MPI-Evaluation

Bericht zum Projekt „Parallelrechnerevaluation“,

von Sibilla Mazza und Houman Ramezani Farid

Inhaltsverzeichnis

1 Ziel des Projekts

Das Ziel des Projektes war, verschiedene Implementationen von MPI (Message Passing Interface), nämlich OpenMPI, MPICH und MVAPICH2 zu evaluieren.

Dafür haben wir die OSU-Benchmarks (http://mvapich.cse.ohio-state.edu/benchmarks/) benutzt.

Mithilfe von Benchmark-Tests kann man die Leistung von den Implementationen ermitteln. Die Benchmarks sind dabei ein Vergleichsmaßstab, mit dem die Ergebnisse ermittelt und verglichen werden können.

Auf der oben genannten Seite sind verschiedene Tests, wie zum Beispiel zur Latenz- und zur Bandbreitenmessung, die wir für die drei verschiedene Implementationen auf dem Cluster laufengelassen haben, um zu sehen, welche Implementation bei welchem Test am besten war. Die Auswertungen haben wir dann grafisch dargestellt.

Für das Projekt haben wir mit

- Cluster
- Spack
- MPI-Implementationen
- Benchmarks
- Job scripts
- Bash scripts
- Python zur automatisierten Auswertung der Daten
- C um die MPI Programme zu schreiben

gearbeitet.

2 Was ist MPI?

2.1 Allgemein und ein Beispiel

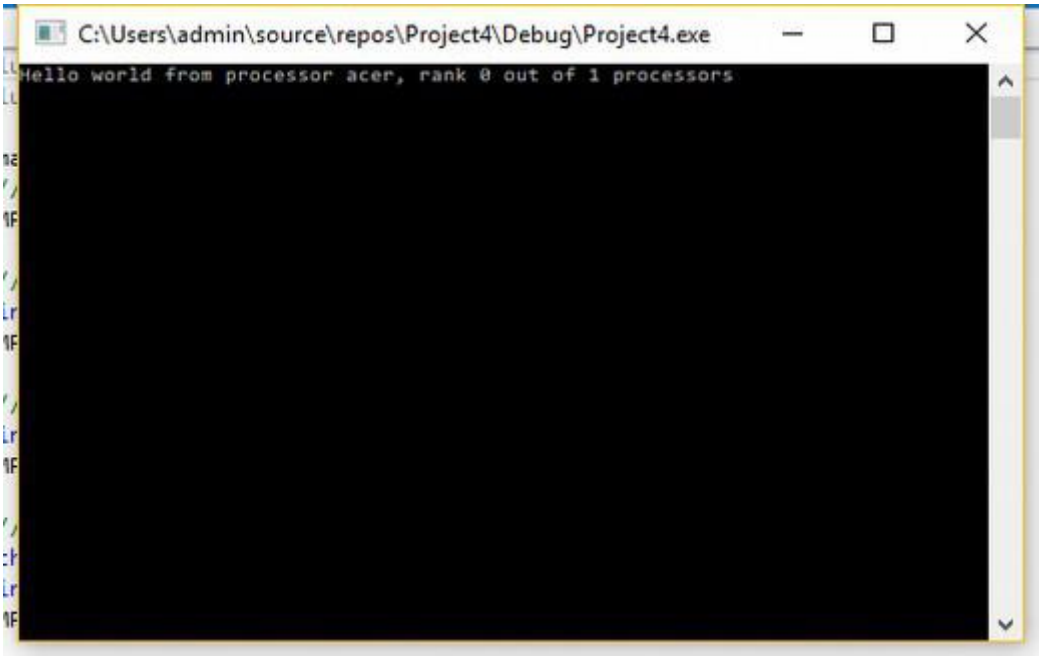
MPI steht für Message Passing Interface und beschreibt die Kommunikation und den Nachrichtenaustausch bei parallelen Berechnungen zwischen verschiedenen Prozessen. Es enthält eine Sammlung von Operationen und ihre Semantik, aber keine Implementierung. Es gibt verschiedene Implementationen von MPI, wie zum Beispiel OpenMPI, MPICH und MVAPICH2.

Es gibt Prozesse, die gemeinsam an einem Problem arbeiten und mithilfe der MPI-Schnittstelle Daten und Informationen austauschen.

Hier ist ein Beispiel für ein MPI „Hello World“ Programm, geschrieben in C:

```
1     int main(int argc, char** argv) {
2           MPI_Init(NULL, NULL);
3
4
5           int world_size;
6           MPI_Comm_size(MPI_COMM_WORLD, &world_size);
7
8
9           int world_rank;
10          MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
11
12
13          char processor_name[MPI_MAX_PROCESSOR_NAME];
14          int name_len;
15          MPI_Get_processor_name(processor_name, &name_len);
16
17
18          printf("Hello world from processor %s, rank %d out of %d
            processors\n", processor_name, world_rank, world_size);
19
20          getchar();
21
22          MPI_Finalize();
23    }
```

Hier kann man die Ausgabe des Programms sehen:



Beschreibung des Codes:

Zuerst braucht man #include <mpi.h> für die nötige Bibliothek für MPI.

Danach wird MPI initialisiert und die Prozesse werden in dem Kommunikator MPI_COMM_WORLD zusammengefasst, wobei ein Kommunikator eine Gruppe von MPI-Prozessen ist.

MPI_Comm _size gibt die Anzahl von Prozessen in dem Kommunikator.

MPI_Comm_rank liest den Rang von den Prozessen aus, von dem der Befehl ausgeführt wird. Jeder Prozess hat einen Rang und es wird bei 0 angefangen. Den Rang braucht man zur Identifizierung bei den Sende- und Empfangsoperationen.

MPI_Get_processor_name gibt den Namen des Prozesses.

Mit dem printf Befehl wird die Nachricht „Hello world“ angezeigt.

Am Ende braucht man MPI_Finalize um die MPI Umgebung zu beenden.

Dies ist ein schönes Beispiel, um sich einmal die besonderen Befehle für MPI-Programme anzusehen. Man kann sehen, dass es etwas komplizierter ist, als das „normale“ „Hello World“ Programm in C. Außerdem sieht man hier schon einige der wichtigen Befehle, die man für viele MPI-Anwendungen benötigt.

Um MPI zu verstehen, ist es wichtig zu wissen, wie MPI-Prozesse miteinander kommunizieren.

Es gibt in MPI zwei verschiedene Arten von Kommunikation:

- Punkt zu Punkt Kommunikation (zwischen zwei Prozessen)
- Globale Kommunikation (alle Prozesse in einer Gruppe kommunizieren)

2.2 Punkt-zu-Punkt Kommunikation

Die Punkt-zu-Punkt Kommunikation ist eine Kommunikation zwischen zwei Prozessen mit einem Sendeprozess und einem Empfangsprozess.

Die Informationen sind in Nachrichten enthalten und es gibt eine Sendeoperation und eine Empfangsoperation.

Für die Sendeoperation gibt es den Befehl MPI_Send mit folgenden Parametern:

buf: Adresse vom Sendepuffer

count: Anzahl der Elemente im Sendepuffer

datatype: Datentyp der Elemente im Sendepuffer

dest: Zielrang

tag: Markierung der Nachricht

comm: Kommunikator

Für die Empfangsoperation gibt es den Befehl MPI_Recv mit folgenden Parametern:

buf: Adresse der Empfangspuffer

count: Anzahl der Elemente im Empfangspuffer

datatype: Datentyp der Elemente im Empfangspuffer

source: Rang des sendenden Prozesses. Mit source=MPI_ANY_SOURCE werden Nachrichten von jedem Prozess empfangen

tag: Markierung der Nachricht. Mit tag=MPI_ANY_TAG werden Nachrichten mit beliebiger Markierung empfangen

comm: Kommunikator

status: Zeiger auf einer Statusstruktur

Man kann zwischen blockierender Kommunikation, nicht-blockierender Kommunikation und synchronisierendem Senden unterscheiden.

- Blockierende Kommunikation: MPI_Send und MPI_Recv halten den Prozess an, bis alle Daten vollständig übertragen worden sind.
- Nicht-blockierende Kommunikation: Die Kommunikation läuft, ohne auf das Ende des Sende-/Empfangsprozesses zu warten.

Die nicht-blockierende Kommunikation hat zwei zusätzliche Parameter: request, um abzufragen ob der Sende-/Empfangsprozess beendet ist und MPI_Wait um auf seine Beendigung zu warten.

- Synchronisierendes Senden: Das Senden wird erst beendet, wenn die Empfangsoperation begonnen wurde.

Hier ist ein Beispiel eines Send/Receive-Programms in C, um die Befehle der Punkt-zu-Punkt Kommunikation zu zeigen:

```c
// Prozess 0 sendet die Zahl -1 an Prozess 1.
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

	MPI_Init(NULL, NULL);
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	int zahl;

	if (world_rank == 0)
	// Die Argumente von MPI_Send sind Zahl, count, datatype, dest, tag,
	comm.
	{
		zahl = -1;
		MPI_Send( &zahl, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
	}
	else if (world_rank == 1)
	// Die Argumente von MPI_Recv sind Zahl, count, datatype, source
	tag, comm, status
	{
		MPI_Recv( &zahl, 1, MPI_INT, 0, 0,
		MPI_COMM_WORLD,MPI_STATUS_IGNORE);

		printf("Prozess 1 hat Zahl %d bekommen vom Prozess 0\n", zahl);
		getchar();
	}

	MPI_Finalize();
}
```

Beschreibung des Codes:

Am Anfang wird wie beim “Hello world” Programm MPI initialisiert, die Prozesse werden in dem Kommunikator MPI_COMM_WORLD zusammengefasst und MPI_Comm_size und MPI_Comm_rank definiert.

Mit der if-Abfrage sendet der Prozess mit Rang 0 die Zahl -1 und der Prozess mit dem Rang 1 empfängt die Zahl.

Beim MPI_Send ist:

count=1, da nur eine Zahl gesendet wird,

datatype ist Integer,

dest=1, da der empfangende Prozess Rang 1 hat,

tag=0,

der Kommunikator ist MPI_COMM_WORLD.

Beim MPI_Recv ist:

count=1, da nur eine Zahl gesendet wird,

datatype ist Integer,

source=0, da der sendende Prozess Rang 0 hat,

tag=0,

der Kommunikator ist MPI_COMM_WORLD.

Im Folgenden ist noch ein Beispiel:

„Potenzen von 2“

```
1     // Initialisierung und Finalisierung weggelassen
2     int number;
3     if (rank == 0) // Falls erster Prozess
4     {
5     number = 2;
6
7           if (size > 1) // Falls es mehr als 1 Prozess gibt
8           {
9                 MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
10                MPI_Recv(&number, 1, MPI_INT, MPI_ANY_SOURCE, 1,
11                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
12          }
13    printf("2^%d equals %d\n", size, number);
14    }
15
16    elseif ((rank == size - 1) && (size > 1)) // Falls Prozess letzter
17    {
18          MPI_Recv(&number, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
19          number = number * 2;
20          MPI_Send(&number, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
21    }
22
23    else // Falls Prozess nicht letzter oder erster
24    {
25          MPI_Recv(&number, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE)
26          number = number * 2;
27          MPI_Send(&number, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
28    }
```

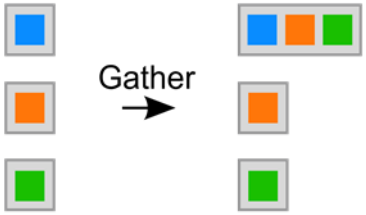
2.3 Globale Kommunikation

Für die globale Kommunikation gibt es folgenden Arten der Kommunikation:

- Broadcast: Ein MPI-Prozess (root) schickt allen Prozessen in seiner Gruppe (comm) die gleichen Daten

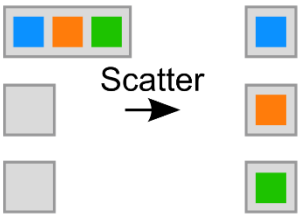
(Quelle: https://de.wikipedia.org/wiki/Message_Passing_Interface#/media/Datei:Mpi_broadcast.svg, Stand: 30.8.2019)

- Gather: Der Prozess root sammelt die Daten der anderen Prozesse ein, die dann nach Rang sortiert im Empfangspuffer abgelegt werden



(Quelle: https://de.wikipedia.org/wiki/Message_Passing_Interface#/media/Datei:Mpi_gather.svg, Stand: 30.8.2019)

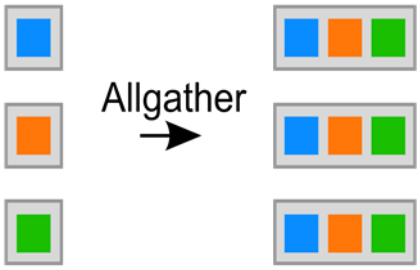
- Scatter: root schickt allen Prozessen ein unterschiedliches, aber gleich großes Element



(Quelle: https://de.wikipedia.org/wiki/Message_Passing_Interface#/media/Datei:Mpi_scatter.svg, Stand:30.8.2019)

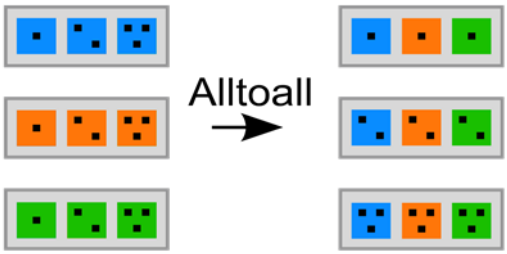
- Akkumulation: Spezielle Form von Gather. Die Daten werden zusätzlich zu einem Datum reduziert

- Allgather: Jeder Prozess schickt an jeden anderen Prozess die gleichen Daten



(Quelle: https://de.wikipedia.org/wiki/Message_Passing_Interface#/media/Datei:Mpi_allgather.svg, Stand: 30.8.2019)

- All-to-all: Ähnlich wie Allgather, aber der i-te Teil des Sendebuffers wird an den i-ten Prozess gesendet



(Quelle: https://de.wikipedia.org/wiki/Message_Passing_Interface#/media/Datei:Mpi_alltoall.svg, Stand: 30.8.2019)

3 Clustervorbereitungen

Bevor die Benchmarks auf dem Cluster ausgeführt werden konnten, mussten Vorbereitungen getroffen werden. Neben den notwendigen Vorbereitungen, wie z.B. das Installieren von Spack, den verschiedenen MPI-Implementationen und der Benchmarks, wurden auch Hilfsmittel erstellt, die eine Ausführung erleichtern bzw. automatisieren. Dazu gehören die Job scripts und die Bash scripts.

3.1 Spack und MPI-Implementationen

Spack ist ein Paketmanager für Cluster zur Unterstützung mehrerer verschiedener Versionen und Konfigurationen von Software, unter anderem Compiler, Architekturen und Bibliotheken. Es wird benötigt, um die drei verschiedenen MPI-Implementationen zu installieren und bei Bedarf abrufen zu können.

Spack war auf dem Cluster bereits als globale Installation vorhanden, jedoch wurde eine für den Benutzer lokale Installation durchgeführt, um Spack und verfügbare Pakete individuell konfigurieren zu können.

Die Installation von Spack war unkompliziert:

```
$ git clone https://github.com/spack/spack
```

Damit bei der Benutzung des Befehls „spack“ nicht die globale Spack-Installation verwendet wurde, musste die lokale Umgebung geladen werden:

```
$ source /PfadZumSpackOrdner/share/spack/setup-env.sh
```

Das dies in einer neuen Sitzung auf dem Cluster nicht automatisch geschah, musste es in jeder weiteren Sitzung erneut durchgeführt werden.

Nun konnten die MPI-Implementationen lokal installiert werden. Die Befehle dazu waren einfach:

```
$ spack install mpich

$ spack install openmpi

$ spack install mvapich2
```

An dieser Stelle trat bereits der erste Fehler auf:

```
==> Error: Conflicts in concretized spec "mvapich2@2.3.1%gcc@8.3.0~alloca
ch3_rank_bits=32 ~cuda~debug fabrics=psm file_systems=auto
process_managers=auto +regcache threads=multiple arch=linux-ubuntu18.04-
westmere/eqlv3qp"

List of matching conflicts for spec:
```

```
    psm@3.3%gcc@8.3.0 arch=linux-ubuntu18.04-westmere
        ^libuuid@1.0.3%gcc@8.3.0 arch=linux-ubuntu18.04-westmere


1. "%gcc@6:" conflicts with "psm@3.3"
```

Wie zu erkennen ist, standen hier zwei Pakete im Konflikt. MVAPICH2 benutzte in der Standardkonfiguration „psm“ als fabric, weshalb hier zusätzlich bei der Installation ein anderes fabric gesetzt wurde:

```
$ spack install mvapich2 fabrics=nemesis
```

OpenMPI ließ sich in Standardkonfiguration zwar Installieren, jedoch trat im Laufe des Projekts bei der Ausführung der Benchmarks folgender Fehler auf, wenn ein Job script ausgeführt wurde:

```
This test requires exactly two processes
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpiexec detected that one or more processes exited with non-zero status,
thus causing the job to be terminated. The first process to do so was:

  Process name: [[36296,1],14]
  Exit code:    1
--------------------------------------------------------------------------
```

Dieser Fehler sagt aus, dass der Test zwei Prozesse benötigt. Merkwürdig hierbei war, dass in demselben Job script die Implementationen MPICH und MVAPICH2 fehlerfrei funktionierten. Bei der manuellen Ausführung mit „mpiexec“ und der Übergabe der Prozessanzahl lief auch OpenMPI problemlos. Die Vermutung war daher, dass OpenMPI die Argumente von Slurm nicht ordnungsgemäß erhielt.

Zunächst wurde versucht, bei der Installation von OpenMPI die Option „+legacylaunchers“ zu setzen. Der gleiche Fehler bestand jedoch weiterhin. Erst mit der Option „schedulers=slurm“ und dazugehörig „+pmi“ funktionierte der Job script mit allen drei Implementationen. Unabhängig davon wurde auch „+thread_multiple“ gesetzt. Die genauen Builds werden in Abschnitt 5 näher aufgeführt.

(Hinweis: Es war geplant, diesen Fehler zum Zeitpunkt der Berichterstellung zu reproduzieren, jedoch konnten auf dem Cluster keinerlei Jobs ausgeführt werden. Es tauchte

folgender Fehler auf: “sbatch: error: Batch job submission failed: Unable to contact slurm controller (connect failure)”; auch sinfo schlägt fehl).

3.2 OSU-Benchmarks

Die OSU-Benchmarks mussten zunächst heruntergeladen und entpackt werden:

```
$ wget http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.6.1.tar.gz

$ tar xzvf osu-micro-benchmarks-5.6.1.tar.gz
```

Die Installation erfolgte mithilfe der ausführbaren Dateien „configure“ und „make“ , die sich im Benchmark-Ordner befanden. Weiterhin wurden als Argumente die Pfade zu den Compilern „mpicc“ und „mpicxx“, sowie optional der Installationspfad (--prefix) benötigt. Bei der ersten Installation wurde der Installationspfad weggelassen, was zu Fehlermeldungen geführt hat, da versucht wurde, auf das Root-Verzeichnis zuzugreifen. Dafür fehlten jedoch dem gewöhnlichen Nutzer die Berechtigungen, weshalb ein lokaler Pfad angegeben wurde. Da jede MPI-Implementation ihre eigenen Varianten der Compiler „mpicc“ und „mpicxx“ besaß, wurde zuvor der Benchmark-Ordner zweimal kopiert, sodass jede Implementation in einem separaten Ordner installiert werden konnte.

Im Folgenden wird die Installation für MPICH beispielhaft dargestellt. Zunächst mussten alle nicht benötigten Module, wie etwa andere MPI-Implementationen, aus der Umgebung entfernt werden. Anschließend wurde die gewünschte Implementation geladen und der Pfad zu den beiden Compilern ermittelt:

```
$ module purge

$ spack load mpich

$ which mpicc
/home/benutzer/spack/opt/spack/linux-ubuntu18.04-westmere/gcc-8.3.0/mpich-3.3-2ld7eyt3jiooqapu3duefu53ysmlwjhb/bin/mpicc

$ which mpicxx
/home/benutzer/spack/opt/spack/linux-ubuntu18.04-westmere/gcc-8.3.0/mpich-3.3-2ld7eyt3jiooqapu3duefu53ysmlwjhb/bin/mpicxx
```

Nun konnten die benötigten Pfade bei der Installation mit configure angegeben werden:

```
$ ./configure CC=/home/benutzer/spack/opt/spack/linux-ubuntu18.04-westmere/gcc-8.3.0/mpich-3.3-2ld7eyt3jiooqapu3duefu53ysmlwjhb/bin/mpicc CXX=/home/benutzer/spack/opt/spack/linux-ubuntu18.04-westmere/gcc-8.3.0/mpich-3.3-2ld7eyt3jiooqapu3duefu53ysmlwjhb/bin/mpicxx --prefix=/home/benutzer/benchmarkOrdner

$ make
```

```
$ make install
```

Die Benchmarks waren somit erfolgreich installiert.

3.3 Job scripts

Job scripts sind Textdateien mit der Endung „.slurm“, die vom Slurm Workload Manager verarbeitet werden können. Der Slurm Workload Manager ist ein Jobmanangementsystem für Linux Cluster, mit dem die Ressourcen des Clusters verwaltet und Aufgaben bzw. Aktionen automatisiert bearbeitet werden können. In einem Job script werden zunächst die für den Job benötigten Argumente festgelegt, wie z.B. Name des Jobs, Anzahl der Knoten bzw. der Prozesse und der Clusterpartition. Anders als bei der manuellen Ausführung, wo die Ausgaben direkt im Terminal ausgegeben werden, können hier auch Pfade angegeben werden, wo die Ausgaben und Fehler protokolliert werden sollen.

Hier ist der Job script für den Bandbreiten-Benchmark beispielhaft zu sehen:

```
1     #!/bin/bash
2
3     #SBATCH --job-name=mpi-bandwith
4     #SBATCH --nodes=2
5     #SBATCH --ntasks=2
6     #SBATCH --ntasks-per-node=1
7     #SBATCH --partition=west
8     #SBATCH --output=/home/benutzer/benchmark-output/bandwith.out
9     #SBATCH --error=/home/benutzer/benchmark-output/bandwith.err
10
11    . /home/benutzer/spack/share/spack/setup-env.sh
12
13    echo 'Bandwith:'
14    echo ''
15    echo 'MPICH:'
16    module purge
17    spack load mpich
18    srun hostname
19    mpiexec /home/benutzer/osu-micro-benchmarks-5.6.1-
      mpich/mpi/pt2pt/osu_bw
20
21    echo '---'
22
23    echo 'OpenMPI:'
24    module purge
25    spack load openmpi
26    srun hostname
27    mpiexec /home/benutzer/osu-micro-benchmarks-5.6.1-
      openmpi/mpi/pt2pt/osu_bw
28
29    echo '---'
30
31    echo 'MVAPICH2:'
```

```
32    module purge
33    spack load mvapich2
34    srun hostname
35    mpiexec /home/benutzer/osu-micro-benchmarks-5.6.1-
      mvapich2/mpi/pt2pt/osu_bw
36
37    echo '==='
38
39    echo 'MPICH:'
40    module purge
41    spack load mpich
42    srun hostname
43    mpiexec /home/benutzer/osu-micro-benchmarks-5.6.1-
      mpich/mpi/pt2pt/osu_bw
44
45    echo '---'

[...]

85    echo 'done'
```

In der ersten Zeile wird festgelegt, dass die Datei Bash-Befehle enthält (Linux Kommandozeile). Anschließend werden mit „#SBATCH“ alle Argumente angegeben. In diesem Fall wurden 2 Knoten der Partition „west“ und 2 Prozesse benötigt, die sich auf die Knoten verteilen sollten. Die Option „—ntasks-per-node“ hätte hier weggelassen werden können, da Slurm automatisch die Prozesse zuerst auf die Knoten verteilt, jedoch wurde es sicherheitshalber angegeben. Mit „--output“ und „--error“ werden die zwei Pfade angegeben, wo jeweils die Ausgabe und Fehlermeldungen geschrieben werden sollten.

Statt die lokale Spack Umgebung in jeder Sitzung neu zu laden, wird dies in Zeile 11 durchgeführt. Dadurch braucht man nur einen Job script zu starten und benutzt automatisch die lokale Spack Installation. Die folgenden „echo“-Befehle dienen nur zur Strukturierung der Ausgabe, indem Begriffe oder Hinweise in die Datei geschrieben werden.

Ab Zeile 16 wird die Ausführung vorbereitet. Mit „module purge“ werden alle Module in der Umgebung verworfen, um anschließend eine der MPI-Implementationen zu laden. „srun hostname“ gibt die Namen der Knoten aus, auf denen das Programm ausgeführt wird. Die eigentliche Benchmark-Ausführung beginnt erstmals in Zeile 27 mit „mpiexec“ und der Pfadangabe zum Benchmark. Hier musste darauf geachtet werden, dass die Ordnerpfade mit der aktuell geladenen Implementation übereinstimmten. Zum Trennen der MPI-Implementationen werden in der Ausgabe drei Trennstriche (---) benutzt. Anschließend folgt der gleiche Ablauf für die anderen beiden Implementationen. Nachdem die Benchmarks mit jeder Implementation einmal ausgeführt wurden, markieren drei Gleichheitszeichen (===) das Ende einer Iteration. Es werden danach zwei weitere male alle Implementationen getestet, sodass insgesamt drei Iterationen entstehen. Ein „done“ markiert das Ende der Datei.

Hier ist ein Teil der Ausgabe eines solchen Job scripts zu sehen:

```
Bandwith:

MPICH:
west3
west4
# OSU MPI Bandwidth Test v5.6.1
# Size      Bandwidth (MB/s)
1                       0.24
2                       0.51
4                       1.03
[...]
4194304               117.65
---
OpenMPI:
west3
west4
# OSU MPI Bandwidth Test v5.6.1
# Size      Bandwidth (MB/s)
1                       0.27
2                       0.54
4                       1.06
[...]
4194304               235.12
---
MVAPICH2:
[...]
===
MPICH:
[...]
```

Für jeden Benchmark wurde ein eigenes Job script erstellt, um bei Bedarf einfacher die Daten eines bestimmten Benchmarks wiederzufinden.

3.4 Bash scripts

Bash scripts sind ähnlich wie Job scripts, enthalten jedoch keine Slurm-spezifischen Informationen, sondern Aneinanderreihungen von Bash-Befehlen und besitzen die Dateiendung „.sh“. Diese wurden genutzt, um alle zuvor erstellten Job scripts nacheinander auszuführen, sodass diese nicht manuell gestartet werden mussten. Zudem können auch in Bash scripts Argumente für die Befehle gesetzt werden, wie im Folgenden erkennbar wird.

Hier ist ein Ausschnitt eines verwendeten Bash scripts:

```
1     #/bin/bash
2
3     sbatch --nodelist=west[2] /home/benutzer/jobscripts/acc_latency.slurm
4     sleep 2s
5     sbatch --nodelist=west[5-7] /home/benutzer/jobscripts/allgather.slurm
6     sleep 2s
```

```
7     sbatch --nodelist=west[5-7] /home/benutzer/
      jobscripts/allgatherv.slurm
8     sleep 2s
      [...]
15    sbatch --nodelist=west[3-4] /home/benutzer/jobscripts/bandwith.slurm
16    sleep 2s
      [...]
```

Auch hier wird zunächst angegeben, dass es sich um ein Bash script handelt. Die Aufgaben dieses Skripts sind einfach: Es wird zuerst mit „sbatch“ ein Job script abgeschickt, woraufhin 2 Sekunden gewartet wird, bevor das Gleiche mit dem nächsten Job script geschieht.

Beim Aufruf von „sbatch“ werden die zu verwendenden Knoten der Partition „west“ mitgegeben. Weiterhin ist es möglich, hier Argumente anzugeben, die bereits im Job script gesetzt wurden, wie z.B. die Anzahl der Knoten und der Prozesse. Als Beispiel wird der bereits aufgeführte Bandbreiten-Benchmark betrachtet. Für diesen galten folgende Werte:

```
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=1
```

Ein Aufruf im Bash script könnte wie folgt aussehen:

```
sbatch --nodes=1 --ntasks=2 --ntasks-per-node=2 --nodelist=west[2]
/home/benutzer/jobscripts/bandwith.slurm
```

Die im Bash script gesetzten Werte besitzen gegenüber denen im Job script Priorität, sodass der Benchmark mit zwei Prozessen auf einem Knoten laufen würde.

Es wurde ein weiterer Bash script benutzt, bei dem von dieser Funktionalität gebrauch gemacht wurde, um im Nachhinein die Werte zu ändern, denn ansonsten hätten alle Job scripts einzeln angepasst werden müssen. Darauf wird in Abschnitt 5 näher eingegangen.

4 Python: Verarbeitung/Visualisierung der Benchmarks

Nachdem alle Benchmarks ausgeführt worden waren, galt es, diese in Graphen zu visualisieren, sodass ihre Performance besser verglichen werden konnte. Hierfür wurde die Programmiersprache Python verwendet, für die bereits ein Modul zum Plotting namens „Matplotlib“ existiert.

Die Funktionsweise des Skripts funktioniert im Allgemeinen wie folgt:
Für jede Benchmark-Ausgabe mit der Endung „.out“ im aktuellen Verzeichnis wird zunächst geprüft, ob eine zugehörige Fehlerdatei mit der Endung „.err“ exisitiert und ob sie leer ist, denn bei vorhandenen Fehlern sollten jene Ausgabedateien übersprungen werden. Das Skript geht dann Zeile für Zeile durch die Ausgabe. Wird die Zeichenkette “OSU” gelesen, startet ab der nächsten Zeile die Extraktion der Daten. Dabei werden die Zeilen in zwei Teile getrennt und jeweils gespeichert. Sie bilden die Basis für die x- und y-Werte der Graphen. Dies wird

für alle Implementationen und Iterationen durchgeführt. Anschließend werden für jede Implementation die Ergebnisse der drei Iterationen betrachtet und für jeden x-Wert jeweils der niedrigste, mittlere und höchste y-Wert berechnet. Mithilfe des Matplotlib-Moduls werden diese Daten in Errorbar-Graphen umgewandelt, die wiederum individuell angepasst werden können. Zuletzt werden die erstellen Graphen in einem Ordner abgespeichert. Befindet sich bereits ein gleichnamiger Plot in dem Ordner, wird dieser gelöscht und der aktuelle gespeichert, wodurch alte Dateien praktisch überschrieben werden.

In den folgenden Abschnitten wird das Skript genauer erklärt.

4.1 Vorbereitungen

```
1      import matplotlib.pyplot as plt
2      import os
3      import time
4      import sys
5
6      parsing_data = {'acc_latency.out':[2], 'get_acc_latency.out':[2],
       'get_bw.out':[2], 'get_latency.out':[2], 'iallgather.out':[2],
       'iallgatherv.out': [2],'iallreduce.out':[2], 'ialltoall.out':[2],
       'ialltoallv.out':[2], 'ialltoallw.out':[2], 'ibcast.out':[2],
       'igather.out':[2], 'igatherv.out':[2], 'ireduce.out':[2],
       'iscatter.out':[2], 'iscatterv.out':[2],
       'multiple_bandwith.out':[2,'# Size','Latency (us)'],
       'put_bibw.out':[2], 'put_bw.out':[2], 'put_latency.out':[2]}
7
8      #Jede Datei mit Endung .out öffnen
9      for filename in os.listdir():
10         if filename[-4:].lower() == '.out':
11             err_file = f'{filename[:-4]}.err'
12             err_file_empty = True
13
14      #Falls dazu eine nicht leere Fehlerdatei existiert, nicht fortfahren
15      if os.path.exists(err_file):
16          with open(err_file) as err_log:
17              if err_log.read().strip() == '':
18                  pass
19              else:
20                  err_file_empty = False
21                  print(f"A non-empty error log file exits for '{filename}'")
```

Am Anfang des Skripts (Z. 1-4) werden alle benötigten Module importiert. In Zeile 6 wird ein Dictionary definiert, welches die Namen einiger Benchmark-Ausgaben als Schlüssel enthält. Sie verweisen jeweils auf eine Liste, in dem der Offset für die Extraktion angegeben ist, also wie viele Zeilen nach dem Auftauchen von „OSU“ übersprungen werden müssen, um an die Daten zu gelangen. Der Grund, weshalb hier Listen verwendet werden, ist der, dass in einem Ausnahmefall die Bezeichner der Werte (Size, MB/s etc.) nicht automatisch extrahiert werden können. So werden bei „multiple_bandwith.out“ die Bezeichner direkt mitgegeben.

Ab Zeile 9 beginnt die Schleife, die über alle Dateien im aktuellen Verzeichnis iteriert. Erst wird der Dateiname ohne die Endung „.out“ übernommen, mit der Endung „.err“ versehen und abgespeichert, sowie ein boolean erstellt, der festhält, ob die Fehlerdatei leer ist. Danach

wird ab Zeile 15 geprüft, ob eine solche Datei existiert. Falls ja, wird sie geöffnet und alle leeren Zeichen entfernt. Nur wenn die Datei leer ist oder nicht existiert, wird die Datei verarbeitet. Ansonsten wird eine Fehlermeldung ausgegeben, die den Namen der Ausgabedatei enthält.

4.2 Parsing

```
22    if err_file_empty:
23          x_values = []
24          y_values = [[[], [], []], [[], [], []], [[], [], []]]
25          #y_values[i][k][n]: i=aktuelle Iteration des Benchmarks,
             k=aktuelle MPI-Implementation, n=Benchmark-Wert
26          x_axis_name = ''
27          y_axis_name = ''
```

Wenn die Bedingungen zum Parsing erfüllt sind, werden alle benötigten Variablen zum Speichern der Daten erstellt. Die Namen der x- und y-Achsen sind einfache Strings, die x-Werte werden in eine Liste gespeichert.
Etwas komplizierter ist die Liste in Zeile 24 zum Speichern der y-Werte aufgebaut. Die Liste ist dreidimensional verschachtelt: In der äußersten Liste gibt es insgesamt 3 Einträge; für jede Iteration eine Liste. Darin befinden sich wiederum je 3 Listen; eine für jede Implementation. Die dort vorliegenden Listen enthalten schließlich eine Liste der x-Werte einer einzigen Benchmark-Ausführung.

```
29          with open(filename) as file:
30              print(filename)
31              current_mpi = 0
32              current_iteration = 0
33              extract_count = 0
34              try:
35                  extract_offset = parsing_data[filename][0]
36                  if len(parsing_data[filename]) > 1:
37                      x_axis_name = parsing_data[filename][1]
38                  if len(parsing_data[filename]) > 2:
39                      y_axis_name = parsing_data[filename][2]
40              except:
41                  extract_offset = 0
42
43              extract_data = False
```

Ab Zeile 29 wird die Datei geöffnet, dessen Name ausgegeben und weitere benötigte Daten werden gesetzt bzw. ermittelt. Die Variablen in Zeile 31-33 dienen als Zähler. Der anschließende try-Block versucht, aus dem in Zeile 6 definierten Dictionary den Offset der Extraktion bzw. der Zeilen abzufragen. Der try-Block ist deshalb notwendig, da nicht jeder Benchmark in dem Dictionary vertreten ist, sondern nur, wenn es einen Offset gibt. Zusätzlich werden in den nächsten Zeilen geprüft, wie lang die gespeicherten Listen sind, für den Fall, dass zusätzlich Achsennamen gespeichert wurden. Ist der Benchmark im Dictionary nicht vorhanden, ist der Offset gleich Null. Zudem wird in Zeile 43 der boolean definiert, der die Extraktion startet.

```
45              #Prüft auf Ende einer Implementation bzw. Iteration
46              for line in file:
47                  if line.strip() == '---':
48                      extract_data = False
49                      current_mpi += 1
50                  elif line.strip() == '===':
51                      extract_data = False
52                      current_mpi = 0
53                      current_iteration += 1
54
55                  #Normalerweise folgen auf Zeile mit 'OSU [Testname]' die
                    Werte, darum ab nächster Zeile extrahieren
56                  elif 'OSU' in line:
57                      if extract_offset > 0:
58                          extract_count = extract_offset
59                      else:
60                          extract_data = True
61
62                  #Falls Offset übergeben wurde, erst runterzählen bevor
                    Extraktion
63                  elif extract_count > 0:
64                      extract_count -= 1
65                      if extract_count == 0:
66                          extract_data = True
67
68                  elif extract_data == True: [...]
```

Der oben dargestellte Code-Abschnitt bestimmt, was in jeder Zeile der Ausgabedatei passieren soll. Es gibt insgesamt fünf größere Fälle.

1. Trennung der Implementationen durch „---“ (Z. 47f.)
2. Trennung der Iterationen durch „===“ (Z. 50f.)
3. „OSU“ in der Zeile vorhanden (Z. 56f.)
4. Extraktionszähler für den Offset ist noch größer als 0 (Z. 63f.)
5. Die Daten sollen extrahiert werden (Z. 68f.)

Betrachtet werden zunächst die ersten vier Fälle.

Im ersten Fall wird der Zähler der aktuellen MPI-Implementation erhöht, um die Werte in die nächste Liste hinzufügen zu können. Im zweiten Fall wird jener Zähler auf 0 zurückgesetzt und der Zähler der aktuellen Iteration erhöht.

Tritt der dritte Fall ein, soll als nächstes die Extraktion anfangen, sofern für den Benchmark kein Offset (größer 0) angegeben ist. Andernfalls wird dieser in einem Zähler festgehalten und muss heruntergezählt werden.

Dies findet im vierten Fall. Solange der Extraktionszähler größer als 0 ist, wird dieser pro Zeile verringert. Bei 0 angekommen, kann die Extraktion beginnen. Es folgt der fünfte Fall:

```
68                  elif extract_data == True:
69                      if line.strip() == '':
70                          extract_data = False
71                      elif x_axis_name == '' and y_axis_name == '': #Namen
                        müssen nur einmal für alle Impl. extrahiert warden
```

```
72                          x_axis_name, y_axis_name = line.split('  ',1)
73                          y_axis_name = y_axis_name.strip().split('  ',1)
                              [0]
74                          x_axis_name = x_axis_name.strip()
75                          y_axis_name = y_axis_name.strip()
76                      elif x_axis_name in line or y_axis_name in line:
77                          pass
78
79                      #Alle anderen Zeilen während Extraktion: 2 Werte pro
                        Zeile zur Liste der aktuellen Impl. Hinzufügen
80                      else:
81                          x, y = line.split('  ',1)
82                          y = y.strip().split('  ',1)[0]
83                          x, y = int(x.strip()), float(y.strip())
84                          if current_mpi == 0:
85                              if current_iteration == 0:
86                                  x_values.append(x)
87                              y_values[current_iteration][current_mpi].append(y)
88                          else:
89                              y_values[current_iteration][current_mpi].append(y)
```

Die eigentliche Extraktion findet in diesem Abschnitt statt. Die Zeilen 69 und 70 behandeln nur einen Sonderfall, bei dem die Zeileninhalt leer ist, da im Nachhinein festgestellt wurde, dass eines der Job scripts am Ende vor dem „done“ noch eine solche Zeile besaß. Statt den Job script zu bearbeiten und neu auszuführen, wurde dieser Sonderfall eingeführt.

Nach der Zeile mit dem Begriff „OSU“ bzw. dem Offset folgen in der Ausgabe die Bezeichner und Einheiten der Werte. Zunächst wird in Zeile 71 geprüft, ob diese nicht bereits gespeichert wurden, da dies pro Benchmark nur einmal notwendig ist und für jede Implementation/Iteration gelten. Falls nicht, wird die Zeile in zwei Teile gespalten, von allen Leerzeichen befreit und gleichzeitig den Variablen der Achsennamen zugeordnet. Als Separator für die Teilung wird nicht nur ein, sondern zwei Leerzeichen benutzt, da viele Bezeichner einzelne Leerzeichen enthalten können (Bsp.: Avg Latency).

Da manche Benchmarks mehr als einen Wert messen und entsprechend mehrere Bezeichner haben können, wird in Zeile 73 sichergestellt, dass nur das zweite Element gespeichert wird. Erreicht wird dies, indem der String der Variable „y_axis_name“ erneut geteilt wird. Das Teilen erzeugt eine Liste mit allen Teilstrings, von dem nur das erste Element ausgelesen wird. Somit funktioniert es auch, wenn es keine weiteren Teilstrings (Werte) gibt und die Liste nur ein Element enthält. Allerdings müssen vor der Teilung alle Leerzeichen entfernt werden, um sicherzustellen, dass das erste Element einen Inhalt besitzt und nicht aus leeren Leerzeichen besteht. Anschließend werden bei beiden Bezeichnern alle Leerzeichen entfernt, die mit der Teilung einhergegangen sind. Wurden die Bezeichner extrahiert, kann in allen weiteren Implementationen und Iterationen mithilfe der Zeilen 76/77 die Extraktion übersprungen werden.

In allen anderen Zeilen der Datei sind während der Extraktion die Werte zu finden. Die Extraktion dieser erfolgt analog zu der Extraktion der Bezeichner. Die einzigen Unterschiede sind, dass in Zeile 83 die extrahierten Werte zunächst Zeichen sind und in Zahlen umgewandelt werden müssen. Die x-Werte werden zu Integers und die y-Werte zu Floats, da letztere nicht zwingend ganzzahlig sind. Auch hier muss die Extraktion der x-Werte nur einmal erfolgen, wodurch sich die Zeilen 85-87 und 88-89 nur darin unterscheiden, dass in

erstere zusätzlich die x-Werte zu einer Liste hinzugefügt werden, wenn erstmalig eine Extraktion stattfindet. Die y-Werte werden in die multidimensionale Liste gespeichert und werden durch die Zähler der Iterationen und Implementationen entsprechend zugeordnet.

4.3 Berechnungen

```
92              #neue Listen für die statistisch relevanten Werte
93              #[i][k][n]: i = durchschnittlicher y-Wert, k=unterer Error-
                 Offset, n=oberer Error-Offset
94              mpich_y_val = [[],[],[]]
95              openmpi_y_val = [[],[],[]]
96              mvapich2_y_val = [[],[],[]]
```

Für die Berechnungen werden für jede Implementation neue verschachtelte Listen erstellt. Die erste Teilliste speichert jeweils alle Mittelwerte; die zweite und dritte Teilliste speichern je die größten Abweichung (Offset) nach unten und oben im Vergleich zu den Mittelwerten. Die Positionen innerhalb der Teillisten sind hierbei ausschlaggebend: Jeder Wert innerhalb einer Teilliste hängt mit den Werten gleicher Position der anderen beiden Teillisten zusammen (Bsp.: 2. Element aller drei Teillisten bilden zusammen den unteren, mittleren und oberen y-Wert für den 2. x-Wert).

```
97              #Durchschnitt, unteren und oberen Error-Offset bilden/speichern
98          for i in range(len(x_values)):
99              mpich_y1, mpich_y2, mpich_y3 = y_values[0][0][i],
                y_values[1][0][i], y_values[2][0][i]
100             openmpi_y1, openmpi_y2, openmpi_y3 = y_values[0][1][i],
                y_values[1][1][i], y_values[2][1][i]
101             mvapich2_y1, mvapich2_y2, mvapich2_y3 =  y_values[0][2][i],
                y_values[1][2][i], y_values[2][2][i]
102
103             mpich_mean = (mpich_y1 +  mpich_y2 +  mpich_y3) / 3
104             openmpi_mean = (openmpi_y1 +  openmpi_y2 +  openmpi_y3) / 3
105             mvapich2_mean = (mvapich2_y1 +  mvapich2_y2 +  mvapich2_y3) / 3
106
107             mpich_y_val[0].append(mpich_mean)
108             mpich_y_val[1].append(abs(min(mpich_y1, mpich_y2, mpich_y3)
                - mpich_mean))
109             mpich_y_val[2].append(abs(max(mpich_y1, mpich_y2, mpich_y3)
                - mpich_mean))
110
101             openmpi_y_val[0].append(openmpi_mean)
102             openmpi_y_val[1].append(abs(min(openmpi_y1, openmpi_y2,
                openmpi_y3) - openmpi_mean))
103             openmpi_y_val[2].append(abs(max(openmpi_y1, openmpi_y2,
                openmpi_y3) - openmpi_mean))
```

```
104
105             mvapich2_y_val[0].append(mvapich2_mean)
106             mvapich2_y_val[1].append(abs(min(mvapich2_y1, mvapich2_y2,
                mvapich2_y3) - mvapich2_mean))
107             mvapich2_y_val[2].append(abs(max(mvapich2_y1, mvapich2_y2,
                mvapich2_y3) - mvapich2_mean))
```

Die Berechnung erfolgt über eine for-Schleife mit einer Anzahl an Durchläufen, die der Anzahl an x-Werten entspricht. Der Vorgang wird beispielhaft für die Implementation MPICH erläutert, wobei in jedem Schritt das gleiche für die anderen Implementationen geschieht. In Zeile 99 werden die y-Werte aus der Liste der Parsing-Phase für eine bessere Verständlichkeit neuen Variablen zugeordnet. Pro Schleifen-Iteration gibt es drei y-Werte: y1, y2 und y3. Sie sind der i-te Wert aller drei Benchmark-Iterationen (in diesem Fall für MPICH).

Der Mittelwert errechnet sich durch das Addieren aller y-Werte und dem Teilen durch die Anzahl der Summanden (Z. 103). Anschließend wird der Mittelwert zur neuen Liste der MPICH-Werte hinzugefügt. Der untere und obere Offset wird errechnet, indem der Mittelwert einmal vom Minimum und einmal vom Maximum der drei Werte abgezogen wird. Davon ist nur der absolute Wert gewollt, da Matplotlib stets einen positiven Offset erwartet.

4.4 Plotting

```
105         plt.figure(figsize=(12,5))
106         plt.errorbar(x_values, mpich_y_val[0], yerr=[mpich_y_val[1],
             mpich_y_val[2]], fmt='r-', capsize=3, capthick=2, label='MPICH')
107         plt.errorbar(x_values, openmpi_y_val[0], yerr=[openmpi_y_val[1],
             openmpi_y_val[2]], fmt='g-.', capsize=3, capthick=2, label='OpenMPI')
108         plt.errorbar(x_values, mvapich2_y_val[0], yerr=[mvapich2_y_val[1],
             mvapich2_y_val[2]], fmt='b--', capsize=3, capthick=2,
             label='MVAPICH2')
109         plt.xlabel(x_axis_name)
110         plt.ylabel(y_axis_name)
111         plt.xscale('log')
112         plt.legend()
113         plt.rc('axes', titlesize=18)
114         plt.rc('axes', labelsize=18)
115         plt.rc('xtick', labelsize=16)
116         plt.rc('ytick', labelsize=16)
117         plt.rc('legend', fontsize=16)
118         plt.tight_layout()
119         plt.grid(True)
```

Ein Großteil der Funktionen beim Plotting dienen zur Individualisierung der Graphen, wie z.B. die Gesamtgröße, Achsenbeschriftungen, Achsenskalierung, Schriftgrößen und Ähnlichem. Die Daten wurden im Vorfeld in einer geeigneten Struktur gespeichert, die von

Matplotlib verarbeitet werden kann. Die Umwandlung der Daten in Graphen erfolgt in den Zeilen 106-108. Die ersten drei Argumente sind die Listen der x-Werte, y-Werte und einer Liste mit den unteren Abweichungen an erster und den oberen Abweichungen an zweiter Stelle. Alle weiteren Argumente haben nur visuelle Änderungen zur Folge.

```
121         file_as_png = f'{filename[:-4]}.png'
122         png_folder = 'benchmark_plots'
123         file_png_path = f'{png_folder}/{file_as_png}'
124 
125         if not os.path.exists(png_folder):
126             os.mkdir(png_folder)
127         if os.path.isfile(file_png_path):
128             os.remove(file_png_path)
129             time.sleep(1)
130         plt.savefig(file_png_path)
131         plt.close()
132 
133     print('done')
```

Zuletzt müssen die Graphen noch gespeichert werden. In den Zeilen 121-123 werden der Dateiname mit „.png“ Endung, der Name des Ordners und der Pfad der Datei mitsamt Ordner festgelegt. Falls der Ordner nicht existiert, wird er erstellt. Sollte bereits eine gleichnamige Datei in dem Ordner vorhanden sein, wird diese entfernt. Der Plot wird dann gespeichert und geschlossen. Eine Ausgabe im Terminal bestätigt das Ende des Skripts.

Einige wenige Benchmarks bestanden aus einer einzigen Messung pro Ausführung (cas_latency, fop_latency, barrier, ibarrier). Bei diesen Benchmarks wurden die Mittelwerte manuell ermittelt und in Matplotlib eingetragen.

5 Testumgebung und Methodik

Um die Ergebnisse besser zu verstehen und die Reproduzierbarkeit zu erhöhen, werden in diesem Abschnitt weitere, möglicherweise relevante Informationen zu den Clusterknoten, den verwendeten Software-Versionen und der Methodik festgehalten.

5.1 Technische Daten der Cluster Nodes

Partition: west

CPU (Ausgabe von lscpu):

```
Architecture: x86_64
CPU op-mode(s): 32 bit, 64 bit
Byte Order: Little Endian
CPU(s): 24
Online CPU(s) list: 0 23
Thread(s) per core: 2
Core(s) per socket: 6
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 44
Model name: Intel(R) Xeon(R) CPU X5650 @2.67GHz
Stepping: 2
CPU MHz: 2399.347
CPU max MHz: 2667,0000
CPU min MHz: 1600,0000
BogoMIPS: 5333.73
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 12288K
NUMA node0 CPU(s): 05,12-17
NUMA node1 CPU(s): 6-11,18-23
```

RAM: 12GB
Netzwerkverbindung: Ethernet

5.2 Software Builds

Für jede MPI-Implementationen werden im Folgenden alle verfügbaren Flags und Argumente, mit denen die Implementationen installiert wurden, aufgelistet. Zusätzlich wird

ein einzigartiger Hash angegeben. Alle Optionen, die zu dem Zeitpunkt von der Standardkonfiguration abwichen und manuell gesetzt wurden, werden blau hervorgehoben.

MPICH:

```
mpich@3.3%gcc  device=ch3 +hydra netmod=tcp
patches=c7d4ecf865dccff5b764d9c66b6a470d11b0b1a5b4f7ad1ffa61079ad6b5dede
+pci pmi=pmi +romio ~slurm ~verbs +wrapperrpath
```

```
Hash: 2ld7eyt3jiooqapu3duefu53ysmlwjhb
```

OpenMPI:

```
openmpi@3.1.4%gcc ~cuda +cxx_exceptions fabrics=none ~java +legacylaunchers
~memchecker +pmi schedulers=slurm ~sqlite3 +thread_multiple +vt
```

```
Hash: ewoohlexvxsop5ok3vto726m6ms4hh3n
```

Für OpenMPI wurde trotz eines möglicherweise verlangsamenden Effekts die Option „thread_multiple“ aktiviert, da es einerseits in realem Gebrauch verwendet werden könnte und da bei MVAPICH2 standardmäßig die Option aktiviert ist, sodass ein besserer Vergleich ermöglicht wurde.

MVAPICH2:

```
mvapich2@2.3.1%gcc ~alloca ch3_rank_bits=32 ~cuda ~debug fabrics=nemesis
file_systems=auto process_managers=auto +regcache threads=multiple
```

```
Hash: izxc4kvxdj4iueizqgugiq7s262bsef6
```

Benchmarks:

OSU-Micro-Benchmarks-5.6.1 (Download: http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.6.2.tar.gz)
Weitere Details zu den Benchmarks sowie Abkürzungen können der Webseite entnommen werden.

5.3 Methodik

Wie in den vorangegangenen Abschnitten deutlich wurde, wurde jeder Benchmark pro MPI-Implementation drei Mal ausgeführt. Dabei wurden die Implementationen bewusst abwechselnd getestet, statt jede Implementation nacheinander abzuarbeiten, damit sich eventuelle Beeinträchtigungen der Knoten (Rechenleistung, Verbindung) mit höherer Wahrscheinlichkeit auf alle Implementationen verteilten und bemerkbar machten.

Es wurden alle Benchmarks zweimal getestet: Einmal mit Prozessen, die global (knotenübergreifend) liefen und einmal mit allen Prozessen lokal auf einem einzigen Knoten.

Letzteres verdeutlicht Unterschiede in der Effizienz der Implementationen unabhängig vom Datenaustausch innerhalb des Netzwerks. Da diese Entscheidung erst im späteren Verlauf des Projekts getroffen wurde, wurde ein weiterer Bash script erstellt, in dem die neuen Argumente für die Benchmarks gesetzt wurden, sodass die in den Job scripts angegebenen Argumente überschrieben wurden und ein einzelner Knoten verwendet werden konnte.

Die One-sided und Point-to-Point Benchmarks wurden stets mit 2 Prozessen gestartet. Die Collective Benchmarks liefen bei globaler Ausführung mit 3 und bei lokaler Ausführung mit 2 Prozessen. Das lokale Ausführen mit 3 Prozessen führte zu folgendem Fehler:

```
Fatal error in MPI_Init: Other MPI error, error stack:
MPIR_Init_thread(490).....:
MPID_Init(395)............: channel initialization failed
MPIDI_CH3_Init(104).......:
MPID_nem_init(272)........:
MPIDI_CH3I_Seg_commit(369): PMI_KVS_Get returned -1
Fatal error in MPI_Init: Other MPI error, error stack:
MPIR_Init_thread(490).....:
MPID_Init(395)............: channel initialization failed
MPIDI_CH3_Init(104).......:
MPID_nem_init(272)........:
MPIDI_CH3I_Seg_commit(369): PMI_KVS_Get returned -1
Fatal error in MPI_Init: Other MPI error, error stack:
MPIR_Init_thread(490).....:
MPID_Init(395)............: channel initialization failed
MPIDI_CH3_Init(104).......:
MPID_nem_init(272)........:
MPIDI_CH3I_Seg_commit(369): PMI_KVS_Get returned -1
```

Weiterhin wurden Benchmarks, die die gleiche Anzahl an Knoten benötigten, stets auf denselben Knoten ausgeführt. Es wurden folgende Knoten benutzt:

1 Knoten: west[2]
2 Knoten: west[3-4]
3 Knoten: west[5-7]

Es hätten auch alle Benchmarks auf den Knoten west[2-4] ausgeführt werden können, jedoch wäre dann eine parallele, zeitsparendere Ausführung kaum möglich gewesen.

6 Ergebnisse

Die Ergebnisse der globalen und lokalen Benchmarks werden separat betrachtet. Zudem wird nicht jeder einzelne Benchmark dargestellt. Stattdessen werden möglichst Graphen mit ähnlichen Verläufen gruppiert und stellvertretend ausgewertet. Im Anhang finden sich alle Graphen in alphabetischer Reihenfolge.

Weiterhin werden hier die Arten der Benchmarks aufgelistet:

One-sided:
acc_latency, cas_latency, fop_latency, get_acc_latency, get_bw, get_latency, put_bibw, put_bw, put_latency

Point-to-Point:

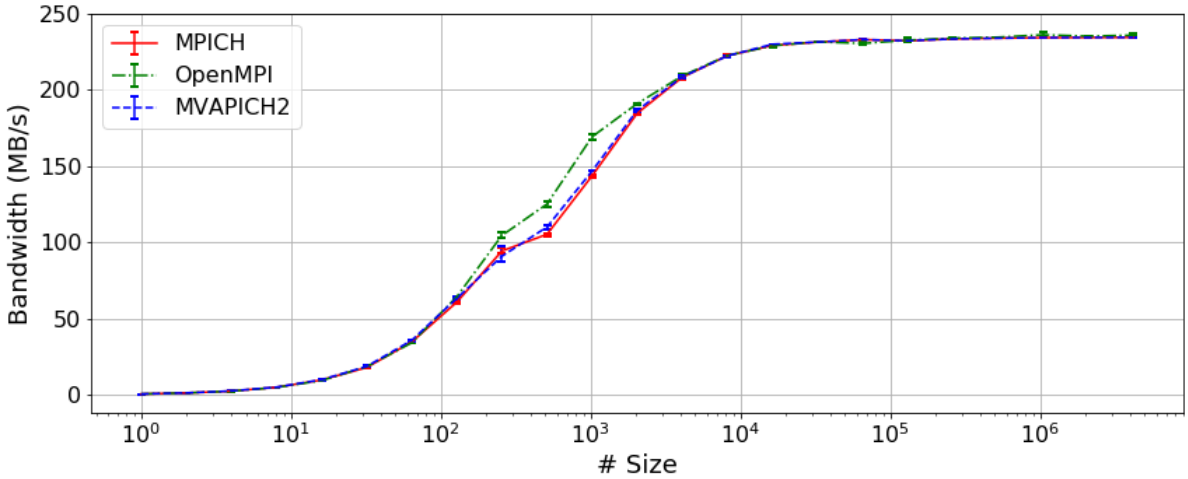
bibw, bw, latency, latency_mt, mbw_mr, multi_lat

Collective:

Blocking: allgather, allgatherv, allreduce, alltoall, alltoallv, barrier. bcast, gather, gather, reduce, reduce_scatter, scatter, scatter
Non-blocking: iallgather, iallgatherv, iallreduce, ialltoall, ialltoallv, ialltoallw, ibarrier, ibcast, igather, igatherv, ireduce, iscatter, iscatterv

6.1 Globale Benchmarks

bidirectional_bandwith (bibw):

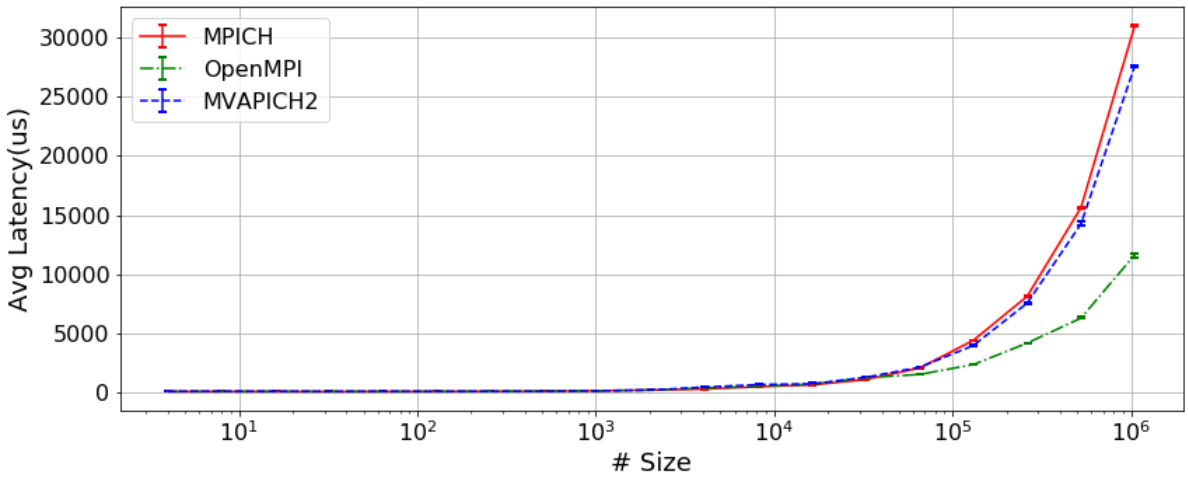


Bei diesem Benchmark ist erkennbar, dass grundsätzlich alle drei MPI-Implementationen gleichschnell sind. Bei Größen im hohen zweistelligen bis dreistelligen Bereich ist OpenMPI minimal, jedoch nicht signifikant schneller. Die Fehlerbalken sind kaum vorhanden und die

Messungen entsprechend stabil. Es kann kein klarer Performancevorteil einer bestimmten MPI-Implementation erkannt werden.

Die folgenden Benchmarks weisen ähnliche Graphen auf: get_acc_latency, iallgather, iallgatherv, ialltoall.
Ihre Fehlerbalken sind unter Umständen etwas größer, aber insgesamt sind die Unterschiede zwischen den Implementationen gering.

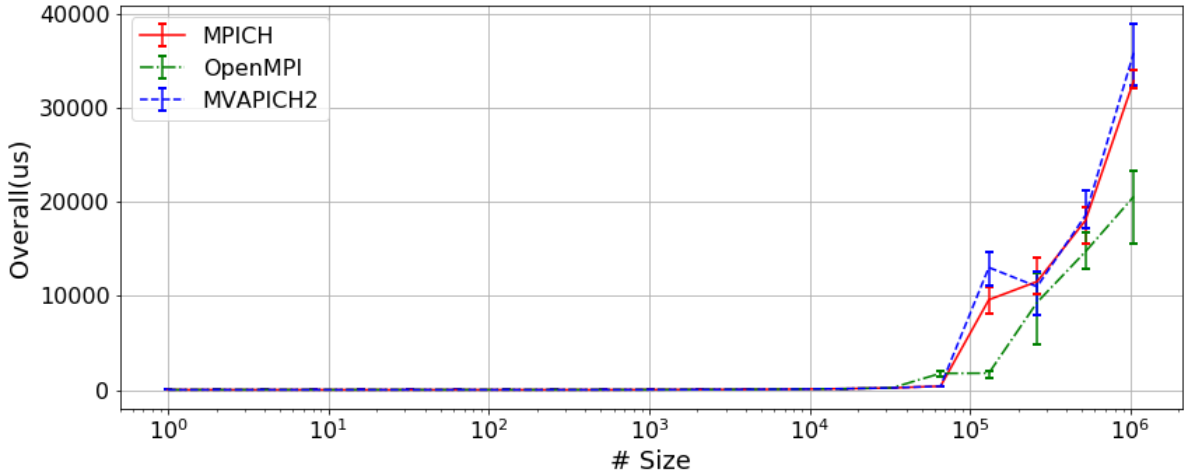
allreduce:



Der Verlauf dieses Graphen ist weitgehend unspektakulär, jedoch zeigen sich zunehmend Unterschiede ab Größen von ca. 10^5. Die Werte von MPICH und MVAPICH2 sind sich sehr ähnlich, die Latenzzeit von OpenMPI hingegen bleibt vergleichsweise gering und erreicht in diesem Fall einen bis zu ca. 2,3-mal besseren Wert, als die anderen beiden Implementationen. Somit kann hier von einem klaren Performancevorteil von OpenMPI ausgegangen werden.

Die Graphen folgender Benchmarks zeigen einen ähnlichen Verlauf: acc_latency, allgather, allgatherv, alltoall, alltoallv, bcast, get_latency, iallreduce, ialltoallv, ialltoallw, ibcast, ireduce, iscatter, iscatterv, latency, latency_mt, multi_lat, put_latency, reduce_scatter, scatter, scatterv.
Die Schwankungen sind meist gering. Die Performancevorteile fallen unterschiedlich hoch aus, jedoch ist immer ein Trend zu Gunsten von OpenMPI zu erkennen.

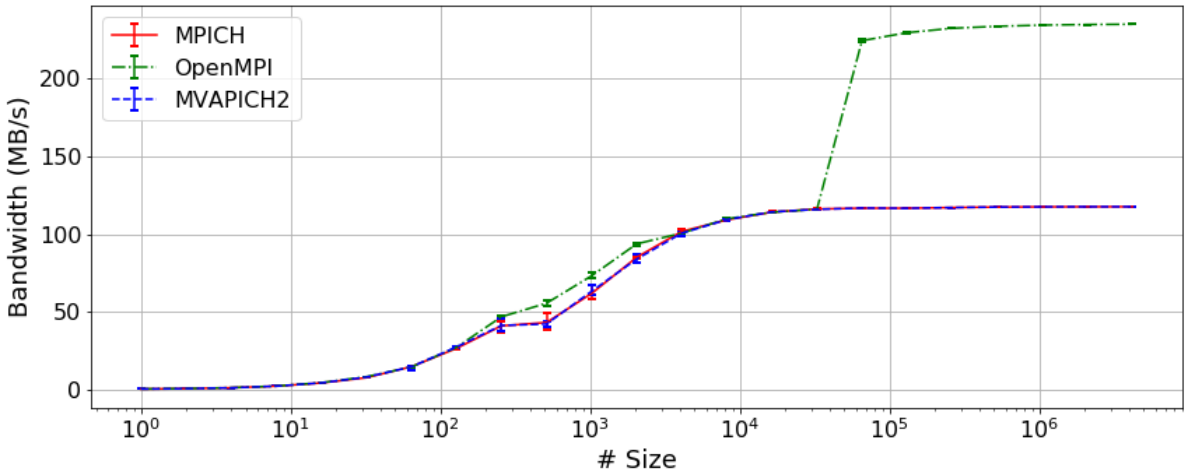
igatherv:



Dieser Graph ist ähnlich zu dem Vorherigen, jedoch sind die Fehlerbalken wesentlich größer. Je nach Schwankung, können die Implementationen gleich oder unterschiedlich schnell sein. Dennoch verläuft der Graph von OpenMPI tendenziell unterhalb derer der anderen Implementationen, insbesondere bei Größen um 10^5 und 10^6. Auch wenn die Performance inkonsistenter ist, ist erneut die beste Performance OpenMPI zuzuschreiben.

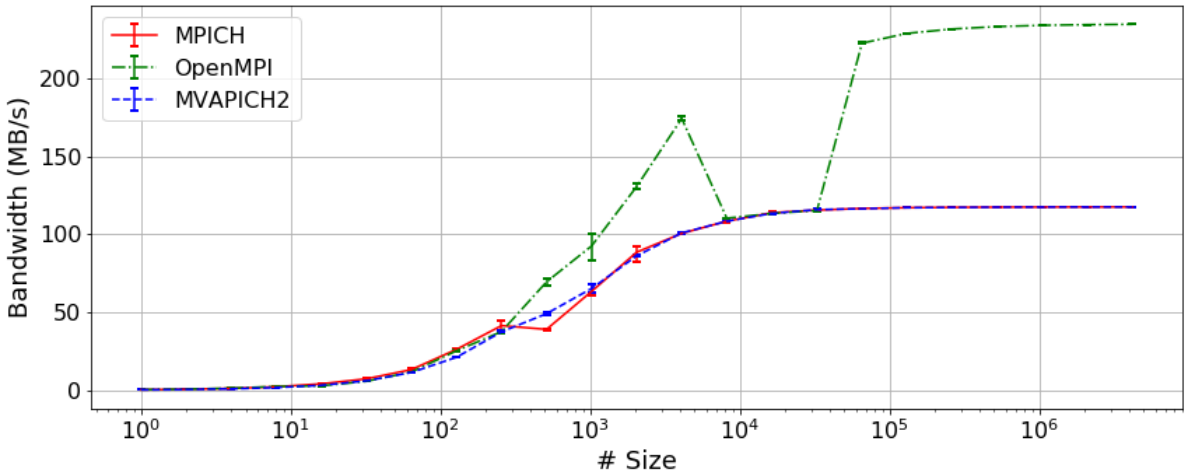
Andere Varianten von gather erzeugen ähnliche Resultate: gather, gatherv, igather.
Bei gatherv ist der Leistungsvorteil von OpenMPI konsistenter und deutlicher. Bei gather und igather ist dieser vorranging im Größenbereich von 10^6 erkennbar.

bandwith (bw):



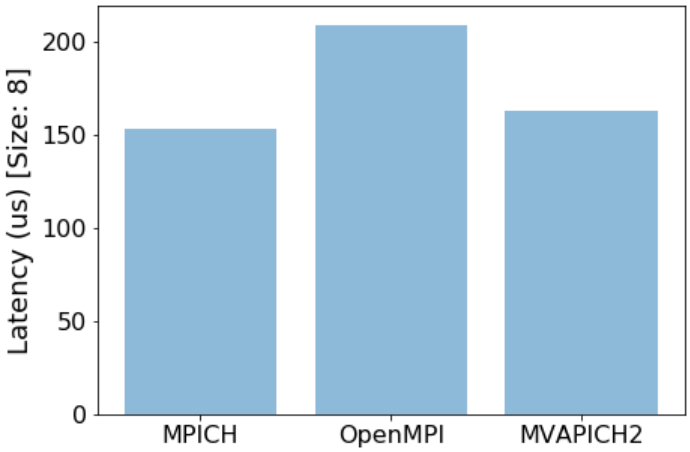
Die Ergebnisse dieses Benchmarks sind etwas interessanter. Wie bei beim ersten Graphen zu „bibw“ sind zunächst kaum Unterschiede zu erkennen. Bei einem Schritt in der Größe von 32768 zu 65536 verdoppelt sich fast plötzlich die Bandbreite von OpenMPI. Hier wird das Hinzuschalten eines zweiten Network-Interface vermutet. Somit hat OpenMPI ab einer bestimmten Größe einen klaren Performancevorteil. Der Graph von „mbw_mr“ verläuft analog zu diesem. Auch „get_bw“ verhält sich ähnlich, wobei der Effekt wesentlich früher einsetzt und gradueller verläuft.

put_bw:



Dieser Graph besitzt ähnliche Merkmale wie der Vorangegangene und sieht fast so aus wie der Graph von „get_bw“. Merkwürdig ist hierbei der plötzliche Leistungsverlust bei OpenMPI bei einer Größe von ca. 10^4, als wäre vorübergehend eine Umstellung von zwei Network-Interfaces auf eines geschehen. Ab einer Größe von 65536 normalisieren sich die Werte wieder und sind ungefähr doppelt so hoch. Warum dies Auftritt, ist nicht klar. Der Benchmark „put-bibw“ verhält sich gleich, allerdings mit etwas größeren Fehlerbalken zum Ende hin.

cas_latency:

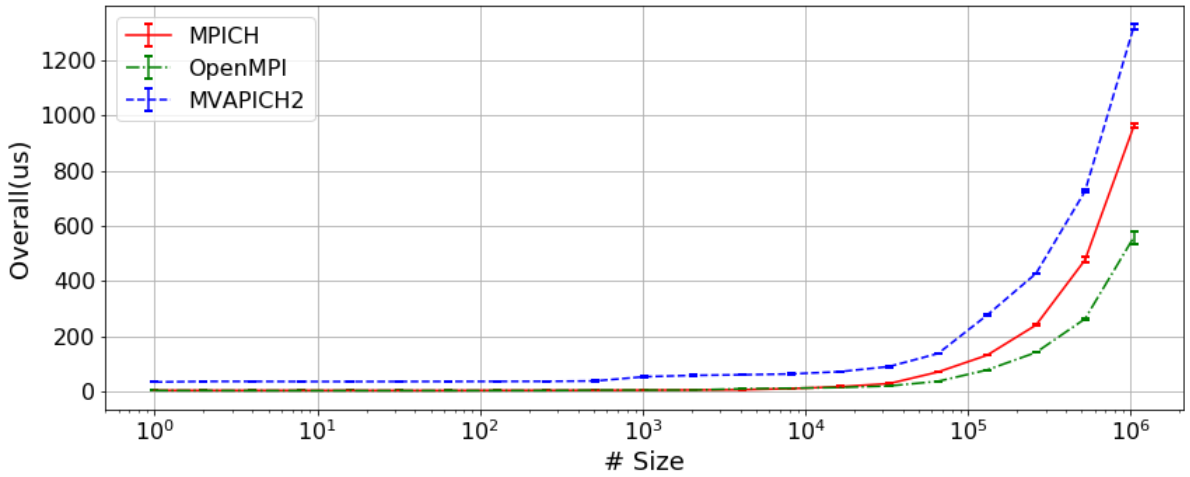


Dies ist ein Benchmark, bei dem nur für eine einzelne Größe eine Messung durchgeführt wird. Auch hier wurden drei Messungen durchgeführt und der Mittelwert bestimmt. In diesem Fall besitzt OpenMPI die höchste Latenz, während MPICH und MVAPICH2 eine knapp 25% geringere Latenz aufweisen und sich nur geringfügig unterscheiden. Weitere Benchmarks mit Balkendiagrammen sind: fop_latency, barrier und ibarrier. Das Diagramm von „fop_latency“ sieht diesem hier ähnlich. Bei „barrier“ und „ibarrier“ hingegen sind alle Implementation fast gleichschnell.

6.2 Lokale Benchmarks

Bei den lokalen Benchmarks gibt es nur einen Benchmark, bei dem alle Implementationen nahezu gleichschnell sind, nämlich „get_acc_latency“. Die Benchmarks „put_bibw“, „put_bw“ und „get_bw“ sehen fast genauso aus wie bei den globalen Messungen. Meist ergibt sich jedoch folgendes Bild:

ialltoall:

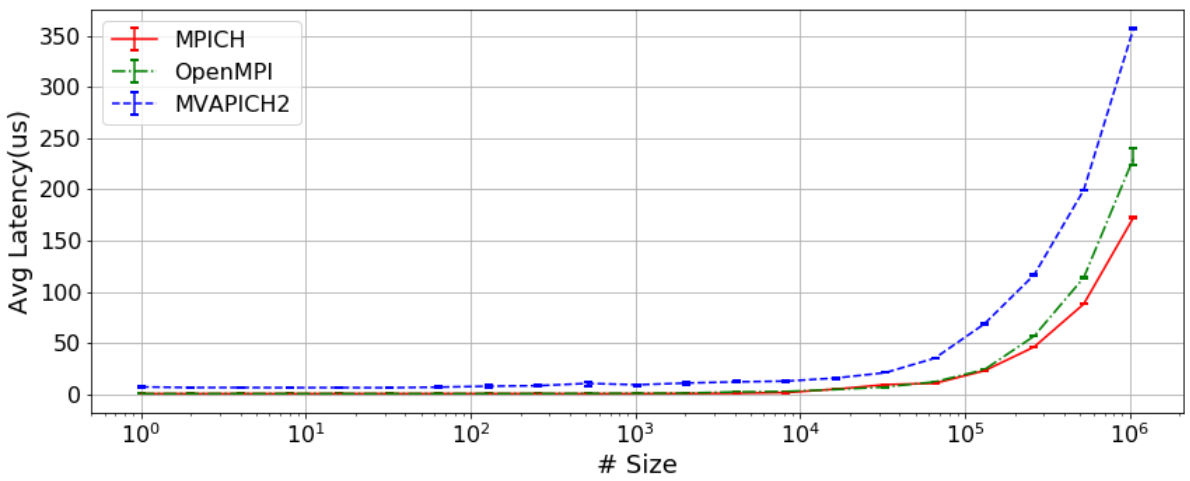


Zu sehen ist hier, dass bereits von Beginn an MPICH und OpenMPI schneller sind, als MVAPICH2. Im weiteren Verlauf werden auch Unterschiede zwischen MPICH und OpenMPI deutlich, sodass OpenMPI klar die besten Werte erreicht.

Einen ähnlichen Verlauf weisen viele der Benchmarks auf: acc_latency, allgather, allgatherv, alltoall, alltoallv, get_latency, iallgather, iallgatherv, ialltoall, ialltoallv, ialltoallw, ibcast, igather, igatherv, ireduce, iscatter, iscatterv, put_latency, reduce.

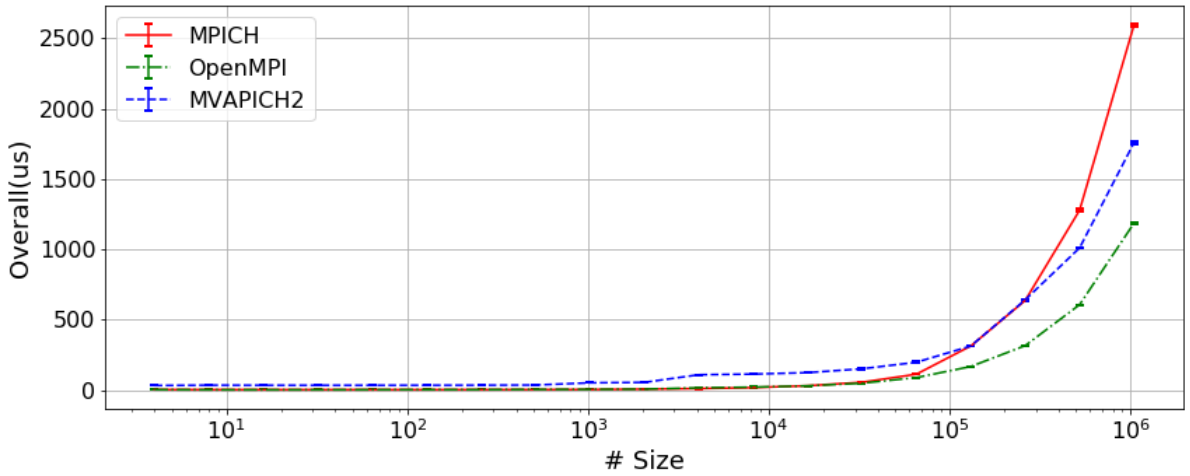
Es ist zwar in vielen, allerdings nicht in allen genannten Benchmark der Fall, dass MPICH schneller ist als MVAPICH2, dafür aber mindestens gleichschnell. Dennoch haben die Benchmarks gemeinsam, dass sie mit OpenMPI insgesamt die beste Performance liefern.

scatter:



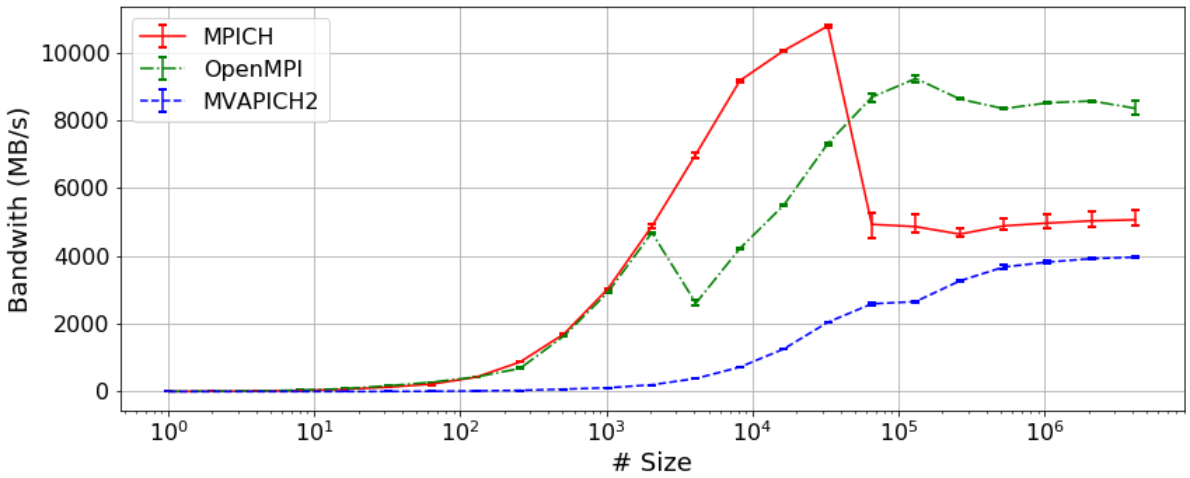
Dies ist einer von zwei Fällen, wo MPICH konsistent eine bessere Performance lieferte, als OpenMPI. Der andere Fall ist beim Benchmark „bcast“. Dennoch kann MPICH keine so großen Performancevorteile aufweisen, wie es OpenMPI in vielen anderen Benchmarks tut.

iallreduce:



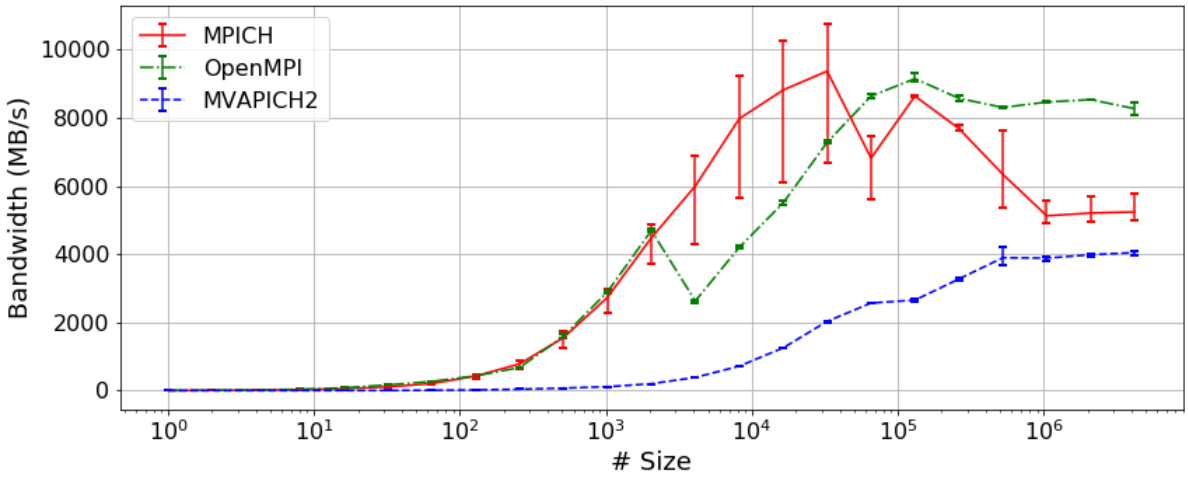
Wie hier zu sehen ist, gibt es auch Benchmarks, in denen MPICH zumindest bei höheren Werten deutlich langsamer ist, als die anderen Implementationen. Der einzige andere Benchmark, wo dies beobachtet werden kann, ist „allreduce“.

mbw_mr:



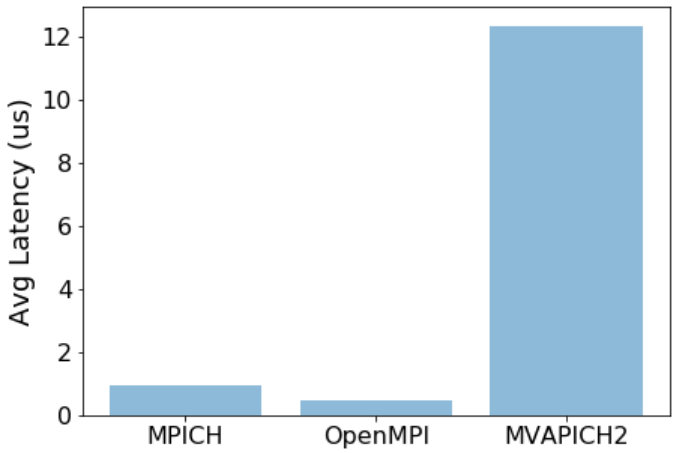
Bei diesem Benchmark werden mitunter die größten Diskrepanzen sichtbar. MVAPICH2 ist bei mittleren Größen die mit Abstand langsamste Implementation. MPICH und OpenMPI haben abwechseln die höhere Performance; MPICH bei mittelhohen Größen, OpenMPI zum Ende hin. Gleiches gilt für „bibw“, wobei dort die Performance von MPICH weniger stark ausreißt als in diesem Beispiel.

Beim Benchmark „bw“ ist ein ähnlicher Trend zu sehen, allerdings treten hier bei MPICH von allen getesteten Benchmarks die größten Fehlerbalken auf:



Da die Implementationen pro Iteration abwechselnd getestet wurden, ist ein temporärer Fehler beim verwendeten Knoten unwahrscheinlich, sodass vermutlich die Implementation eine inkonsistente Leistung erbringt.

barrier:



Bei den Benchmark mit Einzelmessungen sahen „cas_latency“ und „fop_latency“ ähnlich aus wie bei der globalen Messung, allerdings gab es einen signifikanten Unterschied beim hier dargestellten „barrier“ sowie „ibarrier“. Die Implementation MVAPICH2 besitzt eine über 12 Mal langsamere Latenzzeit, wodurch MVAPICH2 mit diesem Benchmark die mitunter schlechteste Performance liefert.

7 Fazit

Es gibt keine Implementation, die konsequent in allen Benchmarks die beste Performance erzielt. Dennoch geht in den meisten Benchmarks OpenMPI als Performancesieger hervor, was wahrscheinlich auch dem Umstand zu verdanken ist, dass OpenMPI scheinbar beide verfügbaren Network-Interfaces benutzt. Die Fälle, in denen MPICH schneller ist als OpenMPI, sind zu selten, als dass sich MPICH als Allround-Implementation lohnen würde. Selbst die Benchmarks, in denen MPICH schneller war, konnten meist geringere Performancesteigerungen erzielt werden, als mit OpenMPI gegenüber der anderen Implementationen.

MVAPICH2 besaß vereinzelt eine gute Latenzzeit bei Benchmarks mit Einzelmessungen (cas_latency, fop_latency), doch insgesamt wies MVAPICH2 die langsamste Performance auf und das sowohl bei den globalen als auch bei den lokalen Benchmarks, was auf eine ineffiziente Implementierung deuten könnte.

Ein Fazit kann nur für die hier angegebene Konfiguration der Implementationen angegeben werden, da sich diese in anderen Konfigurationen drastisch ändern könnte. Es wird deutlich, dass OpenMPI für den alltäglichen Einsatz die beste Wahl ist. Wenn im Voraus bekannt ist, welche Funktionen ein Programm primär benutzt, können andere Implementationen in Erwägung gezogen werden, die in jenem Gebiet eine bessere Leistung bieten.

8 Quellen

- MPI: A Message-Passing Interface Standard Version 3.1, Message Passing Interface Forum, June 4, 2015, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

- Grundlagen zu MPI, Martin Lanser, Mathematisches Institut Universität zu Köln, October 24, 2017, http://www.numerik.unikoeln.de/sites/numerik/uebungen/hpc_1718/Grundlagen_MPI.pdf

- Spack, https://spack.readthedocs.io/en/latest/getting_started.html, 28.09.2019

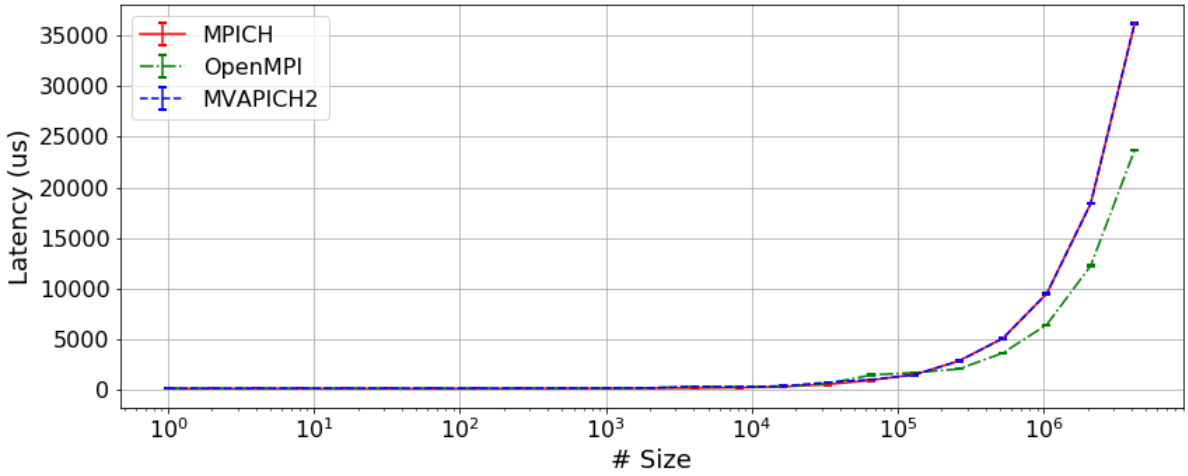
- Github (Spack Wiki), https://github.com/spack/spack/wiki, 28.09.2019

- Slurm, https://slurm.schedmd.com/overview.html, 29.09.2019

9 Anhang

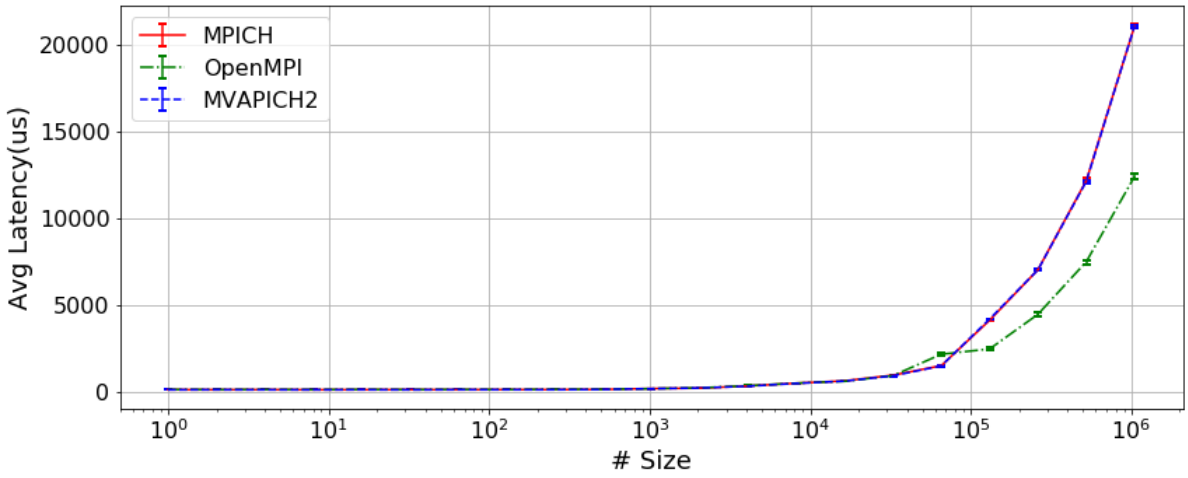
9.1 Graphen global

acc_latency:

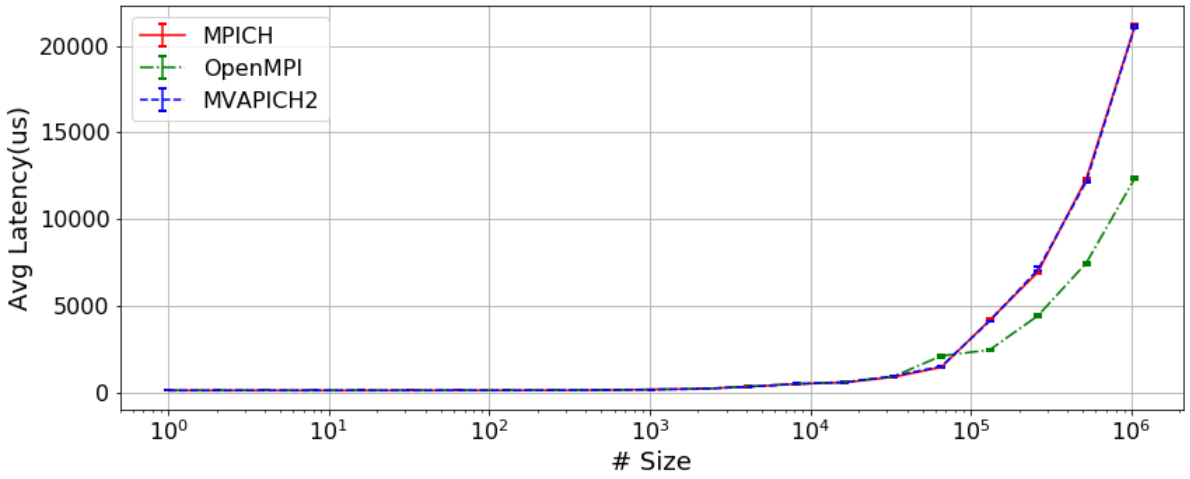


---

allgather:

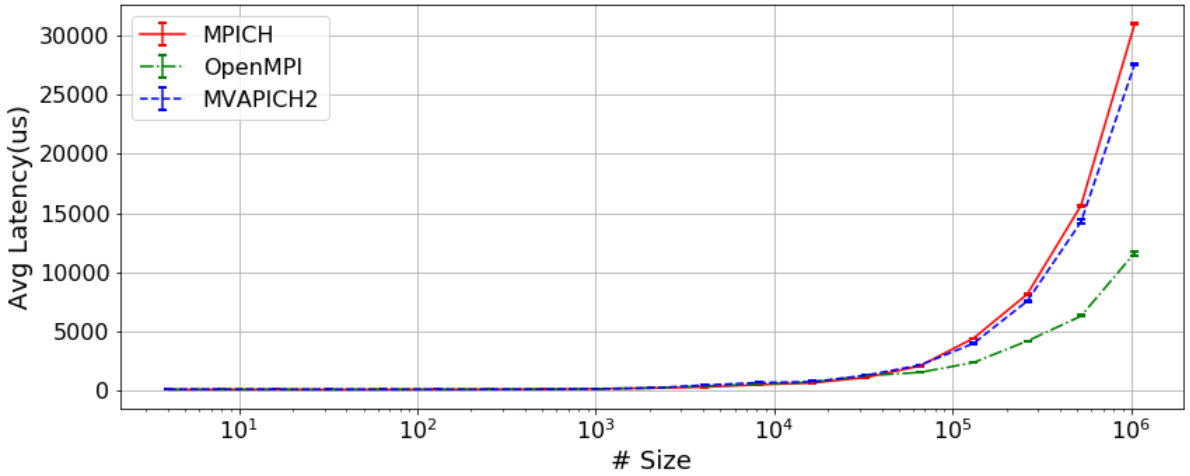


allgatherv:



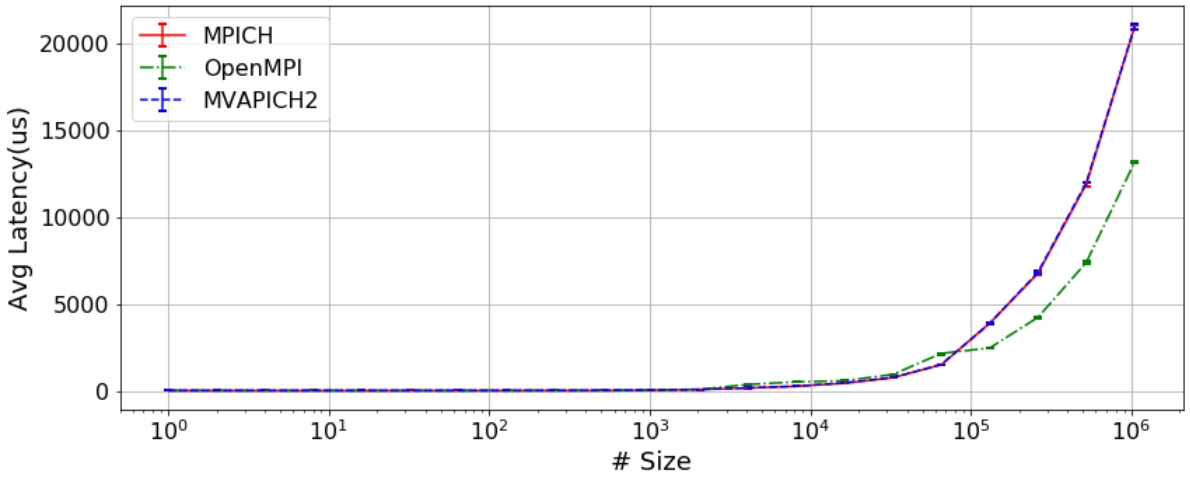
---

allreduce:

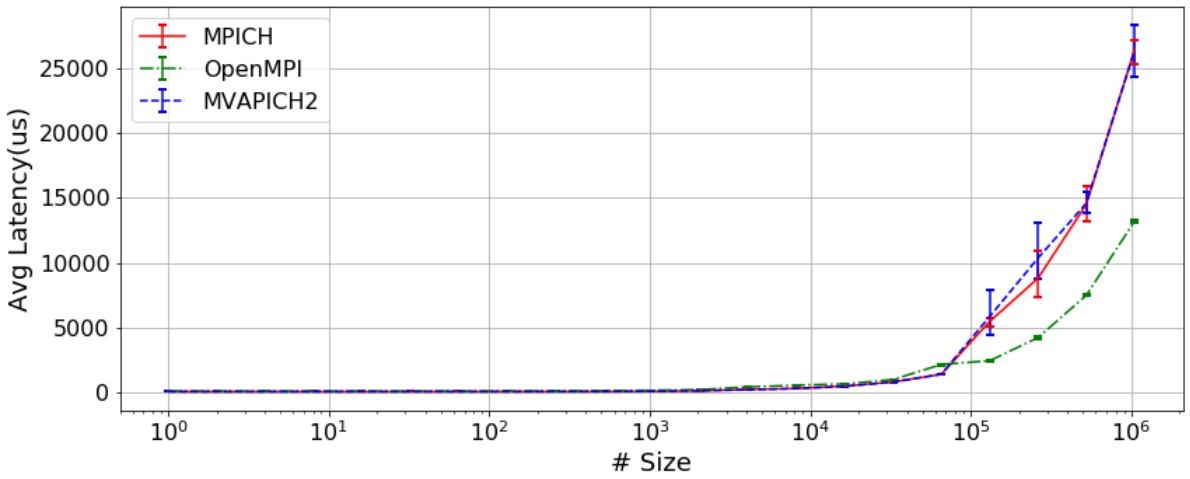


---

alltoall:

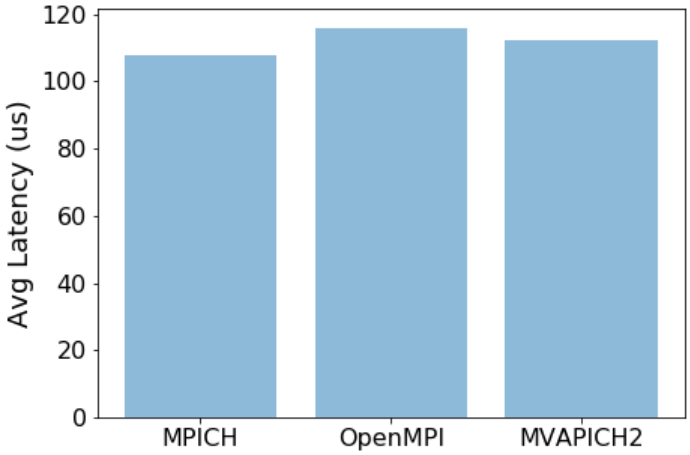


alltoallv:



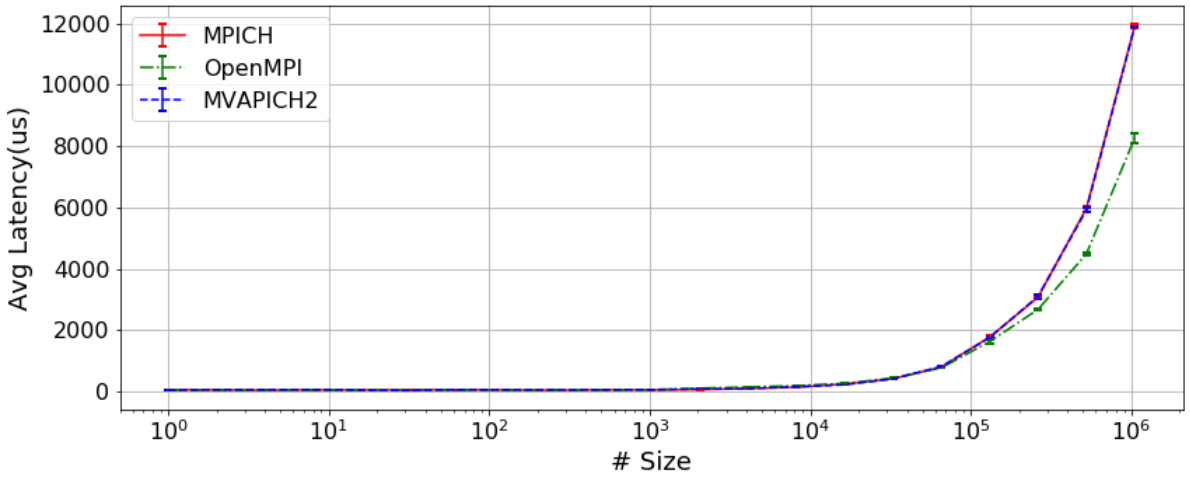
---

barrier:

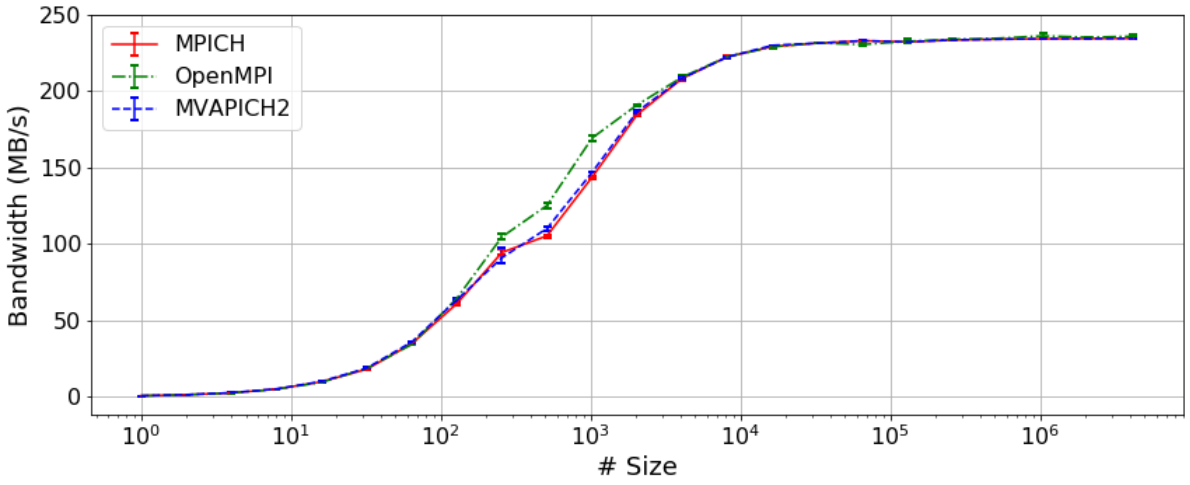


---

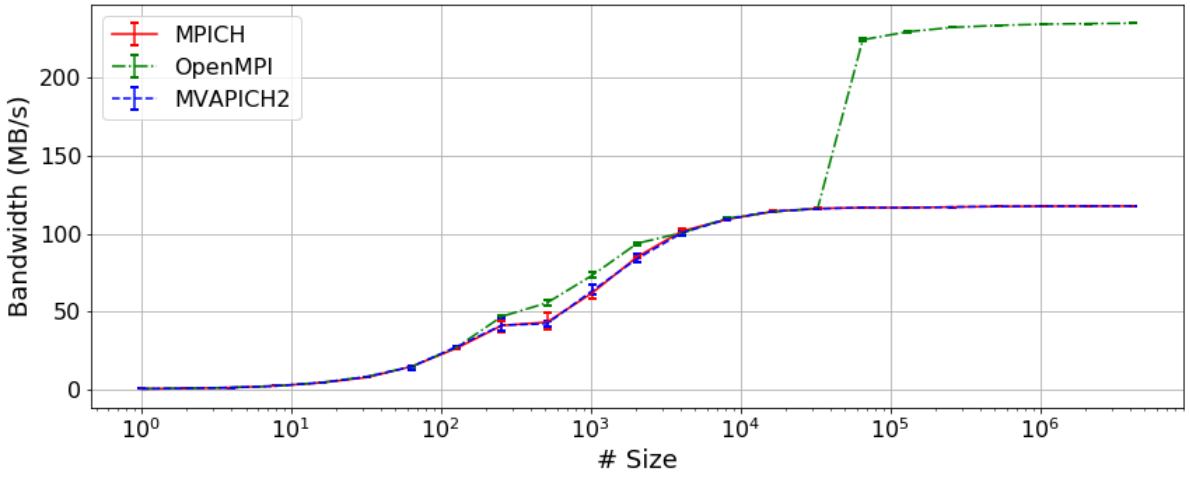
bcast:



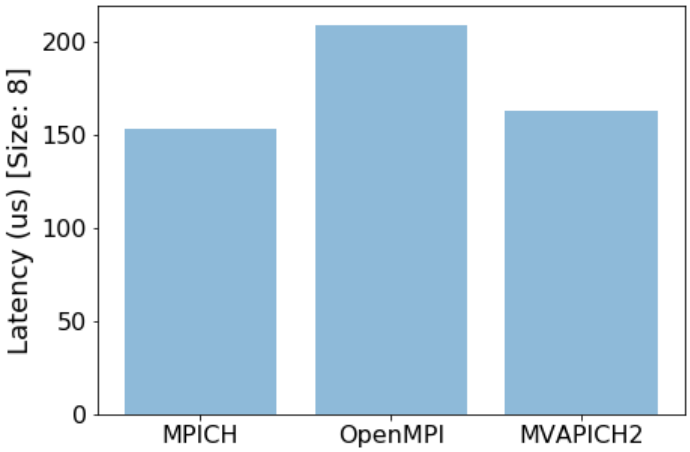
bibw:



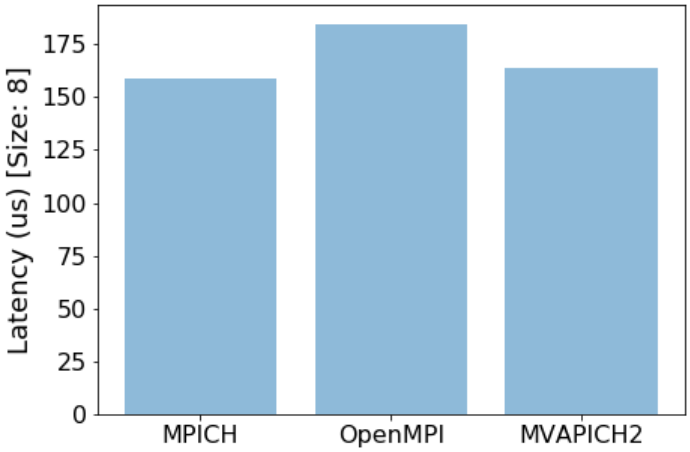
bw:



cas_latency:

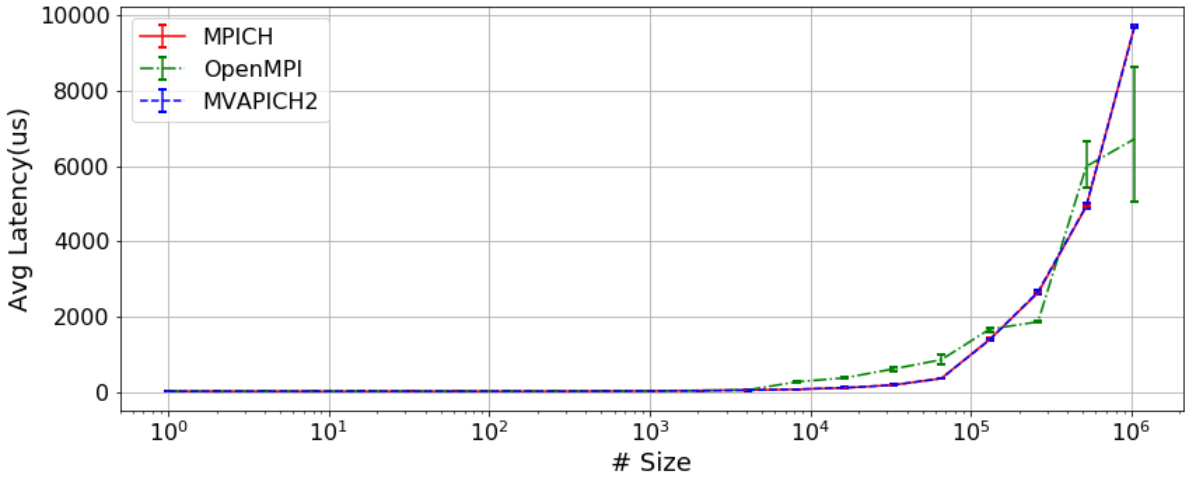


fop_latency:



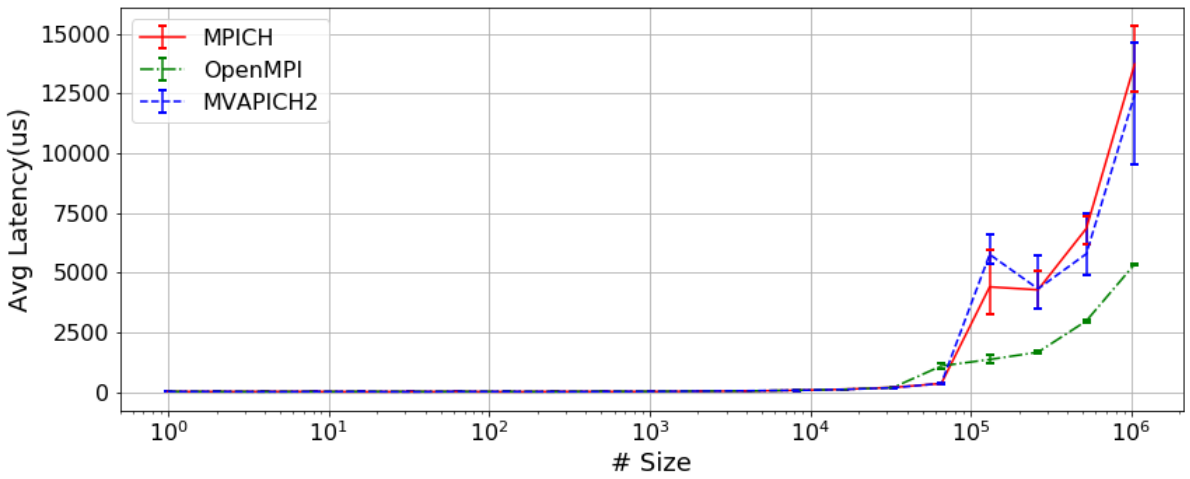
---

gather:

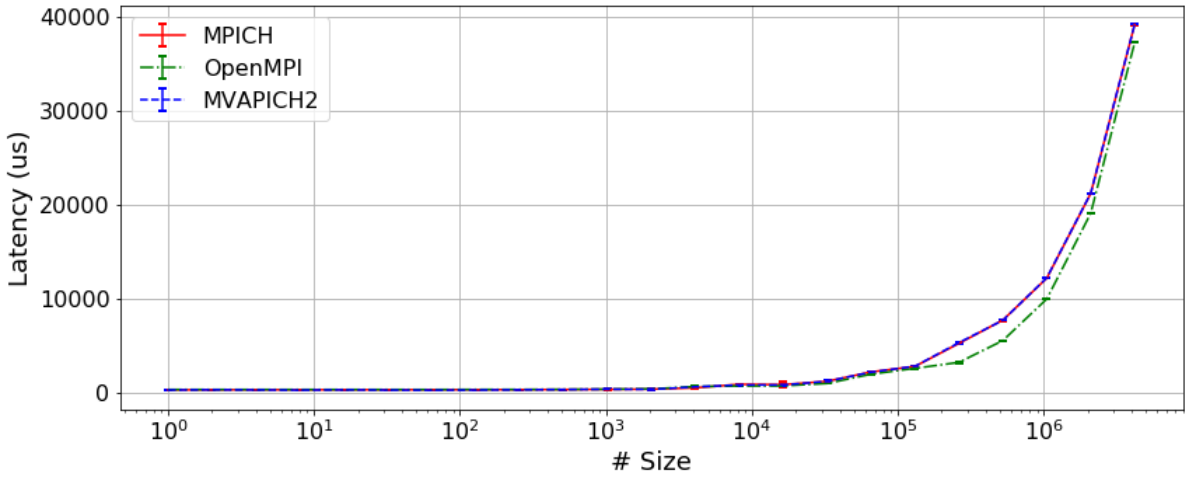


---

gatherv:

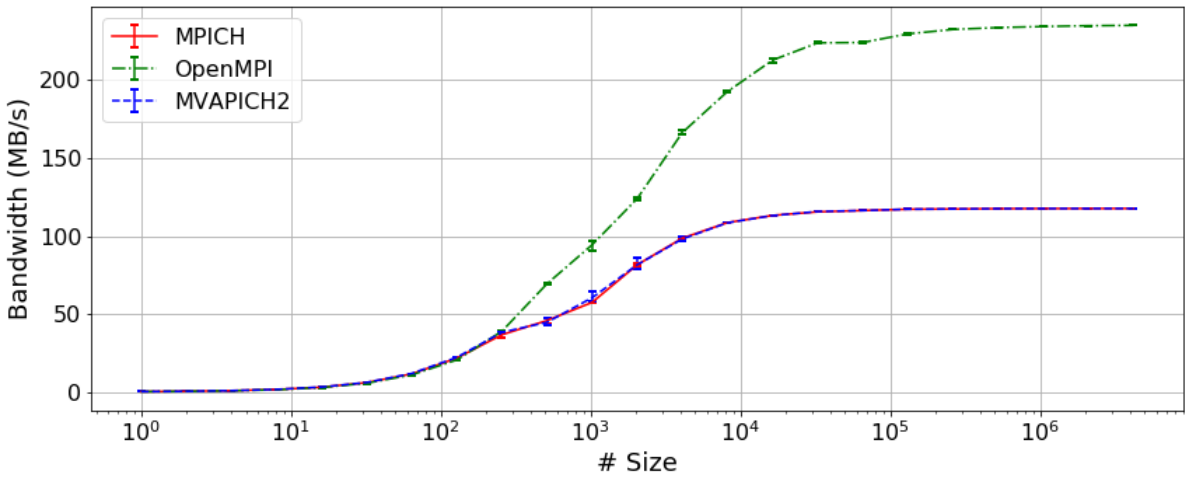


get_acc_latency:



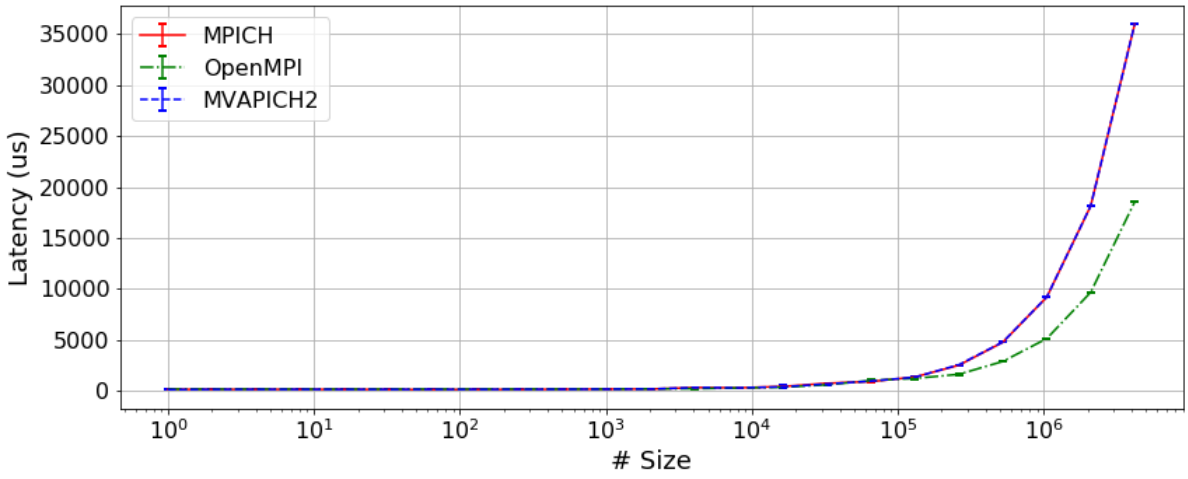
---

get_bw:

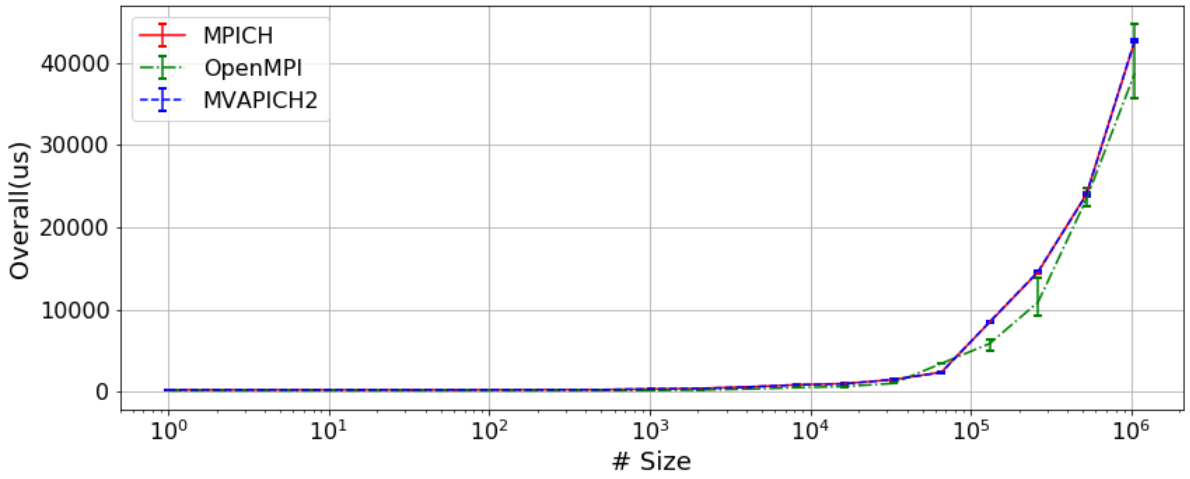


---

get_latency:

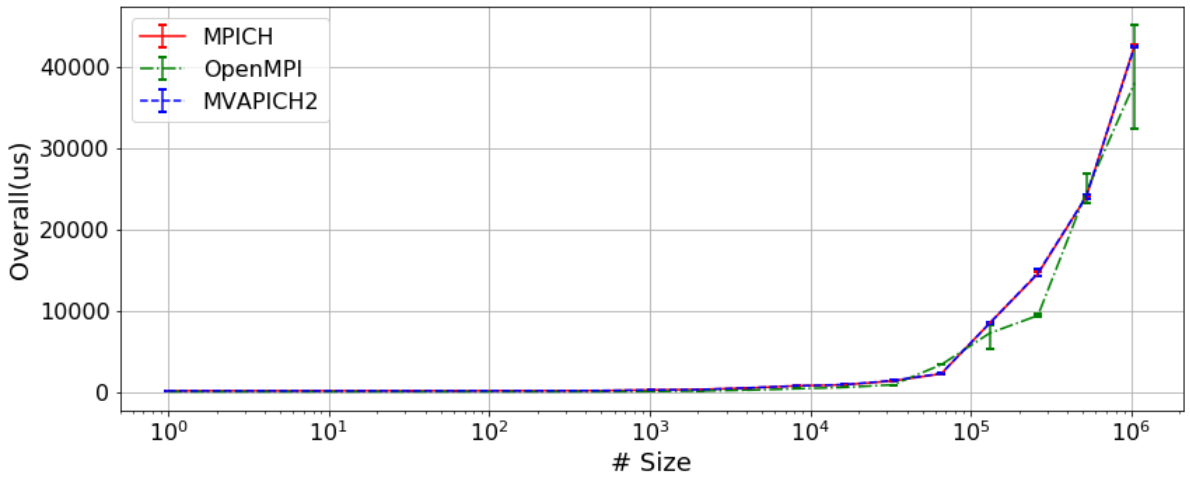


iallgather:



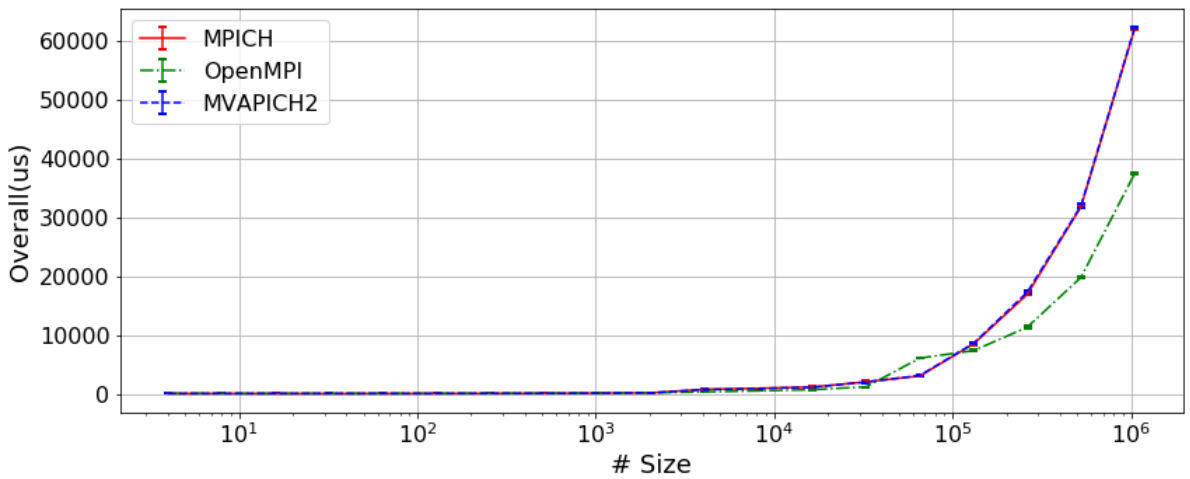
---

iallgatherv:

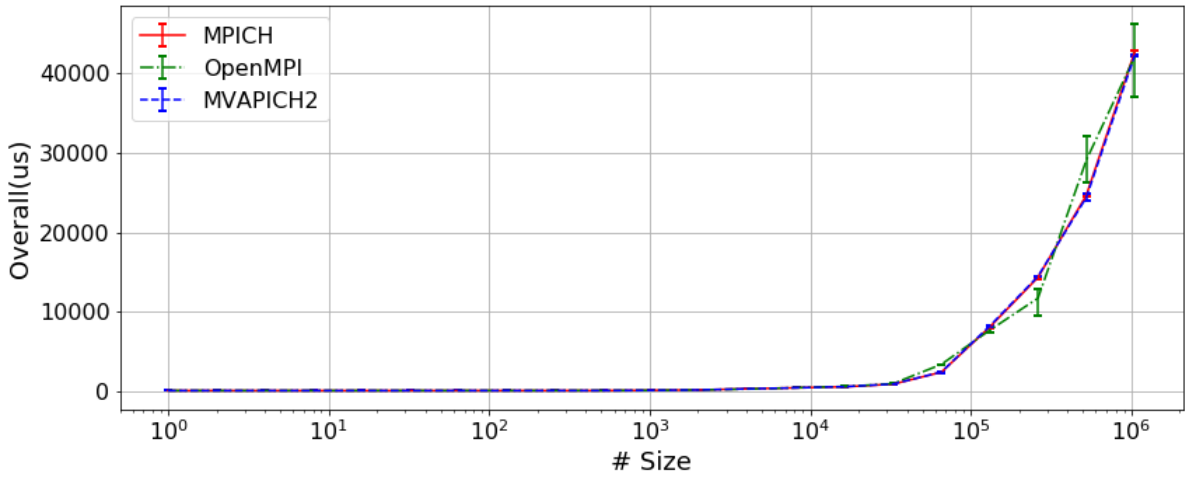


---

iallreduce:

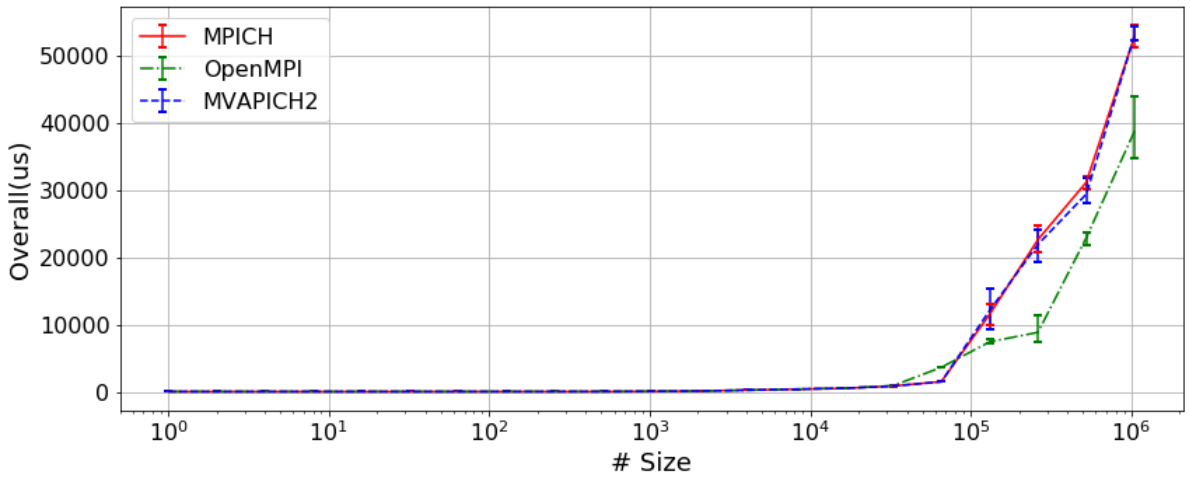


ialltoall:



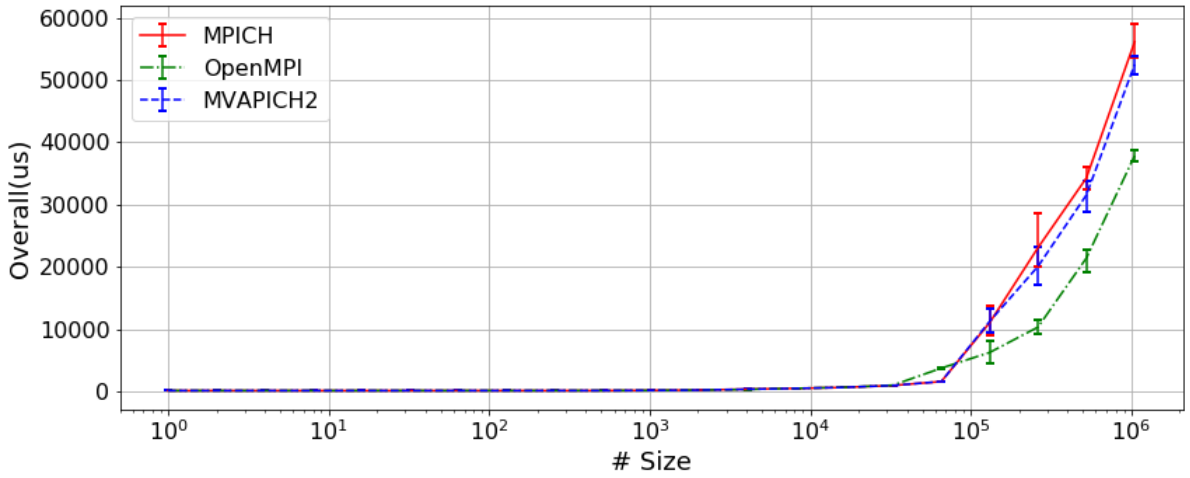
---

ialltoallv:

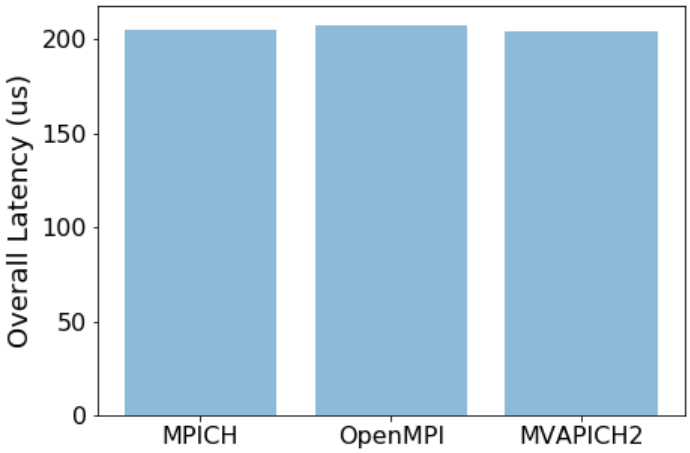


---

ialltoallw:

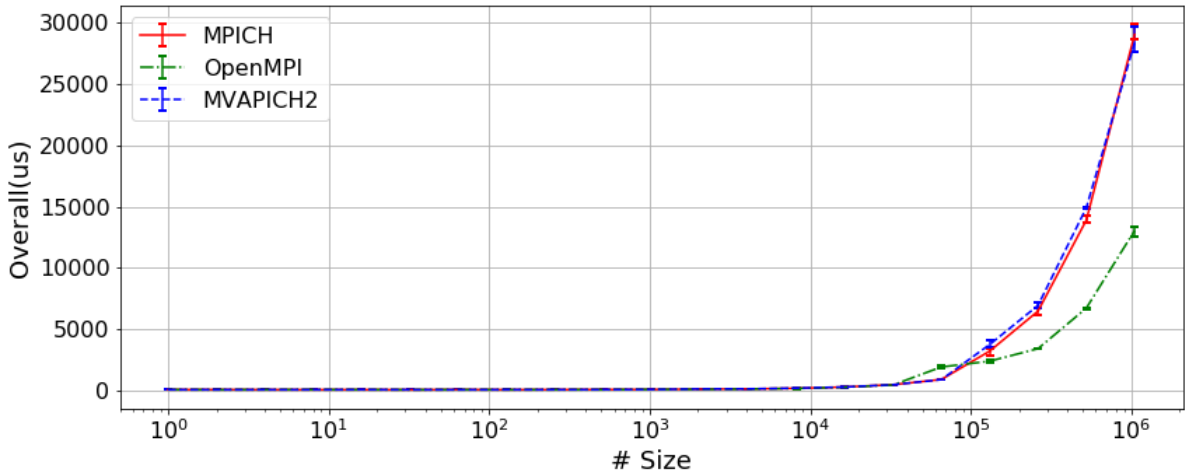


ibarrier:



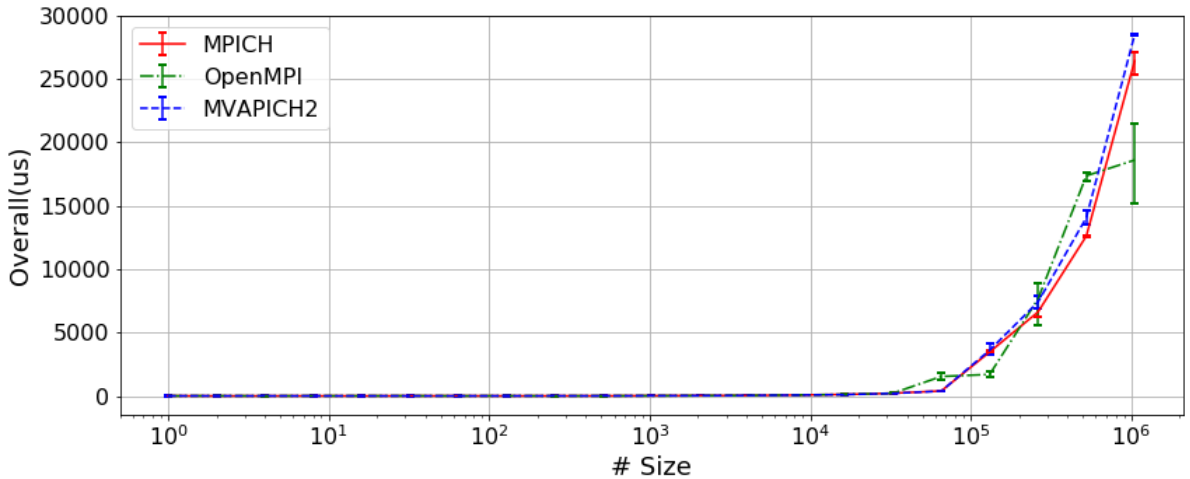
---

ibcast:

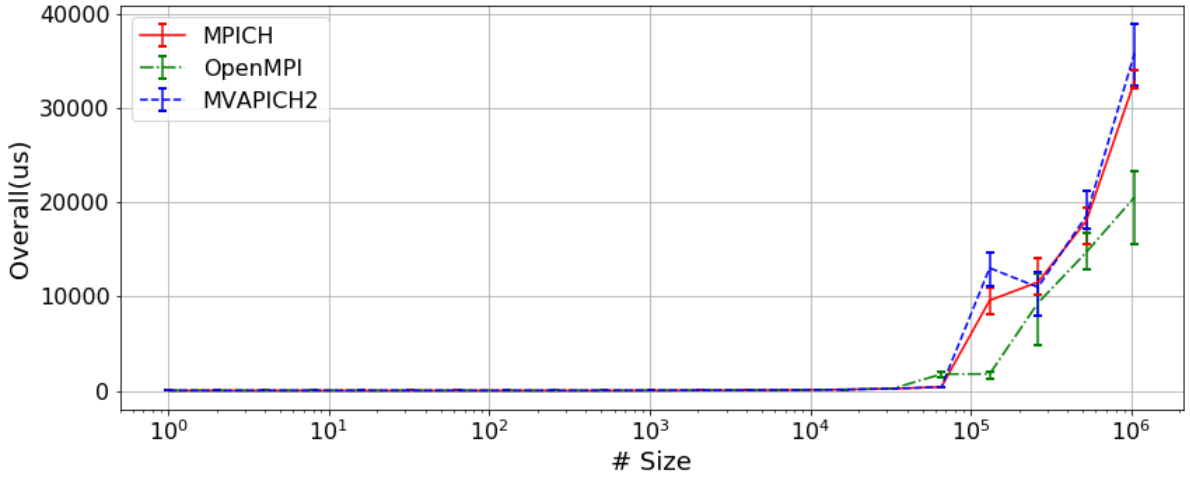


---

igather:

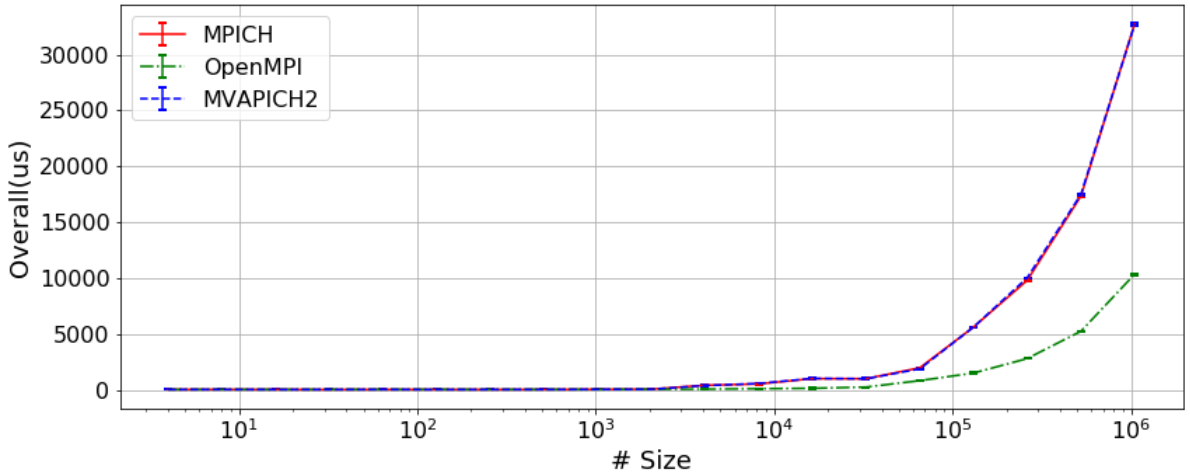


igatherv:



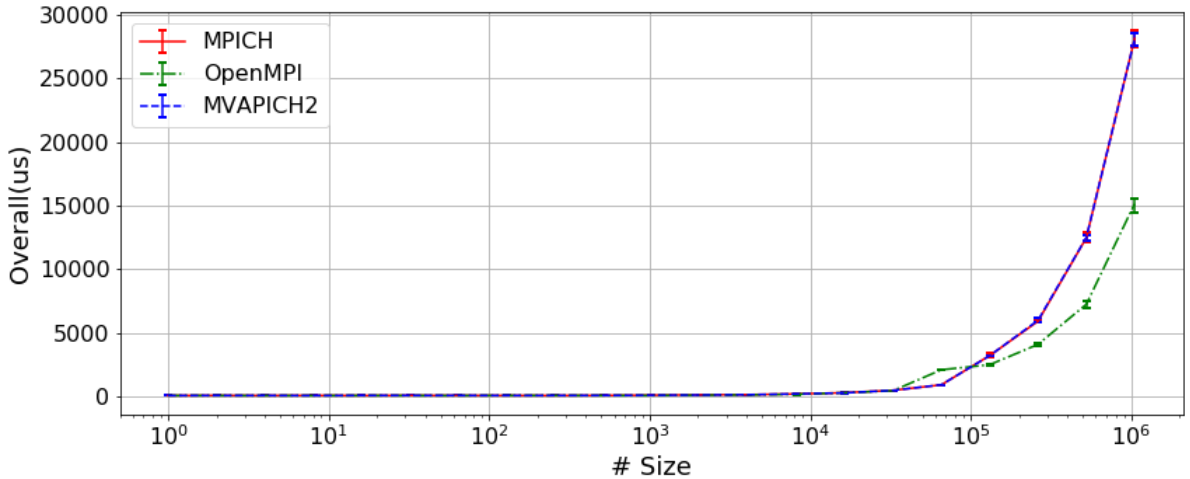
---

ireduce:

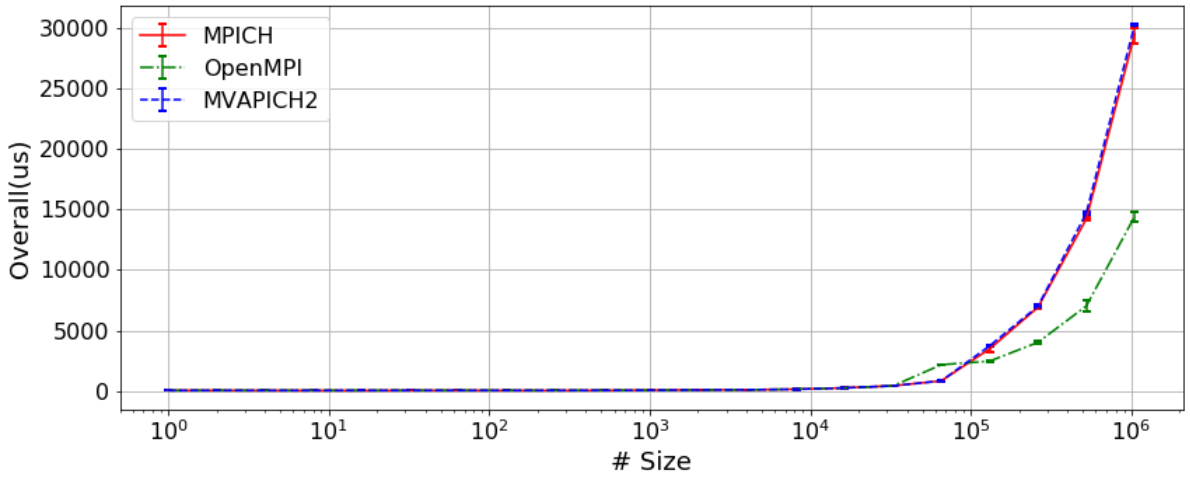


---

iscatter:

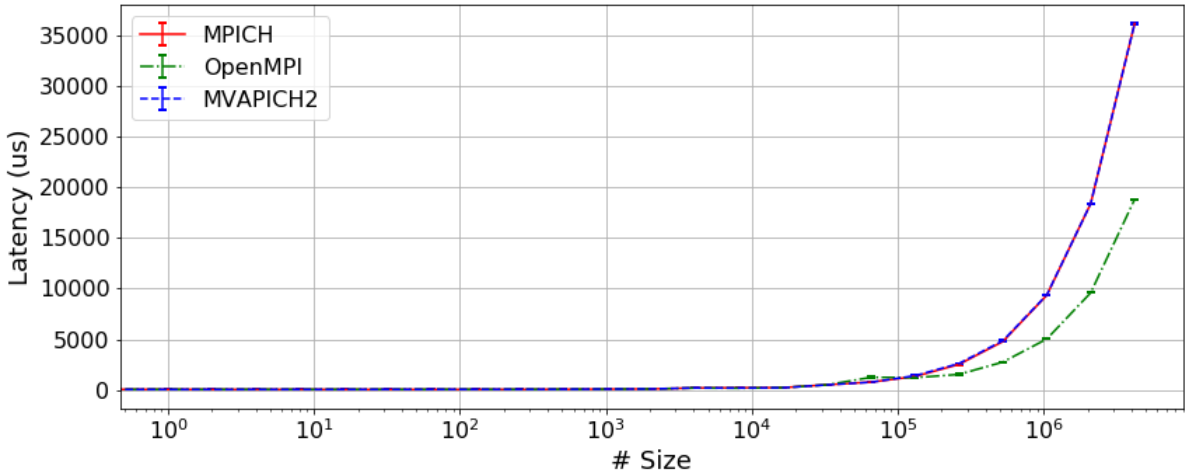


iscatterv:



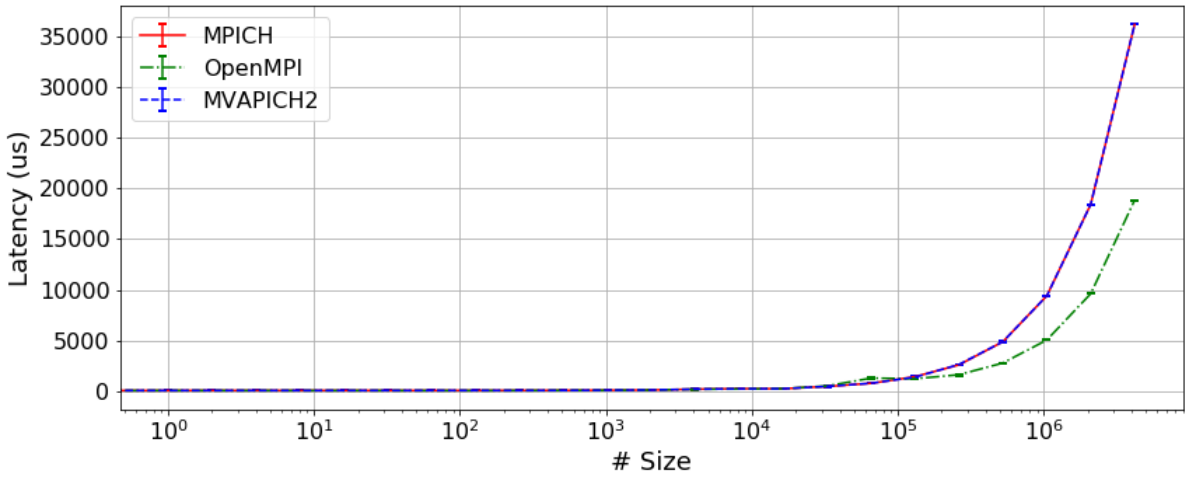
---

latency:

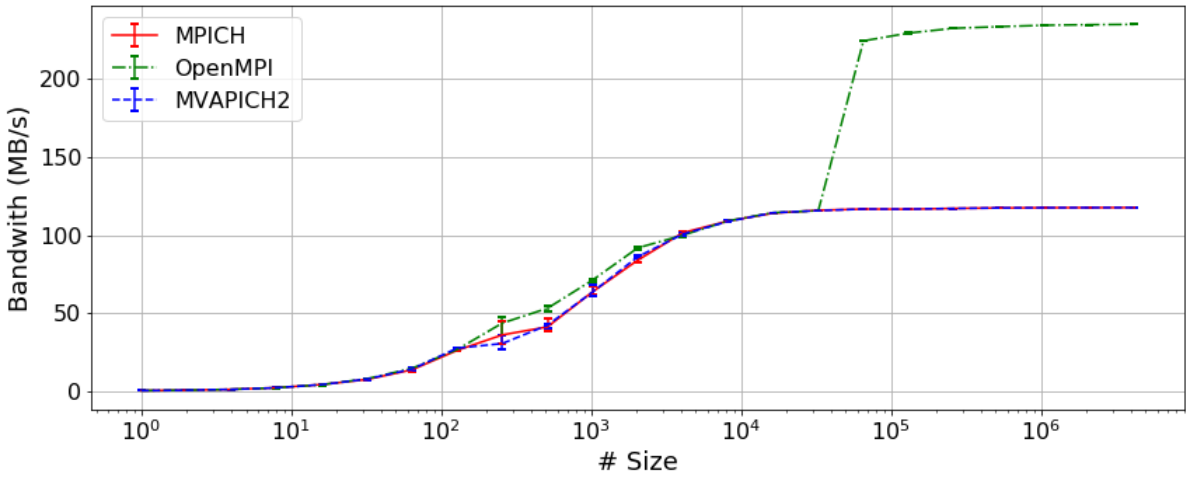


---

latency_mt:

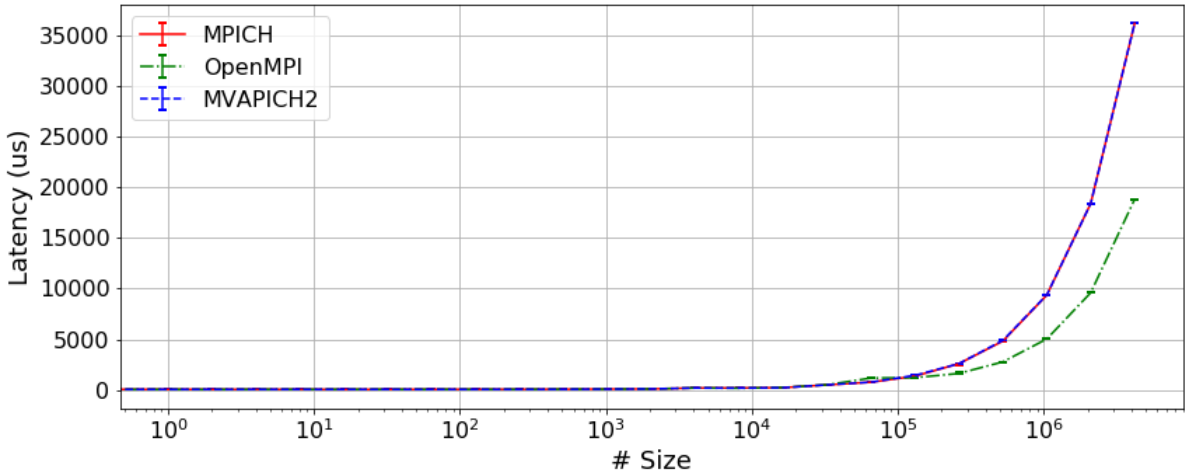


mbw_mr:



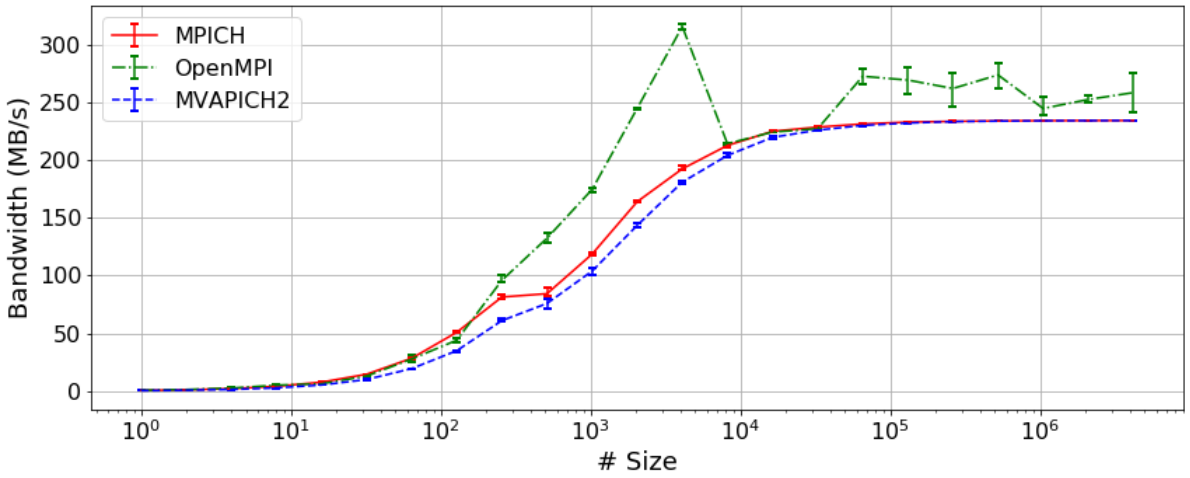
---

multi_lat:

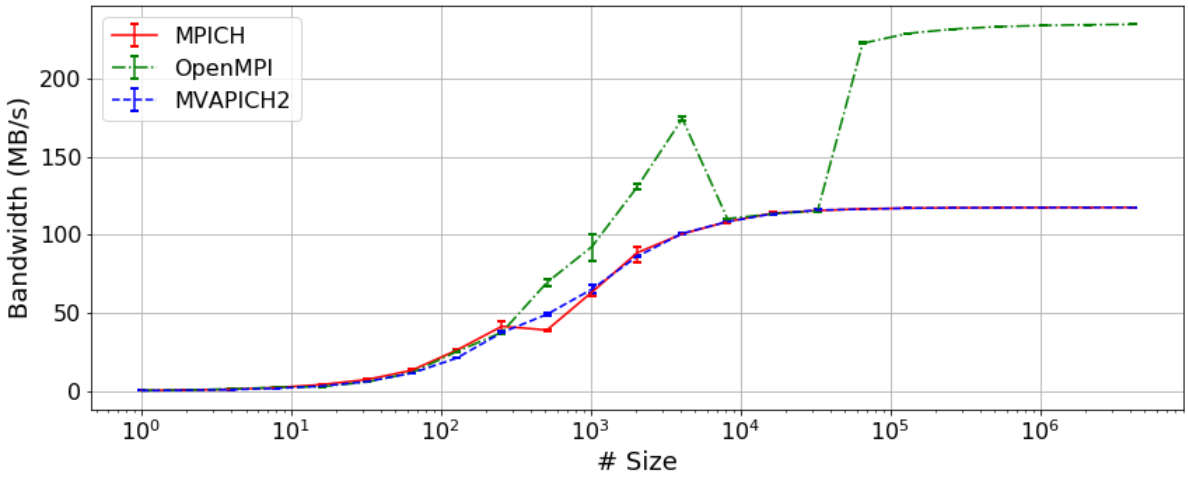


---

put_bibw:

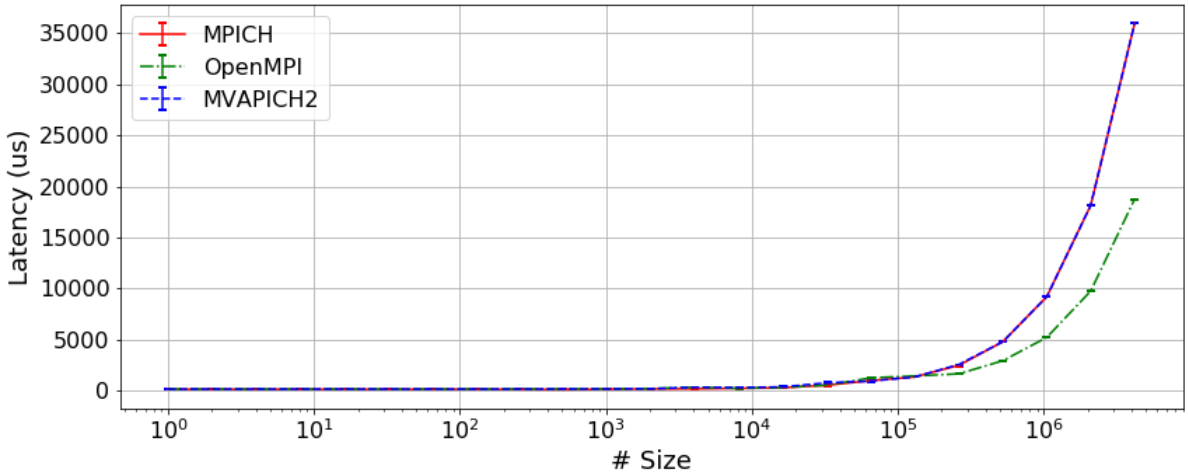


put_bw:



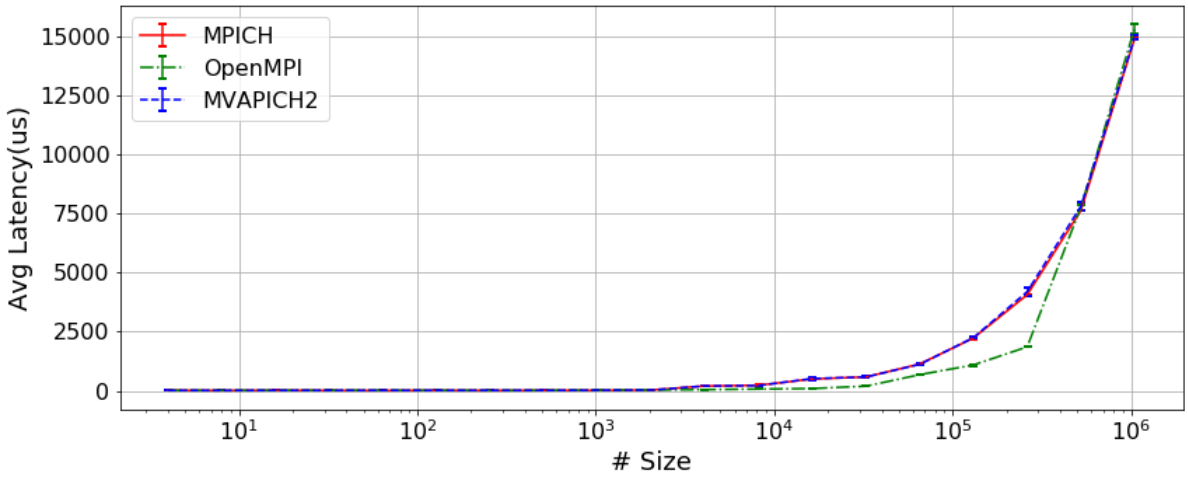
---

put_latency:

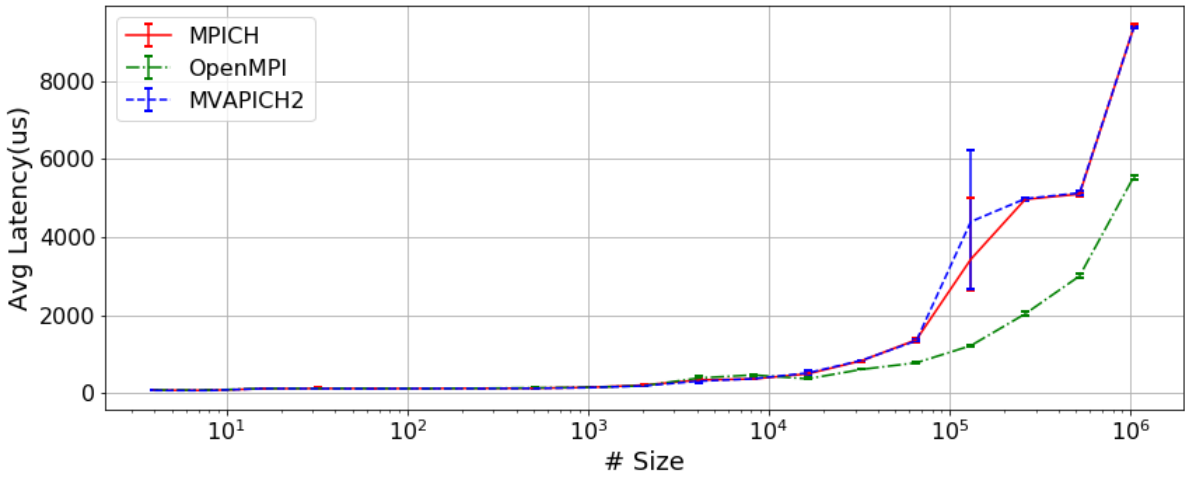


---

reduce:

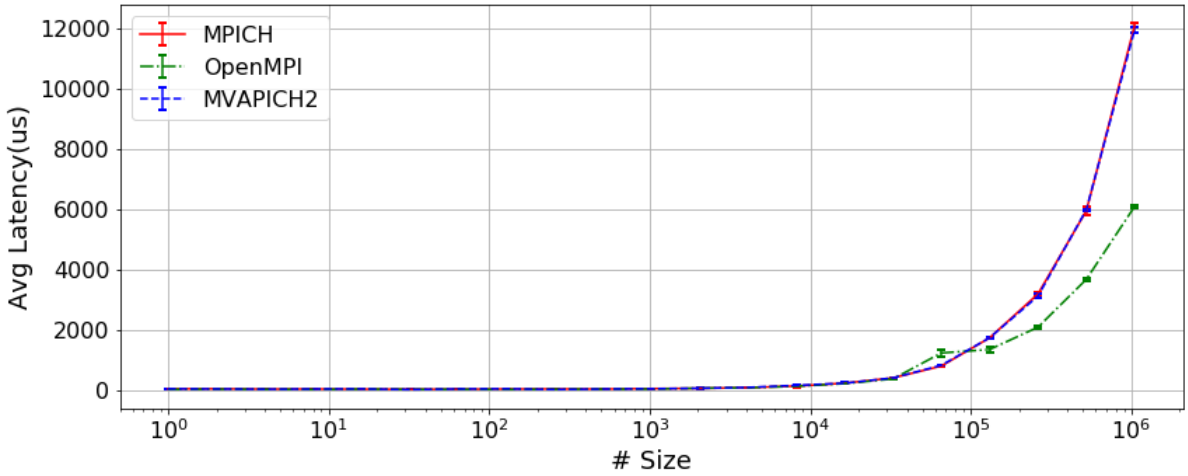


reduce_scatter:



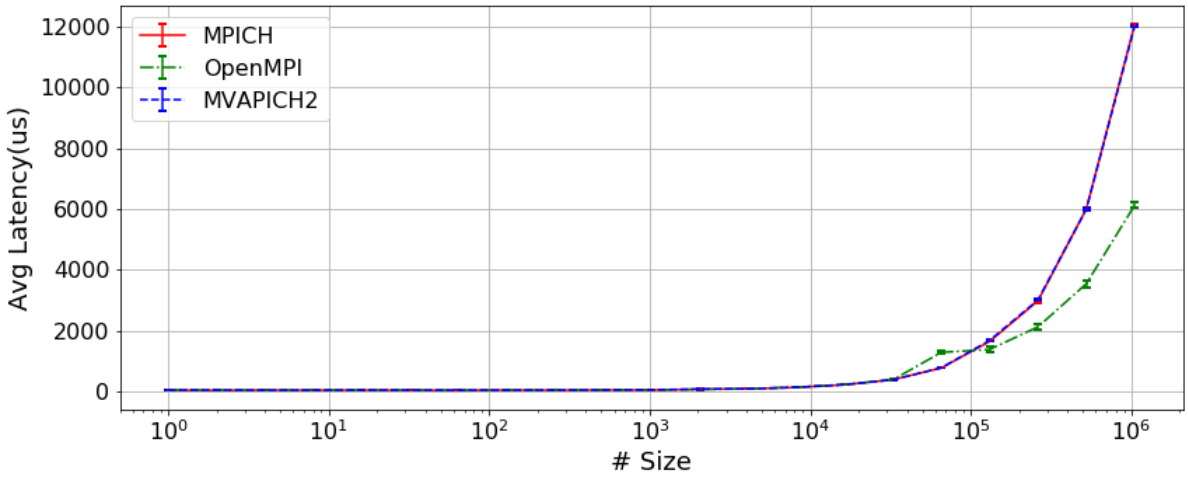
---

scatter:



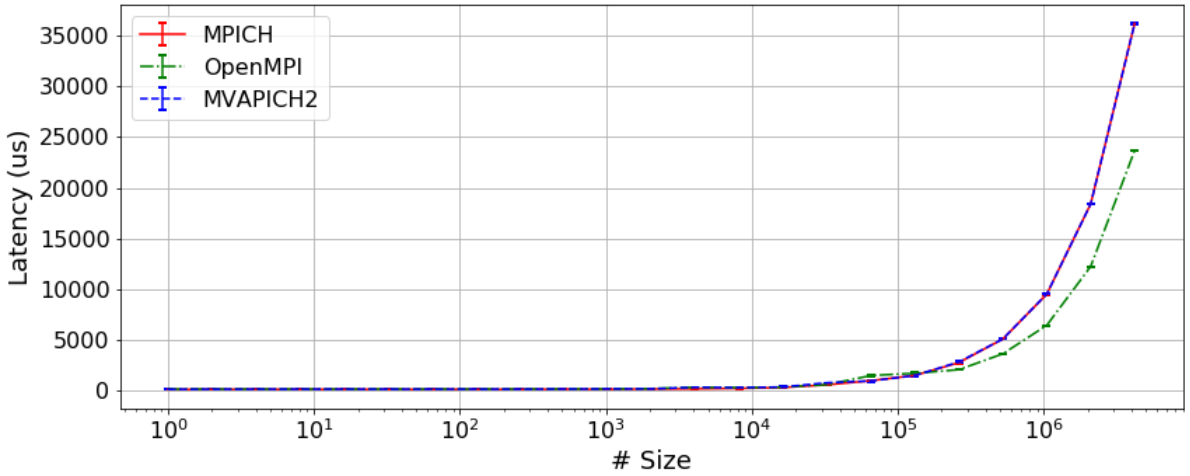
---

scatterv:



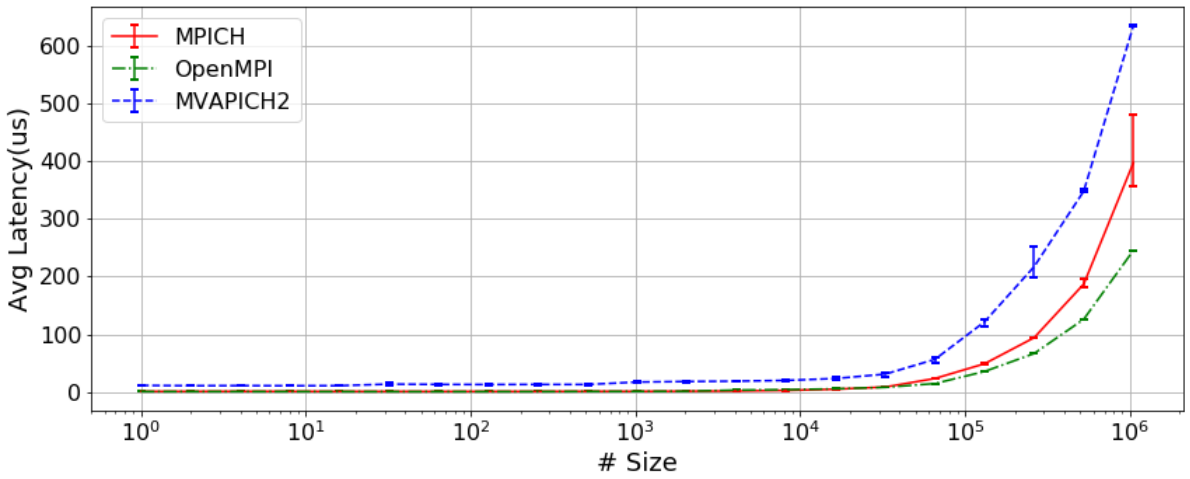
9.2 Graphen lokal

acc_latency:



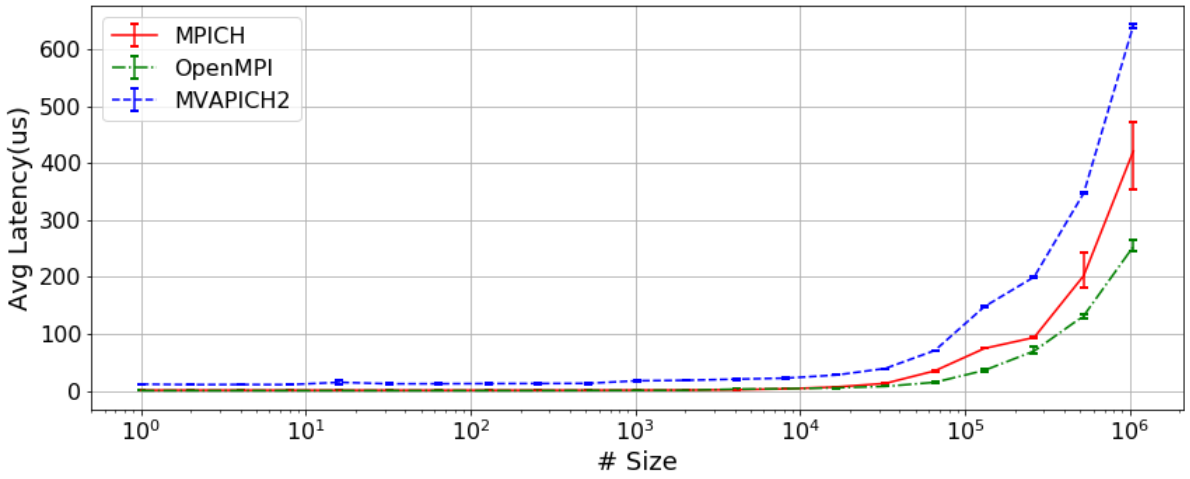
---

allgather:

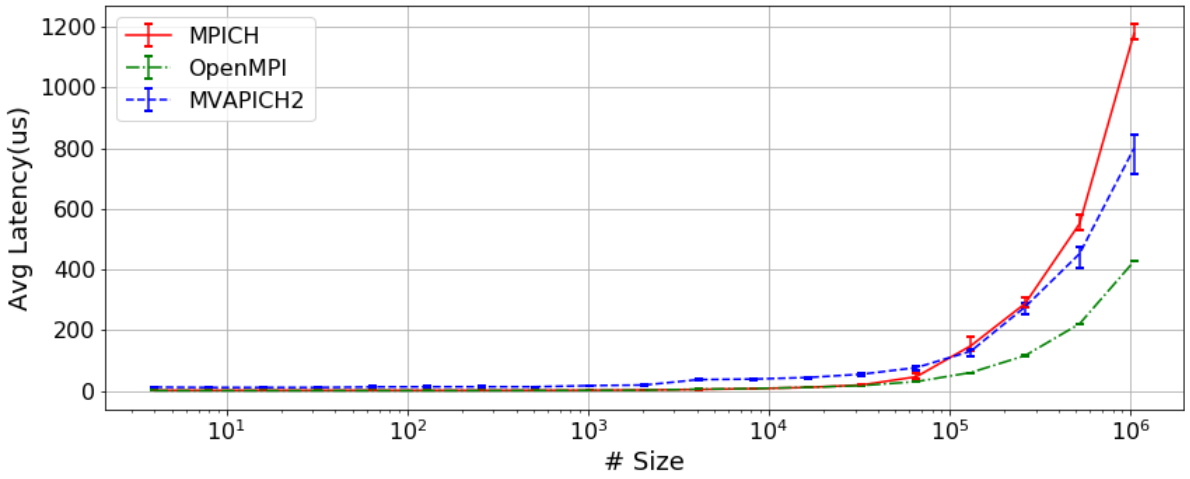


---

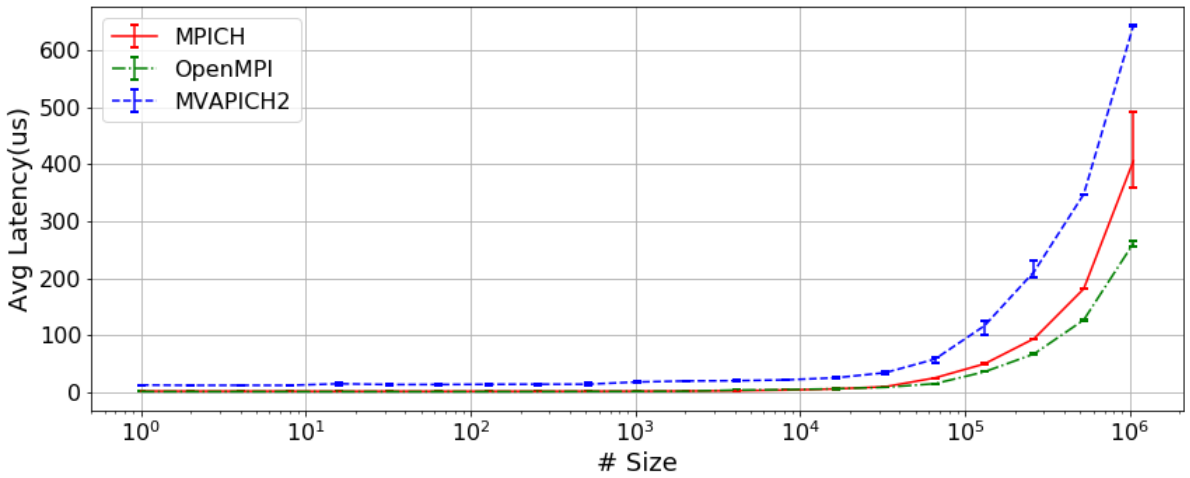
allgatherv:



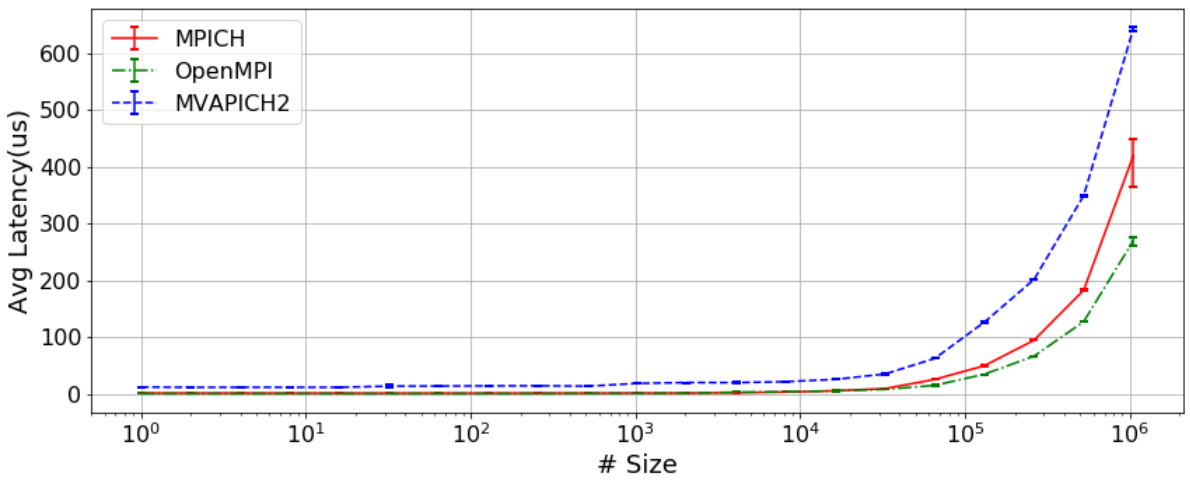
allreduce:



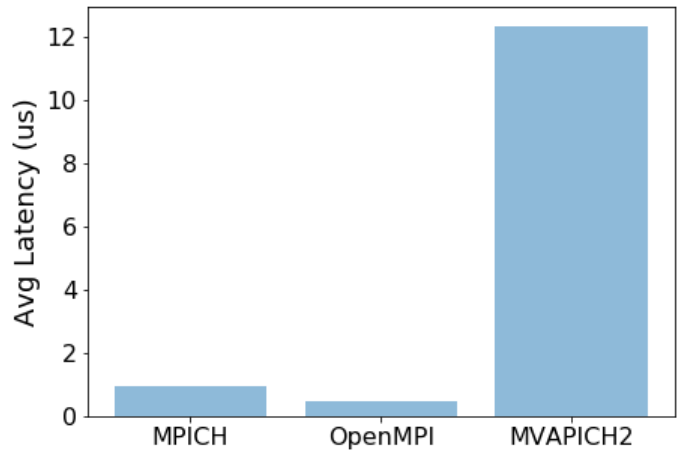
alltoall:



alltoallv:

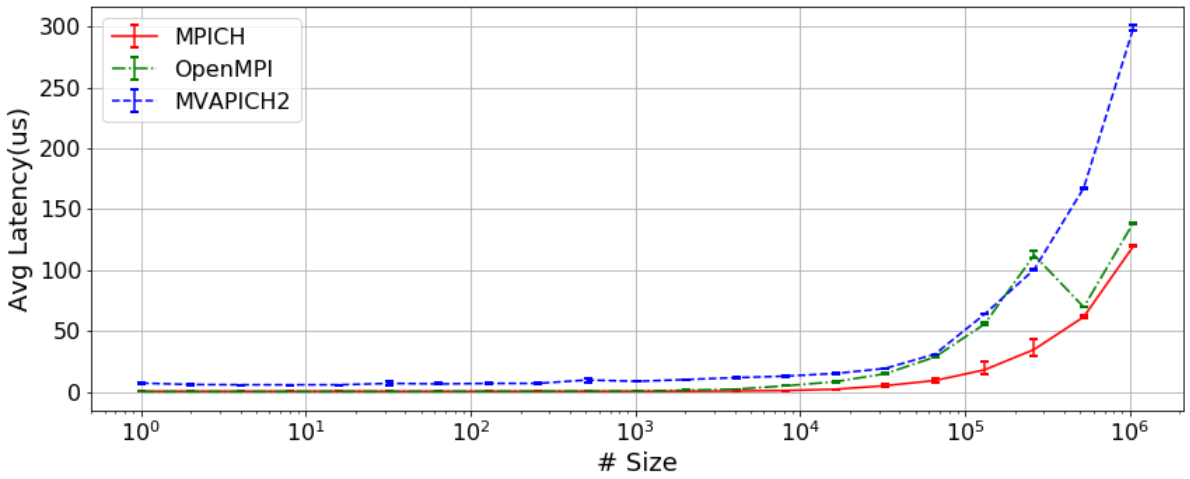


barrier:



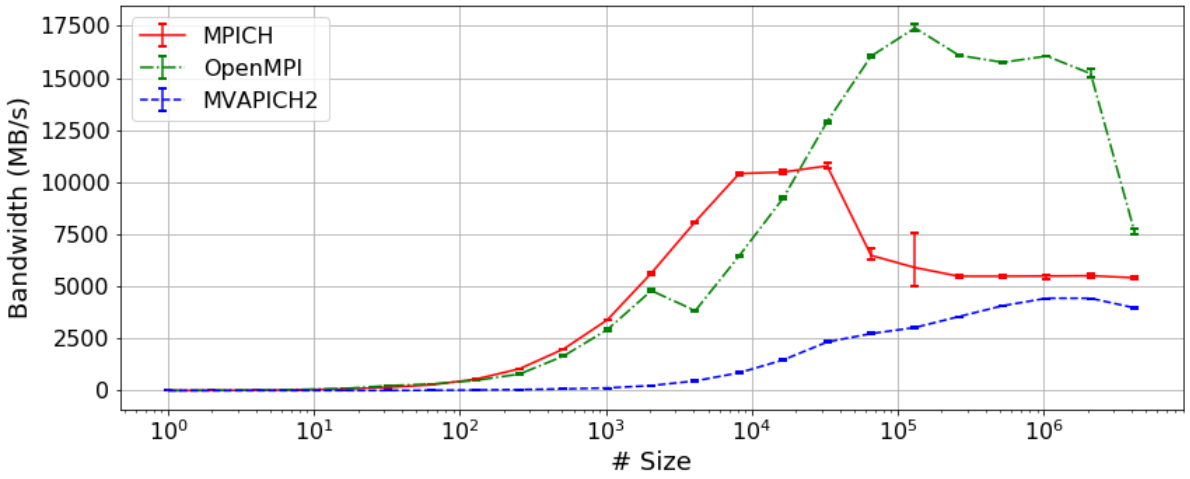
---

bcast:

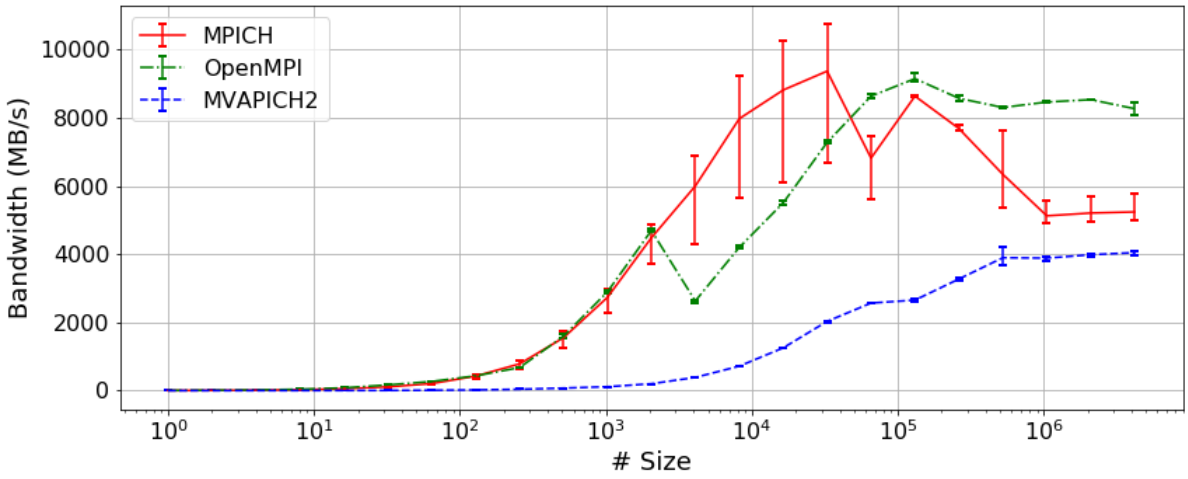


---

bibw:

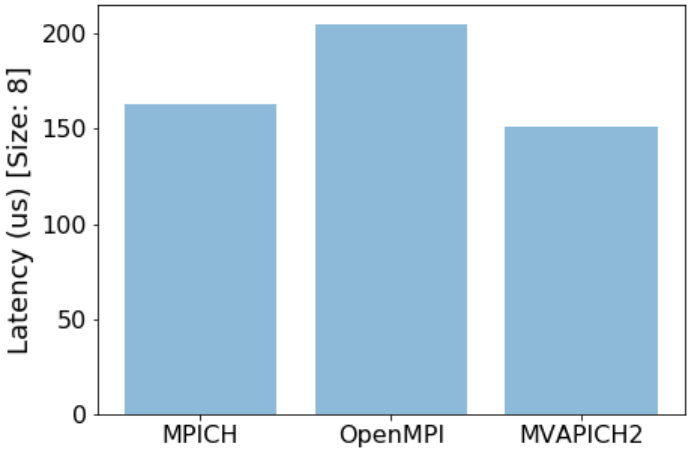


bw:



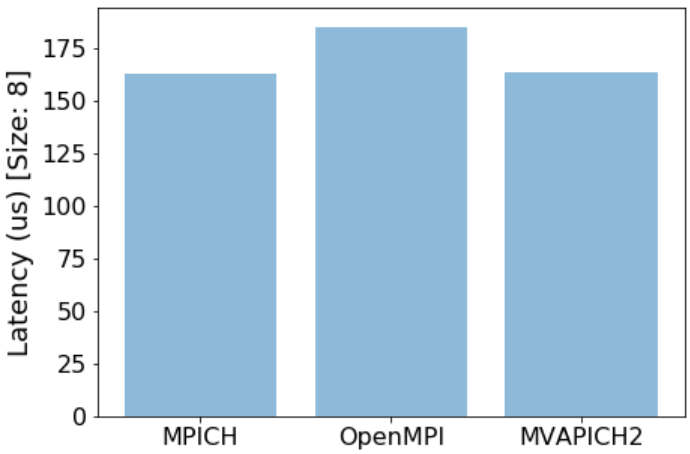
---

cas_latency:

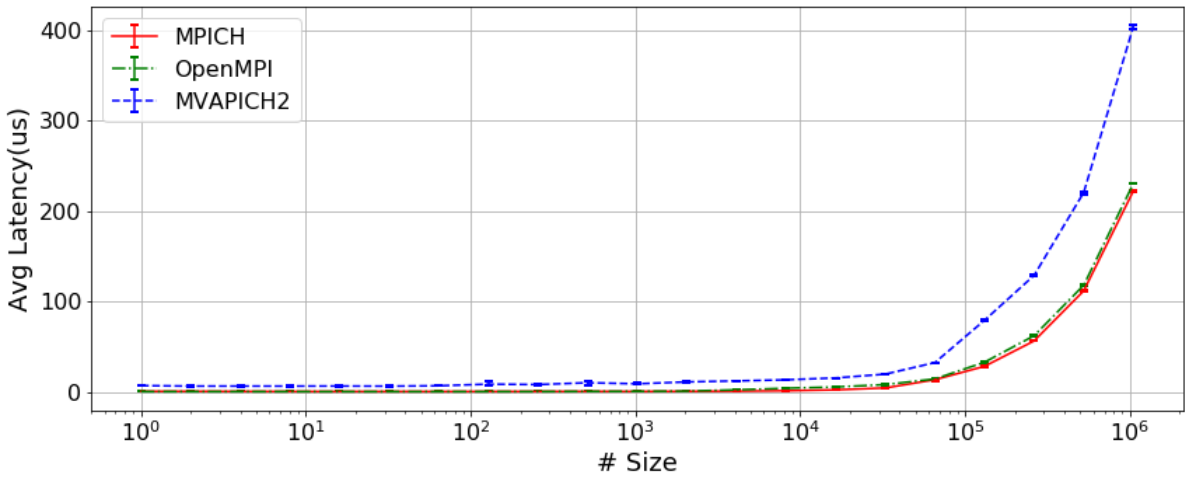


---

fop_latency:

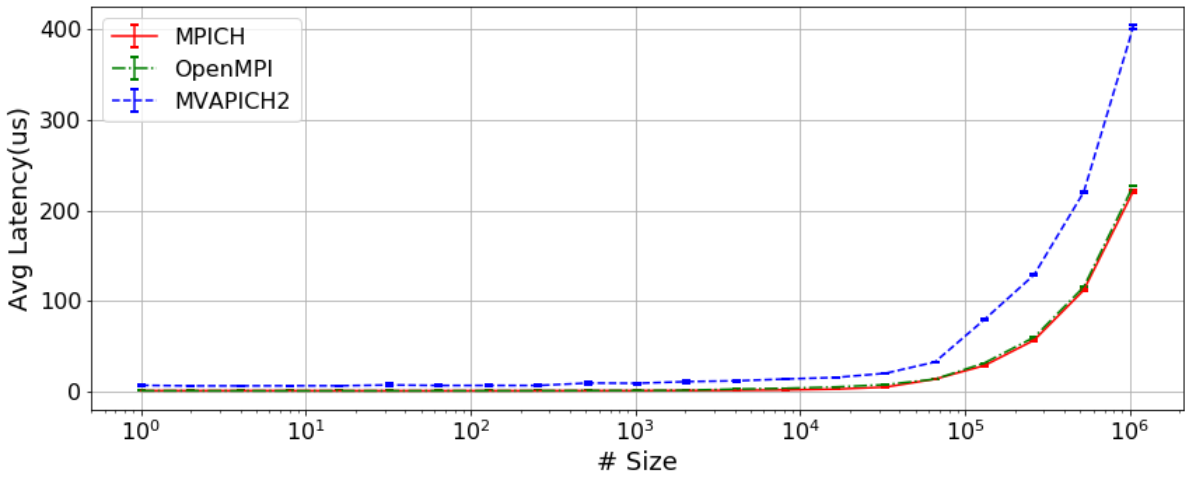


gather:



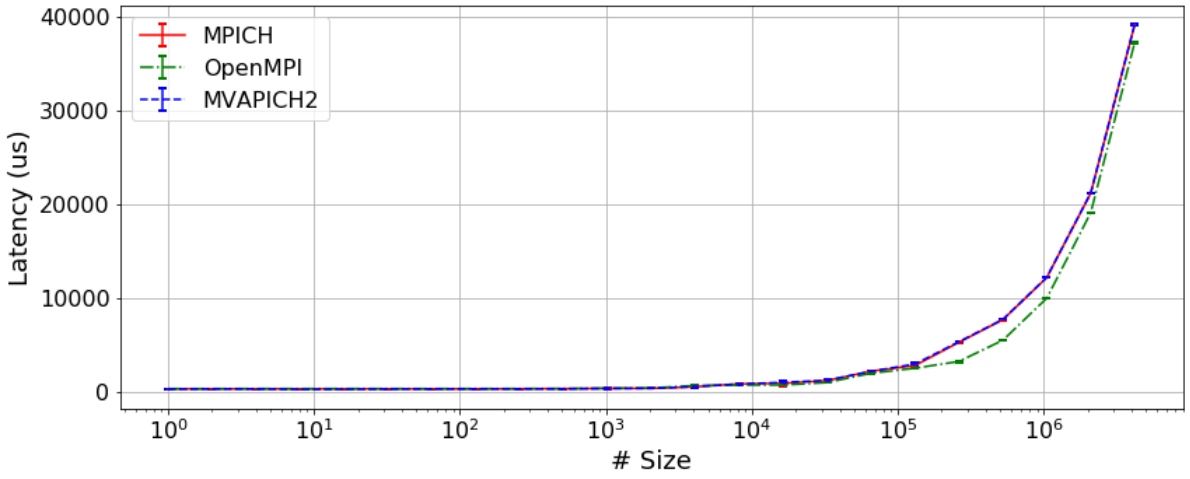
---

gatherv:

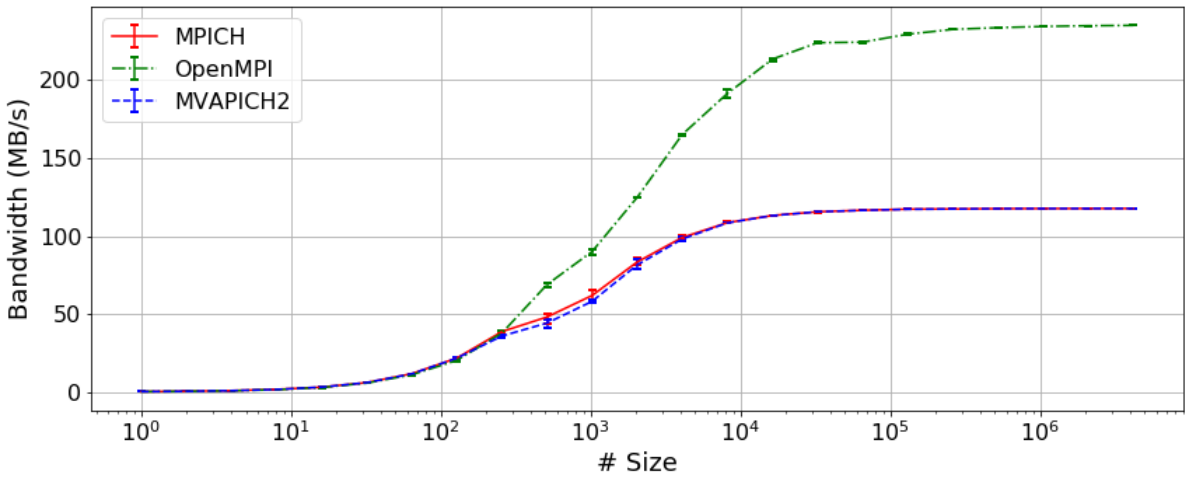


---

get_acc_latency:

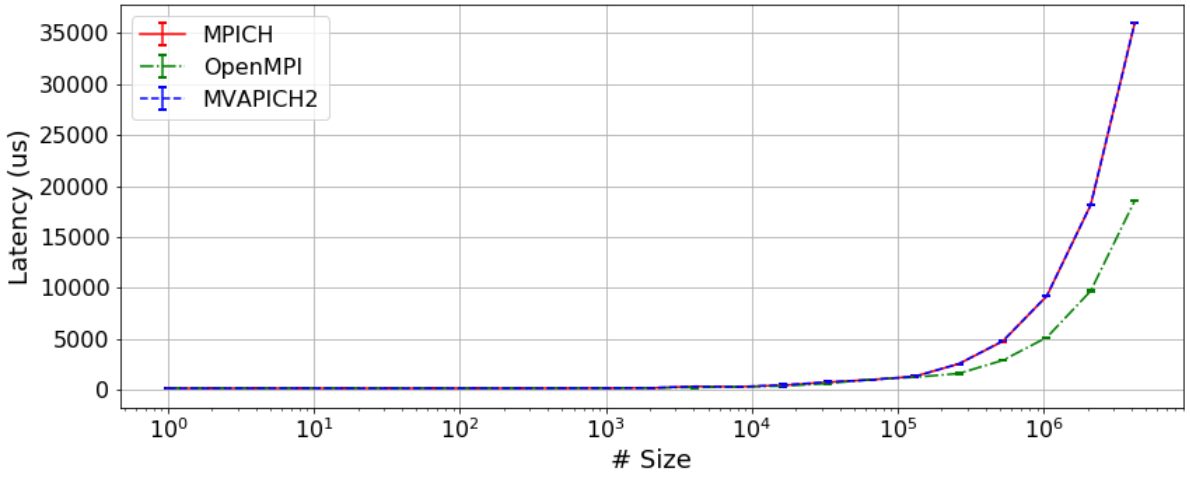


get_bw:



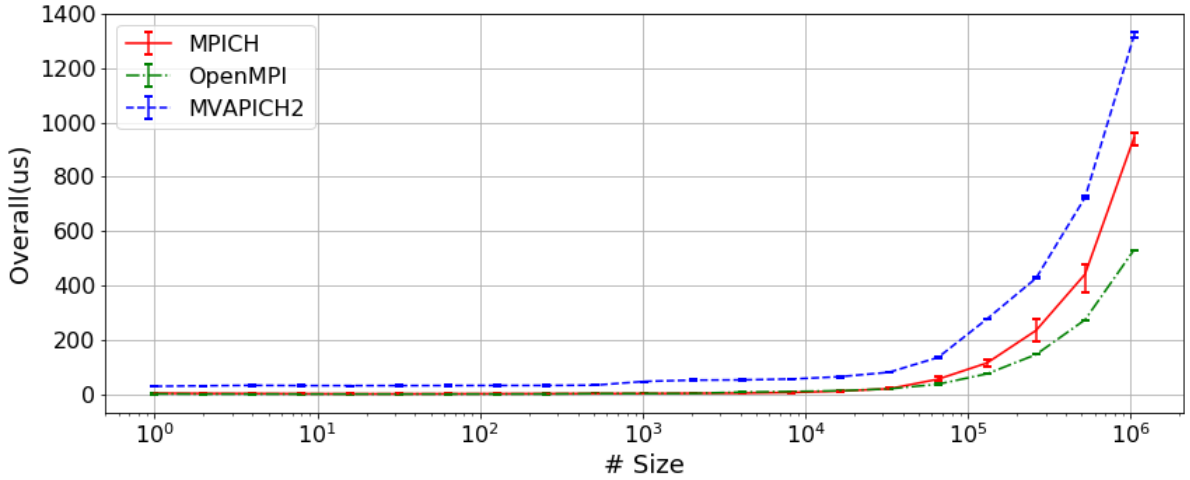
---

get_latency:

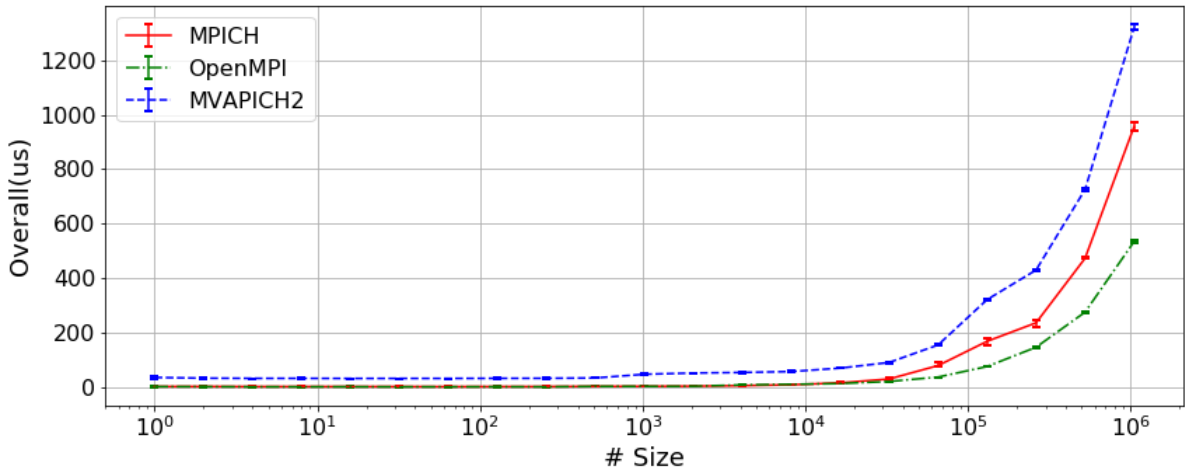


---

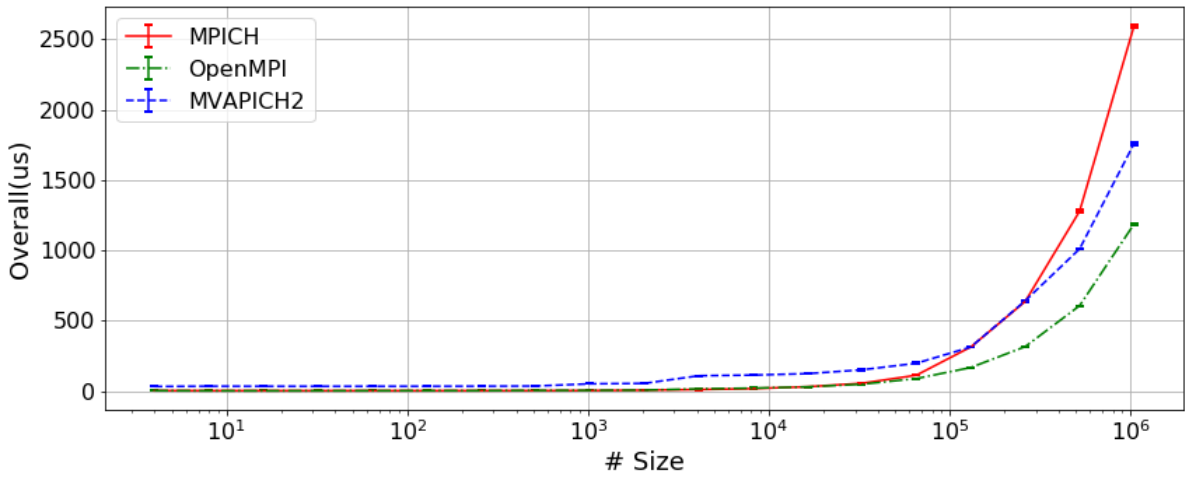
iallgather:



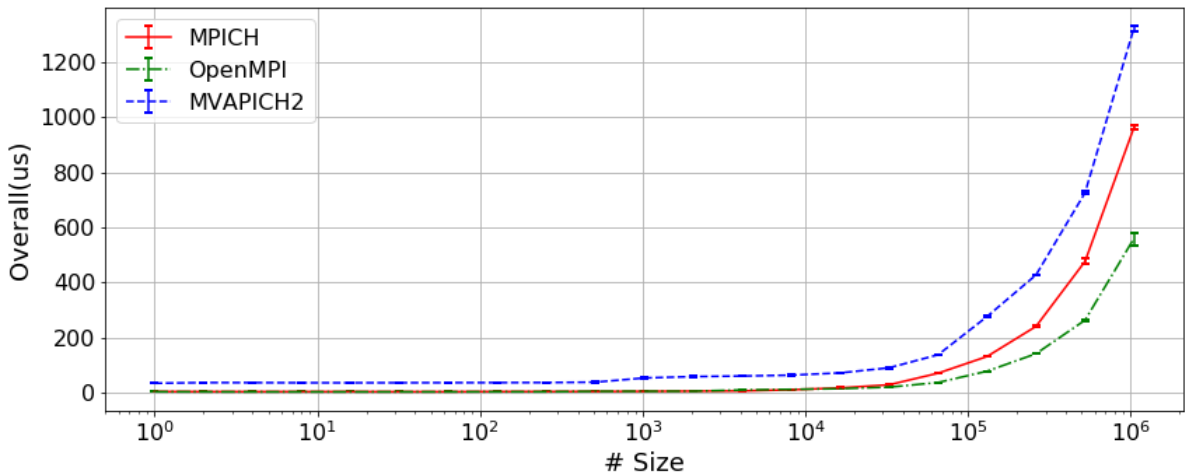
iallgatherv:



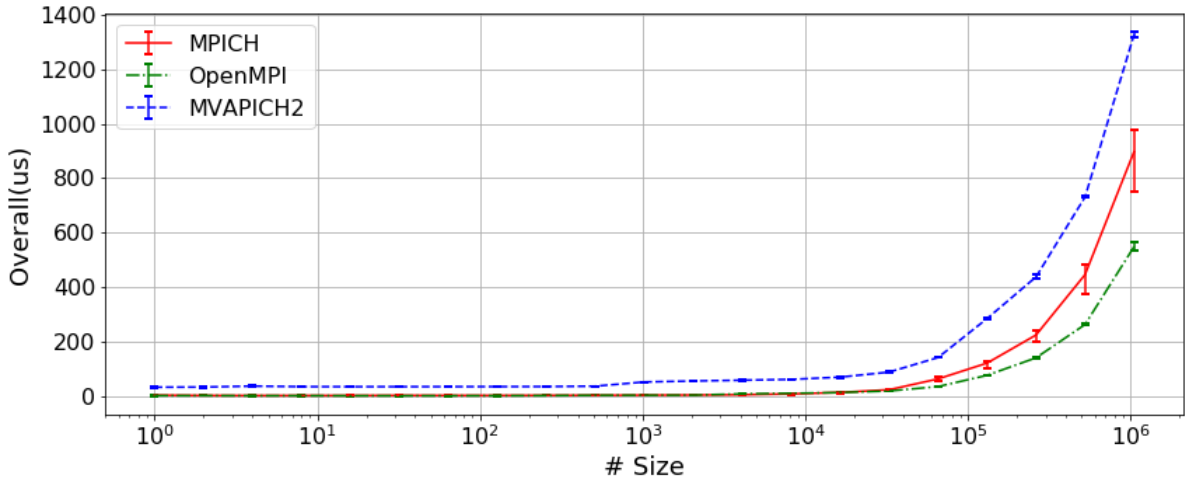
iallreduce:



ialltoall:

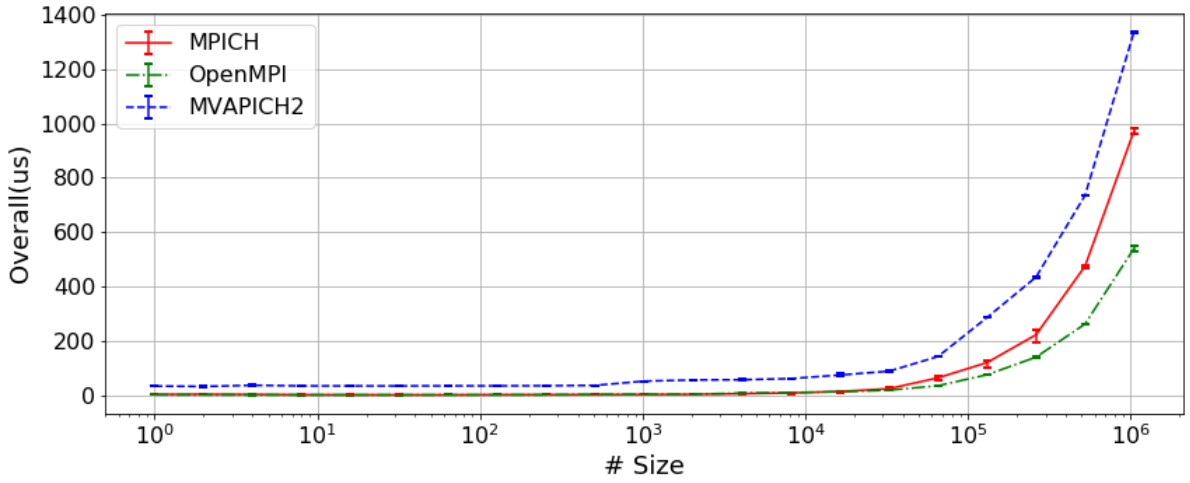


ialltoallv:



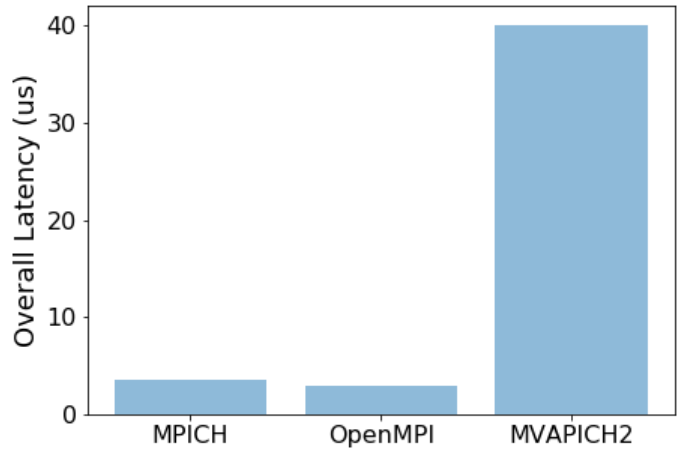
---

ialltoallw:

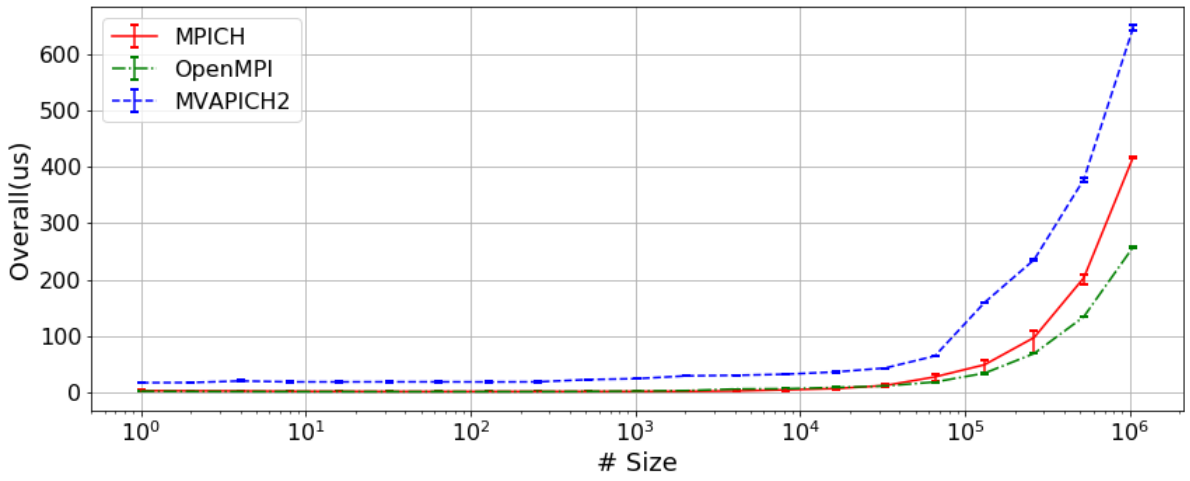


---

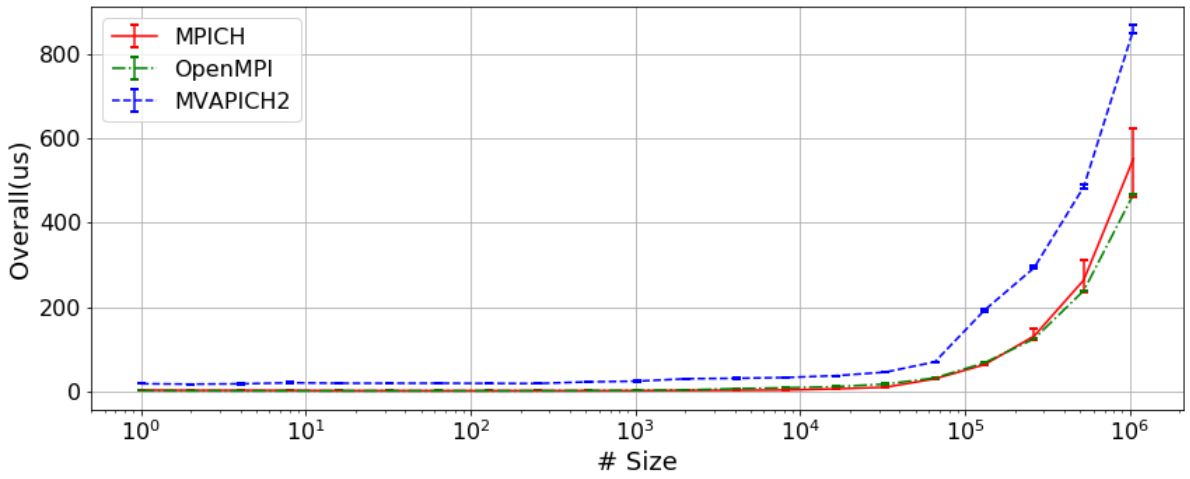
ibarrier:



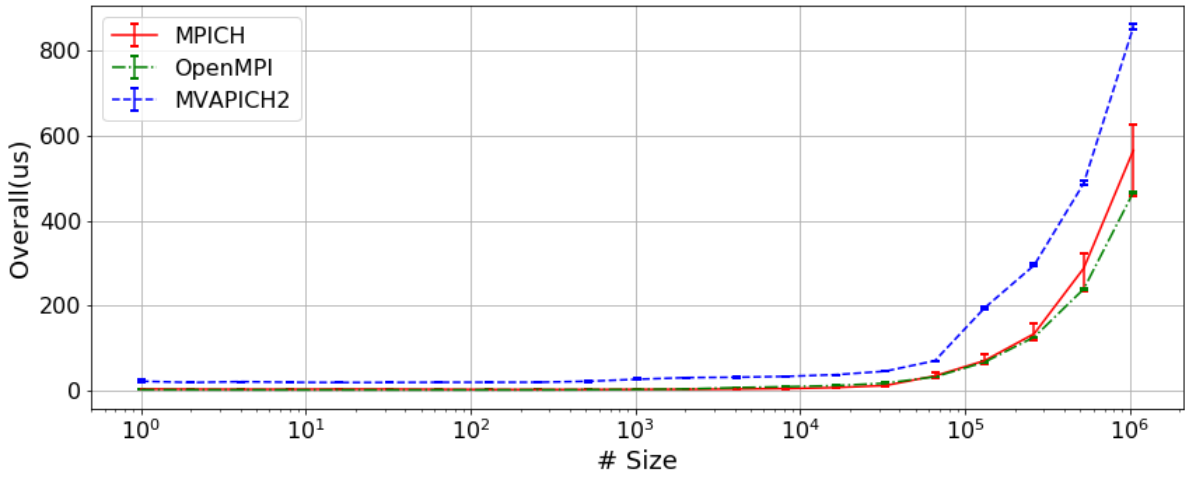
ibcast:



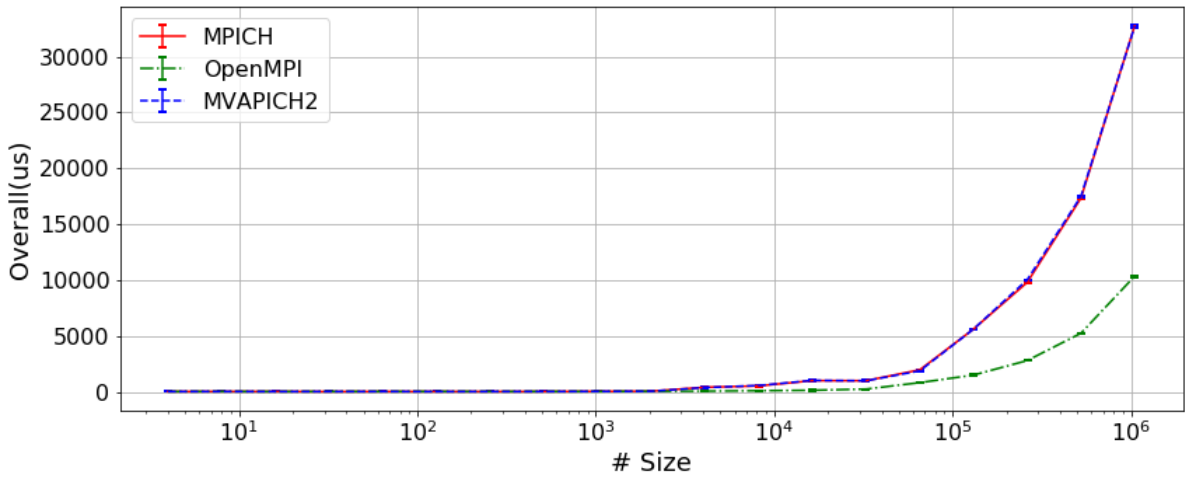
igather:



igatherv:

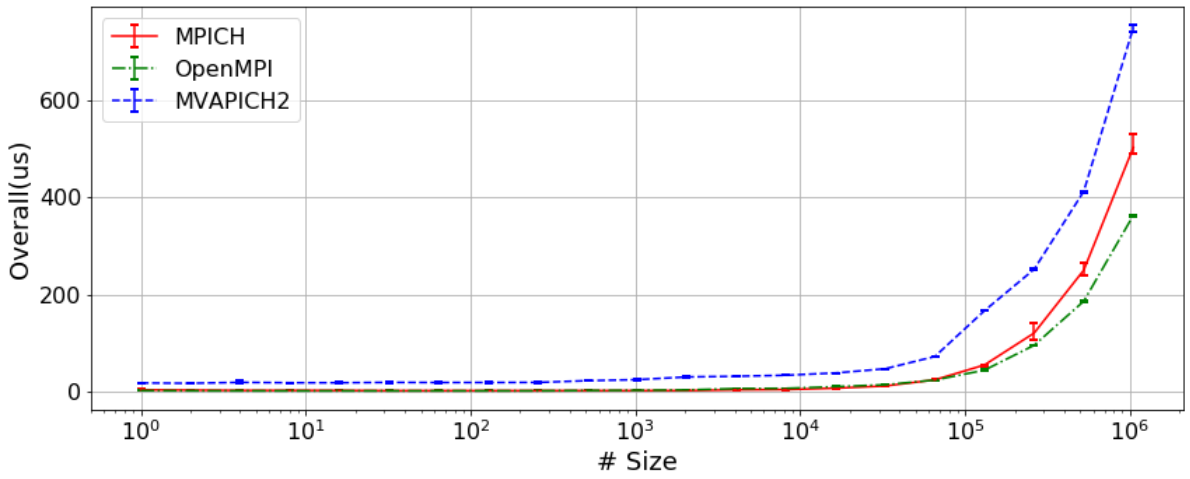


ireduce:



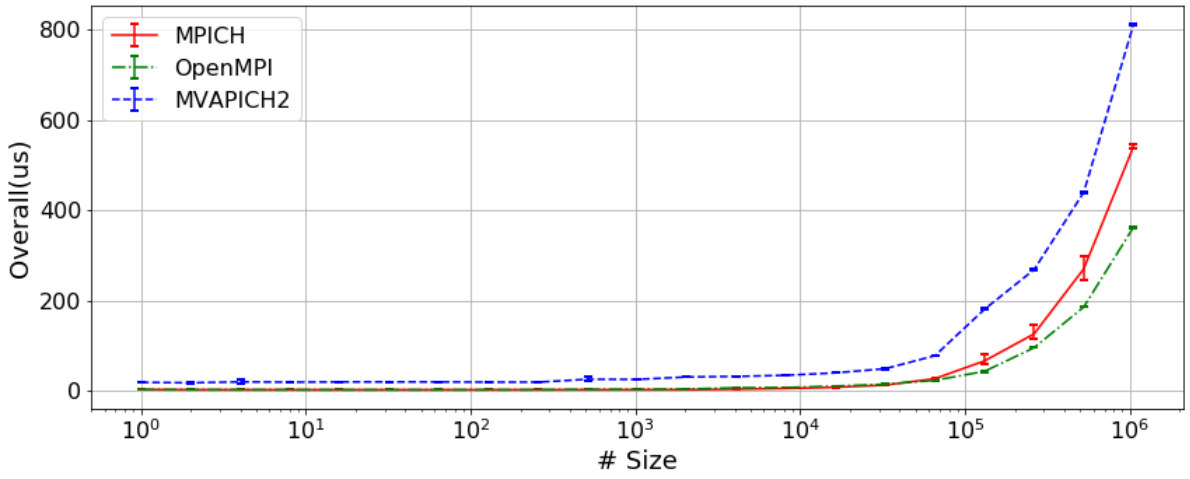
---

iscatter:

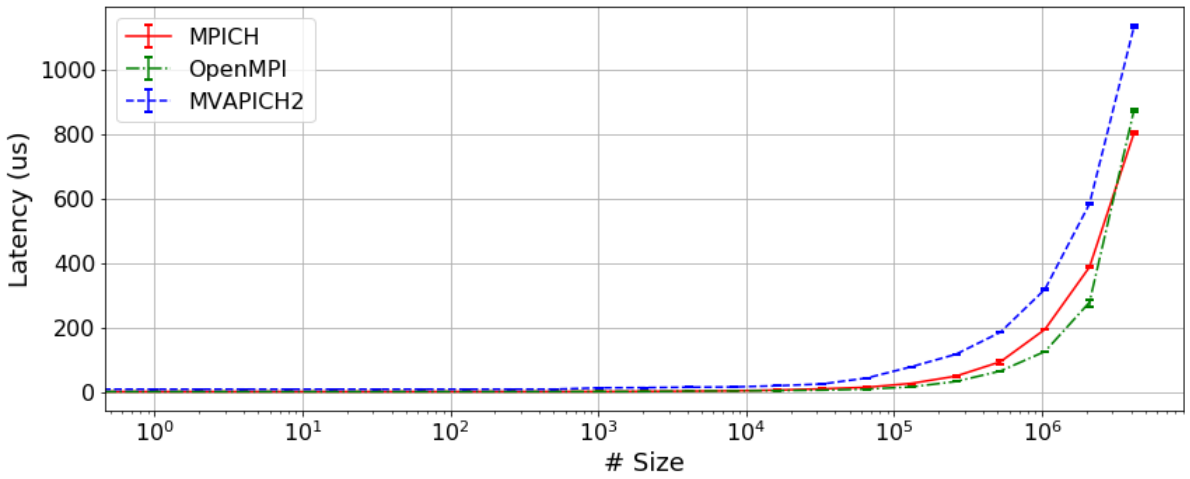


---

iscatterv:

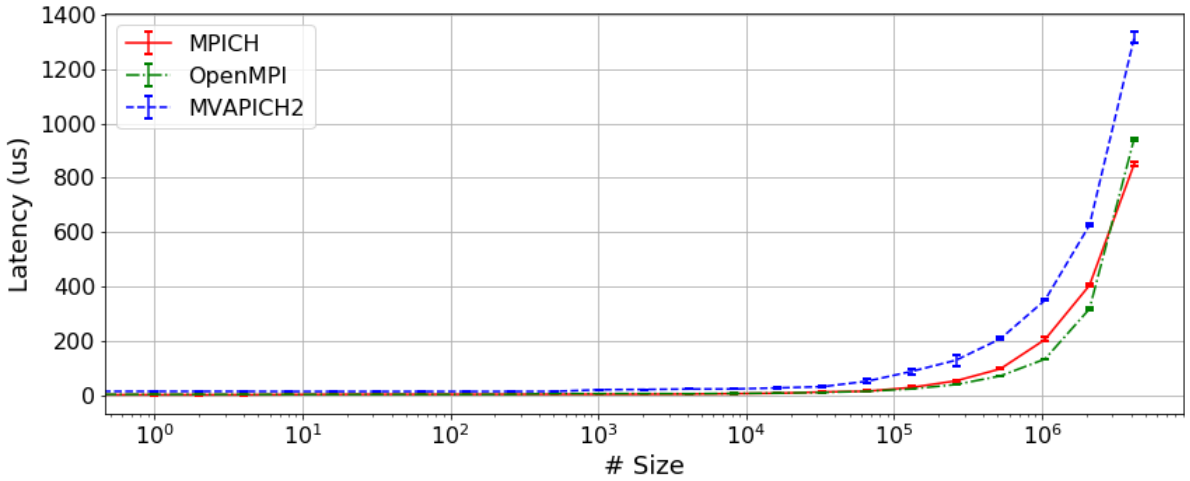


latency:



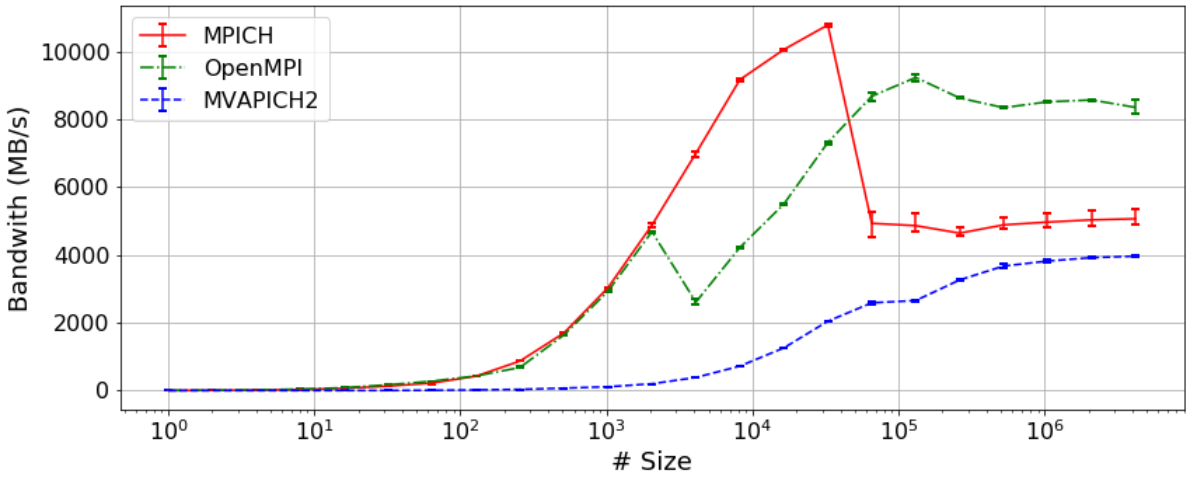
---

latency_mt:

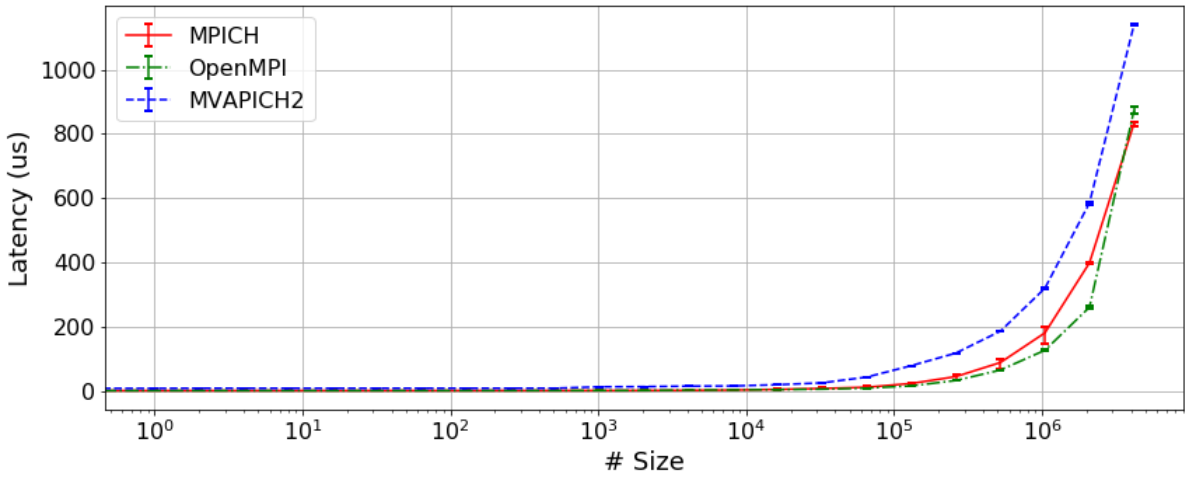


---

mbw_mr:

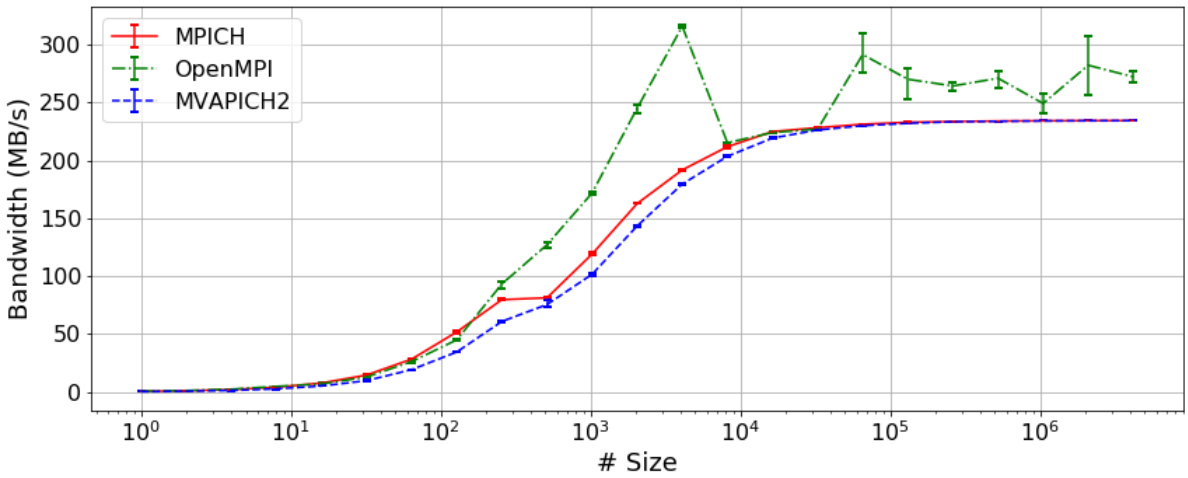


multi_lat:



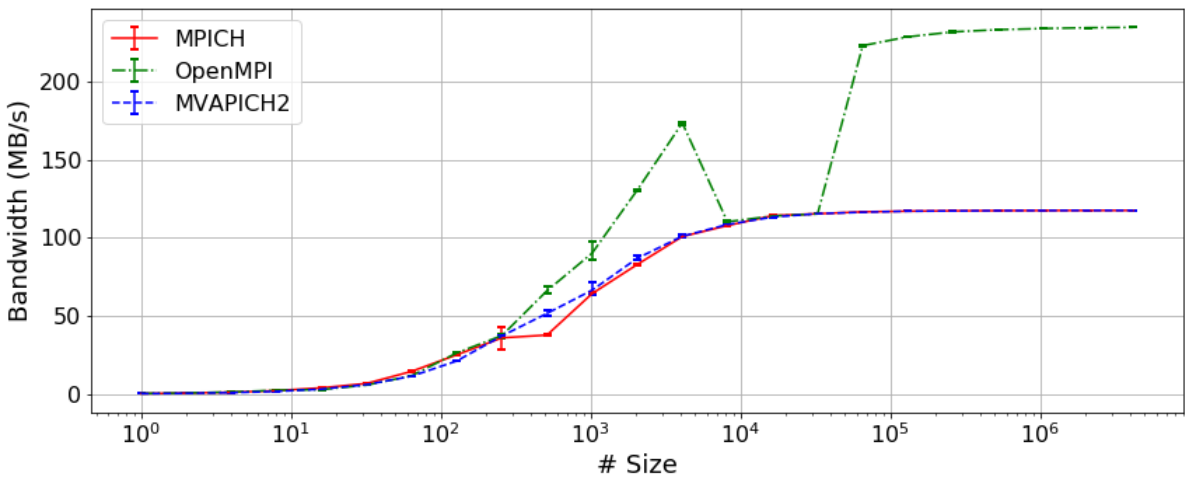
---

put_bibw:

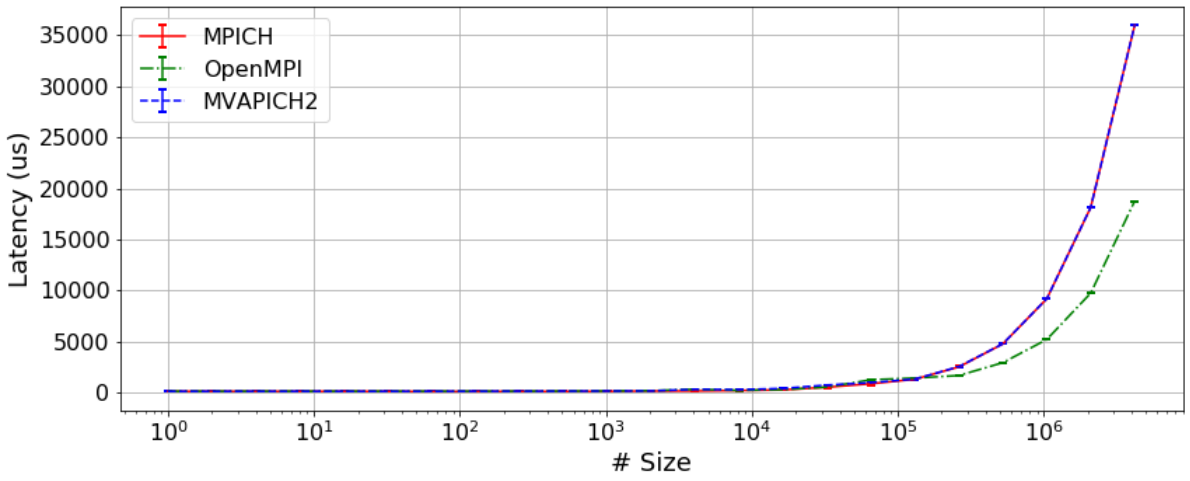


---

put_bw:

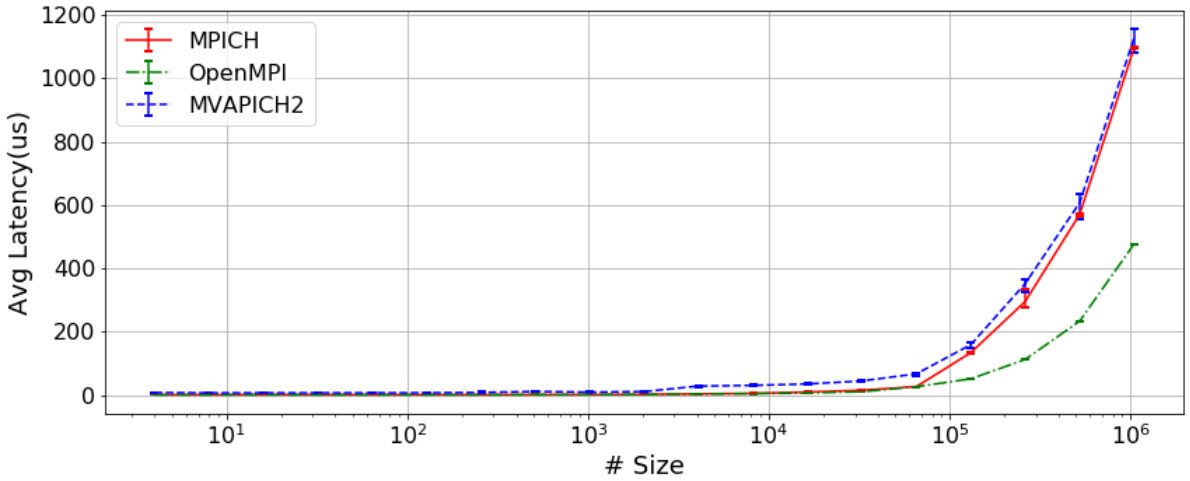


put_latency:



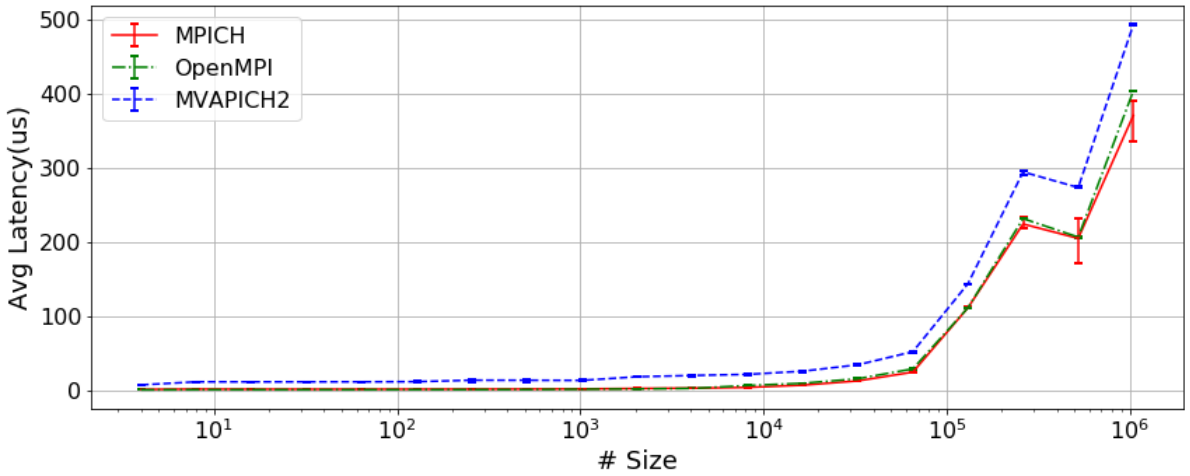
---

reduce:

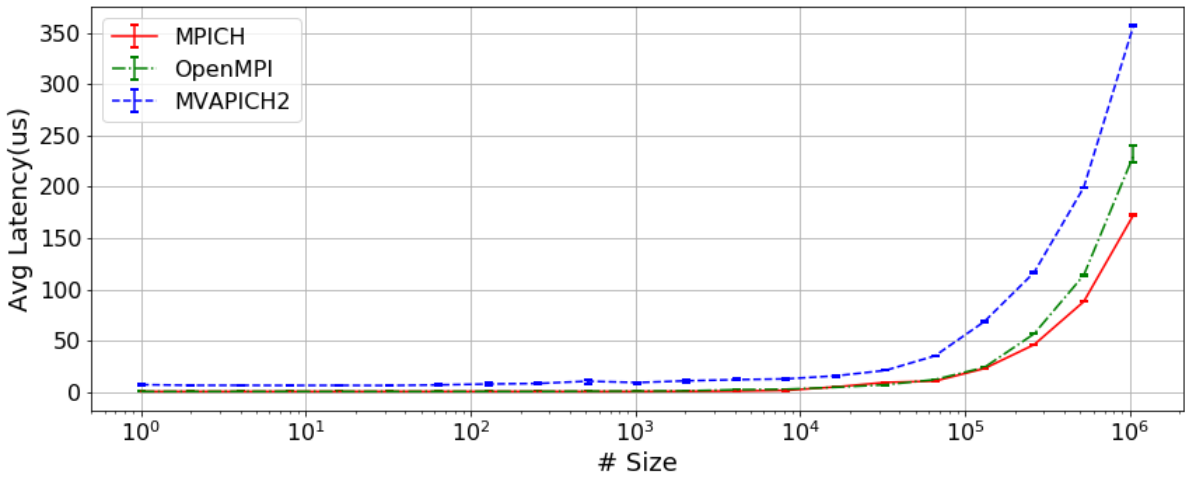


---

reduce_scatter:



scatter:



scatterv: