μπ

A Scalable and Efficient Performance Investigation Simulator for Parallel Applications

Kalyan S. Perumalla, Ph.D.

Oak Ridge National Laboratory

Feb 11, 2009

(Last Updated 6/9/2009 2:32 AM)

Table of Contents

Overview ... 3
1. Introduction ... 3
1.1 Motivation ... 4
1.2 Related Work ... 4
2. Design ... 4
2.1 Grafting ... 4
2.1.1 Source Code-Based Grafting ... 5
2.1.1.1 Source Code- Grafting Usage ... 5
2.1.1.2 Example: C/C++ Program using C Interface of MPI ... 5
2.1.1.3 Example: FORTRAN Program using MPI FORTRAN Interface ... 6
2.1.1.4 Source Code Grafting Implementation ... 6
2.1.1.5 MPI Routines Implemented ... 6
2.1.2 Library Redirection-Based Grafting ... 7
2.1.2.1 Library Grafting Usage ... 7
2.1.2.2 Library Grafting Implementation ... 7
2.1.3 Virtual Machine-Based Grafting ... 7
2.1.3.1 Virtual Machine Grafting Usage ... 7
2.1.3.2 Virtual Machine Grafting Implementation ... 7
2.1.4 Machine Characteristics Customization ... 7
2.2 Virtual Time and Real Wall-Clock Time ... 7
2.3 Virtual Communication, Simulation Events and Real Communication ... 7
3. Implementation ... 7
3.1 Timing Model ... 7
4. Usage ... 8
5. Examples ... 8
6. Experiments ... 8
7. Software ... 8
7.1 Obtaining the Software ... 8
7.2 Installation ... 8
7.3 Simple Example: MPIPING ... 9
7.4 Execution ... 11
7.4.1 How to Run ... 11
7.4.1.1 Verbosity and Debugging ... 11
7.4.1.2 Command-line Arguments ... 12
7.4.1.3 Standard Output ... 12
7.4.2 Example ... 12
7.4.2.1 MPIPING without μπ ... 12
7.4.2.2 MPIPING with μπ ... 13
7.5 Compiling and Running your MPI Application ... 15
7.6 Customization ... 15
8. Summary ... 15
9. References ... 15

μπ

Kalyan S. Perumalla
Oak Ridge National Laboratory
Feb 11, 2009
(Last Updated 6/9/2009 2:32 AM)

Overview

μπ is a highly scalable, transparent system for predicting the performance of parallel programs. Performance can be investigated by executing actual applications on customized configurations of hypothetical parallel machines. The level of detail can be varied for application behavior as well as machine characteristics. Unique features of μπ are repeatability of execution, scalability to very large number of processing elements, portability of the system to a wide variety of platforms, and ease of experimentation with scientific applications with source-code. The design of μπ supports experimentation using thousands of real cores to predict the performance of applications using several fold as many virtual cores as real cores. Low slowdowns can be expected, due to the scalability and efficiency of the underlying simulator, μsik.

1. Introduction

μπ is designed to solve the following problem: Given a (real) parallel application A_α, predict how it would perform when executed on a (real or imaginary) machine M_α. The prediction system, which is a (real) parallel application A_β in its own right, is to be executed on another (real) machine M_β in an as-fast-as-possible manner.

Typical performance metrics of primary interest include computation time, blocked (wasted) time, memory consumption, and network load. Additional performance items of interest include software engineering concerns (e.g., uncovering software shortcomings or other unknown behavior when the application is executed on a larger number of processors than feasible before).

Ideally, the prediction should be (a) repeatable, (b) accurate, and (c) fast, in that order of preference. However, when the complexity of the application A_α and/or the machine M_α increases, it becomes a harder endeavor to satisfy all the preceding three criteria at once. In particular, when the number of processing elements in M_α is on the order of 10^5-10^6 (comparable to existing and emerging peta-scale and exa-scale platforms), a specially designed scalable and efficient prediction system is needed to meet the goals. μπ is designed to fill this need.

Here, we document the design, development and usage of μπ for large-scale performance prediction of complex parallel applications such as Message Passing Interface (MPI)-based parallel scientific codes.

The name μπ is a Greek abbreviation for the English acronym MUPI for **mi**cro parallel **p**erformance **i**nvestigator.

1.1 Motivation

Parallel processing complexity analysis alone is inadequate to deal with the current trends in parallel systems due to increasing scale and widening diversity of hardware platforms, middleware systems, software interfaces, and parallel algorithms. An environment for debugging, testing and customizing existing applications to new parallel hardware and software platforms would result in significantly better utilization of many resources, including time and money. Ideally, an existing application would be minimally modified and executed on a newly envisioned parallel system before that new system is built. Not only can the application can benefit from the performance insights, but also the system designers can use the observed behavior to improve the system design.

1.2 Related Work

There have been several past and ongoing efforts for performance prediction. μπ is focused on experimentation with very large (virtual) parallel computing configurations, especially with hundreds of thousands of (virtual) processor cores. In particular, of specific interest is the ability to sustain experiments with MPI-based programs with 10^5-10^6 MPI tasks within a single experiment of performance prediction. We are not aware of any systems that are able to sustain experiments at such a high level of scale. Additionally, at the higher levels of detail of MPI program execution, parallel execution of the prediction system itself becomes imperative, for which μπ is uniquely equipped for large-scale parallel execution of the *simulation*. This enables μπ to complete the prediction runs within reasonable turnaround times.

2. Design

Recall that the subject application A_α is to be virtually executed on some P_α processors of the imaginary (guest) machine M_α, but needs to execute on (typically much smaller) number of processors P_β of a real (host) machine M_β.

For example, if the application is to be experimented on 100,000 processor cores, but only 10,000 cores are available for experimentation, then the application must be somehow "fooled" into the view that it is executing on 100,000 cores. Since more than one virtual core is "simulated" on each real core, timing results perceived by the application have to be adjusted to erase the effects of multiplexing. Message buffering and re-ordering have to be similarly accurately recreated as they would have occurred in the envisioned machine network, even though messages might arrive faster or slower on the host machine compared to the envisioned machine.

The full design involves three components: (1) grafting (2) time pacing (3) communication buffering and reordering. Each of these components is described next.

2.1 Grafting

The grafting portion of the prediction system deals with the software interface-related mismatch between the guest parallel machine M_α and the host parallel machine M_β. The original software interface (e.g., MPI) that the application uses should be retained, but the implementation must be changed (preferably, transparently) to reflect the behavior of the new machine. We call the redirection as a grafting process. There are multiple grafting methods, each with its own merits and demerits. A source code-based grafting method is

best when the source code of the application is available. A library redirection-based grafting method is best to deal with application functionality that is available only in the form of pre-built object libraries; however, depending on the complexity of the libraries involved, the grafting systems can involve significant amount of systems work. A virtual machine-based grafting is possible for the highest levels of transparency in the grafting process, but also is the most expensive with respect to runtime cost.

2.1.1 Source Code-Based Grafting

In the initial version of μπ, the source code-based grafting method is supported. Its usage and implementation are described here.

2.1.1.1 Source Code- Grafting Usage

When the source code is available for the MPI application, μπ supports grafting with the following steps:

1. In the application source code, the header file **`mupi.h`** is included, right after **`mpi.h`**:
   - **`#include <mpi.h>`**
     `#include <mupi.h>`
   - In fact, instead of an additional include, the original **`#include <mpi.h>`** can simply be changed to **`#include <mupi.h>`**, since **`mupi.h`** already includes **`mpi.h`**.
   - The MUPI include directory is added to the compiler flags
2. The MUPI libraries are added to the list of libraries to which the application is linked.
   - **`-Lmupi/lib -lmupi -lmusik -lsynk`**
3. The application is compiled and linked as usual. The resulting executable will be a μπ simulation, which executes in virtual mode.
   - On platforms on which μπ itself executes over MPI:
     `mpirun –np 4 myprog -nvp 32`
     runs myprog on 32 virtual cores, simulated by μπ on 4 real cores

2.1.1.2 Example: C/C++ Program using C Interface of MPI

To compile **`mpiping.cpp`** (included in **`mupi/examples`**) to use μπ, the following command lines must be executed:

```
% mpicxx -o mpiping.o -DMUPI=1 -I../include -c mpiping.cpp
```

```
% mpicxx -o mpiping mpiping.o -L../lib -lmupi -lmusik -lsynk
```

Note the definition of the macro variable **`MUPI`** on the command-line, to enable the inclusion of **`mupi.h`** which is conditionally included in the **`mpiping.cpp`** source. Note also the specification of μπ's header file directory **`mupi/include`** to find the **`mupi.h`** and its related header files. The μπ libraries are located in **`mupi/lib`**, which is specified by the **`-L`** option.

2.1.1.3 Example: FORTRAN Program using MPI FORTRAN Interface

To compile **mpipingf.f90** (included in **mupi/examples**) to use µπ, the following command lines must be executed:

```
% mpif90 -o mpipingf.o -c mpipingf.f90
```

```
% mpicxx -o mpipingf mpipingf.o -L../lib -lmupi -lmusik -lsynk –lgfortran
```

Note the use of **mpif90** to generate the object file from FORTRAN source, but the use of **mpicxx** for generating the executable. The object file is linked to µπ using **mpicxx**; doing so makes it easier to link, since µπ is a C/C++ library. For the same reason, the FORTRAN library must be explicitly specified to resolve FORTRAN intrinsic routines in the object file.

Depending on your specific platform, the exact same command lines may not work as specified, but the scheme can be employed to customize the compilation and linkage process based on this basic approach.

2.1.1.4 Source Code Grafting Implementation

The **mupi.h** header file captures three items of functionality required for source code-based grafting:

1. The original MPI routines are redefined, via macro substitution, to µπ calls of correspondingly identical signatures. The effect of this substitution is that, wherever the application invokes an MPI routine, a corresponding peer routine of µπ is invoked instead, thus giving µπ the control in catering to that call.
    - Example: **#define MPI_Init(ac,av) MUPI_Init(ac,av)**
2. The µπ routine signatures for the redirected calls of item 1 are declared.
3. The **main()** routine of the application is redefined, via macro substitution, to **MUPI_main()**.

The **MUPI_main()** routine will be automatically invoked by µπ for each virtual MPI task of the application (command line arguments are duplicated for each virtual task invocation, as expected by the original **main()**). µπ allocates and creates a separate stack context for each virtual MPI task, and maintains the full stack context between MPI calls invoked by the task.

2.1.1.5 MPI Routines Implemented

The MPI routines implemented, in C/C++ and FORTRAN, as of this writing are:

MPI_Init(), **MPI_Finalize()**, **MPI_Comm_rank()**, **MPI_Comm_size()**, **MPI_Barrier()**, **MPI_Send()**, **MPI_Recv()**, **MPI_Isend()**, **MPI_Irecv()**, **MPI_Waitall()**, **MPI_Wtime()**.

Only the **MPI_COMM_WORLD** communication group is currently recognized. No other communicator group operations are implemented.

MPI's C++ interface is not yet implemented.

There are a few other symbols that are trapped, but not yet implemented. They include:

MPI_Allreduce(), **MPI_Reduce()**, **MPI_Alltoall()**, **MPI_Bcast()**.

2.1.2 Library Redirection-Based Grafting

2.1.2.1 Library Grafting Usage

2.1.2.2 Library Grafting Implementation

This method of grafting will be documented in greater detail at a later time.

2.1.3 Virtual Machine-Based Grafting

2.1.3.1 Virtual Machine Grafting Usage

2.1.3.2 Virtual Machine Grafting Implementation

This method of grafting will be documented in greater detail at a later time.

2.1.4 Machine Characteristics Customization

The characteristics of the envisioned machine can be specified programmatically, as an object library linked to the μπ simulation.

As a short-term solution, a very simple machine performance model is provided, for expedience (the μπ design allows for far more sophisticated machine models to be added later): a point-to-point bandwidth and a point-to-point latency.

These two values can be specified via two environment variables: `MUPI_BANDWIDTH_BPS` and `MUPI_LATENCY_SEC`. The former is in bits per second (e.g., specify `1e9` for 1Gbps), and the latter is in seconds (e.g., specify `1e-6` for 1 microsecond).

The mpiping example shows the effect of varying the values of these variables. This feature can be easily experimented by executing the mpiping example with μπ with different settings of the bandwidth and latency environment variables.

2.2 Virtual Time and Real Wall-Clock Time

TBC

2.3 Virtual Communication, Simulation Events and Real Communication

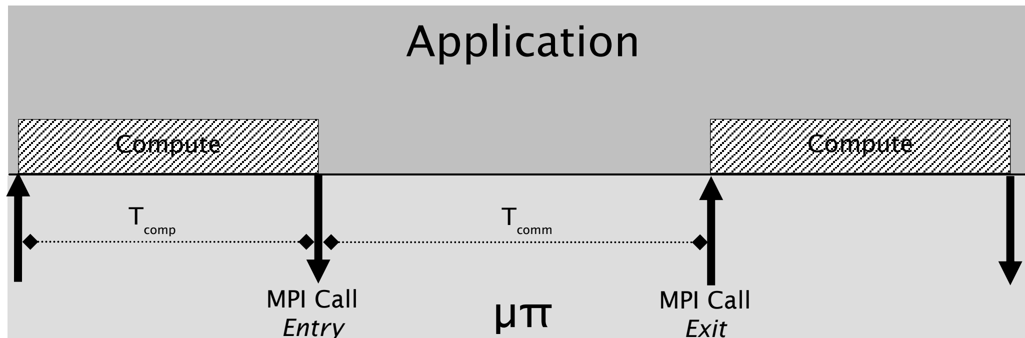
TBC

3. Implementation

TBC

3.1 Timing Model

TBC



TBC

4. Usage

TBC

5. Examples

TBC

6. Experiments

TBC

7. Software

The μπ software is written C/C++, as an application of the μsik parallel discrete event simulation system. μπ and μsik are both portable to a large number of platforms. μsik works on most platforms, from palmtops to supercomputers. μπ has been tested on MPI-based platforms, including Linux, Mac OS X and Cray XT4 and Cray XT5. As of this writing, μsik has been shown to scale to nearly 150,000 cores.

7.1 Obtaining the Software

The μπ homepage includes instructions for downloading the μπ package:
    www.ornl.gov/~2ip/mupi/index.htm .

The most recent version of μπ will be named **mupi-current-*arch*.tar.gz**, where ***arch*** is **linux** for Linux, **macosx** for Mac OS X, **crayxt5** for Cray XT5, and **bgp** for Blue Gene P.

7.2 Installation

Execute the following steps to install μsik and μπ in your home directory:

1. **tar -zxvf mupi-current-*arch*.tar.gz**
   This will create a directory named **mupi**.

2. **cd mupi/examples**
   Change directory to the subdirectory named **mupi/examples**.

3. Customize **mupi/examples/Makefile** to suit your MPI compiler and flags. For example, set the compiler to **mpicxx** on Mac OS X.

4. **make**
   This will build μπ examples.

7.3 Simple Example: MPIPING

A simple **mpiping** example is included in the **mpiping.cpp** file in the **mupi/examples** directory. It is intended to illustrate the power of repeatability, ease of customization of machine characteristics, and the use of actual MPI-based source code of the application used in performance prediction/experimentation.

The example uses **MPI_Send** and **MPI_Recv** to exchange messages in a ring of MPI tasks. The message size is increased by doubling from 8 bytes to 1MB.

Terminal — vim — ttys000 — 80×65 — ⌘1

```c
/*---------------------------------------------------------------------------*/
#include <mpi.h>
#include "mupi.h"

#include <stdio.h>
#include <stdlib.h>
/*---------------------------------------------------------------------------*/
int main( int argc, char *argv[] )
{
    int rank = 0, size = 0, other_rank = 0, msg_size = 0;
    double t1, t2, microsecs;
    double *a, *b;
    MPI_Status status;

    a = (double *) malloc(132000 * sizeof (double));
    b = (double *) malloc(132000 * sizeof (double));

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    other_rank = (rank + 1) % size;

    printf("Hello from %d of %d\n", rank, size); fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);

    for( msg_size = 8;  msg_size <= 1048576;  msg_size *= 2 )
    {
        int i = 0;
        for( i = 0; i < msg_size / 8; i++ ) { a[i] = (double)i; b[i] = 0.0; }

        MPI_Barrier(MPI_COMM_WORLD);
        t1 = MPI_Wtime();
        if( rank % 2 == 0 ) {
            MPI_Send(a, msg_size/8, MPI_DOUBLE, other_rank, 0, MPI_COMM_WORLD);
            MPI_Recv(b, msg_size/8, MPI_DOUBLE, other_rank, 0, MPI_COMM_WORLD,
                     &status);
        } else {
            MPI_Recv(b, msg_size/8, MPI_DOUBLE, other_rank, 0, MPI_COMM_WORLD,
                     &status);
            MPI_Send(a, msg_size/8, MPI_DOUBLE, other_rank, 0, MPI_COMM_WORLD);
        }
        t2 = MPI_Wtime();
        microsecs = 1.e6 * (t2 - t1);
        MPI_Barrier(MPI_COMM_WORLD);

        #define ACCURACY 0.1
        if( microsecs >= ACCURACY ) {
            printf(" %7d bytes took %9.0f usec (%8.3f MB/sec)\n",
                   msg_size, microsecs, 2.0 * msg_size / microsecs);
        } else {
            printf(" %7d bytes took less than %d usec\n", msg_size, ACCURACY);
        }
        fflush(stdout);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}

/*---------------------------------------------------------------------------*/
~
~
```

7.4 Execution

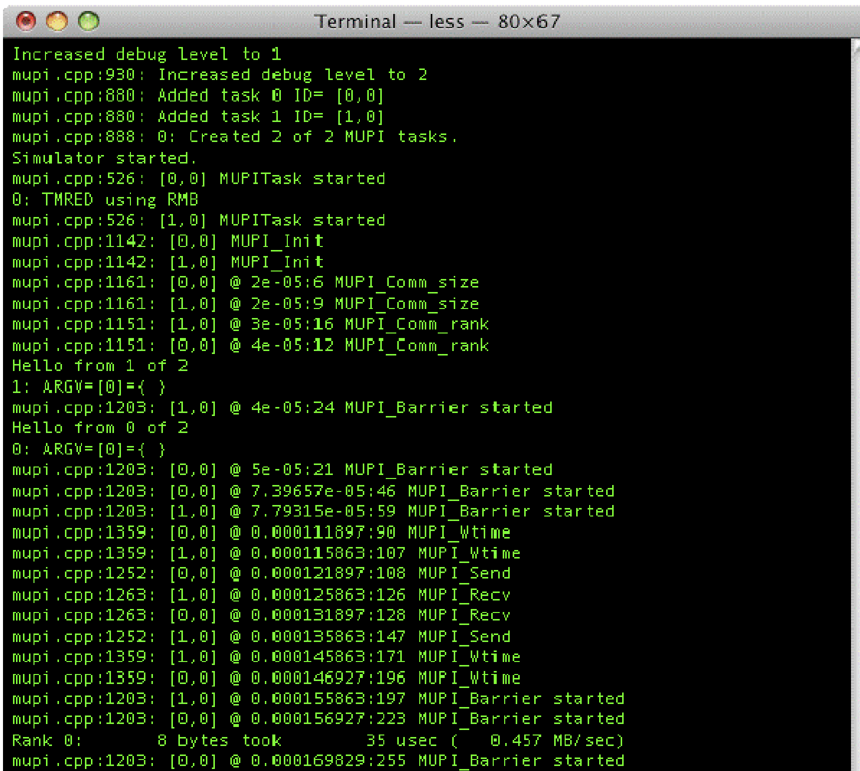
7.4.1 How to Run

A μπ simulation is itself an MPI application, and thus can be spawned as any MPI application, such as using **`mpirun`**. The **`-np`** argument of **`mpirun`** applies to the simulation, not the application. The number of virtual MPI ranks is specified, instead, using the **`-nvp`** argument of the simulation. Thus, **`mpirun -np 2 myapp -nvp 8`** will simulate 8 virtual MPI ranks, and uses 2 cores for the simulation.

7.4.1.1 Verbosity and Debugging

μπ simulations can be made to generate less or more verbose output of their MPI activity and simulation progress, by using the command-line options to μπ, +v, -v, +V and -V. The options with the '+' sign increases the output level while the ones with '-' decrease the output level. The small case increments/decrements by unity, while the capital case increments/decrements by 10. Mutiple v's can follow the sign (e.g., +vv), and the option can be used more than once (e.g., +VV +v).

The default level is 0. Negative levels are allowed (e.g., -v makes the level -1, which makes μπ very silent). A level of 2 (+vv) generates a trace of all MPI calls that have been trapped and simulated by μπ. For example, executing **`mpirun -np 1 mpiping -nvp 2 +vv`** will generate a trace in mupi-0.txt as follows:



```
Terminal — less — 80×67
Increased debug level to 1
mupi.cpp:930: Increased debug level to 2
mupi.cpp:880: Added task 0 ID= [0,0]
mupi.cpp:880: Added task 1 ID= [1,0]
mupi.cpp:888: 0: Created 2 of 2 MUPI tasks.
Simulator started.
mupi.cpp:526: [0,0] MUPITask started
0: TMRED using RMB
mupi.cpp:526: [1,0] MUPITask started
mupi.cpp:1142: [0,0] MUPI_Init
mupi.cpp:1142: [1,0] MUPI_Init
mupi.cpp:1161: [0,0] @ 2e-05:6 MUPI_Comm_size
mupi.cpp:1161: [1,0] @ 2e-05:9 MUPI_Comm_size
mupi.cpp:1151: [1,0] @ 3e-05:16 MUPI_Comm_rank
mupi.cpp:1151: [0,0] @ 4e-05:12 MUPI_Comm_rank
Hello from 1 of 2
1: ARGV=[0]={ }
mupi.cpp:1203: [1,0] @ 4e-05:24 MUPI_Barrier started
Hello from 0 of 2
0: ARGV=[0]={ }
mupi.cpp:1203: [0,0] @ 5e-05:21 MUPI_Barrier started
mupi.cpp:1203: [0,0] @ 7.39657e-05:46 MUPI_Barrier started
mupi.cpp:1203: [1,0] @ 7.79315e-05:59 MUPI_Barrier started
mupi.cpp:1359: [0,0] @ 0.000111897:90 MUPI_Wtime
mupi.cpp:1359: [1,0] @ 0.000115863:107 MUPI_Wtime
mupi.cpp:1252: [0,0] @ 0.000121897:108 MUPI_Send
mupi.cpp:1263: [1,0] @ 0.000125863:126 MUPI_Recv
mupi.cpp:1263: [0,0] @ 0.000131897:128 MUPI_Recv
mupi.cpp:1252: [1,0] @ 0.000135863:147 MUPI_Send
mupi.cpp:1359: [1,0] @ 0.000145863:171 MUPI_Wtime
mupi.cpp:1359: [0,0] @ 0.000146927:196 MUPI_Wtime
mupi.cpp:1203: [1,0] @ 0.000155863:197 MUPI_Barrier started
mupi.cpp:1203: [0,0] @ 0.000156927:223 MUPI_Barrier started
Rank 0:        8 bytes took        35 usec (   0.457 MB/sec)
mupi.cpp:1203: [0,0] @ 0.000169829:255 MUPI_Barrier started
```

7.4.1.2 Command-line Arguments

Command-line arguments to the virtual MPI ranks can be specified between two tokens of **–appargs** as in this example:

```
mpirun -np 2 ./mpiping -nvp 2 –appargs Hello World –appargs
```

This will pass two arguments, "Hello" and "World," to every virtual MPI task simulated by μπ.

<u>Note</u>: As of the μπ version of June 8, 2009, command-line arguments are fully operational for C/C++ applications (*i.e.*, argument vector to `main()` of the application will correctly receive the options specified between `-appargs`), but are not yet implemented for FORTRAN source codes (*i.e.*, `COMMAND_ARGUMENT_COUNT` and `GET_COMMAND_ARGUMENT` do not correctly return the `-appargs` values).

7.4.1.3 Standard Output

When "`mpirun -np S myapp -nvp V`" is executed, μπ creates S output files, named **mupi-*k*.txt**, where $0 \leq k < S$. The standard output of virtual MPI tasks mapped to a real simulator MPI task is redirected to its corresponding output file.

Thus, given "`mpirun –np 2 myapp -nvp 8`", the standard output of the virtual MPI ranks 0-3 appears in `mupi-0.txt`, and those of virtual ranks 4-7 appears in `mupi-1.txt`.

7.4.2 Example

7.4.2.1 MPIPING without μπ

To run the `mpiping` example without μπ, execute the following:

1. **cd mupi/examples**
2. Compile as usual:
   mpicxx –o mpiping mpiping.cpp
3. Execute as usual, say, using **mpirun**:
   mpirun -np 2 ./mpiping

   This will run 2 tasks of the **mpiping** benchmark natively on the host machine's MPI.

```
Terminal — bash — ttys000 — 80×60 — ⌘1
> mpirun -np 2 ./mpiping
Hello from 0 of 2
       8 bytes took       133 usec (   0.120 MB/sec)
      16 bytes took        55 usec (   0.581 MB/sec)
      32 bytes took        13 usec (   4.971 MB/sec)
      64 bytes took        13 usec (   9.942 MB/sec)
     128 bytes took        15 usec (  17.044 MB/sec)
     256 bytes took        15 usec (  34.087 MB/sec)
     512 bytes took        15 usec (  68.174 MB/sec)
    1024 bytes took        15 usec ( 130.547 MB/sec)
Hello from 1 of 2
       8 bytes took        49 usec (   0.327 MB/sec)
      16 bytes took        51 usec (   0.627 MB/sec)
      32 bytes took        12 usec (   5.369 MB/sec)
      64 bytes took        12 usec (  10.737 MB/sec)
     128 bytes took        14 usec (  18.199 MB/sec)
     256 bytes took        13 usec (  39.045 MB/sec)
     512 bytes took        13 usec (  78.090 MB/sec)
    1024 bytes took        14 usec ( 140.102 MB/sec)
>
```

Observe the bandwidth numbers printed by this run.

```
Terminal — bash — ttys000 — 80×60 — ⌘1
> mpirun -np 2 ./mpiping
Hello from 0 of 2
Hello from 1 of 2
       8 bytes took       677 usec (   0.024 MB/sec)
       8 bytes took        73 usec (   0.219 MB/sec)
      16 bytes took       409 usec (   0.078 MB/sec)
      16 bytes took       402 usec (   0.080 MB/sec)
      32 bytes took       380 usec (   0.168 MB/sec)
      32 bytes took       378 usec (   0.169 MB/sec)
      64 bytes took       381 usec (   0.336 MB/sec)
      64 bytes took       378 usec (   0.339 MB/sec)
     128 bytes took       371 usec (   0.690 MB/sec)
     128 bytes took       342 usec (   0.748 MB/sec)
     256 bytes took       386 usec (   1.326 MB/sec)
     256 bytes took       414 usec (   1.236 MB/sec)
     512 bytes took       326 usec (   3.142 MB/sec)
     512 bytes took       312 usec (   3.281 MB/sec)
    1024 bytes took       348 usec (   5.884 MB/sec)
    1024 bytes took       339 usec (   6.045 MB/sec)
>
```

Observe also that the numbers printed by repeated runs will be different across runs.

7.4.2.2 MPIPING with μπ

The same application can be simulated by μπ as illustrated below:

1. Compile with μπ, as described in Section 2.1.1.
2. **`mpirun -np 2 ./mpiping -nvp 2`**

   This will run 2 tasks of the **`mpiping`** benchmark on a controlled guest machine simulated by μsik using 2 MPI tasks.

   Output will appear in **`mupi-0.txt`**, and **`mupi-1.txt`**.

```
> mpirun -np 2 ./mupiping -nvp 2
FMMPI_nodeid=0, FMMPI_numnodes=2 FMMPI_FLCTLK=0
0: Redirecting stdout to "mupi-0.txt"
> grep took mupi-0.txt
       8 bytes took          0 usec (  80.000 MB/sec)
      16 bytes took          0 usec (  80.000 MB/sec)
      32 bytes took          1 usec (  80.000 MB/sec)
      64 bytes took          2 usec (  80.000 MB/sec)
     128 bytes took          3 usec (  80.000 MB/sec)
     256 bytes took          6 usec (  80.000 MB/sec)
     512 bytes took         13 usec (  80.000 MB/sec)
    1024 bytes took         26 usec (  80.000 MB/sec)
> grep took mupi-1.txt
       8 bytes took          0 usec (  80.000 MB/sec)
      16 bytes took          0 usec (  80.000 MB/sec)
      32 bytes took          1 usec (  80.000 MB/sec)
      64 bytes took          2 usec (  80.000 MB/sec)
     128 bytes took          3 usec (  80.000 MB/sec)
     256 bytes took          6 usec (  80.000 MB/sec)
     512 bytes took         13 usec (  80.000 MB/sec)
    1024 bytes took         26 usec (  80.000 MB/sec)
>
```

Observe the bandwidth numbers printed by this run. These numbers will be controlled by the virtual machine specification, and machine characteristics such as bandwidth and latency can be easily changed.

Observe also that repeated execution of the same benchmark will give the same bandwidth results across executions, thus demonstrating repeatability.

3. **`mpirun -np 1 ./mpiping -nvp 2`**

   This will run 2 tasks of the **`mpiping`** benchmark on a controlled guest machine simulated by μπ using 1 MPI tasks.

   Output will appear in **`mupi-0.txt`**.

```
> mpirun -np 1 ./mupiping -nvp 2
FMMPI_nodeid=0, FMMPI_numnodes=1 FMMPI_FLCTLK=0
0: Redirecting stdout to "mupi-0.txt"
> grep took mupi-0.txt
       8 bytes took          0 usec (  80.000 MB/sec)
       8 bytes took          0 usec (  80.000 MB/sec)
      16 bytes took          0 usec (  80.000 MB/sec)
      16 bytes took          0 usec (  80.000 MB/sec)
      32 bytes took          1 usec (  80.000 MB/sec)
      32 bytes took          1 usec (  80.000 MB/sec)
      64 bytes took          2 usec (  80.000 MB/sec)
      64 bytes took          2 usec (  80.000 MB/sec)
     128 bytes took          3 usec (  80.000 MB/sec)
     128 bytes took          3 usec (  80.000 MB/sec)
     256 bytes took          6 usec (  80.000 MB/sec)
     256 bytes took          6 usec (  80.000 MB/sec)
     512 bytes took         13 usec (  80.000 MB/sec)
     512 bytes took         13 usec (  80.000 MB/sec)
    1024 bytes took         26 usec (  80.000 MB/sec)
    1024 bytes took         26 usec (  80.000 MB/sec)
>
```

Observe that, even though the host machine uses fewer tasks for the simulation, the results printed by the benchmark do not change from step 2, just as desired,

demonstrating the ability to use fewer (one) cores/MPI-tasks for prediction than the (two) cores/MPI-tasks in the envisioned machine.

Again, the observed bandwidth and latency can be changed at will, simply by specifying a different set of constants or extrapolation functions using the μπ machine specification.

7.5 Compiling and Running your MPI Application

If you have a simple MPI application you would like to test using μπ, it can be easily done by copying it into the **`mupi/examples`** folder, and adding it to the list of targets in the **`Makefile`**.

If the application is a single file of source code (e.g., **`myapp.cpp`** or **`myapp.f90`**), then, simply adding **`myapp`** to the list of targets will be sufficient. If it's a multi-file source, the general compilation and linking instructions in the earlier sections can be used.

7.6 Customization

TBC

8. Summary

TBC

9. References

TBC