PAW v1.6

Performance Assessment Workbench User Guide
Document Revision 1.0
September 19, 2016

Carlos Rosales Fernández
carlos@tacc.utexas.edu

High Performance Computing
Texas Advanced Computing Center
The University of Texas at Austin

Abstract

The Performance Assessment Workbench (PAW) is an Open Source suite of microbenchmarks designed with simplicity and reliability in mind. This User Guide documents the implementation details and the general usage for all components of PAW. As of version 1.6 the suite contains code designed to test:

- BLAS floating point performance in CPUs, GPUs and Xeon Phi
- MPI latency and bandwidth, including point to point and collective calls
- Data transfer rates between CPU and GPU
- Data transfer rates between CPU and Xeon Phi coprocessor
- Memory bandwidth, IO rates, OMP overhead (extras)

Each of these capabilities is supported by multiple individual codes. Our objective is that each test provided be completely independent of other tests in the suite, so that it can be executed independently of others and its implementation can be reviewed cleanly and without interference.

Excellent microbenchmarks exist already for memory bandwitdth (STREAM, see [1]), OpenMP overheads (EPCC OMP Benchmark, see [2], and Parallel File System IO (IOR, see [3]). Together with PAW, these benchmarks provide an excellent starting point for baseline system evaluation. For your convenience copies of these have been provided in the `extras` directory.

Throughout the text we have used text boxes to highlight important information. These boxes look like this:

> These gray boxes contain highlighted material for each section and chapter.

Our intention is to create a set of simple benchmark implementations that are transparent to use and understand. The hope is that users of this set of tools will actually look at the code implementation and understand exactly what is being measured, without having to peel off multiple layers of obfuscating code. We have made an effort to write simple, reliable, and thoroughly commented C code, but we are well aware that documentation is never perfect or complete. We welcome all feedback. Feedback that includes suggestions for improvement in the usability, reliability, and accuracy of these benchmarks is particularly welcome.

Version 1.6 is a minor update which adds the ability to build a small set of *core* benchmarks, including an MPI latency test, a multitask MPI bandwidth thest for point-to-point communications, the memory bandwidth using STREAM, and the floating point performance using DGEMM.

Contents

1 Installation

The Performance Analysis Workbench is simple to install. In order to use all of its features you will need GNU Make and a C compiler. In order to test GPU performance you will also need the CUDA compiler and CUBLAS linear algebra libraries. If testing the Xeon Phi you will also need a compiler and with support for offload to the Xeon Phi.

First of all obtain the latest version tarball, currently paw-1.5.1.tar.gz, and expand it in a convenient location in your system:

```
tar xzvf ./paw-1.5.1.tar.gz
```

This will create a top level directory called `paw-1.5.1`, with subdirectories `/bin`, `/doc`, and `/src`. Inside the source directory you will also find separate subdirectories for common functions (`aux`) and the floating point (`/blas`), MPI (`/mpi`), GPU (`/gpu`) and Xeon Phi (`/phi`) benchmarking tools.

Change directory to the top level of PAW and edit `Makefile.in` to reflect your compiler and library locations. A default template is provided that uses the Intel compiler, the Intel MKL BLAS libraries, the Mvapich2 MPI libraries and NVIDIA's nvcc compiler.

A set of benchmarks for CPU only will be generated using the traditional `make && make install`. The complete benchmarking suite can be built using:

```
make all
make all-install
```

You may also choose to build only a subset of test codes:

```
make core && make core-install
make blas && make blas-install
make mpi  && make mpi-install
make mpi3 && make mpi3-install
make gpu  && make gpu-install
make phi  && make phi-install
```

Notice that the MPI-3.0 tests have been separated from the rest of MPI tests because many libraries still do not support these functions. They may eventually be merged with the rest of the MPI tests.

The executable tests will be placed in the `/bin` directory unless a `PREFIX=/install/dir/location/` option is passed to Make in the installation step. For example, to build only the tests relevant to regular CPUs and install the resulting executables in the `/home/me/paw/` directory one would execute:

```
make blas && make blas-install PREFIX=/home/me/paw
make mpi  && make mpi-install PREFIX=/home/me/paw
```

2 Core Tests

A small set of core streamlined core tests have been selected. These tests are not intended to provide highly detailed results but rather to act as sanity checks and tests for maximum performance on a system. The core tests are a single size DGEMM, a latency and bandwidth test for MPI, and the STREAM benchmark. Each of these tests is executed for a single payload size and a minimum number of inner loop iterations to ensure they can be run quickly. Together they will present a fairly good overview of a system maximum performance.

blas_dgemm_single

This is a multithreaded single matrix size DGEMM test. Control the number of threads with the `OMP_NUM_THREADS` environmental variable, and the linear size of the matrix with the variable `MAX_BLAS_SIZE` (integer number of elements, NOT a size in bytes). This test should provide yo with a good indication of the achievable peak floating point performance in your system. For further implementation details read Section 4.

latency

This is an MPI point-to-point latency test. By pinning tasks appropriately one can measure same socket, socket-to socket and node-to-node latencies for the MPI stack. For implementation details read Section 6.

p2p_multitask

This is an MPI point-to-point bandwidth test implemented using isend/irecv/wait calls with a single message size that can be controlled with the environmental variable `MAX_P2P_SIZE`. Some processors have cores which are not powerful enough to saturate MPI bandwidth for large messages when using a single MPI task per node. This test allows to measure unidirectional bandwidth with an arbitrary number of tasks. The recommended use is to start with two nodes and one MPI task per node, and see if increasing the number of tasks per node increases the achieved bandwidth.

stream

The STREAM benchmark, created by John McCalpin, is the industry standard for measuring sustained memory bandwidth. It consists of four tests, described in Table 2.1.

Table 2.1: STREAM benchmark operation definitions

Copy	c[i] = a[i]
Scale	b[i] = scalar*c[i]
Add	c[i] = a[i]+b[i]
Triad	a[i] = b[i]+scalar*c[i]

Each test should be executed for an array size that is larger than four times the available cache size to avoid local caching issues. For more details on the STREAM benchmark see reference [1].

3 General Design

Each benchmark consists of an innermost loop of count `NLOOP` that is used to ensure that sufficient time passes between two consecutive timer calls. An outer loop executes `NREPS` repetitions of the measurement for each workload size, and these runs are used to calculate maximum, minimum, mean, and standard deviation values for the measurement. As the size of the problem increases, the innermost loop iteration count is decreased to avoid excessive runtime, but it is always kept over the safety threshold or over `NLOOP_MIN` iterations, whichever is more conservative.

Each benchmark will execute the following steps:

1. Establish timer overhead
2. Determine minimal loop length for timings, `NLOOP`
   - Run warmup test
   - Time execution of smallest workload with default `NLOOP`
   - Reset `NLOOP` to ensure minimum execution time is 10 times larger than timer overhead
3. Execute test for each workload size
   - Run warmup for this workload
   - Repeat test `NREPS` and store timing for each execution
   - Obtain statistics for test
   - Write partial results to file
   - Reset loop length

By default the values of `NLOOP` and `NREPS` have been set to 1000 and 10 respectively, but one should look carefully at the statistics for each measurement to see if there is a need to increase these.

All constant values controlling iteration and size limits for the microbenchmarks reside in the `src/aux/constants.h` file, and their use is documented with comments. Changing `NREPS` requires recompilation of hte suite, but changing the `NLOOP` iteration limit and the size ranges can be done using environmental variables, as described in Table 3.1.

Table 3.1: Environmental variables controlling iteration number and size ranges

Variable	**Description**	**Default**
NLOOP_MAX	Number of iterations for each timing run	1000
NLOOP_PHI_MAX	Number of iterations for each Phi timing run	100
MIN_BLAS_SIZE	Minimum size for BLAS tests	8
MED_BLAS_SIZE	Warm up size for BLAS tests	1024
MAX_BLAS_SIZE	Maximum size for BLAS tests	10000
MIN_COL_SIZE	Minimum size for MPI collective tests	1
MED_COL_SIZE	Warm up size for MPI collective tests	10000
MAX_COL_SIZE	Maximum size for MPI collective tests	100000
MIN_P2P_SIZE	Minimum size for MPI point to point tests	1
MED_P2P_SIZE	Warm up size for MPI point to point tests	20000
MAX_P2P_SIZE	Maximum size for MPI point to point tests	1000000
WINDOW_SIZE	Window size for MPI message rate tests	128
MIN_GPU_BLAS_SIZE	Minimum size for GPU BLAS tests	8
MED_GPU_BLAS_SIZE	Warm up size for GPU BLAS tests	1024
MAX_GPU_BLAS_SIZE	Maximum size for GPU BLAS tests	9192
MIN_GPU_SIZE	Minimum size for GPU data transfer tests	128
MED_GPU_SIZE	Warm up size for GPU data transfer tests	20000
MAX_GPU_SIZE	Maximum size for GPU data transfer tests	200000000
MIN_PHI_SIZE	Minimum size for Phi data transfer tests	256
MED_PHI_SIZE	Warm up size for Phi data transfer tests	8192
MAX_PHI_SIZE	Maximum size for Phi data transfer tests	614400000

4 Floating Point Performance

Floating point performance is the traditional measure of machine performance for High Performance Computing systems, and represents the overall system capability for performing mathematical operations. Typical benchmarks for overall floating point capability are the Linpack benchmark (HPL, used for the TOP500 list, see references [4] and [5]) and matrix matrix multiplication tests. These algebraic tests have highly optimized implementation with high data reuse and fairly limited memory bandwidth requirements, which are capable of achieving floating point performances very close to the theoretical peak performance of the hardware. Floating point performance is critical for scientific workloads, since they very often rely on solving an algebraic system of equations. Even if a scientific workload is not directly solving a system of linear equations it is likely that it will make extensive use of the floating point capability of the system during tasks like post-processing and data analysis. This makes the determination of the floating point capability of a compute system another essential element in an overall understanding of the system performance.

The tests described in this section are limited to single node executions, since the purpose is to evaluate the processor floating point performance in isolation from network communication and other factors present in distributed execution. Codes are provided to measure the multithreaded performance of the BLAS calls DGEMM, SGEMM, DGEMV and SGEMV using the Intel MKL library or similar.

The code executes each BLAS call within a loop for each matrix size, and the average time is recorded. This loop is repeated `NREPS` times and the minimum, maximum, and mean values are written to file. The number of iterations in the innermost loop, `NLOOP`, is set dynamically. The code ensures the minimum number of inner loop repetitions is sufficient to obtain an accurate timing for the smallest matrix size. This is done by determining the minimum time that must pass between consecutive timer calls for the measurement to be valid, multiplying this by 10, and using that as the minimum acceptable time that the innermost loop should take. As the size of the problem increases the innermost loop iteration number is decreased to avoid excessive runtime, but it is always kept over the safety threshold or over `NLOOP_MIN`, whichever is most conservative. A warmup execution is run without being timed before each message size is tested.

The only exception to this setup is the `blas_dgemm_single` test, which is designed to run a teh DGEMM test for a single matrix of size equal to `MAX_BLAS_SIZE`. This is still run `NREP` times.

The setup costs - memory allocation, array initialization, etc - are not considered as part of the timed region. The number of operations required for the matrix matrix multiplication is estimated as $2N^2(N+1)$ for each size N, and the number of operations required for the matrix vector multiplication is estimated as $2N(N+1)$ for each size N.

Each test will output two files, `XXXXX_time-np_YYYY.dat` and `XXXXX_flops-np_YYYY.dat` with execution timings and floating point operations per second respectively, as well as a brief summary to `stdout`. IN the names the pattern `XXXXX` corresponds to the blas call name (dgemm, sgemm, dgemv, sgemv) and the pattern `YYYY` to the number of threads used in the test. Make sure `OMP_NUM_THREADS` is set to the correct number when you execute these tests, as they rely on this number to write the correct file names.

5 MPI Initialization

MPI initialization times are of importance when scaling execution to a large number of tasks. Typically initialization times depend on the number of nodes involved rather than the total number of tasks, but this depends on the algorithm used for initialization. As the number of nodes involved in the execution increases the time it takes to pass the MPI initialization stage grows quickly, and it may lead to timeouts that prevent execution.

MPI initialization times are measured for both pure MPI (`MPI_Init`) and hybrid (`MPI_Init_thread`) runs. The executables are `col_init` and `col_init_th`, and a driver script called `initest` is provided to automate the tests over a number of node sizes. This script may need to be customized for a specific site, but it is extremely simple. Results are written to `time_init.dat` and `time_init_th.dat`, as well as `time_init_<TASKS>.dat` and `time_init_th_<TASKS>.dat` for every independent run.

6 MPI Latency

MPI latency is of importance to codes that exchange small messages, where latency plays a more significant role than bandwidth. Codes with irregular communication patters typically benefit from short latencies. When looking a latency values off node the limiting factors will be the network interconnect and adapter, while on node the limiting factor will be the memory subsystem, including the socket to socket communication pathways.

To measure MPI latency we use a round trip of an `MPI_Send` / `MPI_Recv` call set. We use a payload of 1 byte only and average the time over `NLOOP` such exchanges – after a non-timed warmup exchange. The test is repeated `NREPS` times, and the results saved to a file called `latency.dat`. A summary of the results is also written to `stdout`.

The latency code can only be executed between two tasks. An error will be thrown and execution aborted if execution with a different number of MPI tasks is attempted.

7 MPI Bandwidth

Many scientific codes require some type of distributed memory model because of their scale and complexity. The Message Passing Interface (MPI) framework allows for communication in distributed systems, and is the de facto standard for codes working in distributed memory systems. Codes that have large messages to exchange depend heavily on the MPI bandwidth supported by the system for their performance, especially if they execute mainly blocking calls or collective operations. Understanding the achievable MPI bandwidth in a system is critical for estimating scalability of distributed codes.

The code executes each MPI call (in a ping-pong setup for point to point calls) within a loop of size `NLOOP` for each message size, and the average time is recorded. This loop is repeated `NREPS` times and the minimum, maximum, mean and standard deviation values are written to file. A warmup exchange is run without being timed before each message size is tested.

Each MPI task times the exchange, and an `MPI_reduce` call is used to determine the minimum of the measured times, which is used to calculate the best average time per iteration in the innermost loop. Note that this can cause inaccuracy in the measurements of some collective calls for small messages, as discussed in Section 7.1.

It is important to note that in the test code task 0 communicates with task n/2, task 1 communicates with task n/2+1, and so on and so forth. This is illustrated in Figure 7.1 for an example using 16 tasks. This setup must be taken into account for binding purposes, particularly when trying to determine differences between intra socket and inter socket communication, or for inter node communication across different sockets. This pattern has been made the default so that using multiple nodes results in all communications taking place in between nodes rather than within nodes.

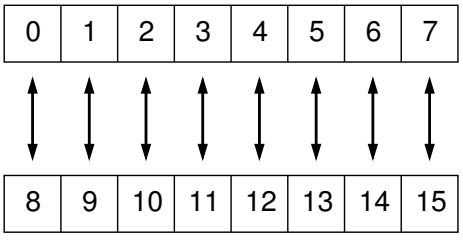

Figure 7.1: MPI task pairing for point to point communication in a 16 task example.

PAW provides independent codes to measure the performance of the following MPI calls:

- Collectives
  - MPI_Accumulate with active synchronization (`col_acc`)
  - MPI_Accumulate with passive synchronization (`col_acc_fence`)
  - MPI_Allgather (`col_agtr`)
  - MPI_Allreduce (`col_ardc`)
  - MPI_Alltoall (`col_a2a`)
  - MPI_Bcast (`col_bct`)
  - MPI_Gather (`col_gtr`)
  - MPI_Reduce (`col_rdc`)
  - MPI_Raccumulate (`col_racc`)
  - MPI_Scatter (`col_sct`)
- Point to Point
  - MPI_Get with active synchronization (`p2p_get`)
  - MPI_Get with passive synchronization (`p2p_get_fence`)
  - MPI_Isend (`p2p_isend`)
  - MPI_Put with active synchronization (`p2p_put`)
  - MPI_Put with passive synchronization (`p2p_put_fence`)
  - MPI_Rget (`p2p_rget`)
  - MPI_Rput (`p2p_rput`)
  - MPI_Send (`p2p_send`)
- Specialized
  - MPI_Isend in ping-ping configuration (`p2p_isend_bidir`)
  - Unidirectional MPI bandwidth using multiple tasks per node (`p2p_multitask`)
  - MPI_Isend using bi-directional communication (`p2p_isend_ping`)
  - Unidirectional MPI message rate (`p2p_mrate`)
  - Bidirectional MPI message rate (`p2p_mrate_bidir`)

Each of the individual tests is described in the sections below.

7.1 MPI-1.0 Collectives

In all collectives tests the call itself is executed by all processes, timed independently, and the minimum value of the timed intervals chosen. Note that this has the serious disadvantage of ignoring issues with messages sent using the eager protocol. An `MPI_Bcast` or `MPI_Scatter` using a small message size may report incorrect timings because it is not forced to wait for a completion signal from any of the other tasks involved. This means that the effective bandwidth may be overestimated for small messages in these cases. In order to minimize this effect we use the maximum measured time across all tasks when reporting timings and bandwidth for these two calls, while we use the minimum time for all others. This is not a perfect solution. If precise measurements of the performance of these calls for small messages are required we suggest altering the code to ensure synchronization has completed within the timing section of the code.

7.2 MPI-2.0 Collectives

7.2.1 col_acc

The `col_acc` benchmark uses the `MPI_Accumulate` call, together with active synchronization calls `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post` and `MPI_Win_wait`. The data is reduced in task 0 using the `MPI_SUM` operation. Within the timed section of the code only the calls shown in Table 7.1 are executed.

Table 7.1: Timed code in `col_acc` test.

Tasks 1...(n-1)	**Task 0**
MPI_Win_start(...);	MPI_Win_post(...);
MPI_Accumulate(...);	MPI_Win_wait(...);
MPI_Win_complete(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `acc_time-np_XXXX.dat` and `acc_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.2.2 col_acc_fence

The `col_acc_fence` benchmark uses the `MPI_Accumulate` call, together with passive synchronization calls to `MPI_Win_fence`. The data is reduced in task 0 using the `MPI_SUM` operation. Within the timed section of the code only the calls shown in Table 7.2 are executed.

Table 7.2: Timed code in `col_acc_fence` test.

Tasks 1...(n-1)	**Task 0**
MPI_Win_fence(...);	MPI_Win_fence(...);
MPI_Accumulate(...);	MPI_Win_fence(...);
MPI_Win_fence(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `accfence_time-np_XXXX.dat` and `accfence_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.3 MPI-3.0 Collectives

The `col_racc` benchmark uses the `MPI_Raccumulate` call, together with synchronization calls to `MPI_Wait` (local synchronization), and `MPI_Win_flush_all` (remote synchronization). The data is reduced in task 0 using the `MPI_SUM` operation. Within the timed section of the code only the calls shown in Table 7.3 are executed.

Table 7.3: Timed code in `p2p_racc` test.

Tasks 1...(n-1)
MPI_Raccumulate(...);
MPI_Wait(...);
MPI_Win_flush_all(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `rget_time-np_XXXX.dat` and `rget_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.4 MPI-1.0 Point to Point

This section describes the setup for all MPI-1.0 point to point communication schemes considered in this test suite.

7.4.1 p2p_isend

The `p2p_isend` benchmark uses the `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` calls. The communication happens in a ping-pong setup, with the first half of the tasks sending data to their partner tasks in the second half, and then reversing the pattern after synchronization. Within the timed section of the code only the calls shown in Table 7.4 are executed.

Table 7.4: Timed code in `p2p_isend` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Isend(...);	MPI_Irecv(...);
MPI_Wait(...);	MPI_Wait(...);
MPI_Irecv(...);	MPI_Isend(...);
MPI_Wait(...);	MPI_Wait(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `isend_time-np_XXXX.dat` and `isend_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.4.2 p2p_send

The `p2p_isend` benchmark uses the `MPI_Send` and `MPI_Recv` calls. The communication happens in a ping-pong setup, with the first half of the tasks sending data to their partner tasks in the second half, and then reversing the pattern after synchronization. Within the timed section of the code only the calls shown in Table 7.5 are executed.

Table 7.5: Timed code in `p2p_send` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Send(...);	MPI_Recv(...);
MPI_Recv(...);	MPI_Send(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `send_time-np_XXXX.dat` and `send_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.5 MPI-2.0 Point to Point

This section describes the setup for all MPI-2.0 point to point communication schemes considered in this test suite.

7.5.1 p2p_get

The `p2p_get` benchmark uses the `MPI_Get` call, together with active synchronization calls `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post` and `MPI_Win_wait`. The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.6 are executed.

Table 7.6: Timed code in `p2p_get` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Win_start(...);	MPI_Win_post(...);
MPI_Get(...);	MPI_Win_wait(...);
MPI_Win_complete(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `get_time-np_XXXX.dat` and `get_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.5.2 p2p_get_fence

The `p2p_get` benchmark uses the `MPI_Get` call, together with passive synchronization calls to `MPI_Win_fence`. The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.7 are executed.

Table 7.7: Timed code in `p2p_get_fence` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Win_fence(...);	MPI_Win_fence(...);
MPI_Get(...);	MPI_Win_fence(...);
MPI_Win_fence(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `getfence_time-np_XXXX.dat` and `getfence_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.5.3 p2p_put

The `p2p_put` benchmark uses the `MPI_Put` call, together with active synchronization calls `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post` and `MPI_Win_wait`. The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.8 are executed.

Table 7.8: Timed code in `p2p_put` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Win_start(...);	MPI_Win_post(...);
MPI_Put(...);	MPI_Win_wait(...);
MPI_Win_complete(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `put_time-np_XXXX.dat` and `put_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.5.4 p2p_put_fence

The `p2p_put` benchmark uses the `MPI_Put` call, together with passive synchronization calls to `MPI_Win_fence`. The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.9 are executed.

Table 7.9: Timed code in `p2p_put_fence` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Win_fence(...);	MPI_Win_fence(...);
MPI_Put(...);	MPI_Win_fence(...);
MPI_Win_fence(...);	

The code writes a summary to `stdout`, and timing and bandwidth results to files named `putfence_time-np_XXXX.dat` and `putfence_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.6 MPI-3.0 Point to Point

This section describes the setup for all MPI-3.0 point to point communication schemes considered in this test suite.

7.6.1 p2p_rget

The `p2p_rget` benchmark uses the `MPI_Rget` call, together with synchronization calls to `MPI_Wait` (local synchronization), and `MPI_Win_flush_all` (remote synchronization). The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.10 are executed.

Table 7.10: Timed code in `p2p_rget` test.

Tasks 0...(n/2-1)
MPI_Rget(...);
MPI_Wait(...);
MPI_Win_flush_all(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `rget_time-np_XXXX.dat` and `rget_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.6.2 p2p_rput

The `p2p_rput` benchmark uses the `MPI_Rput` call, together with synchronization calls `MPI_Win_wait` (local synchronization) and `MPI_Win_flush_all` (remote synchronization). The communication happens in one direction only, with the first half of the tasks pulling data from their partner tasks in the second half. Within the timed section of the code only the calls shown in Table 7.11 are executed.

Table 7.11: Timed code in `p2p_rput` test.

Tasks 0...(n/2-1)
MPI_Rput(...);
MPI_Wait(...);
MPI_Win_flush_all(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `put_time-np_XXXX.dat` and `put_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.7 Specialized

This section describes point to point measurements that do not employ a ping-pong setup, and that are used for specialized purposes.

7.7.1 p2p_isend_bidir

Some high performance interconnects will allow for simultaneous bidirectional traffic, potentially doubling the effective network bandwidth. This test uses non-blocking MPI calls `MPI_Isend` and `MPI_Irecv` completed by `MPI_Waitall` statements. Within the timed section of the code only the calls shown in Table 7.12 are executed.

Table 7.12: Timed code in `p2p_isend_bidir` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Isend(...);	MPI_Irecv(...);
MPI_Irecv(...);	MPI_Isend(...);
MPI_Waitall(...);	MPI_Waitall(...)

The code writes a summary to `stdout`, and timing and bandwidth results to files named `isend_bidir_time-np_XXXX.dat` and `isend_bidir_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.7.2 p2p_isend_ping

The `p2p_isend_ping` benchmark uses the `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` calls. The communication happens in a ping-ping setup, with the first half of the tasks sending data to their partner tasks in the second half. This setup is useful in order to measure asymmetries in the exchange, which would be averaged out by a ping-pong setup. While no asymmetries are expected between identical devices this test can be useful when exchagning data between two different processor types, or between a traditional processor and a coprocessor or accelerator device. Within the timed section of the code only the calls shown in Table 7.13 are executed.

Table 7.13: Timed code in `p2p_isend_ping` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
MPI_Isend(...);	MPI_Irecv(...);
MPI_Wait(...);	MPI_Wait(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `isend_ping_time-np_XXXX.dat` and `isend_ping_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.7.3 p2p_multitask

The `p2p_multitask` benchmark uses the `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` calls. It is a specialized version of the `p2p_multitask` using a single size for the data exchange equal to `MAX_P2P_SIZE`. Within the timed section of the code only the calls shown in Table 7.4 are executed.

This test is provided because the typical ping-pong tests allow for multiple MPI task pair data transfer overlap, taking advantage of the bidirectional nature of most high performance interconnects. A ping-pong test of that type with multiple tasks per node will typically provide a bandwidth that is below the bidirectional limit and above the unidirectional limit. Such a measurement is imprecise and provides no effective information since the overlap is not controlled.

The code writes a summary to `stdout`, and timing and bandwidth results to files named `multitask_time-np_XXXX.dat` and `multitask_bw-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.7.4 p2p_mrate

The `p2p_mrate` benchmark uses the `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` calls. The communication happens in a pipelined ping-ping setup, with the first half of the tasks sending data to their partner tasks in the second half. A user given window size determines how many messages are pipelined one after another before performing a synchronization call, with the aim of determining the sustained message rate of the network. This number is defined in the file `src/aux/constants.h` as `DEFAULT_WINDOW_SIZE` but can be overwritten using the environmental variable `WINDOW_SIZE` Within the timed section of the code only the calls shown in Table 7.14 are executed, with WS standing for the window size value.

Table 7.14: Timed code in `p2p_mrate` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
for(k=0;k<WS;k++){ MPI_Isend(...); }MPI_Waitall(...);	for(k=0;k<WS;k++){ MPI_Irecv(...); }MPI_Waitall(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `mrate_time-np_XXXX.dat` and `mrate_rate-np_XXXX.dat`, where `XXXX` is the number of processors used in the test.

7.7.5 p2p_mrate_bidir

The `p2p_mrate_bidir` benchmark uses the `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` calls. The communication happens in a pipelined bidirectional ping-ping setup, with all tasks sending data to and receiving data from their partner tasks. A user given window size determines how many messages are pipelined one after another before performing a synchronization call, with the aim of determining the sustained message rate of the network. This number is defined in the file `src/aux/constants.h` as `DEFAULT_WINDOW_SIZE` but can be overwritten using the environmental variable `WINDOW_SIZE` Within the timed section of the code only the calls shown in Table 7.15 are executed, with WS standing for the window size value.

Table 7.15: Timed code in `p2p_mrate` test.

Tasks 0...(n/2-1)	**Tasks n/2...(n-1)**
for(k=0;k<WS;k++){ MPI_Isend(...); MPI_Irecv(...); }MPI_Waitall(...);	for(k=0;k<WS;k++){ MPI_Irecv(...); MPI_Isend(...); }MPI_Waitall(...);

The code writes a summary to `stdout`, and timing and bandwidth results to files named `mrate_bidir_time-np_XXXX.dat` and `mrate_bidir_rate-np_XXXX.dat`,

where `XXXX` is the number of processors used in the test.

8 GPU

GPUs can provide a high theoretical peak performance, but can be limited in their usability due to data transfer bottlenecks across the PCIe bus. This makes important the measurement of data transfer rates between CPU and GPU. This section provides a description of the tests for floating point performance and data transfer rates for GPUs (using NVIDIA's CUDA) provided with PAW.

8.1 Floating Point Performance

GPUs traditionally provide a high theoretical peak rate, by including a large number of simple processing units that, on aggregate, yield impressive floating point performance.

PAW provides code designed to test DGEMM and SGEMM calls from the cublas library. As with all PAW benchmarks the timer overhead is taken into account when defining the minimum number of iterations to execute for the smallest workload to be measured, and `NLOOP` is then dynamically adjusted to avoid spending too much time testing a particular workload size. A warmup call is performed outside the timed loop for every workload size. Each size is tested `NREPS` times in order to gather statistics for the timings.

It is important to mention that the `cublasSetVector` calls used to transfer data to the GPU are not timed as part of this measurement. Only the execution of the `cublasDgemm` and `cublasSgemm` calls is actually timed. It would be simple to change the code to include these into the timed section if necessary.

The two executables are simply called `gpu_dgemm` and `gpu_sgemm`

8.2 Data Transfer Rates

To measure the data transfer to and from the GPU a setup similar to that used for measuring MPI data transfers was used. The device memory is allocated with a `cudaMalloc` call, and the transfer performed using `cudaMemcpy`. Multiple messages are sent within a timed loop, and the average value is taken. A warmup message is exchanged outside the timed section of the code, before the loop starts. This measurement is run `NREPS` times and the maximum, minimum and mean values are stored to a file for each message size. As in the MPI bandwidth measurements, care is taken to set the inner loop count to a value such that an accurate timing can be obtained even for the smallest message exchanged.

In the cases where pinned memory is used the `cudaMallocHost` call is used to allocate the host side array.

Several variants of the tests are provided: single direction exchange, round trip, pinned memory and non pinned memory. Table 8.1 describes all variants of the test included with PAW.

Table 8.1: Tests for GPU Data Transfer Rates

Name	**Direction**	**Memory**
`gpu_in_pinned`	CPU to GPU	Pinned
`gpu_inout_pinned`	CPU to GPU + GPU to CPU	Pinned
`gpu_out_pinned`	GPU to CPU	Pinned
`gpu_in_nopin`	CPU to GPU	Not Pinned
`gpu_inout_nopin`	CPU to GPU + GPU to CPU	Not Pinned
`gpu_out_nopin`	GPU to CPU	Not Pinned

9 Xeon Phi

The Intel Xeon Phi is a coprocessor that takes advantage of vectorizable threaded codes to provide high floating point rates. While the Xeon Phi runs its own Operating System, and has a much more flexible execution mode than a GPU, it is still connected to the host CPU by the PCI Express bus, and that can result in a communication bottleneck when large amounts of data must be exchanged often between the two, just like in the case of GPU-enabled codes. One could argue that in this case the CPU could be taken out of the equation, since code can be executed natively on the Xeon Phi, but given the limited amount of on-board memory present on the device most codes will utilize several Phis or multiple Phis and host CPUs. This makes it important to look at data transfer rates between CPU and MIC, as it was done for GPUs.

9.1 Data Transfer Rates

Several factors affect the speed of an offload data transfer to the MIC. By default, when data is transfered to the MIC as part of an offload region, the runtime will allocate memory on the coprocessor, copy the data over, operate on it as needed, transfer it back to the host, and free memory on the coprocessor. Sometimes this is exactly the behavior required by the code but other times it is OK to maintain the allocated memory on the MIC without freeing it on return, and treat it as persistent in the same way that a CUDA memory transfer works for a GPU. The overhead of allocation and deallocation can be particularly noticeable when performing an offload inside a loop. Another issue that could affect data transfers is data alignment. Last, but not least, message size will be important as well.

Several tests are provided to evaluate data transfers considering all these variables. Table 9.1 describes all available tests in PAW.

Table 9.1: Tests for Phi Data Transfer Rates

Name	Direction	Reallocate?	Memory
phi_alloc_al_in	CPU to PHI	YES	Aligned
phi_alloc_al_inout	CPU to PHI + PHI to CPU	YES	Aligned
phi_alloc_al_out	PHI to CPU	YES	Aligned
phi_alloc_noal_in	CPU to PHI	YES	Not Aligned
phi_alloc_noal_inout	CPU to PHI + PHI to CPU	YES	Not Aligned
phi_alloc_noal_out	PHI to CPU	YES	Not Aligned
phi_keep_al_in	CPU to PHI	NO	Aligned
phi_keep_al_inout	CPU to PHI + PHI to CPU	NO	Aligned
phi_keep_al_out	PHI to CPU	NO	Aligned
phi_keep_noal_in	CPU to PHI	NO	Not Aligned
phi_keep_noal_inout	CPU to PHI + PHI to CPU	NO	Not Aligned
phi_keep_noal_out	PHI to CPU	NO	Not Aligned

9.1.1 Dual Phi Systems

In order to investigate the effective transfer rates in dual mic nodes two individual offload tests are executed simultaneously, one to device `mic:0` and one to device `mic:1`. The code uses two OpenMP threads that execute, independently from each other, an offload from thread 0 to mic:0 and an offload from thread 1 to mic:1. Keep in mind that this design is not perfect, and allows for transfers that do not overlap for 100% of the transference time. However, given the simplicity of the test, it is unlikely that the offloads are started with a significant delay with respect to one another. This test also happens to reproduce fairly well a situation that one may expect in a real application code, and we believe the information it provides is relevant and useful.

Every test in Table 9.1 will automatically detect the presence of multiple Phi devices and run this additional test if needed. Output files from these dual Phi tests will be saved with a `_mic0+1` suffix to differentiate them from the single Phi tests.

Bibliography

[1] STREAM is the industry standard memory bandwidth benchmark, created by John D. McCalpin. For more information visit the official STREAM benchmark web site: www.cs.virginia.edu/stream/

[2] The EPCC OpenMP micro-benchmark suite is designed to measure overheads of synchronization, loop scheduling and array operations in the OpenMP runtime library. For more information visit the official EPCC benchmarking website: www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/

[3] The IOR benchmark is designed to measure IO performance at the POSIX and MPI-IO level. For more information visit: github.com/chaos/ior/

[4] J. Dongarra, The LINPACK Benchmark: An Explanation, *Proceedings of the 1st International Conference on Supercomputing*, 456–474 (1988)

[5] The TOP500 list ranks commercially available supercomputers in order of performance, as measured by executing the High Performance Linpack (HPL) benchmark. For more information see: www.top500.org/