MPJ Express: An Implementation of MPI in Java

Linux/UNIX/Mac User Guide

19th March 2014

Document Revision Track

Version	Updates	By
1.0	Initial version document	Aamir Shafi
1.1	A new configuration ‘hybrid configuration’ is added for executing parallel java applications exploiting hybrid parallelism.	Ansar Javed, Mohsan Jameel, Aamir Shafi
1.2	A new configuration ‘Native MPI configuration’ is added for executing parallel java applications on top of a native MPI library.	Bibrak Qamar, Mohsan Jameel, Aamir Shafi

Table of Contents

Table of Figures

1 Introduction

MPJ Express is a reference implementation of the mpiJava 1.2 API, which is an MPI-like API for Java defined by the Java Grande forum. The mpiJava 1.2 API is the Java equivalent of the MPI 1.1 specification document (http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html).

This release of the MPJ Express software contains the core library and the runtime infrastructure. The software also contains a comprehensive test suite that is meant to test the functionality of various communication functions.

MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers. Compute clusters is a popular parallel platform, which is extensively used by the High Performance Computing (HPC) community for large scale computational work. MPJ Express is essentially a middleware that supports communication between individual processors of clusters. The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).

Although MPJ Express is designed for distributed memory machines like network of computers or clusters, it is possible to efficiently execute parallel user applications on desktops or laptops that contain shared memory or multicore processors.

The MPJ Express software can be configured in four ways. The first configuration—known as the Multicore Configuration—is used to execute MPJ Express user programs on laptops and desktops. The second configuration—known as the Cluster Configuration—is used to execute MPJ Express user programs on clusters or network of computers. The third configuration—known as the Hybrid Configuration—is used to execute MPJ Express user programs on clusters of multicore computers. The fourth configuration—known as the Native MPI Configuration—is used to execute MPJ Express user programs on top of a native MPI library (MPICH and MVAPICH or Open MPI etc).

1.1 Configurations

The MPJ Express software can be configured to work on clusters (network of computers) or on laptops/desktops (multicore processors) or on cluster of multicore machines or using a native MPI library for communication.

1.1.1 Multicore configuration

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops—typically such hardware contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We envisage that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platforms including clusters. Also this configuration is preferred for teaching purposes since students can execute message passing code on their personal laptops and desktops. It might be noted that user applications stay the same when executing the code in multicore or cluster configuration.

Under the hoods, the MPJ Express library starts a single thread to represent MPI process. The multicore communication device uses efficient inter-thread mechanism.

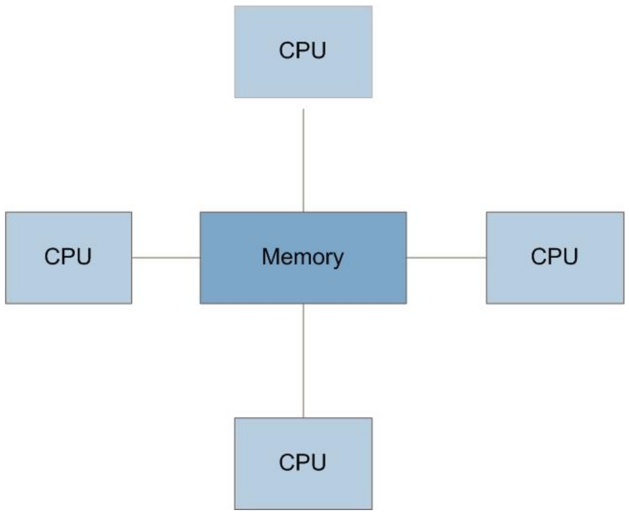

Figure 1: MPJ Express Multicore Configuration Targets the Shared Memory and Multicore Processor Laptops and Desktops

1.1.2 Cluster configuration

The cluster configuration is meant for users who plan to execute their parallel Java applications on distributed memory platforms including clusters or network of computers.

As an example, consider a cluster or network of computers shown in Figure 2 that shows eight compute nodes connected to each other via private interconnect. The MPJ Express cluster configuration will start one MPJ Express process per node, which communicates to each other using message passing.

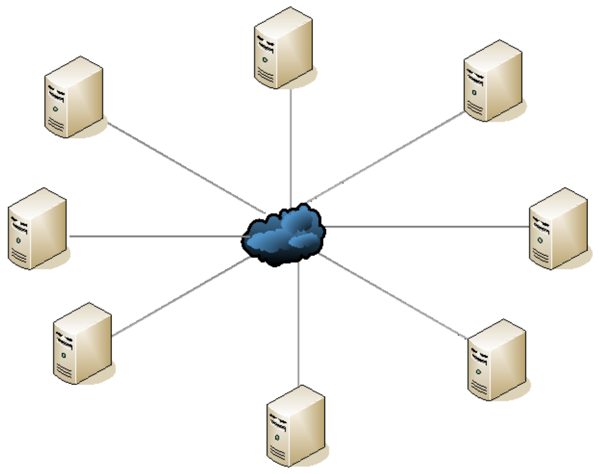

Figure 2: MPJ Express Cluster Configuration Targets the Distributed Memory Platforms Including Clusters and Network of Computers

Application developers can opt to use either of the four communication devices in the cluster configuration:

1. Java New I/O (NIO) device driver known as `niodev`
2. Myrinet device driver known as `mxdev`
3. Hybrid device driver known as `hybdev`
4. Native device driver known as `native`

The Java NIO device driver (also known as `niodev`) can be used to execute MPJ Express programs on clusters or network of computers. The niodev device driver uses Ethernet-based interconnect for message passing. On the other hand, many clusters today are equipped with high-performance low-latency networks like Myrinet. MPJ Express also provides a communication device for message passing using Myrinet interconnect—this device is known as mxdev and is implemented using the Myrinet eXpress (MX) library by Myricom. These communication drivers can be selected using command line switches.

Modern HPC clusters are mainly equipped with multicore processors (Figure 3). The hybrid configuration is meant for users who plan to execute their parallel Java applications on such a cluster of multicore machines. Hybrid configuration transparently uses both multicore configuration and cluster configuration for intra-node communication and cluster configuration (NIO device only) for inter-node communication, respectively.

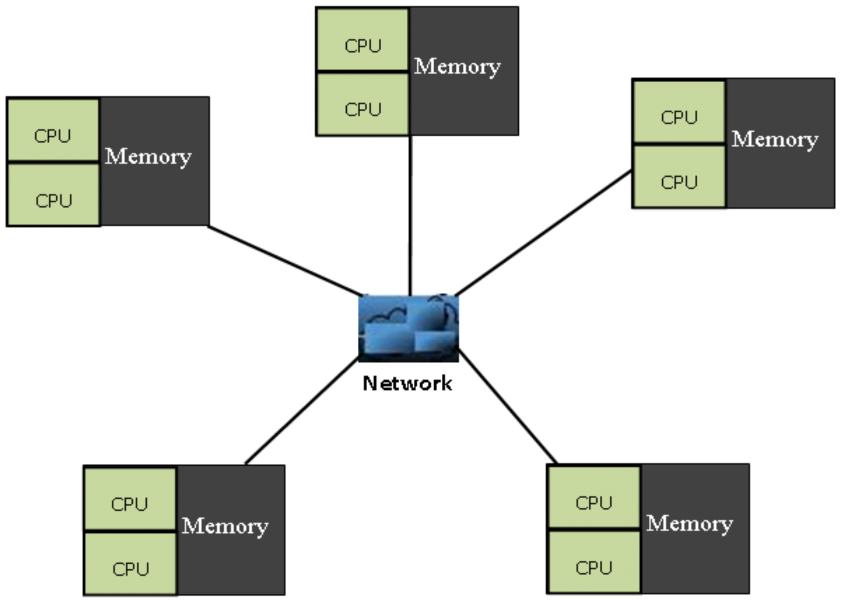

Figure 3: MPJ Express Hybrid Configuration targeting cluster of multicore machines

The fourth device—the Native MPI configuration—is meant for users who plan to execute their parallel Java applications using a native MPI implementation for communication. In this configuration the bulk of messaging logic is offloaded to the underlying MPI library. This is attractive because MPJ Express can exploit latest features, like support for new interconnects and efficient collective communication algorithms, of the native MPI library.

2 Getting Started with MPJ Express

This section shows how MPJ Express programs can be executed in the multicore, cluster and the native MPI configuration.

2.1 Pre-requisites

1. Java 1.6 (stable) or higher
2. Apache ant 1.6.2 or higher (For those who are interested in Compiling source code)
3. Perl (Optional): MPJ Express needs Perl for compiling source code because some of the Java code is generated from Perl templates. The build file will generate Java files from Perl templates if it detects perl on the machine. It is a good idea to install Perl if you want to do some development with MPJ Express.

4. A native MPI library such as MPICH, MVAPICH or Open MPI (For those who are interested in running MPJ Express in native configuration)
5. cmake: MPJ Express needs cmake to generate shared library (libnativempjdev.so) for running parallel Java programs in the native MPI configuration.

2.2 Running MPJ Express in the Multi-core Configuration

This section outlines steps to execute parallel Java programs in the multicore configuration.

1. Download MPJ Express and unpack it
2. Set MPJ_HOME and PATH variables
   a. `export MPJ_HOME=/path/to/mpj/`
   b. `export PATH=$PATH:$MPJ_HOME/bin`

   These lines may be added to ".bashrc" file. However make sure that the shell in which you are setting variables is the 'default' shell. For example, if your default shell is 'bash', then you can set environment variables in .bashrc. If you are using 'tcsh' or any other shell, then set the variables in the respective files.
3. Create a new working directory for MPJ Express programs. This document assumes that the name of this directory is mpj-user.
4. Compile the MPJ Express library (Optional): `cd $MPJ_HOME; ant`
5. Running test cases
   a. Compile (Optional): `cd $MPJ_HOME/test;ant`
   b. Execute: `mpjrun.sh -np 2 -jar $MPJ_HOME/lib/test.jar`
6. Write Hello World MPJ Express program and save it as `HelloWorld.java`

```
import mpi.*;

public class HelloWorld {

        public static void main(String args[]) throws Exception {
                MPI.Init(args);
                int me = MPI.COMM_WORLD.Rank();
                int size = MPI.COMM_WORLD.Size();
                System.out.println("Hi from <"+me+">");
                MPI.Finalize();
        }
}
```

7. Compile: `javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java`

8. Execute: `mpjrun.sh -np 2 HelloWorld`

9. JVM arguments: JVM arguments may be specified to the mpjrun script that passes these directly to the executing MPJ Express processes. For example, the following command modifies the JVM heap size: `mpjrun.sh -np 2 -Xms512M HelloWorld`

10. Application Arguments: Users may pass arguments to their parallel applications by specifying them after `"-jar <jarname>"` or `"classname"` in the mpjrun script:

    a. The user may pass three arguments "a", "b", "c" to the application as follows: `mpjrun.sh -np 2 HelloWorld a b c`

    b. Application arguments can be accessed in the program by calling the `String[] MPI.Init(String[] args)` method. The returned array stores user arguments [a,b,c].

2.3 Running MPJ Express in the Cluster Configuration

This section outlines steps to execute parallel Java programs in the cluster configuration with four communication device drivers including `niodev`, `mxdev`, `hybdev` and `native`.

2.3.1 Cluster Configuration with niodev

1. Download MPJ Express and unpack it

2. Set `MPJ_HOME` and `PATH` variables

```
a. export MPJ_HOME=/path/to/mpj/
b. export PATH=$PATH:$MPJ_HOME/bin
```

   These lines may be added to ".bashrc" file. However make sure that the shell in which you are setting variables is the 'default' shell. For example, if your default shell is 'bash', then you can set environment variables in .bashrc. If you are using 'tcsh' or any other shell, then set the variables in the respective files.

3. Create a new working directory for MPJ Express programs. This document assumes that the name of this directory is mpj-user.

4. Write a machines file stating machine name, IP addresses, or aliases of the nodes where you wish to execute MPJ Express processes. Save this file as '`machines`' `in mpj-user` directory. This file is used by scripts like `mpjboot`, `mpjhalt`, `mpjrun.bat` and `mpjrun.sh` to find out which machines to contact.

Suppose you want to run a process each on 'machine1' and 'machine2', then your machines file would be as follows

```
machine1
machine2
```

Note that in real world, 'machine1' and 'machine2' would be fully qualified names, IP addresses or aliases of your machine

5. Start daemons: `mpjboot machines`

   This should work if `$MPJ_HOME/bin` has been successfully added to `$PATH` variable. This script will `SSH` into each of the machine listed in machines file, change directory to `$MPJ_HOME/bin`, and execute `mpjdaemon` start script to start the daemon. Each daemon produces a MPJ-Daemon<machine_name>.pid file in `$MPJ_HOME/bin` directory and it is used to check the status of daemon. If logging is enabled then each daemon also produces a log file named `daemon-<machine_name>.log` in `$MPJ_HOME/logs` directory.

6. Compile the MPJ Express library (Optional): `cd $MPJ_HOME; ant`
7. Running test cases
   a. Compile (Optional): `cd $MPJ_HOME/test;ant`
   b. Execute: `mpjrun.sh -np 2 -dev niodev -jar $MPJ_HOME/lib/test.jar`
8. Write Hello World MPJ Express program and save it as `HelloWorld.java`

```
import mpi.*;

public class HelloWorld {

        public static void main(String args[]) throws Exception {
                MPI.Init(args);
                int me = MPI.COMM_WORLD.Rank();
                int size = MPI.COMM_WORLD.Size();
                System.out.println("Hi from <"+me+">");
                MPI.Finalize();
        }
}
```

9. Compile: `javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java`
10. Execute: `mpjrun.sh -np 2 -dev niodev HelloWorld`
11. Stop the daemons: `mpjhalt machines`

    It is not required to stop daemons after every execution. Daemons are ready to launch another job after clean exit of application.

12. JVM arguments: JVM arguments may be specified to the mpjrun script that passes these directly to the executing MPJ Express processes. For example, the following command modifies the JVM heap size: `mpjrun.sh -np 2 -dev niodev -Xms512M HelloWorld`

13. Application Arguments: Users may pass arguments to their parallel applications by specifying them after `"-jar <jarname>"` or `"classname"` in the mpjrun script:

    a. The user may pass three arguments “a”, “b”, “c” to the application as follows: `mpjrun.sh -np 2 -dev niodev HelloWorld a b c`

    b. Application arguments can be accessed in the program by calling the `String[] MPI.Init(String[] args)` method. The returned array stores user arguments [a,b,c].

2.3.2 Cluster Configuration with mxdev

Under the cluster configuration, the MPJ Express software also works on Myrinet based clusters. For this purpose, MPJ Express has a communication device that runs on top of Myrinet eXpress (MX) library. Steps for compiling and executing user applications are same as outlined in section 2.3.1. The following steps must be performed additionally to use MPJ Express on Myrinet:

1. Export the `MX_HOME` variable. Assuming the Myrinet eXpress (MX) dirver is in /opt/mx, the variable is exported as follows:

   ```
   export MX_HOME=/opt/mx
   ```

2. Edit build.xml (in $MPJ_HOME) and change the following line:

   ```
   <target name="all" depends="compile,jars,java-docs,clean" >
   ```

   to

   ```
   <target name="all" depends="compile,mxlib,jars,java-docs,clean" >
   ```

   Note that we have added mxlib in the value of "depends" attribute. Being in `$MPJ_HOME` directory, run the command "ant". You will see some funny warning messages from gcc but things will work. The native libraries *.so and JAR files are produced in `$MPJ_HOME/lib` directory.

3. Now write machines file. Basically for this, run "`mx_info`" command in your terminal (assuming $MX_HOME/bin is in the $PATH variable) you'll get something like this:

   ```
   MX Version: 1.1.7rc3cvs1_1_fixes
   MX Build: @indus1:/opt/mx2g-1.1.7rc3 Thu May 31 11:03:00 PKT 2007
   ```

```
2 Myrinet boards installed.
The MX driver is configured to support up to 4 instances and 1024 nodes.
[ .. ]
ROUTE COUNT INDEX MAC ADDRESS HOST NAME P0
----- ----------- --------- ---
0) 00:60:dd:47:ad:7c indus1:0 1,1
1) 00:60:dd:47:ad:68 indus4:0 1,1
[ .. ]
```

Depending upon the machines having Myrinet, write your machines file

4. The device can be used by executing:

```
mpjrun.sh -np 2 -dev mxdev -Djava.library.path=$MPJ_HOME/lib HelloWorld
```

   This command is assuming the Myrinet NICs with id 0 are used, this may be changed by using the `mpjrun` switch called "`-mxboardnum`"

2.3.3 Cluster Configuration with hybdev

This section outlines steps to execute parallel Java programs in the hybrid configuration using multicore and cluster configurations. Hybrid configuration depends on Multicore configuration and Cluster configuration. Make sure that document sections **2.2** and **2.3.1** are completed successfully.

1. Start daemons. `mpjboot machines`
2. Compile the MPJ Express library (Optional): `cd $MPJ_HOME; ant`
3. Running test cases
   c. Compile (Optional): `cd $MPJ_HOME/test;ant`
   d. Execute: `mpjrun.sh -np 2 -dev hybdev -jar $MPJ_HOME/lib/test.jar`

4. Write Hello World MPJ Express program and save it in HelloWorld.java

```
import mpi.*;

public class HelloWorld {

        public static void main(String args[]) throws Exception {
                MPI.Init(args);
                int me = MPI.COMM_WORLD.Rank();
                int size = MPI.COMM_WORLD.Size();
                System.out.println("Hi from <"+me+">");
                MPI.Finalize();
        }
}
```

5. Compile: `javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java`

6. Execute: `mpjrun.sh -np 4 -dev hybdev HelloWorld`

7. Stop daemons: `mpjhalt machines`

   It is not required to stop daemons after every execution. Daemons are ready to launch another job after each clean exit of job.

8. JVM arguments: JVM arguments may be specified to the mpjrun script that passes these directly to the executing MPJ Express processes. For example, the following command modifies the JVM heap size: `mpjrun.sh -np 2 –dev hybdev -Xms512M HelloWorld`

9. Application Arguments: Users may pass arguments to their parallel applications by specifying them after `"-jar <jarname>"` or `"classname"` in the mpjrun script:

   a. The user may pass three arguments “a”, “b”, “c” to the application as follows: `mpjrun.sh -np 2 –dev hybdev HelloWorld a b c`

   e. Application arguments can be accessed in the program by calling the `String[] MPI.Init(String[] args)` method. The returned array stores user arguments [a,b,c].

      ```
      String appArgs[] = MPI.Init(args);
      ```

2.3.4 Cluster Configuration with native MPI

This section outlines steps to execute parallel Java programs in the Native MPI configuration.

1. Download MPJ Express and unpack it

2. Set `MPJ_HOME` and `PATH` variables

   a. `export MPJ_HOME=/path/to/mpj/`
   b. `export PATH=$PATH:$MPJ_HOME/bin`

   These lines may be added to “.bashrc” file. However make sure that the shell in which you are setting variables is the ‘default’ shell. For example, if your default shell is ‘bash’, then you can set environment variables in .bashrc. If you are using ‘tcsh’ or any other shell, then set the variables in the respective files.

3. Create a new working directory for MPJ Express programs. This document assumes that the name of this directory is mpj-user.

4. Write a machines file stating machine name, IP addresses, or aliases of the nodes where you wish to execute MPJ Express processes. Save this file as 'machines' in `mpj-user` directory. This file is used by `mpjrun.sh` and `mpirun` to find out which machines to contact.

   Suppose you want to run a process each on 'machine1' and 'machine2', then your machines file would be as follows

   ```
   machine1
   machine2
   ```

   Note that in real world, 'machine1' and 'machine2' would be fully qualified names, IP addresses or aliases of your machine

5. Check if your native MPI library works. It is assumed that the user has installed and tested the native MPI library. Currently MPJ Express is only tested on MPICH 3.0.4, MVAPICH 2.2 and Open MPI 1.7.4.

6. Compile the MPJ Express library (Optional): `cd $MPJ_HOME; ant`

7. Compile the JNI wrapper library (Mandatory)

   a. Make sure `cmake` (2.6 or above) is installed on the system.

   b. Create build directory: `cd $MPJ_HOME/src/natmpjdev/lib; mkdir build`

   c. Generate Makefile using cmake: `cd $MPJ_HOME/src/natmpjdev/lib/build; cmake ..`

   d. make: `cd $MPJ_HOME/src/natmpjdev/lib/build; make`

   e. install: `cd $MPJ_HOME/src/natmpjdev/lib/build; make install`

      i. This creates a shared library with the name `“libnativempjdev.so”` in `“$MPJ_HOME/lib”`

8. Running test cases

   a. Compile : `cd $MPJ_HOME/test/nativetest; ./compile.sh`

   b. Execute: `cd $MPJ_HOME/test/nativetest; ./runtest.sh`

      i. To supply a machine file provide full path in the first argument of this script: `./runtest.sh /full/path/to/machinefile`

Write Hello World MPJ Express program and save it as `HelloWorld.java`

```
import mpi.*;

public class HelloWorld {

	public static void main(String args[]) throws Exception {
		MPI.Init(args);
		int me = MPI.COMM_WORLD.Rank();
		int size = MPI.COMM_WORLD.Size();
		System.out.println("Hi from <"+me+">");
		MPI.Finalize();
	}
}
```

9. Compile: `javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java`
10. Execute: `mpjrun.sh -np 2 -dev native HelloWorld`
11. JVM arguments: Please refer to section 2.3.4.1 for details on how to pass JVM arguments like `-cp`, `-Xms512M` and `-Djava.library.path` and more.
12. Application Arguments: Users may pass arguments to their parallel applications by specifying them after `"-jar <jarname>"` or `"classname"` in the mpjrun script:

    a. The user may pass three arguments “a”, “b”, “c” to the application as follows: `mpjrun.sh -np 2 -dev native HelloWorld a b c`

    f. Application arguments can be accessed in the program by calling the `String[] MPI.Init(String[] args)` method. The returned array stores user arguments [classname, a,b,c].

    `String appArgs[] = MPI.Init(args);`

2.3.4.1 Advanced Options:

Running directly with mpirun to use options provided by native MPI library

This is for the advanced user who wants to run parallel Java programs using custom options for the native MPI library.

The `mpjrun.sh` script provides a wrapper to native `mpirun` command. The user can bypass `mpjrun.sh` and directly call `mpirun` using the following template.

```
mpirun -np <number of processes> -machinefile </path/to/file/filename> java -cp
$MPJ_HOME/lib/mpj.jar:. -Djava.library.path=$MPJ_HOME/lib HelloWorld 0 0 native
userarg1 userarg2 userarg3
```

The above template consists of three parts: **mpirun**, **java** and **user application**. In this way the user has flexibility to supply three kinds of options:

1. mpirun: these are supplied to native MPI library bootstrapping framework a.k.a `mpirun`, for example `-np` and `-machinefile`
2. java: these are supplied to the JVM for example `-cp`, `-Xms512M` and `-Djava.library.path` and more.
3. user application: these are supplied to the user application for example `userarg1 userarg2 userarg3` in the above template. The three arguments `0 0 native` following user application (classname or jar) are reserved for MPJ Express and are to be kept intact. MPJ Express for conventional reasons searches for device name on argument index 3 (i.e `args[2]`).

3 MPJ Express Debugging

This section shows how to debug various modules of the MPJ Express software. It is possible to debug MPJ Express on three levels:

1. The `mpjrun` Script: This script allows bootstrapping MPJ Express programs in cluster of multicore configuration.
2. Core Library: Internals of the MPJ Express Software
3. MPJ Express Daemons: While running the cluster configuration, daemons execute on compute nodes and are responsible for starting and stopping MPJ Express processes when contacted by the `mpjrun` script.

3.1 The `mpjrun` Script

To turn ON debugging for the `mpjrun` script, follow these steps:

1. Edit `$MPJ_HOME/src/runtime/starter/MPJRun.java` and change the value of `static boolean DEBUG` flag to `true`
2. Recompile the code: `cd $MPJ_HOME ; ant`
3. The `mpjrun` script relevant log file is $`MPJ_HOME/logs/mpjrun.log` file

3.2 Core Library

To turn ON debugging for the core library, follow these steps:

1 Edit `$MPJ_HOME/src/mpi/MPI.java` and change value of `static boolean DEBUG` flag to `true`

2 Recompile the code: `cd $MPJ_HOME ; ant`

3 If the total number of MPJ Express processes is two, then the relevant log files will be `$MPJ_HOME/logs/mpj0.log` and `$MPJ_HOME/logs/mpj1.log` for processes 0 and 1 respectively.

3.3 MPJ Express Daemons (Cluster configuration only)

The MPJ Express daemons running on compute nodes can be debugged in two steps outlined below:

4.3.1 Step One: Modifying the wrapper.conf file

1. Edit `$MPJ_HOME/conf/wrapper.conf` file.
2. Change the value of `wrapper.logfile.loglevel` from "`NONE`" to "`DEBUG`".
3. Now the output of `mpjboot`, `mpjhalt`, and other daemon activities can be seen in `$MPJ_HOME/logs/wrapper.log` file. This information is pretty useful for diagnosing and fixing daemons errors.

4.3.2 Step Two: Modifying MPJDaemon.java file

1. Edit `$MPJ_HOME/src/runtime/daemon/MPJDaemon.java` file.

2. Change the value of `static boolean DEBUG` flag to `true`

4 Recompile the code: `cd $MPJ_HOME ; ant`

3. Now log files can be seen in `$MPJ_HOME/logs/daemon-<machine_name>.log` file.

Additional Optional Information: Running daemons in console mode on compute nodes

For debugging purposes, sometimes it is useful to run the daemons in console mode on compute nodes. This can be achieved in the following way:

1. *`cd $MPJ_HOME/bin`*

2. *Execute `./mpjdaemon_linux_x86_32` . Here we are starting the daemon on a 32 bit x86 processor. Choose the appropriate script for your machine.*

4 Known Issues and Limitations

A list of known issues and limitations of the MPJ Express software are listed below.

1. There is a known (up to some extent) problem on Windows and Solaris that results in hanging MPJ processes. Normally this will be observed when MPJ test-cases will hang, as result, not completing or throwing any error message.

   We partially understand the problem but if some user encounters this problem, we would request some more debugging information. The required information can be obtained as follows. Edit `$MPJ_HOME/src/xdev/niodev/NIODevice.java` and goto line 3693 and uncomment the line "`ioe1.printStackTrace() ;`". The line 3693 is in the MPJ Express release 0.34 and it might change in the future. The general code snippet is like this:

```
       catch (Exception ioe1) {
         if(mpi.MPI.DEBUG && logger.isDebugEnabled() )  {
           logger.debug(" error in selector thread "
+ ioe1.getMessage());         }                      //ioe1.printStackTrace() ;
         } //end catch(Exception e) ...
        if(mpi.MPI.DEBUG && logger.isDebugEnabled()) {
 logger.debug(" last statement in selector thread");       }
    } //end run()
  }; //end selectorThread which is an inner class
```

   As a result now, when test-cases are executed again, users will see stacks periodically. Most of these are related to socket closed exceptions that are normal. If the code hangs now, the latest stack trace that is not about socket being closed is perhaps the reason of this hanging behavior. We would request the users to kindly email us the output so that we can fix the problem. A stack trace that leaves MPJ Express hanging on Solaris is as follows:

```
java.nio.channels.CancelledKeyException
at sun.nio.ch.SelectionKeyImpl.ensureValid(SelectionKeyImpl.java:55)
at sun.nio.ch.SelectionKeyImpl.readyOps(SelectionKeyImpl.java:69)
at java.nio.channels.SelectionKey.isAcceptable(SelectionKey.java:342)
at xdev.niodev.NIODevice$2.run(NIODevice.java:3330)
at java.lang.Thread.run(Thread.java:595)
```

2. Some users have noticed that it takes a long time to bootstrap MPJ Express processes. For example,

```
user@machine:~/mpj-user> mpjrun.sh -np 6 -jar $MPJ_HOME/lib/test.jar
16:15:43.400 EVENT  Starting Jetty/4.2.23
16:15:43.415 EVENT  Started HttpContext[/]
```

```
16:15:43.419 EVENT  Started SocketListener on 0.0.0.0:15000
16:15:43.419 EVENT  Started org.mortbay.http.HttpServer@23ac23ac
16:15:43.420 EVENT  Starting Jetty/4.2.23
16:15:43.420 EVENT  Started HttpContext[/]
16:15:43.421 EVENT  Started SocketListener on 0.0.0.0:15001
16:15:43.421 EVENT  Started org.mortbay.http.HttpServer@50265026
[ pause for a minute or two ]

Starting process <0> on  Starting process <1> on

[ pause for a minute or two ]
Starting process <2> on  Starting process <3> on

[ pause for a minute or two ]
Starting process <4> on  Starting process <5> on
[ job starts ]
```

Thanks to Andy Botting who is one of the users that identified this problem. This problem is perhaps related to name resolution and we are currently working to fix it.

3. The merge operation is implemented with limited functionality. The processes in local-group and remote-group *have* to specify 'high' argument. Also, the value specified by local-group processes should be opposite to remote-group processes.

4. Any message sent with `MPI.PACK` can only be received by using `MPI.PACK` as the datatype. Later, `MPI.Unpack(..)` can be used to unpack different datatypes

5. Using 'buffered' mode of send with `MPI.PACK` as the datatype really does not use the buffer specified by `MPI.Buffer_attach(..)` method.

6. `Cartcomm.Dims_Create(..)` is implemented with limited functionality. According to the MPI specifications, non-zero elements of 'dims' array argument will not be modified by this method. In this release of MPJ Express, all elements of 'dims' array are modified without taking into account if they are zero or non-zero.

7. `Request.Cancel(..)` is not implemented in this release.

8. MPJ applications should not print more than 500 characters in one line. Some users may use `System.out.print(..)` to print more than 500 characters. This is not a serious problem, because printing 100 characters 5 times with `System.out.println(..)` will have the same effect as printing 500 characters with one `System.out.print(..)`

9. Some users may see this exception while trying to start the `mpjrun` module. This can happen when the users are trying to run `mpjrun.bat` script. The reason for this error is that the `mpjrun` module cannot contact the daemon and it tries to clean up the resources it has. In doing so, it tries to delete a file named 'mpjdev.conf' using `File.deleteOnExit()` method. This method appears not to work on Windows possibly because of permission issues.

```
Exception in thread "main" java.lang.RuntimeException: Another mpjrun module is
already running on this machine
at runtime.starter.MPJRun.(MPJRun.java:135)
at runtime.starter.MPJRun.main(MPJRun.java:925)
```

This issue can be resolved by deleting mpjdev.conf file. This file would be present in the directory, where your main class or JAR file is present. So for example, if the users are trying to run `"-jar ../lib/test.jar"`, then this file would be present in `../lib` directory.

10. The MPJ Express infrastructure does not deal with security. The MPJ Express daemons could be a security concern, as these are Java applications listening on a port to execute user-code. It is therefore recommended that the daemons run behind a suitably configured firewall, which only listens to trusted machines. In a normal scenario, these daemons would be running on the compute-nodes of a cluster, which are not accessible to outside world. Alternatively, it is also possible to start MPJ Express processes 'manually', which could help avoid runtime daemons. In addition, each MPJ Express process starts at least one server socket, and thus is assumed to be running on machine with configured firewall. Most MPI implementations assume firewalls as protection mechanism from the outside world

11. One of the known issues of MPJ Express in cluster configuration is incorrect working directory. This issue is reported on cluster build using Rocks clusters. MPJRun module of MPJ Express reads the current directory i.e. user directory using System.getProperty("user.dir"). It should return same path as Unix 'pwd' command. But it is not giving same result.

    ```
    java.io.IOException: Cannot run program "java" (in directory
    "/state/partition1/home/aamir/projects/mpj-user"): error=2, No such file or
    directory
      at java.lang.ProcessBuilder.start(ProcessBuilder.java:1029)
      at runtime.daemon.MPJDaemon.<init>(MPJDaemon.java:398)
      at runtime.daemon.MPJDaemon.main(MPJDaemon.java:1144)
    Caused by: java.io.IOException: error=2, No such file or directory
      at java.lang.UNIXProcess.forkAndExec(Native Method)
    ```

    As a manual work around for this issue is to use -wdir switch in mpjrun command and giving path to the current directory where HelloWorld is placed i.e.

    ```
    mpjrun.sh -np 4 -dev niodev -wdir /export/home/aamir/projects/mpj-user/
    HelloWorld
    ```

5 Contact and Support

For help and support, join and post on the MPJ Express mailing list (https://lists.sourceforge.net/lists/listinfo/mpjexpress-users). Alternatively, you may also contact us directly:

1. Aamir Shafi (aamir.shafi@seecs.edu.pk)
2. Mohsan Jameel (mohsan.jameel@seecs.edu.pk)
3. Bryan Carpenter (bryan.carpenter@port.ac.uk)
4. Mark Baker (http://acet.rdg.ac.uk/~mab)
5. Guillermo Lopez Taboada (http://www.des.udc.es/~gltaboada)

6 Appendices

Appendix A: Running MPJ Express without the runtime (manually)

There are two fundamental ways of running MPJ Express applications. The first, and the recommended way is using the MPJ Express runtime infrastructure, alternatively the second way involves the 'manual' start-up of MPJ Express processes. We do not recommend starting programs manually as normal procedure. This section documents the procedure for manual start-up, mainly to allow developers the flexibility to create their own initiation mechanisms for MPJ Express programs. The runmpj.sh script can be considered one example of such a mechanism.

1. `cd mpj-user`
2. This document is assuming mpj-user as the working directory for users. The name mpj-user itself has no significance.
3. Write a configuration file called 'mpj.conf' as follows.
   a. A typical configuration file that would be used to start two MPJ Express processes is as follows. Note the names 'machine1' and 'machine2' would be replaced by aliases/fully-qualified-names/ IP-addresses of the machines where you want to start MPJ Express processes

```
# Number of processes
2
Protocol switch limit
131072
```

```
Entry in the form of machinename@port@rank
machine1@20000@0
machine2@20000@1
```

b. The lines starting with '#' are comments. The first entry which is a number ('2' above) represents total number of processes. The second entry, which is again a number ('131072' above) is the protocol switch limit. At this message size, MPJ Express changes its communication protocol from eager-send to rendezvous. There are a couple of entries, one for each MPJ Express process, each in the form of machine name (OR)IP@PORT_NUMBER@RANK. Using this, the users of MPJ Express can control where each MPJ Express process runs, what server port it uses, and what should be the rank of each process. The rank specified here should exactly match the rank argument provided while manually starting MPJ Express processes (using java command). When the users decide to run their code using `mpjrun`, this file is generated programmatically.

c. Sample configuration files can be found in `$MPJ_HOME/conf` directory. If you wish to start MPJ processes on `localhost`, see `$MPJ_HOME/conf/local2.conf` file.

d. Each MPJ process uses two ports. Thus, do not use consecutive ports if you are trying to execute multiple MPJ Express processes on same node. A sample file for running two MPJ Express processes on same machine would be

```
# Number of processes
2
# Protocol switch limit
131072
# Entry in the form of machinename@port@rank
localhost@20000@0
localhost@20002@1
```

4. Running your MPJ Express program.

a. The script `runmpj.sh` requires password-less SSH access to machines listed in the configuration file. This script will not work if your machines are not setup for this. You may get some guidance here regarding setting up SSH so that no password/passphrase is required at login. This is the only script in this software which requires password-less access. An alternative to using `runmpj.sh` is the manual start-up (using java command directly -- see directions below)

b. Running class files

```
runmpj.sh mpj.conf World
```

For all the machines listed in mpj.conf, login to each machine, change directory to `$MPJ_HOME`

```
java -cp .:$MPJ_HOME/lib/mpj.jar World <rank> mpj.conf niodev
```

The <rank> argument should be 0 for process 0 and 1 for process 1. This should match to what has been written in configuration file (mpj.conf). Check the entry format in the configuration file to be sure of the rank

c. Running JAR files

```
runmpj.sh mpj.conf hello.jar
```

Windows and Linux:

For all the machines listed in mpj.conf, login to each machine

```
java -jar hello.jar <rank> mpj.conf niodev
```

The <rank> argument should be 0 for process 0 and 1 for process 1. This should match to what has been written in configuration file (mpj.conf). Check the entry format in the configuration file to be sure of the rank.

d. Passing arguments to the JVM running MPJ Express program

Edit `$MPJ_HOME/bin/runmpj.sh` shell script to pass the arguments to the JVM.

e. Passing arguments to MPJ Express application.

Edit `$MPJ_HOME/bin/runmpj.sh` shell script to pass the arguments to the application. `MPI.Init(String[] args)` returns a String array that contains user specified arguments. If the user has specified two arguments, then `MPI.Init(..)` returns an array which has length 2

Appendix B: Changing protocol limit switch

MPJ Express uses two communication protocols: the first is 'eager-send', which is used for transferring small messages. The other protocol is rendezvous protocol useful for transferring large messages. The default protocol switch limit is 128 KBytes. This can be changed prior to execution in following ways depending on whether you are running processes manually or using the runtime.

1. Running MPJ Express applications manually (without using runtime): The users may edit configuration file (for e.g. `$MPJ_HOME/conf/mpj2.conf`) to change protocol switch limit. Look at the comments in this configuration file. The second entry, which should be 131072 if you have not changed it, represents protocol switch limit

2. Running MPJ Express applications with the runtime: Use `-psl <val>` switch to change the protocol switch limit

Appendix C: MPJ Express Testsuite

MPJ Express contains a comprehensive test suite to test the functionality of almost every MPI function. This test suite consists mainly of mpiJava test cases, MPJ JGF benchmarks, and MPJ microbenchmarks. The mpiJava test cases were originally developed by IBM and later translated to Java. As this software follows the API of mpiJava, these test cases can be used with a little modification. MPJ JGF benchmarks are developed and maintained by EPCC at the University of Edingburgh. MPJ Express is redistributing these benchmarks as part of its test suite. The original copyrights and license remain intact as can be seen in source-files of these benchmarks in $MPJ_HOME/test/jgf_mpj_benchmarks. Further details about these benchmarks can be seen here. MPJ Express also redistributes micro-benchmarks developed by Guillermo Taboada. Further details about these benchmarks can be obtained here

Compiling source code and Testsuite

1. Compiling MPJ Express source code

   a. Being in `$MPJ_HOME` directory, execute `ant`

      Produces `mpj.jar`, `daemon.jar`, and `starter.jar` in `lib` directory

2. Compiling MPJ Express test-code

   a. `cd test`
   b. `ant`

   This produces `test.jar` in `lib` directory.

Running Testsuite

The suite is located in `$MPJ_HOME/tests` directory. The test cases have been changed from their original versions, in order to automate testing. `TestSuite.java` is the main class that calls each of the test case present in this directory. The build.xml file present in test directory, compiles all test cases, and places test.jar into the lib directory. By default, JGF MPJ benchmarks and MPJ micro-benchmarks are disabled. Edit `$MPJ_HOME/test/TestSuite.java` to uncomment these tests and execute them. Note, after changing `TestSuite.java`, you will have to recompile the testsuite by executing `'ant'` in test directory.

```
1. cd mpj-user
```

With Runtime

2. Write a machines file

3. `mpjrun.sh -np 2 -jar $MPJ_HOME/lib/test.jar`

Without Runtime

1. Write a configuration file called 'mpj.conf'. Further details about writing configuration file and its format can be found here

a. Start the tests

```
runmpj.sh mpj.conf $MPJ_HOME/lib/test.jar
```

'runmpj.sh' requires password-less SSH access to machines in the configuration file. To see how this can be done, look here

For all the machines listed in mpj.conf, login to each machine, type,

```
java -jar $MPJ_HOME/lib/test.jar <rank> mpj.conf niodev
```

The <rank> argument should be 0 for process 0 and 1 for process 1. This should match to what has been written in configuration file (mpj.conf). Check the entry format in the configuration file to be sure of the rank.