMPI_sbrealloc() allocation and to also discard any modifications that may have been made to them. Next time the application access any of the memory in the discarded region it will be treated as if it correspondint to freshly allocated memory (i.e., its values will be undefined).

Parameters

in	*ptr*	is a pointer to a memory location previously allocated by BDMPI_sbmalloc() or BDMPI_sbrealloc().
in	*size*	specifies the number of bytes starting at `ptr` that will be discarded.

References

[1] Dominique LaSalle and George Karypis. BDMPI: Conquering BigData with small clusters using MPI. In *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, 2013. 1, 2

[2] Dominique LaSalle and George Karypis. MPI for Big Data: New Tricks for an Old Dog. Technical Report 13–032, Department of Computer Science & Engineering, University of Minnesota, 2013. 1, 2

Index

BDMPI_Comm_lrank
    Functions for further communicator interrogation, 24
BDMPI_Comm_lsize
    Functions for further communicator interrogation, 24
BDMPI_Comm_nrank
    Functions for further communicator interrogation, 24
BDMPI_Comm_nsize
    Functions for further communicator interrogation, 24
BDMPI_Comm_rrank
    Functions for further communicator interrogation, 25
BDMPI_Entercritical
    Functions for critical sections, 26
BDMPI_Exitcritical
    Functions for critical sections, 26
BDMPI_sbdiscard
    Functions for storage-backed memory allocation, 29
BDMPI_sbfree
    Functions for storage-backed memory allocation, 27
BDMPI_sbload
    Functions for storage-backed memory allocation, 27
BDMPI_sbloadall
    Functions for storage-backed memory allocation, 27
BDMPI_sbmalloc
    Functions for storage-backed memory allocation, 27
BDMPI_sbrealloc
    Functions for storage-backed memory allocation, 27
BDMPI_sbunload
    Functions for storage-backed memory allocation, 27
BDMPI_sbunloadall
    Functions for storage-backed memory allocation, 29

Functions for critical sections, 26
    BDMPI_Entercritical, 26
    BDMPI_Exitcritical, 26
Functions for further communicator interrogation, 24
    BDMPI_Comm_lrank, 24
    BDMPI_Comm_lsize, 24
    BDMPI_Comm_nrank, 24
    BDMPI_Comm_nsize, 24
    BDMPI_Comm_rrank, 25
Functions for storage-backed memory allocation, 27
    BDMPI_sbdiscard, 29
    BDMPI_sbfree, 27
    BDMPI_sbload, 27
    BDMPI_sbloadall, 27
    BDMPI_sbmalloc, 27
    BDMPI_sbrealloc, 27
    BDMPI_sbunload, 27
    BDMPI_sbunloadall, 29

List of implemented MPI functions, 21