

# S T A G E - M P I © 1.0

## User Manual

Ahmad Faraj  
Department of Computer Science,  
Florida State University  
Tallahassee, FL 32306  
[faraj@cs.fsu.edu](mailto:faraj@cs.fsu.edu)

January 1, 2006

# 1 Introduction

STAGE-MPI (Static Tuning and Automatic Generation of Efficient MPI collective communication routines) is a system that automatically tunes and produces customized high performance MPI collective communication routines for Ethernet switched clusters with any physical topology. STAGE-MPI 1.0 achieves the best results on Ethernet clusters with single CPU nodes, although it also works on other types of nodes (e.g. SMP nodes) and other types of networks. More support for SMP/multi-core nodes and other types of high speed system area networks will be added in later releases.

The MPI routines supported by STAGE-MPI 1.0 are *MPI\_Alltoall*, *MPI\_Alltoallv*, *MPI\_Allgather*, *MPI\_Allgatherv*, *MPI\_Allreduce*, *MPI\_Bcast*, *MPI\_Gather*, *MPI\_Scatter*, and *MPI\_Reduce*. Currently, STAGE-MPI maintains extensive sets of algorithms (including automatic routine generators that produce topology specific algorithms) for *MPI\_Alltoall*, *MPI\_Alltoallv*, *MPI\_Allgather*, *MPI\_Allgatherv*, *MPI\_Allreduce*, and *MPI\_Bcast*, and can generally get good results for these routines. Although *MPI\_Reduce*, *MPI\_Scatter*, and *MPI\_Gather* are supported by the system, the algorithms for these operations are still incomplete and the system may not obtain good results for these operations. STAGE-MPI 1.0 runs on MPICH 2. This document describes how to install and use the system.

## 2 Installation

A version of MPICH 2 must be installed before STAGE-MPI can be installed (STAGE-MPI runs on top of MPICH 2). Refer to <http://www-unix.mcs.anl.gov/mpi/mpich2> for information about installing MPICH 2.

To simplify the illustration, we will assume that the MPICH bin path is `'/usr/local/mpich2/bin'` and STAGE-MPI is to be installed under directory `'/home/example'`. The installation consists of the following three steps:

- Step 1: Move the tar file (`stage-mpi.tar.gz`) to `'/home/example'` and unpack the package with the following commands.

```
/home/example> gunzip stage-mpi.tar.gz
/home/example> tar xvf stage-mpi.tar
```

- Step 2: Setup environment variables: `STAGE_HOME` (home directory of STAGE-MPI), `MPICH2_BIN` (the MPICH 2 bin path), `MPICC` (the path for `mpicc` in MPICH 2), `MPIRUN` (the path for `mpiexec` in MPICH 2), `BUFF_SIZE` (the maximum message buffer size). `BUFF_SIZE` is the product of the number of processors in the system and the maximum message size for an all-to-all operation. For example, if users wish to tune *MPI\_Alltoall* with a message size of 512000 on 32 processors, then `BUFF_SIZE` should be set to  $512000 * 32 = 16384000$ . Lastly, add paths `'${STAGE_HOME}/scripts'` and `'${STAGE_HOME}/bin'` to the `PATH` variable. Examples of the commands to set-up the environment variables follow (these lines may be copied into the `.tcshrc` file):

```
/home/example> setenv STAGE_HOME /home/example/STAGE-MPI
```

```

/home/example> setenv MPICH2_BIN /usr/local/mpich2/bin
/home/example> setenv MPICC /usr/local/mpich2/bin/mpicc
/home/example> setenv MPIRUN /usr/local/mpich2/bin/mpiexec
/home/example> setenv BUFF_SIZE 16000000
/home/example> setenv PATH ${STAGE_HOME}/scripts:${STAGE_HOME}/bin:${PATH}

```

- Step 3: Go to the STAGE-MPI home directory and issue './configure' and then change directory to \${STAGE\_HOME}/tuning and type 'make'.

```

/home/example> cd STAGE-MPI
/home/example/STAGE-MPI> ./configure
.....
/home/example/STAGE-MPI> cd tuning
/home/example/STAGE-MPI/tuning> make
.....

```

If everything goes ok, an executable called 'tune' will be produced in the \${STAGE\_HOME}/tuning directory.

### 3 Running STAGE-MPI

The MPICH MPDs must be established before the STAGE-MPI can be started. The process to establish MPICH MPDs for STAGE-MPI is similar to that for running a regular MPICH program. A hostfile must be created first. For example, assume that 4 machines: beowulf1, beowulf2, beowulf3, and beowulf4, are used to run the program, the content in the hostfile is

```

beowulf1
beowulf2
beowulf3
beowulf4

```

Once the hostfile is created. Booting MPICH MPDs for STAGE-MPI uses the command `boot-mpich nprocs hostfilename`, which is a script produced by our package. `boot-mpich` calls `mpdboot` to establish the MPDs. In addition, it also creates some necessary files for STAGE-MPI. An example for using `boot-mpich` is as follows:

```

/home/example/STAGE-MPI/tuning> boot-mpich 4 hostfile

```

Here, the number of processors is 4 and the hostfile name is 'hostfile' (assumed to be in /home/example/STAGE-MPI/tuning). `boot-mpich` only needs to be called once for each configuration.

Now, you are ready to run the tuning system. Change directory to \${STAGE\_HOME}/tuning and type 'tune' to start tuning.

```
/home/example/STAGE-MPI> cd tuning
/home/example/STAGE-MPI/tuning> tune
```

The system will prompt the user for a series of inputs:

1. The number of processors to tune for. This number must be less than or equal to the number of processors used to boot the MPICH MPDs (`nprocs` used in `boot-mpich`). The generated routines are optimized for the number of processes (and its implicit topology).
2. Whether to use the network topology information. If the topology information is not available, only general purpose algorithms will be used in the tuning. The system implicitly assumes that all nodes are connected by one switch. If the topology information is available, the user will need to input the network type and the topology file that contains the topology information. The format of the topology file will be discussed in the next section. Currently, the only network type supported by our system is 'ethernet'. In this case, the system will automatically produce the topology specific algorithms and use them in the tuning.
3. The MPI collective communication routine to tune. The user may choose to tune a particular routine or all non-v MPI routines. If the user chooses to tune a v-version routine (*MPI\_Alltoallv*, *MPI\_Allgatherv*, etc), the system will ask for a message size pattern file and produce the routine that is optimized for that particular message pattern. The format of the message size pattern file will be discussed in the next section.
4. Timing mechanism. The system maintains a set of timing mechanisms, which decide how the performance is measured. The user can choose one mechanism from the set. There are three timing mechanisms provided by the system: `mpptest-barrier`, `mpptest`, and `OL`. Please refer to Section 5 for the details of these timing mechanisms.
5. Message size. The user has two options: tuning for a specific message size, or for a range of message sizes. If user picks the second option, he/she will be prompted to enter the lower and upper bounds for the message size range.
6. Whether to use the log file. If, for some reason, the tuning was not completed for a previous execution, choosing to use the log file will continue the tuning process from the breakpoint.

Following is an example for running the tuning program. In this example, the *MPI\_Alltoall* routine for all message sizes is tuned on 8 nodes with no topology information (no topology information assumes all nodes are connected by a single switch).

```
/home/example/STAGE-MPI/tuning> tune
Enter the number of processors in the system: 8
Do you have topology information (y or n): n
```

- 0) alltoall
- 1) allgather
- 2) allreduce
- 3) bcast
- 4) alltoallv
- 5) allgatherv
- 6) gather
- 7) scatter
- 8) reduce
- 9) gatherv
- 10) scatterv

Select a number from above to tune an MPI routine (-1 to tune all): 0

- 0) mpptest-barrier
- 1) mpptest
- 2) ol

Please pick from the above timing mechanisms: 1

Do you want to tune for a specific message size (y or n): n

Enter message range, there are 14 message sizes points:

- 0: 0
- 1: 1
- 2: 64
- 3: 256
- 4: 1K
- 5: 2K
- 6: 4K
- 7: 8K
- 8: 16K
- 9: 32K
- 10: 64K
- 11: 128K
- 12: 256K
- 13: ( > 256K) INFINITY

Enter lower bound for message range (0 to 12): 0

Enter upper bound for message range (1 to 13): 13

Should the system read any log files (y or n): n

Following is an example for running the tuning program with topology information. In this example, the *MPI\_Alltoall* routine for all message sizes is tuned on an 8-node cluster whose topology is described in file topo2sw.txt.

```
/home/example/STAGE-MPI/tuning> tune
```

Enter the number of processors in the system: 8  
 Do you have topology information (y or n): y  
 Enter platform (ethernet, infiniband, ..etc) you are running on: ethernet  
 Please enter topology information file name: topo2sw.txt  
 0) alltoall  
 1) allgather  
 2) allreduce  
 3) bcast  
 4) alltoallv  
 5) allgatherv  
 6) gather  
 7) scatter  
 8) reduce  
 9) gatherv  
 10) scatterv  
 Select a number from above to tune an MPI routine (-1 to tune all): 0  
  
 0) mpptest-barrier  
 1) mpptest  
 2) ol  
  
 Please pick from the above timing mechanisms: 1  
 Do you want to tune for a specific message size (y or n): n  
 Enter message range, there are 14 message sizes points:  
 0: 0  
 1: 1  
 2: 64  
 3: 256  
 4: 1K  
 5: 2K  
 6: 4K  
 7: 8K  
 8: 16K  
 9: 32K  
 10: 64K  
 11: 128K  
 12: 256K  
 13: ( > 256K) INFINITY  
  
 Enter lower bound for message range (0 to 12): 0  
 Enter upper bound for message range (1 to 13): 13  
 Should the system read any log files (y or n): n

After the tuning, all related files including the final routines and the routines selected by the final routines, and test drivers are placed in `${STAGE-MPI}/tuning/TUNED`. For example, if *MPI\_Alltoall* is tuned, the file `alltoall-tuned.c` contains the tuned routine and `alltoall-tuned-driver.c` is an example driver program that uses the tuned all-to-all routine. Note that when the user tunes for a specific message size or range, then the driver should also be given the same message size or a size within the same range that was tuned. The following command can be used to run the example driver, which gives the performance information about the tuned routine.

```
/home/example/STAGE-MPI/tuning> cd TUNED
.../STAGE-MPI/tuning/TUNED> mpicc -lm alltoall-tuned-driver.c -I. -DVERBOSE
.../STAGE-MPI/tuning/TUNED> mpiexec -machinefile ../hostfile -n 8 a.out 10 100
```

## 4 Input file formats

We will describe the formats of the topology file and the message size pattern file. Some examples of the files are given in the `${STAGE-MPI}/examples` directory. Note that if the cluster of workstations is connected by a single switch, then it does not make a difference if users have topology information or not.

### 4.1 Format of the topology file

The format of the topology file is related to but slightly different from the hostfile used by `boot-mpich`. The file first specifies how each node is connected to a switch. Each of this is specified by a node-switch connection line as follows.

```
hostname switchname
```

The *hostname* is the real name of the node and the *switchname* is switch name you assign to the switches. **The switch name has the format *s#* (e.g. *s0*, *s1*, etc), and must start from 's0' and be continuous.** For example, the topology cannot have switches *s0*, *s1*, and *s4*. In this case, *s4* must be renamed to *s2*. The machine order of these lines must be exactly the same as the the the machine order in the hostfile. For example, assume that the content of the hostfile is:

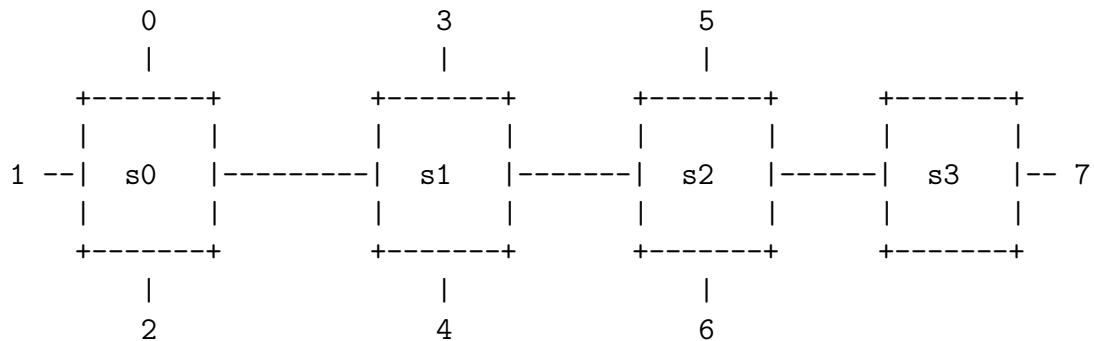
```
beowulf0
beowulf1
beowulf2
beowulf3
beowulf4
beowulf5
beowulf6
beowulf7
```

The node-switch connection lines in the topology file may look like:

```
beowulf0 s0
beowulf1 s0
beowulf2 s0
beowulf3 s1
beowulf4 s1
beowulf5 s2
beowulf6 s2
beowulf7 s3
```

Notice the exact match of the sequence of machine names in the two files. After the node-switch connection lines, a line that only contains only dashes '-----' is followed. After that, the switch-switch connection lines are specified. All the connections are bidirectional and each connection should only be specified once. Also the topology must be a tree (as Ethernet topology is always a tree). Following is a topology and its corresponding file.

Topology:

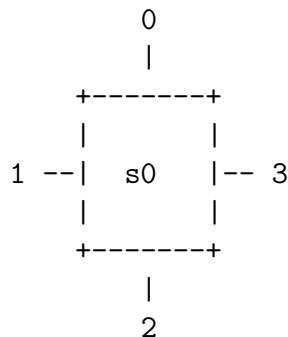


Topology file:

```
beowulf0 s0
beowulf1 s0
beowulf2 s0
beowulf3 s1
beowulf4 s1
beowulf5 s2
beowulf6 s2
beowulf7 s3
-----
s1 s0
s1 s2
s2 s3
```



An example topology that involves 4 machines connected to a single switch is shown below. Notice that the dashes are still needed even in the one switch case.



Topology file:

```

beowulf0 s0
beowulf1 s0
beowulf2 s0
beowulf3 s0
-----

```

## 4.2 Format of the message size pattern file

When the user selects to tune v-version MPI routines such as alltoallv, allgatherv, gatherv, and scatternv, files describing the message size patterns are needed. The files give the send counts, send displacements, receive counts, and receive displacements arrays in the corresponding MPI routine. One way to get these arrays is to profile a particular application.

The system, however, has a corresponding **size-pattern generator** for each of these collectives, which is included in the **bin** directory. For example, assume we want to tune alltoallv. Issuing:

```
/home/example/STAGE-MPI/tuning> generate-alltoallv-size-pattern 4 1
```

generates a size-pattern file that includes 1 size patterns on 4 nodes (the generated file is named “alltoallv-4-nodes-size-pattern.txt”). Users can generate more size patterns by changing the second parameter on the command line from 1 to 12. The format of that file is as follows:

```

total patterns: 1
64 64 64 64
0 64 128 192
64 64 64 64

```

```

0 64 128 192
64 64 64 64
0 64 128 192
64 64 64 64
0 64 128 192
64 64 64 64
0 64 128 192
64 64 64 64
0 64 128 192
64 64 64 64
0 64 128 192
64 64 64 64

```

The first four rows correspond to the send counts, send displacements, receive counts, and receive displacements arrays for the first node. The next four rows are assigned for the second node, and so forth. If users profile their applications, the output of their profiler must match the one included in these files.

The following shows the format for the MPI\_Allgatherv size pattern for four nodes. The first line is the send count for all processors (in this case, each node sends a count = 1). The second two lines are the recv counts and displacements arrays, respectively, for the first processor. The second two lines are the recv counts and displacements arrays for the second processor, and so forth.

```

1 1 1 1
1 1 1 1
0 1 2 3
1 1 1 1
0 1 2 3
1 1 1 1
0 1 2 3
1 1 1 1
0 1 2 3

```

The format for the MPI\_Gatherv is shown next on four processors. The first line represents the send count for each of the four processors. The remaining two lines are the recv counts array and the displacements array for the root processor.

```

1 1 1 1
1 1 1 1
0 1 2 3

```

Finally, the MPI\_Scatterv format for four processors is shown below. The first two lines are the send counts array and the displacements array for the root processor, and the last

line is the recv count for each processor.

```
1 1 1 1
0 1 2 3
1 1 1 1
```

## 5 Timing mechanisms

STAGE-MPI comes with different timing mechanisms to measure the performance of the individual communication algorithms. Users are given the choice of selecting one of the three following mechanisms: `mpptest`, `mpptest-barrier`, and `ol` (overhead latency). In all mechanisms, the MPI collective routine is executed a number of iterations. The following shows a code segment of the `mpptest`:

```
MPI_Barrier(...)
start = MPI_Wtime();
For (i = 0; i < ITER; i++) {
    MPI_Alltoall(...);
}
elapsed = MPI_Wtime() - start;
```

The `mpptest-barrier` is similar to the `mpptest` mechanism except that a barrier synchronization is inserted in each iteration before the MPI collective routine to eliminate the pipelining effect. In both `mpptest` and `mpptest-barrier`, the per iteration elapsed time is reported at the end as the performance measurement. The following shows a code segment of the `mpptest-barrier`:

```
MPI_Barrier(...)
start = MPI_Wtime();
For (i = 0; i < ITER; i++) {
    MPI_Barrier(...);
    MPI_Alltoall(...);
}
elapsed = MPI_Wtime() - start;
```

Figure 1 describes the `ol` timing mechanism. This method tries to measure the collective operation latency indirectly by measuring the operation latency for individual nodes and taking the maximum of these measurements as a reasonable approximation of the entire operation latency. The algorithm first computes the average latency of acknowledgments sent from a node  $i$  to the root, which is simply equal to the overhead of a single point-to-point message from  $i$  to the root. In the second step, the root does not start the collective

operation until it receives an acknowledgment from current node  $i$ , which avoids the pipeline effect. This step repeats some  $M$  number of iterations where the root averages the elapsed time  $E_i$  over  $M$ . Then, the  $OL_i$  is computed by simply subtracting the acknowledgment overhead from the elapsed time for a node  $i$ . Finally, the maximum of  $OL_i$ ,  $0 \leq i < P$  is reported.

Root (node 0):	Current node $i$ :	All other nodes:
$t_1 = \text{MPI\_Wtime}()$	for $x = 1 \dots M$	$\text{MPI\_Barrier}$
for $x = 1 \dots M$	$\text{MPI\_Recv}$ from root	for $x = 1 \dots M + 1$
$\text{MPI\_Send}$ to $i$	$\text{MPI\_Send}$ to root	$\text{MPI\_Bcast}$
$\text{MPI\_Recv}$ from $i$	$\text{MPI\_Barrier}$	
$t_2 = \text{MPI\_Wtime}()$	for $x = 1 \dots M + 1$	
$\text{RTL}_i = (t_2 - t_1) / M$	$\text{MPI\_Bcast}$	
$\text{MPI\_Barrier}$	$\text{MPI\_Send ACK}$ to Root	
$\text{MPI\_Bcast}$		
$\text{MPI\_Recv ACK}$ from $i$		
$t_1 = \text{MPI\_Wtime}()$		
for $x = 1 \dots M$		
$\text{MPI\_Bcast}$		
$\text{MPI\_Recv ACK}$ from $i$		
$t_2 = \text{MPI\_Wtime}()$		
$E_i = (t_2 - t_1) / M$		
$OL_i = E_i - (\text{RTL}_i / 2)$		
Report $\max(OL_i)$		

Figure 1: OL timing mechanism

## 6 Trouble-shooting

### TUNING SESSION HANGS:

In some cases, users may experience that some tuning runs are taking a much longer time than usual. This may indicate that the system is hanging. It is not clear to us what is causing the system to hang at this time. We found that the MPICH library itself may hang in some cases. We suspect that this problem may be caused by the underlying point-to-point communication system.

#### FIX:

The first step is to kill the tuning process, and we urge NOT to use CTRL-C. To do this, issue:

```
/home/example/STAGE-MPI/tuning> killall tune
```

In the second step, isolate the machine that is causing the hanging behavior. This can be done by pinging or ssh-ing all the machines in the hostfile. The one that cannot be reached must be rebooted. Note that sometimes more than a machine is hanging, and thus, more machines will need to be rebooted. Finally, kill all the current MPDs running on the remaining nodes, and issue the “boot-mpich” command again.

**REMINDER:**

If for some reason the system hangs or users had to abort their tuning session earlier, there is no need to restart the tuning run or session from scratch. The STAGE-MPI has a logging capability that makes the tuning system pick up the tuning process at the last point just before it was stopped.

## 7 Contacting the developers

Please contact Ahmad Faraj ([faraj@cs.fsu.edu](mailto:faraj@cs.fsu.edu)) or Xin Yuan ([xyuan@cs.fsu.edu](mailto:xyuan@cs.fsu.edu)) for problems with the system.

## 8 COPYRIGHTS

Copyright (c) 2006, Ahmad Faraj & Xin Yuan,  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Florida State University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*\*\*\*\*  
Any results obtained from executing this software require the acknowledgment and citation of the software and its owners. The full citation is given below:

A. Faraj and X. Yuan. "Automatic Generation and Tuning of MPI Collective Communication Routines." The 19th ACM International Conference on Supercomputing (ICS), Cambridge, Massachusetts, June 20-22, 2005.

\*\*\*\*\*