Open University of Catalonia

# Evaluation of MPI Collectives for HPC Applications on Distributed Virtualized Environments

Gabriel Díaz López de la Llave

gdiazlo@uoc.edu

Advisor: Iván Rodero

# Contents

# Figures

# Tables

# 1
# Introduction

*1. Introduction*

With virtualization technologies becoming mainstream due to its high flexibility and low costs, the interest in running distributed applications, which traditionally runs on physical cluster systems, is rising. However, there are still questions to be answered before these distributed applications could run in cloud computing environments.

One of these questions is what is the impact of the cloud computing overhead for common distributed applications.

The most common way to distribute an application across computing nodes is through Message Passing Interface (MPI) library implementations. We intend to explore how the MPI collective communication is affected by this abstraction and where the bottlenecks occur. This would help in understanding which short-comings need to be addressed in introducing HPC applications to distributed virtualized environments (i.e. Clouds) where the scalability and on-demand resource provisioning may prove vital for urgent need.

We used the Weather Research and Forecasting (WRF) model, which is a state-of-the-art numerical modeling application developed by the National Center for Atmospheric Research (NCAR) for both operational forecasting and atmospheric research, as the main use case of a distributed application

WRF simulations demand a large amount of computational power, and they need to execute as rapidly as possible, however, on distributed virtualized environments the overhead of communication increases due to the abstraction layer.

*1.1 Intellectual Merit*

Cloud computing has grown significantly in recent years, however, more research is necessary in order to assess the viability of providing HPC as a service in a cloud. Even with high-speed, gigabit-scale network connectivity and more recently, infiniband, the time delay of communication on distributed virtualized environments is high in comparison to Grids and Supercomputers.[1]

Through our work we will extensively study the necessary requirements to deploy HPC applications in a virtualized environment complementing exiting research on FutureGrid[2] , taking into account key metrics (e.g. throughput, communication latency, etc.) and more detailed analysis of MPI programs.

This will help us understand where clouds stand in relation to grids/supercomputers thus allowing us to focus our efforts on improving the current cloud infrastructure to perform on-par with conventional supercomputing facilities. Our motivation is to develop a new standard for HPC Clouds where the large scalability, high availability and fault-tolerance would provide

scientists with a unique advantage when running simulations.

### *1.2 Related works*

Research about cloud computing performance is becoming more common as its popularity raises. For example work done by S. Ostermann, A. Iosup, N. Yigibasi, R. Prodan, T. Fahringer, and D. Epema, *An early performance analysis of cloud computing services for scientific computing* in 2008 and other related works[3] [4] [5] [6] carried out in the last years.

In contrast with these studies, our aim is to study the behavior of MPI collective operations on production-grade applications and see how it behaves in different scenarios.

In this study we didn't measure the times needed to bring up each environment (physical vs. virtual) which could be determinant in some situations to choose one or another, like S. Ostermann et al. did. We do not focus on the virtual infrastruture per se, only in the performance achieved by the application and the MPI collective calls.

## *2. Objetives*

Through this study, we will measure how the collective MPI operations behaves in virtual and physical clusters, and its impact on the application performance. As we stated before, we will use as a test case the Weather Research and Forecasting simulations.

For that purpose, we will use the FutureGrid facilities to run, measure and analyze WRF simulations, as it is the best platform available which has virtual and physical clusters. Also, we will use Tunning and Analysis Utilities (TAU) to measure the execution of the simulation.

## *3. Plan*

We have divided the project in five phases:

- **Plan** In this phase a plan is established to measure progress and to ensure the achievement of the objectives. In order to do an accurate plan we will do a background review of the technologies and methodologies involved.
- **Environment set up** Resources within FutureGrid are complex and need to be adapted to our needs. During this phase we will create an environment on which WRF tests cases can be executed and measured on physical and virtual clusters.
- **Test execution** Once the environment is prepared to run the WRF tests cases, we will execute various tests measuring performance of MPI calls.
- **Analysis** We will compare the measured results, such as execution times and latency, in order to see the differences between virtualized and physical clusters.
- **Report** All the results will be collected as a final report.

The Gantt diagram of the Figure 1 describes the plan in context with the time.

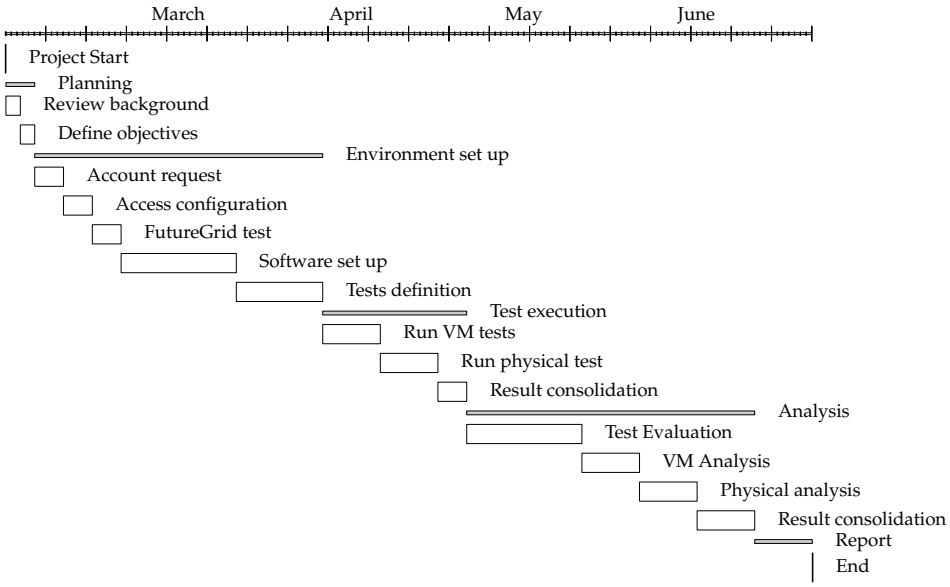



**Figure 1.** Project plan

*4. Organization*

This document is organized in five chapters which covers the most relevant topics for this work. In Chapter 2, we describe the FutureGrid facilities and services, on which we will execute the tests. In Chapter 3, we introduce MPI, focusing on the collective functions. In Chapter 4, we describe the profiling tools and define the analysis methodology and tests to be executed. In Chapter 5 we will describe the environment set up for the experiment, taking in account the tests requirements. In Chapter 6 we summarize the analyzed data and state the conclusion of the analysis. Finally, in Chapter 7 we included a summary of all measures taken during the experiment execution, and in Chapter 8, the bibliographic references.

# 2
# Futuregrid

*1. Introduction*

The FutureGrid Project provides a capability that makes it possible for researchers to tackle complex research challenges in computer science related to the use and security of grids and clouds.

This capability has been materialized in a a geographically distributed set of heterogeneous computing systems, a data management system and a network, which together provides the infrastructure foundation of many experiments.

The FutureGrid platform is composed of a set of hardware and software resources developed to help the researchers manage virtual machines and physical resources easily, providing batch, grid and cloud computing capabilities.

One of the main goals of the project is to make possible the measurement of the overhead of cloud technology by requesting linked experiments on both virtual and bare-metal systems.

*2. Accesing FutureGrid*

All users must register in the web portal[1] Once the registration process is complete, the user must join a project before being able to use FutureGrid resources.

The cloud services are provided by the Nimbus platform. The system requires a SSH key, which must be uploaded to the web portal, to authenticate users to the log in servers. When an user upload a new SSH key, the portal provision it to the appropriate resources, and this process can last between 24 or 48 hours.

The HPC services are provided by reserved hardware, called partitions, on each site. With the default log in account created in the portal, users can connect to login servers to launch HPC jobs using the installed job scheduler.

Both services are accessible connecting to the log in servers, named after the site's name. Although Nimbus services could be accessed using the Nimbus Cloud Client and the credentials issued by the FutureGrid portal.

*3. Infrastructure*

FutureGrid infrastructure is composed of various hardware resources located in various universities across the United States. Each facility has a number of nodes for each of the services

1. Web Portal URL `https://portal.futuregrid.org/`

provided by FutureGrid.

The following table shows a summary of the resources allocated on each site:

| Name | System type | # Nodes | # CPUs | # Cores | Site |
|---|---|---|---|---|---|
| india | IBM iDataPlex | 128 | 256 | 1024 | IU |
| hotel | IBM iDataPlex | 84 | 168 | 672 | UC |
| sierra | IBM iDataPlex | 84 | 168 | 672 | SDSC |
| foxtrot | IBM iDataPlex | 32 | 64 | 256 | UF |
| alamo | Dell PowerEdge | 96 | 192 | 768 | TACC |
| xray | Cray XT5m | 1 | 168 | 672 | IU |
| bravo | HP Proliant | 16 | 32 | 128 | IU |

**TABLE 1.** FutureGrid computing resources summary

All the infrastructure is monitored so the users are able to view the status of all the elements. Once in the portal, you can see the status of the system configuration, network and cloud services using the following in the status web page[1] Each system has a login server from which users can manage each location resources, named after the system name, for example to access hotel login server, a user just need to connect using a ssh client to `hotel.futuregrid.org`

Each site distributes its resources in partitions[2] for its use with Nimbus, HPC, OpenStack and Eucalyptus cloud management software. This means, that the same hardware configuration is used to execute programs in virtual and physical environments, which is key for the success of our analysis.

| Resource | HPC | Eucalyptus | Nimbus | Openstack |
|---|---|---|---|---|
| IU-INDIA (1208 cores) | 37.7% (456 cores) | 17.9% (216 cores) | | 9.3% (112 cores) |
| IU-XRAY (392 cores) | 100% (392 cores) | | | |
| TACC-ALAMO (542 cores) | 94.5% (512 cores) | | 2.6% (14 cores) | |
| UC-HOTEL (568 cores) | 53.5% (304 cores) | | 45.1% (256 cores) | |
| UCSD-SIERRA (648 cores) | 38.3% (248 cores) | 17.3% (112 cores) | 22.2% (144 cores) | 3.7% (24 cores) |
| UFL-FOXTROT (200 cores) | | | 0,96 (192 cores) | |

**TABLE 2.** FutureGrid Resources partitions for each location

*4. Nimbus cloud environment*

The Nimbus framework enable providers of resources, like FutureGrid, to build IaaS clouds. The

1. Status page URL: `https://portal.futuregrid.org/status`

2. Partition table URL: `http://inca.futuregrid.org:8080/inca/jsp/partitionTable.jsp`

Nimbus Workspace Service provides an implementation of a compute cloud allowing users to lease computational resources by deploying virtual machines (VMs) on those resources.

It provides an integrated set of tools designed to enable users to move to the cloud quickly and effortlessly, automating and facilitating much of the process. It provides a bridge allowing a user to overlay familiar concepts, such as virtual clusters, onto the resources provisioned in the cloud, but with the flexibility to support many different configurations and requirements.

The FutureGrid Nimbus platform can be accessed using the Nimbus Cloud Client, which is a command line utility to manage Nimbus cloud resources, basically groups of virtual machines.

### *4.1 Virtual clusters*

A virtual cluster is a set of virtual machines which work together to solve computing problems. Nimbus is able to manage virtual clusters easily through its cloud client tool.

In order to tell the Nimbus system how must build our cluster, we need to create a cluster definition file[1] indicating how many machines we want to launch, if we will use the context broker service, etc.

Each cluster runs its own configuration which is called a context. The context broker service is in charge to bring up the machines and configure them with a new set of credentials. The usage of the context server is optional, so users are able to configure clusters using their own resources.

With the cluster definition file, we can use the cloud client tool to bring a cluster up using the option `--cluster`. For example:

```
$ ./bin/cloud-client.sh --run --hours 1 --cluster samples/base-cluster.xml

SSH known_hosts contained tilde:
  - '~/.ssh/known_hosts' --> '$HOME/.ssh/known_hosts'

Requesting cluster.
  - head-node: image 'base-cluster-cc14.gz', 1 instance
  - compute-nodes: image 'base-cluster-cc14.gz', 2 instances

Context Broker:
    https://svc.uc.futuregrid.org:8443/wsrf/services/NimbusContextBroker

Created new context with broker.

Workspace Factory Service:
    https://svc.uc.futuregrid.org:8443/wsrf/services/WorkspaceFactoryService

Creating workspace "head-node"... done.
  - 149.165.148.144 [ vm-144.uc.futuregrid.org ]

Creating group "compute-nodes"... done.
  - 149.165.148.145 [ vm-145.uc.futuregrid.org ]
  - 149.165.148.146 [ vm-146.uc.futuregrid.org ]

Launching cluster-003... done.
```

**Figure 2.** Nimbus cluster launch

1. http://www.nimbusproject.org/docs/2.4/clouds/clusters2.html

The sample file `base-cluster.xml` can be found in a default Nimbus client installation.

#### *4.1.1 Cluster images*

Virtual machines in FutureGrid Nimbus use file disk images as their virtual hard disks. These images contains the operating system and, may be, the applications to executed in the nodes of the cluster.

FutureGrid provides a set of images that are to use or adapt to the user needs, like adding software packages, upgrading the operating system, etc.

## *5. HPC cloud environment*

Several of the clusters that are part of FutureGrid have partitions that operate as High Performance Computing (HPC) systems. These partitions are composed of a set of computing nodes able to run scheduled jobs. Also, these nodes are not virtualized, and this means they a fixed operating system and configuration, so users must adapt their applications to execute on them because they can't change its configuration dynamically.

Users willing to execute jobs on these clusters must use the TORQUE[1] Resource Manager. This software is in charge to schedule and run the jobs submitted by all the users.

In HPC terminology, a job is the execution of one or various programs under the same environment, with a defined set of resources and during an established time window.

In order to submit a job using TORQUE, we need to create a job definition file with the following structure:

```
#!/bin/bash
#PBS -N hostname_test
#PBS -o hostname.out
#PBS -e hostname.err
#PBS -q short
#PBS -l nodes=1
#PBS -l walltime=00:20:00

/bin/hostname
```

**Figure 3.** HPC Job definition file sample

The `#PBS` section describe various parameters related to the TORQUE system itself and the general execution parameters of the job, while the rest is a standard shell script that should contain the program to be executed.

The options are passed to Torque on lines that begin with #PBS. The options above are:

- An optional job name.
- The name of the file to write stdout to.
- The name of the file to write stderr to.
- The queue to submit the job to.

1. TORQUE stands for Terascale Open-Source Resource and QUEue Manager

- The resources needed by the job. In the case above, 1 node for 20 minutes.

This script must prepare the environment for the program execution including all the resources on which the program depends like libraries, data files, or configuration files. All the configuration options are documented in the official TORQUE manuals at its official web site[2]

### *5.1 HPC Queues*

The TORQUE scheduler is organized around the queues concept. Each scheduler has different queues of jobs to be launched. Each queue have unique properties which impact in how and when the jobs are executed. Typically, there are queues for long jobs, for batch jobs or for ad hoc jobs, and each one with a set of resources assigned, with time waits and other limitations in place.

Its important to select the correct queue in order to be able to get the requested resources and to be able to run the job in the time needed.

2. TORQUE web site URL: `http://www.adaptivecomputing.com/products/open-source/torque/`

# 3
# Message Passing Interface (MPI)

*1. Introduction*

Message Passing Interface (MPI) is an international standard, created by the Message Passing Interface Forum (MPIF), with the goal of developing a widely used standard for writing message-passing programs in a practical, portable, efficient, and flexible way[7] [8]

The paradigm of communication though a message-passing approach ensures wide portability across diferent computer achitectures, the only requisite is to honor the messaging format. It could also be abstracted from transport protocols, adding enough flexibility to support from distributed-memory multicomputers, shared-memory multiprocessors, networks of workstations, and combinations of all of these. Also, this independence makes possible the implementation of the standard in many platforms, and because it is platform-agnostic, the evolution of computing architectures doesn't invalidates the paradigm including new technologies.

In this Chapter we will describe the main MPI concepts to define what are the MPI collective calls. Also we will include the standard definition of those calls, including its call signature and purpose as stated in the standard.

*2. MPI Conventions*

In order to understand how MPI collective operation works, we first need to introduce some concepts core to the MPI standard like processes, communicators, groups, etc.

In the following section we will provide a brief of this concepts based on the publicly available MPI standard.

*2.1 Processes*

An MPI program consists of autonomous processes, executing their own code. Each process need not be identical and communicate via calls to MPI communication primitives. Normally, each process executes in its own address space, although there are shared-memory implementations of MPI.

MPI does not specify the execution model for each process and allows the use of threads which might execute concurrently. This is possible because MPI does use implicit state for its work, and allows a thread blocked by an MPI call to be scheduled away in favor or non blocked threads.

Also, MPI does not provide facilities to do the initial allocation of processes to a given set of MPI computation nodes, so each MPI implementation should provide its own means (like the Hydra process manager from MPICH2 that we will see later on this chapter).

Finally, MPI identifies each process according to their relative rank in a group, which is an integer

starting from 0 to the number of elements of the group (for a group of ten, we will have process 0 through process 9).

### *2.2 Division of Processes*

MPI provides facilities to allow different groups of processes work with in dependency of others. For example, applications that want to use 30% of its processes to work with new data while the other 70% of processes work to analyze data already processed. And in case there is no new data, dedicate all processes to analyze already processed data.

This capability relies no only in the MPI implementation but in the application it self, as it need to be able to divide the processes into independent subsets, taking care of the process ranking as there should be always a root process on each group.

This facilities are also used by applications that use collective operations on a subset of processes. The time to complete most collective operations increases with the number of processes, so its important to execute this kind of operations in the optimal group to get a much better scaling behavior than if are executed for all the nodes. For example, if a matrix computation needs to broadcast information along the diagonal of a matrix, only the processes containing diagonal elements should be involved.

### *2.3 Communicators*

MPI communicators are opaque objects that encompass several fundamental ideas in MPI, and its importance also relies on the fact that almost all MPI calls has a communicator as an argument.

This object has a number of attributes and rules that govern itself, determining the scope and domain in which a point-to-point or collective operation operates.

Each communicator contains a group of participant processes in which the source and destination of a message is identified by the process rank within the group.

There are intracommunicators, used for communicating within a single group of processes, and intercommunicators, used for communicating within two or more groups of processes. In MPI-2, the intercommunicator can be used for collective communication within two or more groups of processes.

#### *2.3.1 Groups*

A group is an ordered set of process identifiers (henceforth processes) that cannot be transfered between processes as are represented by opaque implementation-dependent objects. Each process in a group is associated with an rank which is an contiguous integer starting from zero.

There is a special pre-defined group: `MPI_GROUP_EMPTY`, which is a group with no members, but valid to handle to an empty group, and `MPI_GROUP_NULL` which is the value used for invalid group handles. `MPI_GROUP_EMPTY` should not be confused with `MPI_GROUP_NULL`, which is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, in not a valid argument

#### *2.3.2 Communicators*

A communicator is an opaque object with a number of attributes, together with simple rules that govern its creation, use and destruction. The communicator specifies a communication domain which can be used for communications. There are two types of communicators:

intracommunicators and intercomuunicators.

An intracommunicator is used for communicating within a single group of processes; we call such communication intra-group communication. An intracommunicator has two fixed attributes, the process group and the topology describing the logical layout of the processes in the group. Intracommunicators are also used for collective operations within a group of processes.

An intercommunicator is used for point-to-point communication between two disjoint groups of processes. We call such communication inter-group communication. The fixed attributes of an intercommunicator are the two groups. No topology is associated with an intercommunicator. In addition to fixed attributes a communicator may also have user-defined attributes which are associated with the communicator using MPI's caching mechanism.

The table below summarizes the differences between intracommunicators and intercommunicators.

| Functionality | Intracommunicator | Intercommunicator |
|---|---|---|
| #of groups | 1 | 2 |
| Communication Safety | Yes | Yes |
| Collective Operations | Yes | No |
| Topologies | Yes | No |
| Caching | Yes | Yes |

**TABLE 3.** Intra vs. inter communicator

*2.4 Communicator domains*

A communication domain is a set of processes communicators. Any point-to-point or collective communication occurs in MPI within a communication domain. Each communicator is the local representation of the global communication domain. If this domain is for intra-group communication then all the communicators are intracommunicators, and all have the same group attribute. Each communicator identifies all the other corresponding communicators.

A communicator can be viewed as an array of links to other communicators where:

- their links form a complete graph: each communicator is linked to all communicators in the set, including itself; and
- links have consistent indexes: at each communicator, the i-th link points to the communicator for process i.

This distributed data structure is illustrated below , for the case of a three member group.

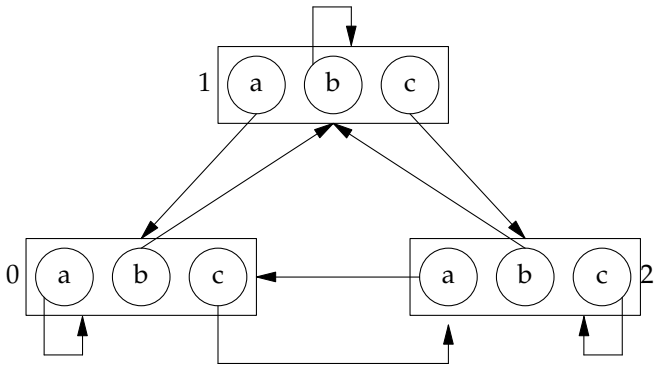



**Figure 4.** Distributed data structure for intra-communication domain.

*2.5 Intracommunication*

In point-to-point communication, matching send and receive calls should have communicator arguments that represent the same communication domains. The rank of the processes is interpreted relative to the group, or groups, associated with the communicator. Thus, in an intra-group communication domain, process ranks are relative to the group associated with the communicator.

Similarly, a collective communication call involves all processes in the group of an intra-group communication domain, and all processes should use a communicator argument that represents this domain. Intercommunicators may not be used in collective communication operations.

Each communicator is a different opaque object, local to a process, and communicators that represent the same communication domain may have different attribute values attached to them at different processes.

*2.6 MPI Call types*

When discussing MPI procedures, we can use the following classification, depending on the way an MPI call behaves:

- **Local:** If the completion of the procedure depends only on the local executing process. Such an operation does not require an explicit communication with another user process. MPI calls that generate local objects or query the status of local objects are local.
- **Non-local:** If completion of the procedure may require the execution of some MPI procedure on another process. Many MPI communication calls are non-local.
- **Blocking:** If return from the procedure indicates the user is allowed to re-use resources specified in the call. Any visible change in the state of the calling process affected by a blocking call occurs before the call returns
- **Non-blocking:** If the procedure may return before the operation initiated by the call completes, and before the user is allowed to re-use resources (such as buffers) specified in the call. A nonblocking call may initiate changes in the state of the calling process that actually take place after the call returned: e.g. a nonblocking call can initiate a receive operation, but the message is actually received after the call returned.

- **Collective:** If all processes in a process group need to invoke the procedure.

In the following sections we describe these collective operations, which are the object of this study.

*3. Collective communications*

Collective communications transmit data among all processes in a group specified by an intra-communicator object. One function, the barrier, serves to synchronize processes without passing data. The MPI collective communication functions are described in the following sections.

*3.1 MPI collective calls*

In the MPI-1 standard (Section 4.2), collective operations only apply to intracommunicators; however, most MPI collective operations can be generalized to intercommunicators. To understand how MPI can be extended, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see [9] ):

- **All-To-All:** All processes contribute to the result. All processes receive the result.
  - `MPI_Allgather`
  - `MPI_Allgatherv`
  - `MPI_Alltoall`
  - `MPI_Alltoallv`
  - `MPI_Allreduce`
  - `MPI_Reduce_scatter`
- **All-To-One:** All processes contribute to the result. One process receives the result.
  - `MPI_Gather`
  - `MPI_Gatherv`
  - `MPI_Reduce`
- **One-To-All:** One process contributes to the result. All processes receive the result.
  - `MPI_Bcast`
  - `MPI_Scatter`
  - `MPI_Scatterv`
- **Other:** Collective operations that do not fit into one of the above categories.
  - `MPI_Scan`
  - `MPI_Barrier`

IF the root process sends data to all processes (itself included), the one-to-all procedures, broadcast (`MPI_Bcast`) and scatter (`MPI_Scatter`, `MPI_Scatterv` procedures will be used. Similarly, all-to-oneg procedires, gather (`MPI_Gather`, `MPI_Gatherv`), will be called when root receives data from all processes (itself included). Finally, if each process communicates with each process (itself included), the all-to-all procedures will be used.

We've included each MPI collective call following the MPI Standard v. 2 in the next sections, with

its signature and purpose, as a reference to be able to understand the analysis section.

*3.1.1 Broadcast*

```
MPI_BCAST( buffer, count, datatype, root, comm )
       INOUT    buffer                 starting address of buffer (choice)
       IN       count                  number of entries in buffer (non-negative integer)
       IN       datatype               data type of buffer (handle)
       IN       root                   rank of broadcast root (integer)
       IN       comm                   communicator (handle)
```

**Figure 5.** MPI_BCAST signature

If `comm` is an intracommunicator, `MPI_BCAST` broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for `comm` and `root`. On return, the content of root's buffer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The *in place* option is not meaningful here.

If `comm` is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value `MPI_ROOT` in root. All other processes in group A pass the value `MPI_PROC_NULL` in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

*3.1.2 Gather*

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

   IN       sendbuf                starting address of send buffer (choice)
   IN       sendcount              number of elements in send buffer (non-negative integer)
   IN       sendtype               data type of send buffer elements (handle)
   OUT      recvbuf                address of receive buffer (choice, only for root)
   IN       recvcount              number of elements for any single receive (non-negative integer)
   IN       recvtype               data type of recv buffer elements (handle)
   IN       root                   rank of reciving process
   IN       comm                   communicator
```

**Figure 6.** MPI_GATHER signature

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to `MPI_Send(sendbuf, sendcount, sendtype, root, ...)`, and the root had executed n calls to `MPI_Recv(recvbuf+i-recvcount-extent(recvtype), recvcount,`

0. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

recvtype, i ,...), where extent(recvtype) is the type extent obtained from a call to MPI_Type_extent().

An alternative description is that the *n* messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount-n, recvtype, ...).

The type signature of sendcount and sendtype on process *i* must be equal to the type signature of recvcount and recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the recvcount argument at the root indicates the number of items it receives from each process, not the total number of items it receives.

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                    displs, recvtype, root, comm)
  IN       sendbuf          starting address of send buffer
  IN       sendcount        number of elements in send buffer
  IN       sendtype         datatype of send buffer elements
  OUT      recvbuf          address of receive buffer
  IN       recvcounts       integer array
  IN       displs           integer array of displacements
  IN       recvtype         data type of recv buffer elements
  IN       root             rank of receiving process
  IN       comm             communicator
```

**Figure 7.** MPI_GATHERV signature

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs.

The outcome is as if each process, including the root process, sends a message to the root, MPI_Send(sendbuf, sendcount, sendtype, root, ...) and the root executes n receives, MPI_Recv(recvbuf+displs[i]-extent(recvtype), recvcounts[i], recvtype, i, ...).

The data sent from process *j* is placed in the jth portion of the receive buffer recvbuf on process root. The jth portion of recvbuf begins at offset displs[j] elements (in terms of recvtype) into recvbuf.

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount and sendtype on process *i* must be equal to the type signature implied by recvcounts[i] and recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example .

All arguments to the function are significant on process root, while on other processes, only

arguments `sendbuf`, `sendcount`, `sendtype`, `root`, and `comm` are significant. The argument root must have identical values on all processes, and `comm` must represent the same intragroup communication domain.

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous. On the other hand, the successive displacements in the array `displs` need not be a monotonic sequence.

### *3.1.3 Scatter*

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                    recvtype, root, comm)

  IN       sendbuf          address of send buffer
  IN       sendcount        number of elements send to
                                    each process
  IN       sendtype         datatype of send buffer elements
  OUT      recvbuf          address of receive buffer
  IN       recvcount        number of elements in
                                    receive buffer
  IN       recvtype         data type of recv buffer elements
  IN       root             rank of sending process
  IN       comm             communicator
```

**Figure 8.** MPI_SCATTER signature

`MPI_SCATTER` is the inverse operation to `MPI_GATHER`.

The outcome is as if the root executed n send operations, `MPI_Send(sendbuf+i-sendcount-extent(sendtype), sendcount, sendtype, i,...)`, *i=0 to n-1*. and each process executed a receive, `MPI_Recv(recvbuf, recvcount, recvtype, root,...)`.

An alternative description is that the root sends a message with `MPI_Send(sendbuf, sendcount-n, sendtype, ...)`. This message is split into *n* equal segments, the *ith* segment is sent to the *ith* process in the group, and each process receives this message as above.

The type signature associated with `sendcount` and `sendtype` at the root must be equal to the type signature associated with `recvcount` and `recvtype` at all processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments `recvbuf`, `recvcount`, `recvtype`, `root`, `comm` are significant. The argument `root` must have identical values on all processes and `comm` must represent the same intragroup communication domain. The send buffer is ignored for all non-root processes.

The specification of counts and types should not cause any location on the root to be read more than once.

### *3.1.4 MPI_SCATTERV*

```
MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                    recvtype, root, comm)

  IN       sendbuf          address of send buffer
```

```
IN       sendcounts       integer array
IN       displs           integer array of displacements
IN       sendtype         datatype of send buffer elements
OUT      recvbuf          address of receive buffer
IN       recvcount        number of elements in
                                    receive buffer
IN       recvtype         data type of recv buffer elements
IN       root             rank of sending process
IN       comm             communicator
```

**Figure 9.** MPI_SCATTERV signature

`MPI_SCATTERV` is the inverse operation to `MPI_GATHERV`. It extends the functionality of `MPI_SCATTER` by allowing a varying count of data to be sent to each process, since `sendcounts` is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing the new argument, `displs`.

The outcome is as if the root executed n send operations, `MPI_Send(sendbuf+displs[i]-extent(sendtype), sendcounts[i], sendtype, i,...)`,*i=0 to n-1,* and each process executed a receive, `MPI_Recv(recvbuf, recvcount, recvtype, root,...)`.

The type signature implied by `sendcount[i]` and `sendtype` at the root must be equal to the type signature implied by `recvcount` and `recvtype` at process *i*. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments `recvbuf`, `recvcount`, `recvtype`, `root`, `comm` are significant. The arguments root must have identical values on all processes, and comm must represent the same intragroup communication domain. The send buffer is ignored for all non-root processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

*3.1.5 Gather-to-all*

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                    recvtype, comm)

IN       sendbuf          starting address of send buffer
IN       sendcount        number of elements in send buffer
IN       sendtype         datatype of send buffer elements
OUT      recvbuf          address of receive buffer
IN       recvcount        number of elements received
                                 from any process
IN       recvtype         data type of recv buffer elements
IN       comm             communicator
```

**Figure 10.** MPI_ALLGATHER signature

`MPI_ALLGATHER` can be thought of as `MPI_GATHER`, except all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer `recvbuf`.

The type signature associated with `sendcount` and `sendtype` at a process must be equal to the type signature associated with `recvcount` and `recvtype` at any other process.

The outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm), for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

```
MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
       void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
       MPI_Comm comm)
```

**Figure 11.** MPI_ALLGATHERV signature

MPI_ALLGATHERV can be thought of as MPI_GATHERV, except all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount and sendtype at process j must be equal to the type signature associated with recvcounts[j] and recvtype at any other process.

The outcome is as if all processes executed calls to MPI_GATHERV( sendbuf, sendcount, sendtype,recvbuf,recvcounts,displs, recvtype,root,comm), for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

*3.1.6 All-to-All Scatter/Gather*

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
    IN       sendbuf          starting address of send buffer
    IN       sendcount        number of elements sent to each process
    IN       sendtype         data type of send buffer elements
    OUT      recvbuf          address of receive buffer
    IN       recvcount        number of elements received from any process
    IN       recvtype         data type of receive buffer elements
    IN       comm             communicator
```

**Figure 12.** MPI_ALLTOALL signature

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

The type signature associated with sendcount and sendtype at a process must be equal to the type signature associated with recvcount and recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

The outcome is as if each process executed a send to each process (itself included) with a call to, MPI_Send(sendbuf+i-sendcount-extent(sendtype), sendcount, sendtype, i, ...), and a receive from every other process with a call to, MPI_Recv(recvbuf+i- recvcount-extent(recvtype), recvcount, i,...), where i = 0, ..., n - 1.

All arguments on all processes are significant. The argument comm must represent the same intragroup communication domain on all processes.

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recv-
```

```
type, comm)
    IN       sendbuf          starting address of send buffer (choice)
    IN       sendcounts       non-negative integer array (of length group size) specifying
                              the number of elements to send to each processor
    IN       sdispls          integer array (of length group size). Entry j specifies
                              the displacement (relative to sendbuf from which to
                              take the outgoing data destined for process j
    IN       sendtype         data type of send buffer elements (handle)
    OUT       recvbuf         address of receive buffer (choice)
    IN       recvcounts       non-negative integer array (of length group size) specifying
                              the number of elements that can be received
                              from each processor
    IN       rdispls          integer array (of length group size). Entry i specifies
                              the displacement (relative to recvbuf at which to place
                              the incoming data from process i
    IN       recvtype         data type of receive buffer elements (handle)
    IN       comm             communicator (handle)
```

**Figure 13.** MPI_ALLTOALLV signature

`MPI_ALLTOALLV` adds flexibility to `MPI_ALLTOALL` in that the location of data for the send is specified by `sdispls` and the location of the placement of the data on the receive side is specified by `rdispls`.

The jth block sent from process i is received by process j and is placed in the ith block of `recvbuf`. These blocks need not all have the same size.

The type signature associated with `sendcount[j]` and sendtype at process i must be equal to the type signature associated with `recvcount[i]` and `recvtype` at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to process i with `MPI_Send( sendbuf + displs[i]-extent(sendtype), sendcounts[i], sendtype, i, ...)`, and received a message from process i with a call to `MPI_Recv( recvbuf + displs[i]-extent(recvtype), recvcounts[i], recvtype, i, ...)`, where i = 0 ... n - 1.

All arguments on all processes are significant. The argument comm must specify the same intragroup communication domain on all processes.

Rationale. The definition of MPI_ALLTOALLV gives as much flexibility as one would achieve by specifying at each process n independent, point-to-point communications with the exceptions that all messages use the same datatype.

```
MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recv-
types, comm)
    IN       sendbuf          starting address of send buffer (choice)
    IN       sendcounts       non-negative integer array (of length group size) specifying
                              the number of elements to send to each processor
    IN       sdispls          integer array (of length group size). Entry j specifies
                              the displacement in bytes (relative to sendbuf) from
                              which to take the outgoing data destined for process
                              j (array of integers)
    IN       sendtypes        array of datatypes (of length group size). Entry j
                              specifies the type of data to send to process j (array
                              of handles)
    OUT      recvbuf          address of receive buffer (choice)
    IN       recvcounts       non-negative integer array (of length group size) specifying
                              the number of elements that can be received
```

```
                        from each processor
    IN       rdispls    integer array (of length group size). Entry i specifies
                        the displacement in bytes (relative to recvbuf) at which
                        to place the incoming data from process i (array of
                        integers)
    IN       recvtypes  array of datatypes (of length group size). Entry i
                        specifies the type of data received from process i (array
                        of handles)
    IN       comm       communicator (handle)
```

**Figure 14.** MPI_ALLTOALLW signature

MPI_ALLTOALLW is the most general form of complete exchange. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum exibility, the displacement of blocks within the send and receive buffers is specified in bytes.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with MPI_Send(sendbuf + sdispls[i], sendcounts[i], sendtypes[i], i, ...), and received a message from every other process with a call to MPI_Recv(recvbuf + rdispls[i], recvcounts[i], recvtypes[i];¡, i, ...).

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

Like for MPI_ALLTOALLV, the *in place* option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts and recvtypes arrays, and is taken from the locations of the receive buffer specified by rdispls.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

### *3.1.7 Reduce*

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
    IN       sendbuf    address of send buffer (choice)
    OUT      recvbuf    address of receive buffer (choice, significant only at root)
    IN       count      number of elements in send buffer (non-negative integer)
    IN       datatype   data type of elements of send buffer (handle)
    IN       op         reduce operation (handle)
    IN       root       rank of root process (integer)
    IN       comm       communicator (handle)
```

**Figure 15.** MPI_REDUCE signature

If comm is an intracommunicator, `MPI_REDUCE` combines the elements provided in the input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments `sendbuf`, `count` and `datatype`; the output buffer is defined by the arguments `recvbuf`, `count` and `datatype`; both have the same number of elements, with the same type.

The routine is called by all group members using the same arguments for `count`, `datatype`, `op`, `root` and `comm`. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is `MPI_MAX` and the send buffer contains two elements that are floating point numbers (`count = 2` and `datatype = MPI_FLOAT`), then `recvbuf(1) = global max(sendbuf(1))` and `recvbuf(2) = global max(sendbuf(2))`.

```
MPI_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)
    IN       sendbuf          starting address of send buffer (choice)
    OUT      recvbuf          starting address of receive buffer (choice)
    IN       counts           number of elements in send buffer (non-negative integer)
    IN       datatype         data type of elements of send buffer (handle)
    IN       op               operation (handle)
    IN       comm             communicator (handle)
```

**Figure 16.** MPI_ALLREDUCE signature

If `comm` is an intracommunicator, `MPI_ALLREDUCE` behaves the same as `MPI_REDUCE` except that the result appears in the receive buffer of all the group members.

The *in place* option for intracommunicators is specified by passing the value `MPI_IN_PLACE` to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

*4. MPI Process manager*

As we already stated, MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their binding to physical processors. It is expected that vendors will provide mechanisms to do so either at load time or at run time. We have selected the MPICH2 library, which use the Hydra process management framework for this purpose: allocate processes in computation nodes.

*4.1 Hydra process management framework*

The Hydra framework is composed of various components which work together with a defined interface. This approach gives the framework the flexibility needed to be adapted to a lot of environments. The following list describe its basic components:

- User Interface (UI, e.g., mpiexec)
- Resource Management Kernel (RMK)

- Process manager
- Bootstrap server (e.g., ssh, fork, pbs, slurm, sge)
- Process Binding (e.g., plpa)
- Communication Subsystem (e.g., IB, MX)
- Process Management proxy
- I/O demux engine

And the next diagram show the relations between each of the components:

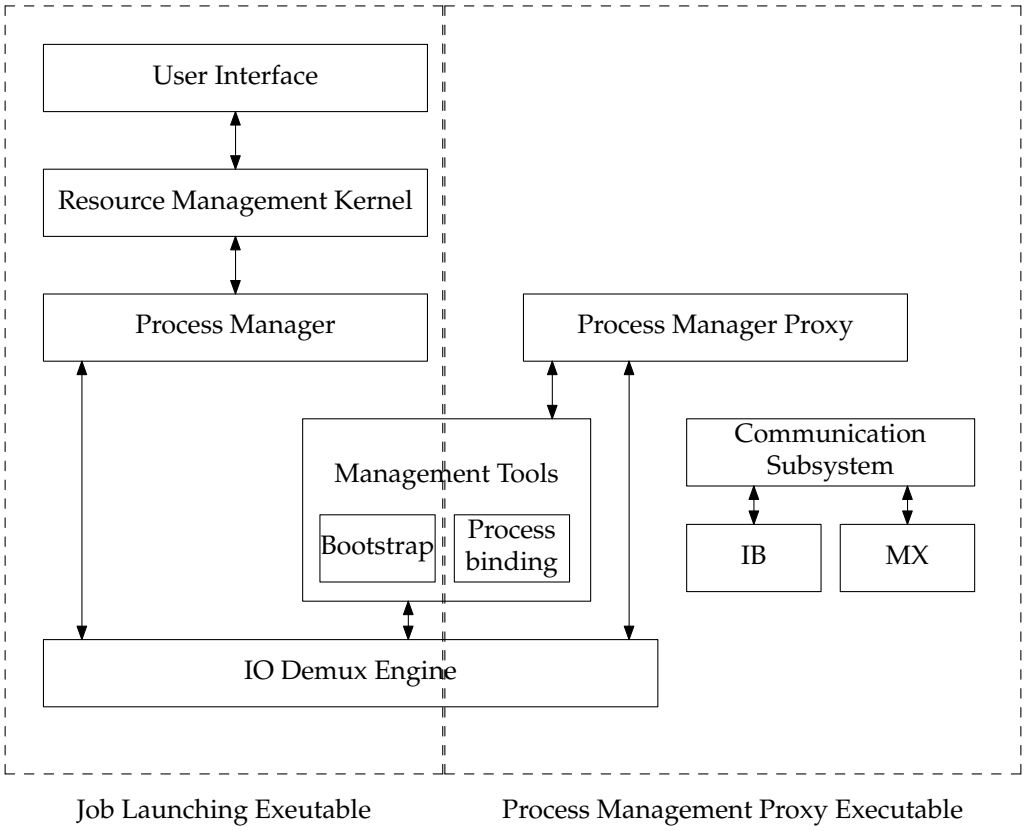



**Figure 17.** Hydra process manager architecture

- **User Interface:** The main responsibility of this layer is to collect information from the user with respect to the application, where to launch the processes, mappings of processes to cores, reading stdin and forwarding it to the appropriate process(es), reading stdout/stderr from different processes and directing it appropriately.
- **Resource Management Kernel (RMK):** The RMK provides plug-in capabilities to interact with resource managers (e.g., Torque). For example, if the application needs to allocate nodes on a system before launching a job, the RMK plays this part. Similarly, the RMK can also allow for decoupled job launching in cases where a single system reservation is to be used for multiple jobs.

- **Process Manager:** The process manager deals with providing processes with the necessary environment setup as well as the primary process management functionality. For example, the "pmiserv" process manager provides the MPICH2 PMI (Process Management Interface) functionality.
- **Process Management Proxy:** The process management proxy is a helper agent that is spawned at each node on the system to assist the process manager with process spawning, process cleanup, signal forwarding, I/O forwarding, and any process manager specific functionality as well. It can basically perform any task the process manager can do, thus, even a hierarchy of process management proxies can be created, where each proxy acts as a process manager for its sub-tree.
- **Bootstrap Server:** The bootstrap server mainly functions as a pre-configured daemon system that allows the upper layers to launch processes throughout the system. For example, the ssh bootstrap server forks off processes, each of which performs an ssh to a single machine to launch one process. The SSH back end have been used in the virtual environment experiments as a bootstrap server, instead of the Torque resource manager.
- **Processing Binding:** Deals with extracting the system architecture information (such as the number of processors, cores and SMT threads available, their topology, shared cache, etc.) and binding processes to different cores in a portable manner.
- **Communication Subsystem:** The communication sub-system is a way for different proxies to communicate with each other in a scalable manner. This is only relevant for the pre-launched and pre-connection proxies that are described below. This component provides a scalable communication mechanism irrespective of the system scale (e.g., InfiniBand UD based or Myrinet MX based).
- **I/O Demux Engine:** This engine acts as a centralized event management mechanism, where different components can "register" their file-descriptors and the demux engine can wait for events on any of these descriptors. The I/O demux engine uses a synchronous callback mechanism, which implies that with each file descriptor, the calling processes provides a function pointer to the function that has to be called when an event occurs on the descriptor. The demux engine blocks for events on all its registered file descriptors and calls the appropriate callbacks whenever there is an event.

#### *4.1.1 Implementation and control flow*

There are three basic executable classes in the current model: UI process, process management proxy and the application process. The following diagram shows a typical structure of how these classes communication is organized.

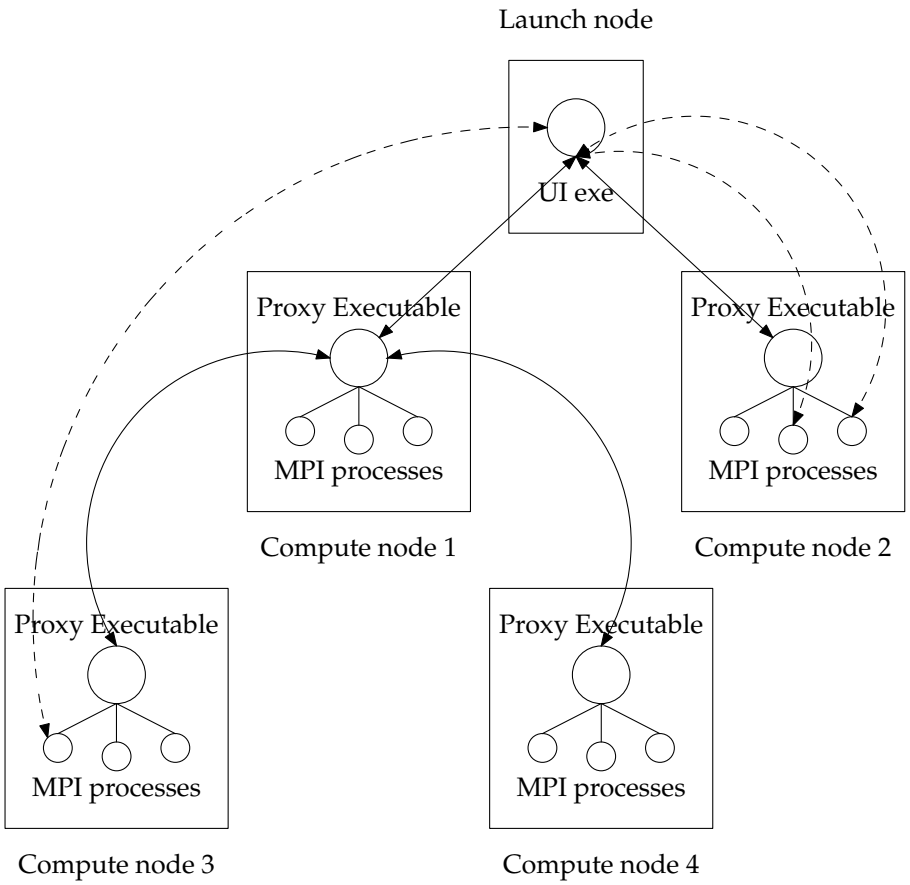



**Figure 18.** Hydra processes communications

- **UI Process:**
  1. The UI process (e.g., mpiexec) takes the user arguments, environment settings, information about nodes where the process has to be launched and other such details provided by the user and passes them to the RMK.
  2. The RMK, if appropriate, interacts with the resource manager to either verify access to the nodes on which the processes need to be launched or to allocate the nodes, and passes this information to the process manager.
  3. The process manager creates an executable argument list to launch the proxy containing information about the application itself and its arguments, the environment that needs to be setup for the application and the partition of processing elements that this proxy would be responsible for. The environment also contains additional information that can allow the PMI client library on the application to connect to the process manager.
  4. The executable argument list is passed down to the bootstrap server with information on where all to launch this executable; in this case, the executable that is handed over to the bootstrap server is the proxy, but the bootstrap server itself does not interpret this information and can be used transparently for the proxy or the application executable directly. -LI The bootstrap server launches the processes on the appropriate nodes, and keeps track of the stdout, stderr and stdin end points for these processes. This information is passed back to the upper layers.

5. Once the processes have been launched, the UI executable registers the stdout, stderr, stdin file descriptors with the demux engine and waits for events on them. Finally, when all these sockets have been closed, it passes the control back to the bootstrap server that waits for each of the spawned processes to terminate and returns the exit statuses of these processes.

- **Process Management Proxy:** The process management proxy just reads the environment that needs to be setup for each process, forks off the processes and sets up the required environment. The proxy use the bootstrap server itself to launch processes in a tree-like architecture. The proxy forwards I/O information to/from the application processes to its parent proxy or the main process manager.
- **Process Cleanup During Faults:** The Hydra framework contemplates two types of faults: an application processes dies or calls an abort, or the timeout specified by the user expires. In both cases, Hydra tries to clean up all the remaining application processes, using the capabilities of the bootstrap server to forward signals, or sending the appropriate exit code though the process management proxy.

# 4
# Measuring MPI performance

*1. Profiling tools*

Profiling tools have the ability to measure how a program behave, so the user can identify possible performance problems within their applications.

There are two main approaches to measure and profile HPC applications: through static instrumentation, on which the programmer or the compiler insert into the program calls to the profiling library, or dynamically through run-code analysis and breakpoints.

The static instrumentation normally requires a special compiler which is in charge of inserting the profiling calls automatically, given a configuration criteria, for example, profile all MPI calls. This method is able to generate a lot of profiling data of any part of the application, given that the source code is modified with calls to the profiling library.

On the other hand, dynamic profiling is able to perform measures even if we do not have the source code of the application. We have evaluated VampirTrace and TAU profiling environments, selecting TAU due to its free analysis tools, as VampirTrace uses the Vampir analysis tools which is only available under commercial license.

We have chosen the TAU utilities to collect performance and usage statistics for each simulation. We did it because it supports all the software we use, and all its components are freely available, from the instrumentation library to the visualization tools. Also supports dynamic instrumentation of MPI (using library preloading techniques) but does not have external requirements, unlike VampirTrace, which requires another components.

*2. Tuning and Analysis Utilities (TAU)*

TAU is a program and performance analysis tool framework being developed jointly by the University of Oregon, Los Alamos National Laboratory, and Research Centre Jülich, ZAM, Germany.

It provides a suite of static and dynamic tools with an integrated analysis environment for parallel Fortran, C++, C, Java, and Python applications. TAU is capable of instrumenting source code or analyzing dynamically a program in run time.

The model that TAU uses to profile parallel, multi-threaded programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for C++ functions, methods, basic blocks, and statement execution at these levels.

From the profile data collected, TAU's profile analysis procedures can generate a wealth of

performance information for the user. It can show the exclusive and inclusive time spent in each function with nanosecond resolution. For templated entities, it shows the breakup of time spent for each instantiation. Other data includes how many times each function was called, how many profiled functions did each function invoke, and what the mean inclusive time per call was. Time information can also be displayed relative to nodes, contexts, and threads. Instead of time, hardware performance data can be shown. Also, user-level profiling is possible.

Paraprof, TAU's profile visualization tool, provides graphical displays of all the performance analysis results, in aggregate and per node/context/thread form. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface.

### *2.1 Instrumentation*

To perform measurements with TAU, the user's application program needs to be instrumented, i.e., at specific points of interest (called events) TAU measurement calls have to be activated. As an example, common events are, amongst others, entering and leaving of functions as well as sending and receiving of MPI messages.

TAU handles this automatically by default. In order to enable the instrumentation of function calls, the user only needs to replace the compiler and linker commands with TAUS's wrappers.

Also, TAU supoport dynamic instrumentation. This means TAU is able to capture trace events without recompiling the application. Using the `tauex` tool, taking measures of an applications is as easy as execute `$ tauex application`.

### *2.2 Measurements*

TAU provides the ability to measure how many time a program uses to execute monitored calls. This will show us how this time grows as the cluster grow, and if it grows more in a virtualized environment than on a physical one.

The following table contains the execution times of the function calls instrumented for analysis, in this case only MPI calls were instrumented.

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 100.0 | 514429 | 1850781 | 4 | 2.86801E+06 | 462695243 | .TAU application |
| 68.5 | 1267600 | 1267600 | 772080 | 0 | 1642 | MPI_Wait() |
| 2.0 | 36476 | 36476 | 43528 | 0 | 838 | MPI_Bcast() |
| 0.9 | 15848 | 15848 | 386040 | 0 | 41 | MPI_Isend() |
| 0.4 | 7123 | 7123 | 26460 | 0 | 269 | MPI_Gatherv() |
| 0.2 | 3795 | 3795 | 4 | 0 | 948632 | MPI_Init() |
| 0.2 | 2883 | 2883 | 26744 | 0 | 108 | MPI_Gather() |
| 0.1 | 1272 | 1272 | 386040 | 0 | 3 | MPI_Irecv() |
| 0.0 | 399 | 399 | 400004 | 0 | 1 | MPI_Cart_shift() [T] |
| 0.0 | 361 | 361 | 400004 | 0 | 1 | MPI_Comm_rank() [T] |
| 0.0 | 333 | 333 | 400004 | 0 | 1 | MPI_Cart_coords() [T] |
| 0.0 | 216 | 216 | 284 | 0 | 760 | MPI_Scatterv() |
| 0.0 | 33.4 | 33.4 | 26764 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 4.47 | 4.47 | 8 | 0 | 559 | MPI_Allreduce() |
| 0.0 | 3.15 | 3.15 | 16 | 0 | 197 | MPI_Allgather() |
| 0.0 | 2.87 | 2.87 | 4 | 0 | 717 | MPI_Finalize() |
| 0.0 | 2.03 | 2.03 | 8 | 0 | 254 | MPI_Comm_split() |
| 0.0 | 0.707 | 0.707 | 4 | 0 | 177 | MPI_Comm_dup() |
| 0.0 | 0.013 | 0.013 | 16 | 0 | 1 | MPI_Type_size() |

**TABLE 4.** Call spent time sample

*3. Analysis methodology*

Our tests aim to the MPI collective operations mentioned in chapter three. We want to measure how many calls the program did, how much time was spent executing them and how does scale when the number of nodes of a cluster grow. And we will compare the results of those measures in physical and virtual environments of FutureGrid.

We executed the same simulation test using WRF in different clusters to collect performance and MPI usage traces. For the purpose of this experiment we choose `em_b_wave`[10] simulation.

This is the simplest simulation available in the WRF suite, but it uses MPI intensively to coordinate the parallel execution of the simulation.

We have run the simulation test with the following clusters set up:

| Type | # nodes | # Cores | # Processes |
|---|---|---|---|
| A | 2 | 1 | 2 |
| B | 4 | 1 | 4 |
| C | 8 | 1 | 8 |
| D | 16 | 1 | 16 |

**TABLE 5.** Cluster types

Each configuration (through A to D) have been executed under physical (P) and virtual (V) clusters.

In the analysis we will see how MPI scales from P/V-A to P/V-D and if it behaves different in

P-[A-D] in comparison with V-[A-D].

*4. Executions in physical clusters*

In Futuregrid, jobs can be submitted using the TORQUE tool `$ qsub job_script` from a login server with HPC/TORQUE capabilities.

We have elaborated the following job script to execute the test P-A:

```
#!/bin/bash

#PBS -N P-A
#PBS -o P-A_$PBS_JOBID.out
#PBS -e P-A_$PBS_JOBID.err
#PBS -q batch
#PBS -l nodes=2:ppn=1
#PBS -l walltime=01:00:00

# MPI & TAU environment set up
export PATH=$PATH:$HOME/hpc/bin/mpich2/bin:
               $HOME/hpc/src/tau-2.21.2/x86_64/bin

export LD_LIBRARY_PATH=$HOME/hpc/bin/gmp/lib/:
               $HOME/hpc/bin/mpfr/lib:$HOME/hpc/bin/mpc/lib:
               $HOME/hpc/bin/gcc/lib:$HOME/hpc/bin/gcc/lib64:
               $HOME/hpc/bin/gcc/libexec:$HOME/hpc/bin/mpich2/lib:
               $HOME/hpc/bin/papi/lib:
               $HOME/hpc/bin/netcdf/lib:
               $HOME/hpc/src/tau-2.21.2/x86_64/lib/

export WAVE=$HOME/hpc/src/WRFV3/test/em_b_wave

export EXP=$HOME/hpc/exp/P-A/
mkdir -p $EXP

export TAU_COMM_MATRIX=1
export PROFILEDIR=$EXP
export TRACEDIR=$EXP
export EVENTDIR=$EXP

cd $EXP
for i in ‘ls -1 $WAVE‘; do
        ln -s $WAVE/$i $i
done

mpirun -np 2 tauex ./wrf.exe
```

**Figure 19.** TORQUE job submission script

As we can see in the `PBS` options, we have selected to use one core per machine, and we need to indicate `mpirun` to use one process per available core, which means one MPI process on each host. Besides the last line, the script just define environment variables to execute the `mpirun`, `tauex` and `wrf.exe`.

The file system is shared across all nodes, this is convenient in order to provide an homogeneous file system view to all nodes where all the binaries and data are reachable in the same location. But with this configuration, if two instances of the same application are run on the same location, the output of each execution can potentially conflict with others. And the same could happen to the TAU utilities.

In order to be able to queue various tests at the same time, we created a `$EXP` folder for each simulation. We link all the WRF binaries in that folder to execute the simulation from it. This way, all the executions generate the TAU trace files in different folders, making possible the queue and execution of various experiments at the same time.

*5. Executions in virtual clusters*

Following the work done for the job submission script for TORQUE, we have elaborated the following script to launch the MPI processes across the VMs deployed with Nimbus:

```
#!/bin/bash

# MPI & TAU environment set up
export PATH=$PATH:$HOME/hpc/bin/mpich2/bin:
               $HOME/hpc/src/tau-2.21.2/x86_64/bin

export LD_LIBRARY_PATH=$HOME/hpc/bin/gmp/lib/:
               $HOME/hpc/bin/mpfr/lib:$HOME/hpc/bin/mpc/lib:
               $HOME/hpc/bin/gcc/lib:$HOME/hpc/bin/gcc/lib64:
               $HOME/hpc/bin/gcc/libexec:$HOME/hpc/bin/mpich2/lib:
               $HOME/hpc/bin/papi/lib:
               $HOME/hpc/bin/netcdf/lib:
               $HOME/hpc/src/tau-2.21.2/x86_64/lib/

export WAVE=$HOME/hpc/src/WRFV3/test/em_b_wave

export EXP=$HOME/hpc/exp/P-A/
mkdir -p $EXP

export TAU_COMM_MATRIX=1
export PROFILEDIR=$EXP
export TRACEDIR=$EXP
export EVENTDIR=$EXP

cd $EXP

mpirun -np 2 tauex ./wrf.exe
```

**Figure 20.** MPI job execution script

In contrast with the HPC environment, the cloud environment does not have a shared file system by default, so we generated a virtual machine image containing all the folders needed for the defined tests.

# 5
# Environment set up

*1. Software components*

The latest version of the Weather Research Framework model is 3.4. Unfortunately, this versions do not support the compilers installed in the FutureGrid physical clusters, which are based on RedHat 5.8 and have the GCC suite from 2008. To overcome this handicap we will install our own compiler suite based on GCC 4.6.3 from 2012.

Also we will use the latests versions of the profiling tools TAU as of March 2012, which supports the GCC compiler suite.

Summarizing, we will use the following software packages:

- GNU compiler suite version 4.6.3 and gfortran dependencies
- mpich2 library version 1.4.1p1 stable
- TAU utilities version 2.21
- NetCDF version 4.1.3
- WRF version 3.4

*2. General development environment*

We prepared the environment to be self contained, and stored in a file systems structure (`$HOME/hpc`) composed of various directories.

Before starting to compile and install programs, we needed to set up different environment variables to avoid typing errors and to add clarity to the commands to be executed.

- This sets up the base directory and will contain sources and binaries of all programs. Copying this directory to another machine with the same architecture and operating should provide the new machine with all the environment needed to compile and run tests. This allowed us to be sure the very same software is running in virtual and physical clusters.
- This folder contains all the binaries, programs and libraries, organized in subdirectories. For example should contain MPICH2 executables, while should contain MPICH2 libraries.
- This folder contains all the sources of the software used for the experiment.
- This folder contains all the job launch scripts to execute the experiments.
- This folder contains all the data gathered of each experiment.

More environment variables will be added as the software is installed because programs expect different environments for compilation and execution.

After the initial set up is complete, we create the folders using the following commands:

```
$ cd $HOME
$ mkdir -p $SRC # programs will be compiled here
$ mkdir -p $BIN # programs will be installed here
$ mkdir -o $JOBS # job scripts
$ mkdir -p $EXP # data from experiments
```

**Figure 21.** Make file system folder structure

*3. Installing the software*

The process of compiling and installing is the same for all the software for this experiment:

- Download the source code from the vendor's public repository
- Uncompress it in the `$SRC` folder
- Set up the environment for the compilation, with the options we need. Most notably the option to install binaries under `$BIN` and the compilers and libraries to be included as dependencies.
- Compile the software using the vendor's provided build script
- Install the software using the vendor's provided install script

*3.1 GCC and related utilities*

The GNU Compiler Collection is a suite of compilers built and distributed by the GNU project[1] and includes compilers for a lot of languages (such as C, C++, Fortran, etc.) and their runtime tools and libraries. This compiler suite is used not only in the free software community but by a lot of commercial vendors such as IBM and Apple.

The fact that is a complete, modern and free compiler suite makes it the perfect choice for experimentation and development.

To build the software for these experiments we need the gfortran compiler from the GCC suite, which need the following libraries to e build:

- The GNU Multiple Precision Arithmetic Library (gmp-5.0.5): GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functi ons, and the functions have a regular interface.
- TThe GNU Multiple-Precision Floating-Point library (mpfr-3.1): the main goal of MPFR is to provide a library for multiple-precision floating-point computation which is both efficient and has a well-defined semantics.
- Mpc is a C library for the arithmetic of complex numbers with arbitrarily high precision and correct rounding of the result. It is built upon and follows the same principles as Mpfr (mpc-0.9)

All of them are installed following the `configure`, `make`, `make install` procedure, with no other configure options besides the `--prefix` to specify the install directory.

1. About GCC: `http://gcc.gnu.org`

On the other hand, the gcc compiler was configured with the following options:

```
--enable-languages=c,c++,fortran
--with-gmp=$BIN/gmp
--with-mpfr=$BIN/mpfr
--with-mpc=$BIN/mpc
--with-cpu=generic
--enable-shared
--enable-threads=posix
--enable-bfd-assembler
```

**Figure 22.** GCC configure command

The compilation and installation process (`make`, `make install`) was executed without errors.

After the installation, we needed to modify the `PATH` variable to find first out new installed compiler suite, instead of the system's default.

```
$ export PATH=$BIN/gcc/bin:$PATH
```

**Figure 23.** Set new system path

*3.2 MPICH2*

There are multiple implementations of the MPI standard, and one of the most used implementations, used as a reference by others, is the MPICH2 from the Argonne National Laboratory[2] MPICH2 is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2). The goals of MPICH2 are: to provide an MPI implementation that efficiently supports different computation and communication platforms to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations.

MPICH2 is distributed as source with an open-source, freely available license and it has been tested on several platforms, which makes it a very good candidate for these experiments.

The following commands implement the install process for the MPICH2 library and tools:

```
$ cd $SRC
$ wget "http://www.mcs.anl.gov/research/projects/mpich2/downloads/
       tarballs/1.4.1p1/mpich2-1.4.1p1.tar.gz"
$ tar zxf mpich2-1.4.1p1.tar.gz
$ cd  mpich2-1.4.1p1
$ export MPICH2LIB_CFLAGS=-fPIC
$ export MPICH2LIB_CPPFLAGS=-fPIC
$ ./configure --prefix=$BIN/mpich2 --enable-shared
$ make
$ make install
```

**Figure 24.** MPICH2 install commands

After installing MPICH2, we will set up some environment variables, needed to build MPI programs:

```
$ export MPI=$BIN/mpich2
$ export MPICC=$BIN/mpich2/bin/mpicc
$ export MPICXX=$BIN/mpich2/bin/mpicxx
```

---

2. About MPICH2: `http://www.mcs.anl.gov/research/projects/mpich2/about/index.php?s=about`

```
$ export MPIF77=$BIN/mpich2/bin/mpif77
$ export MPIF90=$BIN/mpich2/bin/mpif90
$ export PATH=$PATH:$BIN/mpich2/bin
```

**Figure 25.** MPICH2 environment variables

### *3.3 Tunning and Analysis Utilities (TAU)*

The Tunning and Analysis Utilities framework is a set of tools being developed for the DOE Office of Science, ASC initiatives at LLNL, the ZeptoOS project at ANL, and the Los Alamos National Laboratory[3]. TAU provides a suite of static and dynamic tools that provide graphical user interaction and inter-operation to form an integrated analysis environment for parallel Fortran, C++, C, Java, and Python applications. It has a a robust and open source performance profiling facility able to profile though various instrumentation methods, such as the library interposition, which makes it able to measure MPI calls without recompiling the application or the MPI library.

The following commands build and install the TAU binaries and libraries:

```
$ cd $SRC
$ wget "http://tau.uoregon.edu/tau.tgz"
$ tar zxf tau.tgz
$ cd tau-2.21.2
$ ./configure -mpi -MPITRACE -pthread -papi=$BIN/papi
$ make
$ make install
```

**Figure 26.** TAU install commands

#### *3.3.1 NetCDF*

NetCDF[4] is a library used by WRF to read and write data. This library is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for various languages. The netCDF libraries support a machine-independent format for representing scientific data which is widely used by applications such as WRF.

NetCDF is a requisite of WRF, but we do not need support for version 4 features nor the documentation, so we disable them on the configuration script:

```
$ cd $SRC
$ wget "http://www.unidata.ucar.edu/downloads/netcdf/ftp/
       netcdf-4.1.3.tar.gz"
$ tar zxf netcdf-4.1.3.tar.gz
$ cd netcdf-4.1.3
$ ./configure --prefix=$BIN/netcdf --disable-netcdf-4  --disable-doxygen
$ make
$ make install
```

**Figure 27.** NetCDF install commands

After installing netCDF, we will set up the `NETCDF` variable, needed by WRF compile script:

```
$ export NETCDF=$BIN/netcdf
```

**Figure 28.** Export NetCDF variable

---

3. About TAU: `http://www.cs.uoregon.edu/Research/tau/about.php`

4. About NetCDF: `http://www.unidata.ucar.edu/software/netcdf/docs/faq.html#whatisit`

### *3.3.2 WRF*

The Weather Research and Forecasting (WRF) Model[5] is a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. It is composed of several simulation programs which are engineered to allow computational parallelism and system extensibility.

The configure script of the WRF is not based on autogen tools, so we will select the option to compile using GCC / gfortran and dmpar (for MPI). Also, the build script is not based on the autogen tools, so instead of calling `make`, we need to execute the `compile` script.

```
$ cd $SRC
$ wget "http://www.mmm.ucar.edu/wrf/src/WRFV3.4.TAR.gz"
$ tar zxf WRFV3.4.TAR.gz
$ cd WRFV3
$ ./configure
$ ./compile em_wave &> compile.log
```

**Figure 29.** WRF install commands

*4. HPC environment*

The HPC environment is already set up in FutureGrid, so the only task pending is the set up of the executing environment, including the script to launch the job for each experiment.

In the `$JOBS` folder of the FutureGrid login server, we created four scripts to launch each of the tests:

- `P-A.sh` to execute the simulation in two nodes, using one core on each one.
- `P-B.sh` to execute the simulation in four nodes, using one core on each one.
- `P-C.sh` to execute the simulation in eight nodes, using one core on each one.
- `P-D.sh` to execute the simulation in sixteen nodes, using one core on each one.

All the scripts use the same template, modifying the number of reserved nodes appropriately for each test. The template was shown in Chapter 4 as the *TORQUE job submission script*

*5. VM environment*

The virtual machine environment is composed by the VM image, containing the operating system (OS) and the experiment environment already described, and the scrips needed to make them work together through SSH (which works as a back-end to the Hydra process manager.

### *5.1 Virtual Machine image*

In order to build a binary-compatible virtual environment, we created a Virtual Machine image based on the same version of the operating system used in the HPC environment (RedHat/CentOS 5.8).

For its creation we used as a base an image of CentOs 5.7. The first step done was the OS update up to version 5.8 and the install off all the packages needed for the programs we compiled for the HPC environment. The second step was to copy the compiled binaries and scripts from the HPC

5. About WRF: `http://www.wrf-model.org/index.php`

environment to the VM image.

The file system used in the VM was configured to have the same layout of the HPC system using folders and symlinks, so all the programs and scripts were able to run without modifications.

### *5.2 Trust & collect*

In the time of the execution of this experiments, the Context Server was having problems. In order to avoid such problems we configured the machines manually to trust each other using a SSH certificate.

The script `trust.sh` was used to install a SSH identity on each node to trust each other. Also, SSH client was configured to trust any remote SSH host key, so there is no need to accept the key interactively, which was needed by the HYDRA MPI process manager.

```
#!/bin/bash

HOSTFILE=fl
HEADNODE=$1

ssh root@$HEADNODE 'ssh-keygen -t rsa -N "" -f /root/.ssh/id'
scp root@$HEADNODE:/root/.ssh/id* .
for h in `cat $HOSTFILE`; do
        cat id.pub | ssh root@$h "cat >> ~/.ssh/authorized_keys"
done
```

**Figure 30.** trust.sh script

Also, the machines did not share any network file system, so we needed t gather all the results manually from each machine. To automate the process we created the `collect.sh` script.

```
#!/bin/bash

HOSTFILE=fl
HEADNODE=$1

for h in `cat $HOSTFILE`; do
        scp root@$h:/root/hpc/exp/$1/tautrace* $HOME/Dropbox/pfc/exp/$1/
        scp root@$h:/root/hpc/exp/$1/profile* $HOME/Dropbox/pfc/exp/$1/
        scp root@$h:/root/hpc/exp/$1/events* $HOME/Dropbox/pfc/exp/$1/
done
```

**Figure 31.** collect.sh script

## *6. Instrumentation*

To take measures of MPI functions for each execution using the TAU utilities there is no need for instrumentation in compile time. We used `tauex` utility to measure the MPI usage and behaviour automatically without recompiling or dynamically modifying binaries.

# 6
# Results

*1. Introduction*

In this chapter we present the measures taken on each experiment. For each case we first present a summary table indicating the total time spent per MPI call, and then, a graphical representation of the time spent per node and per call, giving an overview of the symmetry of some calls, and the asymmetry of the collective communications, on which some processes send, and some gather.

Also, we included the TAU profile of each environment, showing the different configurations of physical and virtual environments. The virtual machines had one processing core, while the physical ones had four cores. In our tests, only one core of those four were used, to make a fair comparison between environments.

*2. Virtual environment tests*

The TAU profile for this experiment series was:

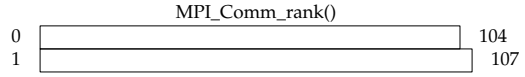
```
CPU Cores                  : 1
CPU Type                   : Intel(R) Xeon(R) CPU L5420 @ 2.50GHz
CPU Vendor                 : GenuineIntel
CWD                        : /root/hpc/exp/V-A
Cache Size                 : 6144 KB
Command Line               : ./wrf.exe
Executable                 : /root/hpc/src/WRFV3/main/wrf.exe
File Type Index            : 1
File Type Name             : Tau profiles
Local Time                 : 2012-06-07T03:39:54-04:00
Memory Size                : 2097532 kB
OS Machine                 : x86_64
OS Name                    : Linux
OS Release                 : 2.6.18-xen
OS Version                 : #2 SMP Wed Apr 16 12:47:36 CDT 2008
TAU Architecture           : x86_64
TAU Config                 : -mpi -MPITRACE -pthread -papi=/N/u/gdiaz/hpc/bin/papi
TAU Makefile               : /N/u/gdiaz/hpc/src/tau-2.21.2/x86_64/
                                                    lib/Makefile.tau-papi-mpi-pthread-mpitrace
TAU MetaData Merge Time    : 8.8E-05 seconds
TAU Version                : 2.21.2
TAU_CALLPATH               : off
TAU_CALLPATH_DEPTH         : 2
TAU_CALLSITE_LIMIT         : 1
TAU_COMM_MATRIX            : on
TAU_COMPENSATE             : off
TAU_CUPTI_API              : runtime
TAU_EBS_KEEP_UNRESOLVED_ADDR: off
TAU_IBM_BG_HWP_COUNTERS    : off
TAU_PROFILE                : on
```

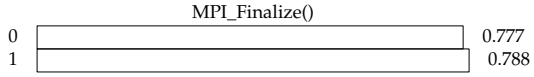
```
TAU_PROFILE_FORMAT         : profile
TAU_SAMPLING               : off
TAU_SIGNALS_GDB            : off
TAU_SUMMARY                : off
TAU_THROTTLE               : on
TAU_THROTTLE_NUMCALLS      : 100000
TAU_THROTTLE_PERCALL       : 10
TAU_TRACE                  : off
TAU_TRACK_HEADROOM         : off
TAU_TRACK_HEAP             : off
TAU_TRACK_IO_PARAMS        : off
TAU_TRACK_MEMORY_LEAKS     : off
TAU_TRACK_MESSAGE          : on
TAU_TRACK_SIGNALS          : off
UTC Time                   : 2012-06-07T07:39:54Z
username                   : root
```

*2.1 Two virtual machines (V-A)*

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 100.0 | 691989 | 986629 | 2 | 708290 | 493314476 | .TAU |
| 27.6 | 272240 | 272240 | 126760 | 0 | 2148 | MPI_Wait() |
| 1.0 | 9773 | 9773 | 63380 | 0 | 154 | MPI_Isend() |
| 0.7 | 7039 | 7039 | 21764 | 0 | 323 | MPI_Bcast() |
| 0.2 | 2127 | 2127 | 13230 | 0 | 161 | MPI_Gatherv() |
| 0.1 | 1340 | 1340 | 13372 | 0 | 100 | MPI_Gather() |
| 0.1 | 696 | 696 | 2 | 0 | 348003 | MPI_Init() |
| 0.1 | 638 | 638 | 63380 | 0 | 10 | MPI_Irecv() |
| 0.0 | 381 | 381 | 200002 | 0 | 2 | MPI_Cart_shift() |
| 0.0 | 211 | 211 | 126584 | 0 | 2 | MPI_Comm_rank() |
| 0.0 | 115 | 115 | 66264 | 0 | 2 | MPI_Cart_coords() |
| 0.0 | 59.0 | 59.0 | 142 | 0 | 415 | MPI_Scatterv() |
| 0.0 | 15.8 | 15.8 | 13382 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 1.56 | 1.56 | 2 | 0 | 782 | MPI_Finalize() |
| 0.0 | 0.932 | 0.932 | 8 | 0 | 116 | MPI_Allgather() |
| 0.0 | 0.622 | 0.622 | 4 | 0 | 156 | MPI_Comm_split() |
| 0.0 | 0.495 | 0.495 | 4 | 0 | 124 | MPI_Allreduce() |
| 0.0 | 0.289 | 0.289 | 2 | 0 | 144 | MPI_Comm_dup() |
| 0.0 | 0.011 | 0.011 | 8 | 0 | 1 | MPI_Type_size() |

**TABLE 6.** V-A Totals in miliseconds



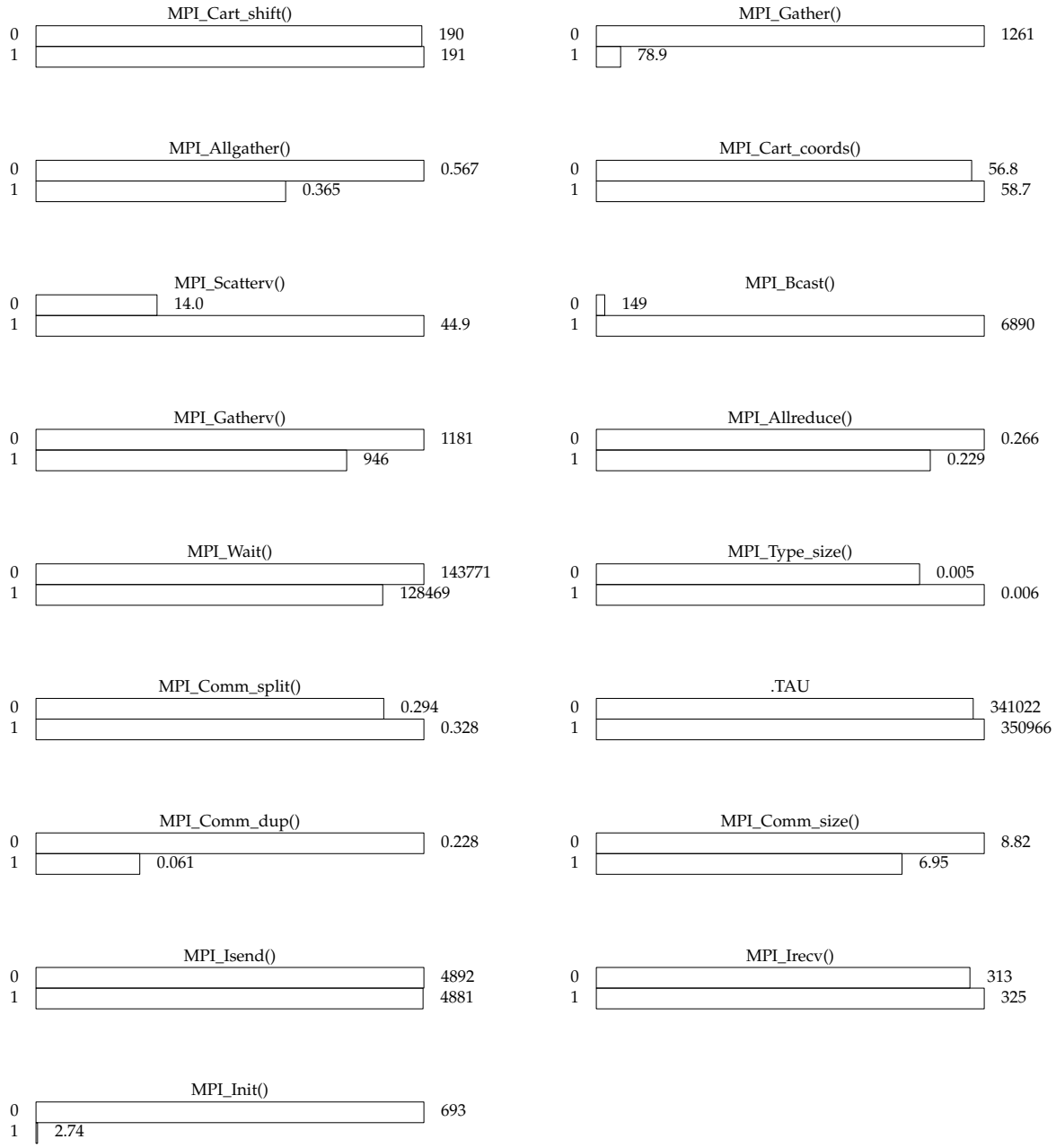
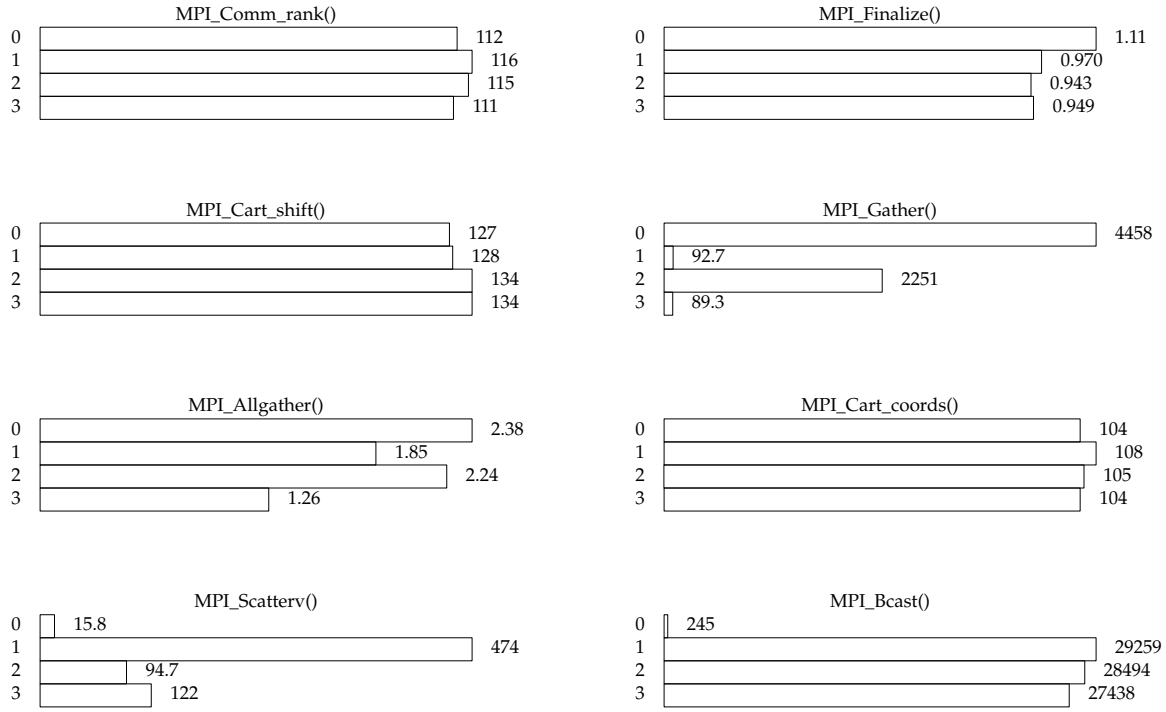



**Figure 32.** V-A Time distribution graph

*2.2 Four virtual machines (V-B)*

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 64.1 | 1917016 | 1917016 | 772080 | 0 | 2483 | MPI_Wait() |
| 100.0 | 918128 | 2991922 | 4 | 2.86801E+06 | 747980615 | .TAU |
| 2.9 | 85436 | 85436 | 43528 | 0 | 1963 | MPI_Bcast() |
| 1.0 | 30538 | 30538 | 386040 | 0 | 79 | MPI_Isend() |
| 1.0 | 28758 | 28758 | 26460 | 0 | 1087 | MPI_Gatherv() |
| 0.2 | 6891 | 6891 | 26744 | 0 | 258 | MPI_Gather() |
| 0.1 | 2206 | 2206 | 386040 | 0 | 6 | MPI_Irecv() |
| 0.0 | 779 | 779 | 4 | 0 | 194862 | MPI_Init() |
| 0.0 | 707 | 707 | 284 | 0 | 2490 | MPI_Scatterv() |
| 0.0 | 523 | 523 | 400004 | 0 | 1 | MPI_Cart_shift() |
| 0.0 | 454 | 454 | 400004 | 0 | 1 | MPI_Comm_rank() |
| 0.0 | 421 | 421 | 400004 | 0 | 1 | MPI_Cart_coords() |
| 0.0 | 44.8 | 44.8 | 26764 | 0 | 2 | MPI_Comm_size() |
| 0.0 | 7.73 | 7.73 | 16 | 0 | 483 | MPI_Allgather() |
| 0.0 | 4.19 | 4.19 | 8 | 0 | 524 | MPI_Comm_split() |
| 0.0 | 3.97 | 3.97 | 4 | 0 | 993 | MPI_Finalize() |
| 0.0 | 2.59 | 2.59 | 8 | 0 | 324 | MPI_Allreduce() |
| 0.0 | 0.994 | 0.994 | 4 | 0 | 248 | MPI_Comm_dup() |
| 0.0 | 0.021 | 0.021 | 16 | 0 | 1 | MPI_Type_size() |

**TABLE 7.** V-B Totals in miliseconds



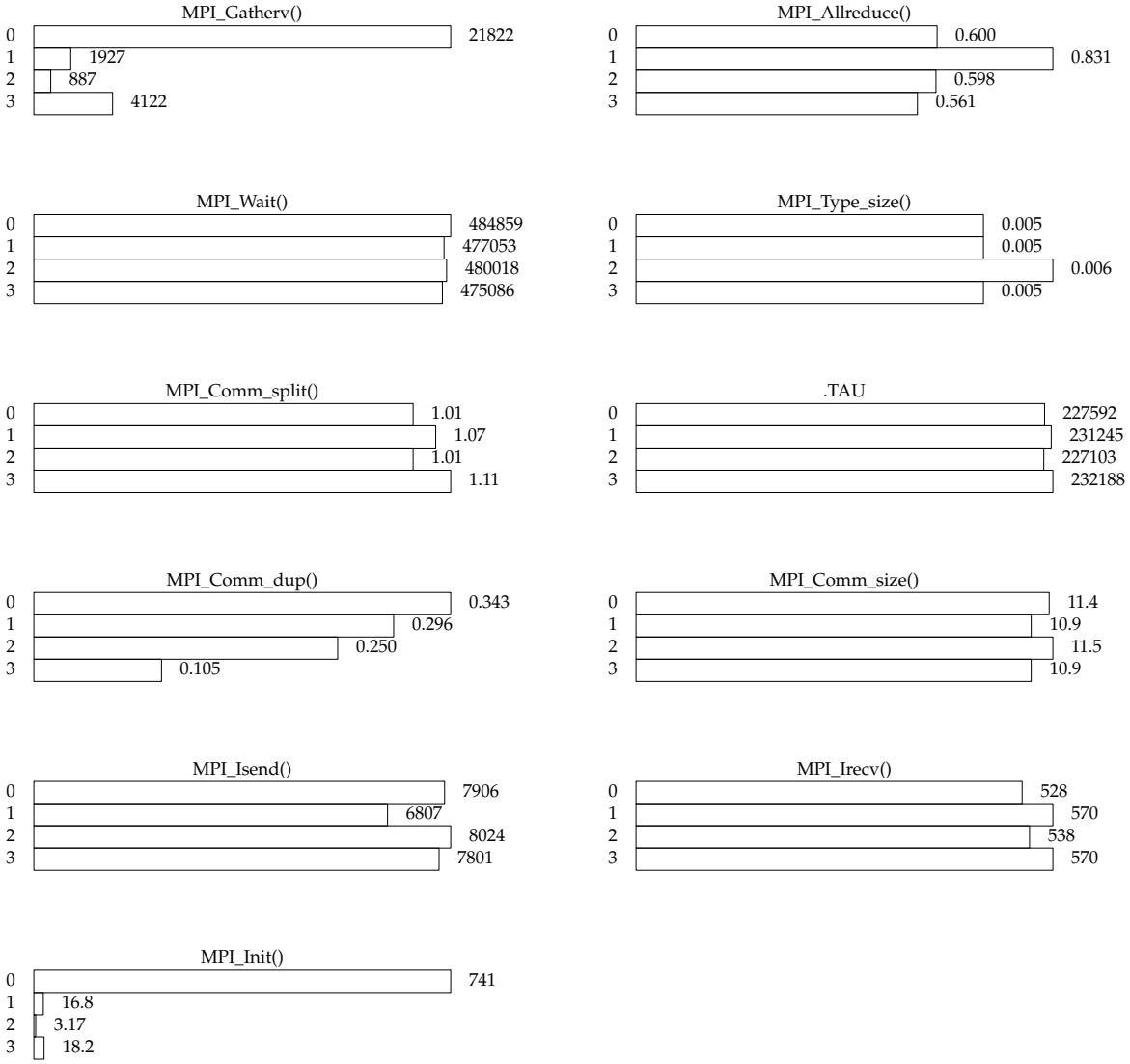
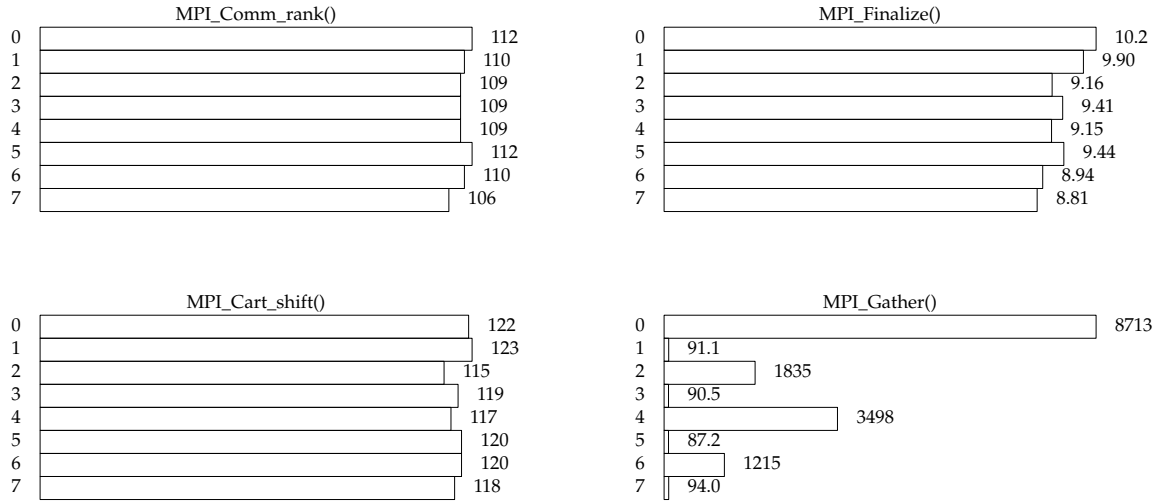


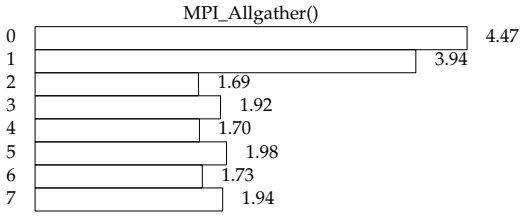

**Figure 33.** V-B Time distribution graph

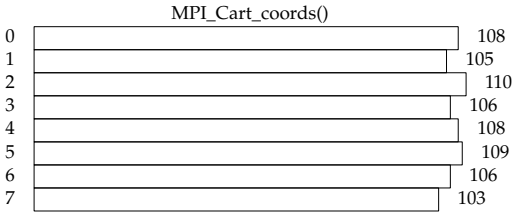
### *2.3 Eight virtual machines (V-C)*

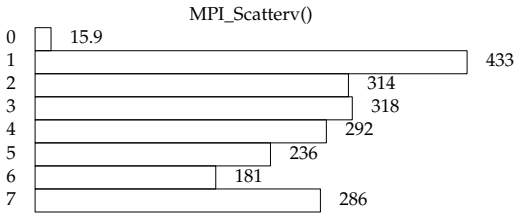
| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 84.1 | 5901471 | 5901471 | 1.79768E+06 | 0 | 3283 | MPI_Wait() |
| 100.0 | 851562 | 7016448 | 8 | 6.13027E+06 | 877056021 | .TAU |
| 2.4 | 171057 | 171057 | 87056 | 0 | 1965 | MPI_Bcast() |
| 0.8 | 56391 | 56391 | 898840 | 0 | 63 | MPI_Isend() |
| 0.2 | 15624 | 15624 | 53488 | 0 | 292 | MPI_Gather() |
| 0.2 | 10786 | 10786 | 52920 | 0 | 204 | MPI_Gatherv() |
| 0.0 | 2076 | 2076 | 568 | 0 | 3656 | MPI_Scatterv() |
| 0.0 | 1588 | 1588 | 8 | 0 | 198557 | MPI_Init() |
| 0.0 | 1532 | 1532 | 400004 | 0 | 4 | MPI_Irecv() |
| 0.0 | 1456 | 1456 | 386040 | 0 | 4 | MPI_Irecv() |
| 0.0 | 954 | 954 | 800008 | 0 | 1 | MPI_Cart_shift() |
| 0.0 | 878 | 878 | 800008 | 0 | 1 | MPI_Comm_rank() |
| 0.0 | 856 | 856 | 800008 | 0 | 1 | MPI_Cart_coords() |
| 0.0 | 81.8 | 81.8 | 53528 | 0 | 2 | MPI_Comm_size() |
| 0.0 | 75.0 | 75.0 | 8 | 0 | 9371 | MPI_Finalize() |
| 0.0 | 25.6 | 25.6 | 16 | 0 | 1602 | MPI_Allreduce() |
| 0.0 | 19.4 | 19.4 | 32 | 0 | 605 | MPI_Allgather() |
| 0.0 | 13.1 | 13.1 | 16 | 0 | 821 | MPI_Comm_split() |
| 0.0 | 3.68 | 3.68 | 8 | 0 | 460 | MPI_Comm_dup() |
| 0.0 | 0.041 | 0.041 | 32 | 0 | 1 | MPI_Type_size() |

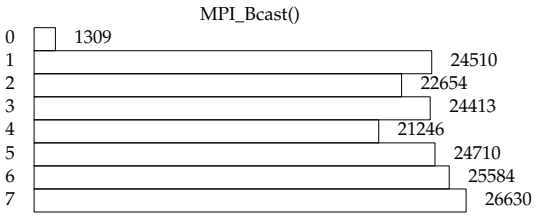
**TABLE 8.** V-C Totals in miliseconds

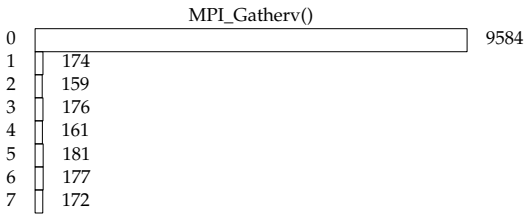


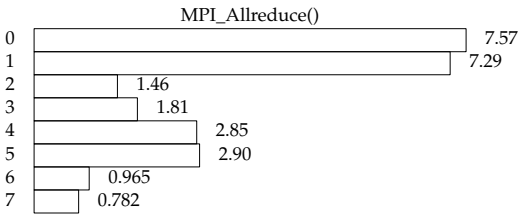

MPI_Allgather()
0 4.47
1 3.94
2 1.69
3 1.92
4 1.70
5 1.98
6 1.73
7 1.94

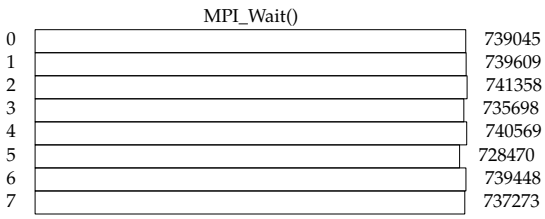

MPI_Cart_coords()
0 108
1 105
2 110
3 106
4 108
5 109
6 106
7 103

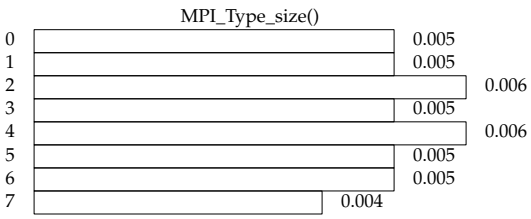

MPI_Scatterv()
0 15.9
1 433
2 314
3 318
4 292
5 236
6 181
7 286

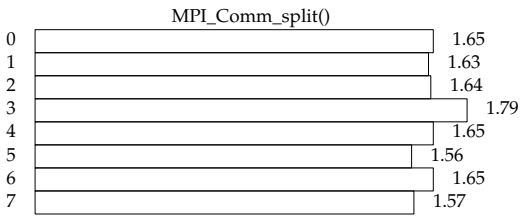

MPI_Bcast()
0 1309
1 24510
2 22654
3 24413
4 21246
5 24710
6 25584
7 26630

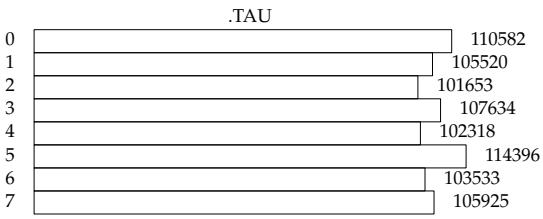

MPI_Gatherv()
0 9584
1 174
2 159
3 176
4 161
5 181
6 177
7 172

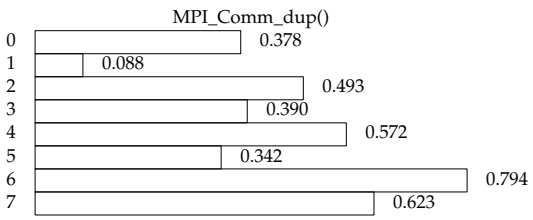

MPI_Allreduce()
0 7.57
1 7.29
2 1.46
3 1.81
4 2.85
5 2.90
6 0.965
7 0.782

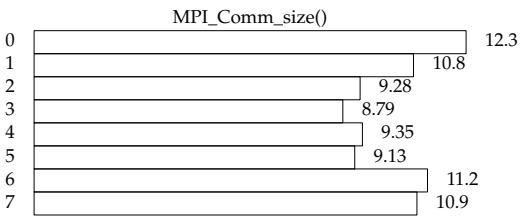

MPI_Wait()
0 739045
1 739609
2 741358
3 735698
4 740569
5 728470
6 739448
7 737273


MPI_Type_size()
0 0.005
1 0.005
2 0.006
3 0.005
4 0.006
5 0.005
6 0.005
7 0.004


MPI_Comm_split()
0 1.65
1 1.63
2 1.64
3 1.79
4 1.65
5 1.56
6 1.65
7 1.57


.TAU
0 110582
1 105520
2 101653
3 107634
4 102318
5 114396
6 103533
7 105925


MPI_Comm_dup()
0 0.378
1 0.088
2 0.493
3 0.390
4 0.572
5 0.342
6 0.794
7 0.623


MPI_Comm_size()
0 12.3
1 10.8
2 9.28
3 8.79
4 9.35
5 9.13
6 11.2
7 10.9

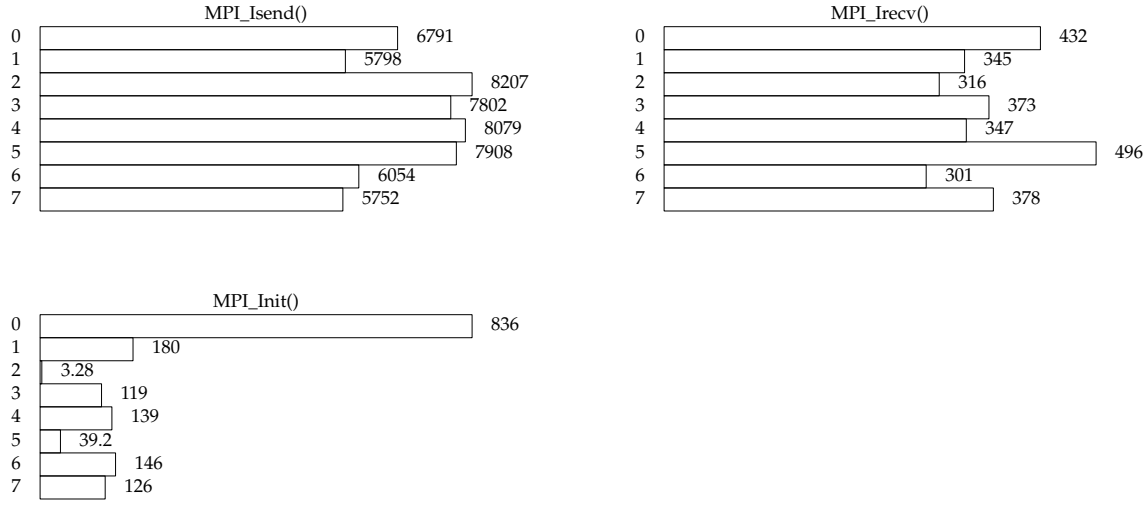
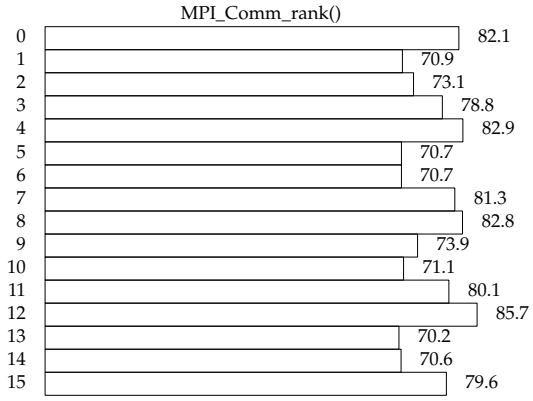


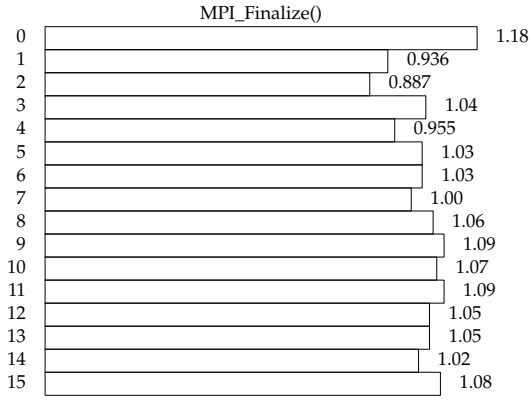

**Figure 34.** V-C Time distribution graph

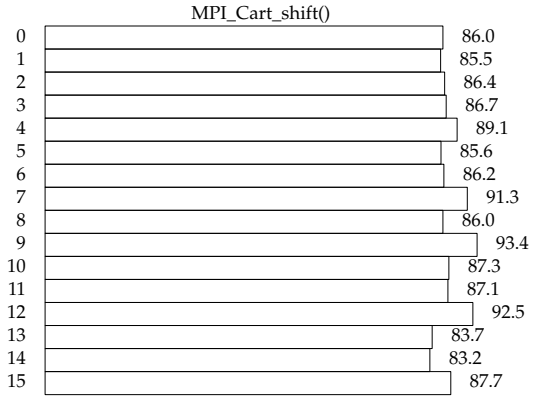
*2.4 Sixteen virtual machines (V-D)*

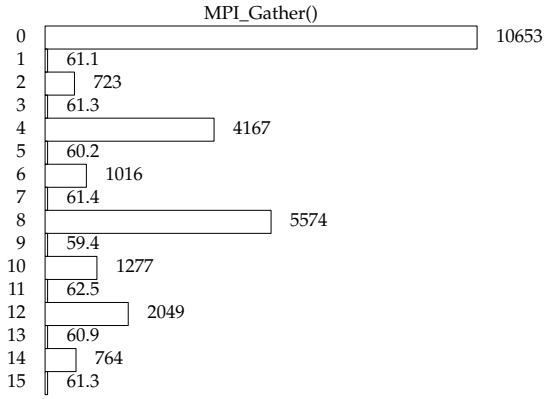
| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 90.4 | 14053278 | 14053278 | 3.57232E+06 | 0 | 3934 | MPI_Wait() |
| 100.0 | 730553 | 15551161 | 16 | 1.22202E+07 | 971947568 | .TAU |
| 3.8 | 595749 | 595749 | 174112 | 0 | 3422 | MPI_Bcast() |
| 0.4 | 54779 | 54779 | 1.78616E+06 | 0 | 31 | MPI_Isend() |
| 0.3 | 44441 | 44441 | 16 | 0 | 2777544 | MPI_Init() |
| 0.2 | 26710 | 26710 | 106976 | 0 | 250 | MPI_Gather() |
| 0.2 | 26439 | 26439 | 105840 | 0 | 250 | MPI_Gatherv() |
| 0.1 | 9881 | 9881 | 1136 | 0 | 8698 | MPI_Scatterv() |
| 0.0 | 2272 | 2272 | 766320 | 0 | 3 | MPI_Irecv() |
| 0.0 | 2247 | 2247 | 800008 | 0 | 3 | MPI_Irecv() |
| 0.0 | 1398 | 1398 | 1.60002E+06 | 0 | 1 | MPI_Cart_shift() |
| 0.0 | 1241 | 1241 | 1.60002E+06 | 0 | 1 | MPI_Cart_coords() |
| 0.0 | 1225 | 1225 | 1.60002E+06 | 0 | 1 | MPI_Comm_rank() |
| 0.0 | 643 | 643 | 32 | 0 | 20085 | MPI_Allreduce() |
| 0.0 | 144 | 144 | 64 | 0 | 2255 | MPI_Allgather() |
| 0.0 | 119 | 119 | 107056 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 20.8 | 20.8 | 32 | 0 | 649 | MPI_Comm_split() |
| 0.0 | 16.6 | 16.6 | 16 | 0 | 1035 | MPI_Finalize() |
| 0.0 | 5.35 | 5.35 | 16 | 0 | 334 | MPI_Comm_dup() |
| 0.0 | 0.093 | 0.093 | 64 | 0 | 1 | MPI_Type_size() |

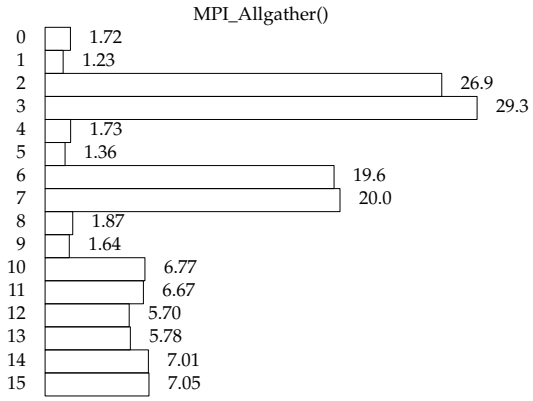
**TABLE 9.** V-D Totals in miliseconds

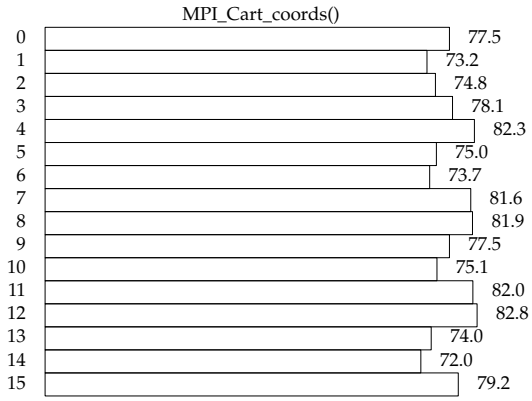

MPI_Comm_rank()
0 82.1
1 70.9
2 73.1
3 78.8
4 82.9
5 70.7
6 70.7
7 81.3
8 82.8
9 73.9
10 71.1
11 80.1
12 85.7
13 70.2
14 70.6
15 79.6

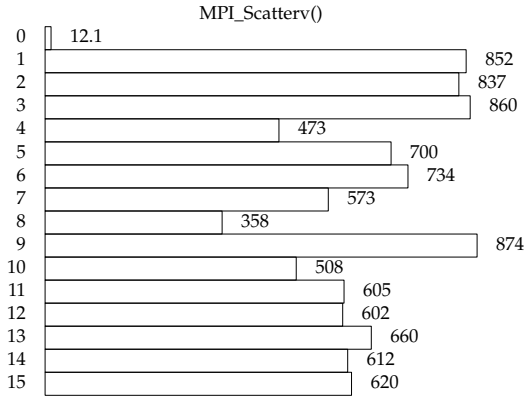

MPI_Finalize()
0 1.18
1 0.936
2 0.887
3 1.04
4 0.955
5 1.03
6 1.03
7 1.00
8 1.06
9 1.09
10 1.07
11 1.09
12 1.05
13 1.05
14 1.02
15 1.08

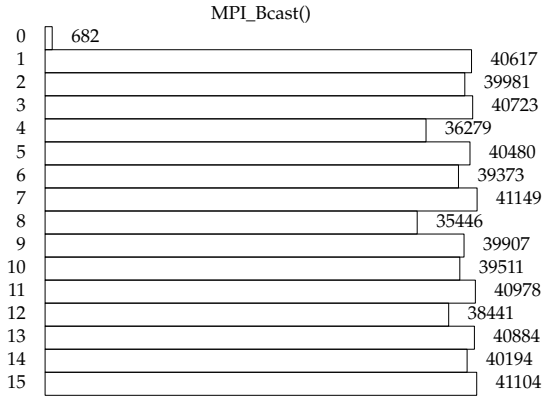

MPI_Cart_shift()
0 86.0
1 85.5
2 86.4
3 86.7
4 89.1
5 85.6
6 86.2
7 91.3
8 86.0
9 93.4
10 87.3
11 87.1
12 92.5
13 83.7
14 83.2
15 87.7

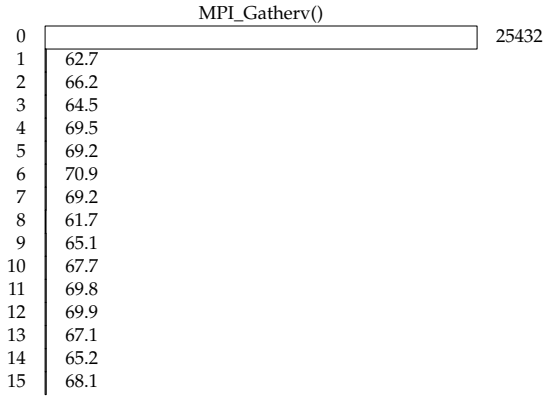

MPI_Gather()
0 10653
1 61.1
2 723
3 61.3
4 4167
5 60.2
6 1016
7 61.4
8 5574
9 59.4
10 1277
11 62.5
12 2049
13 60.9
14 764
15 61.3

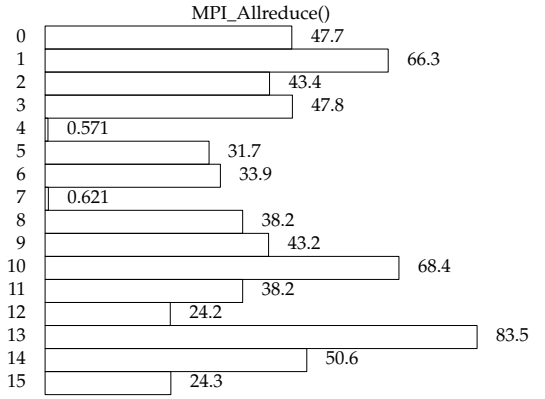

MPI_Allgather()
0 1.72
1 1.23
2 26.9
3 29.3
4 1.73
5 1.36
6 19.6
7 20.0
8 1.87
9 1.64
10 6.77
11 6.67
12 5.70
13 5.78
14 7.01
15 7.05

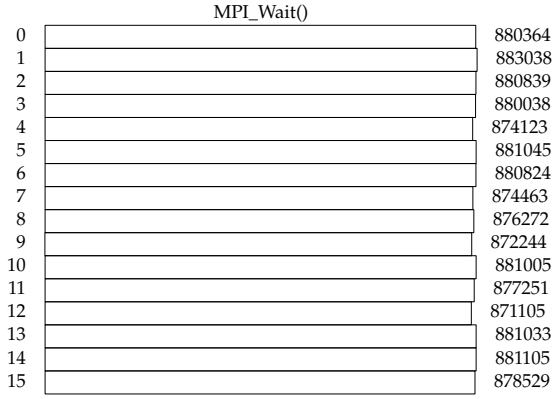

MPI_Cart_coords()
0 77.5
1 73.2
2 74.8
3 78.1
4 82.3
5 75.0
6 73.7
7 81.6
8 81.9
9 77.5
10 75.1
11 82.0
12 82.8
13 74.0
14 72.0
15 79.2

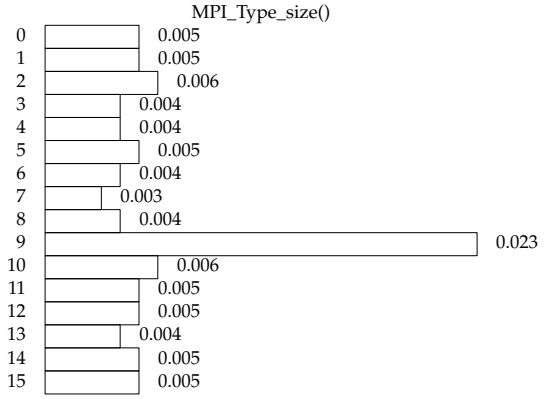

MPI_Scatterv()
0 12.1
1 852
2 837
3 860
4 473
5 700
6 734
7 573
8 358
9 874
10 508
11 605
12 602
13 660
14 612
15 620

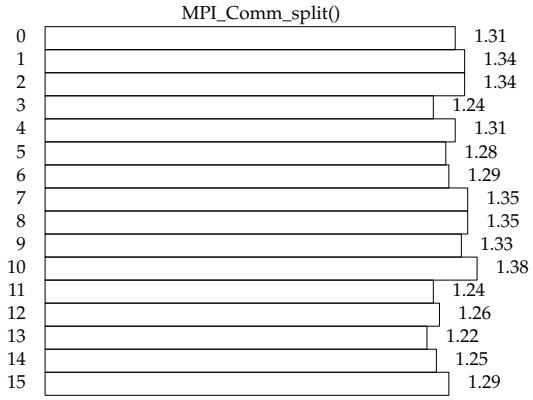

MPI_Bcast()
0 682
1 40617
2 39981
3 40723
4 36279
5 40480
6 39373
7 41149
8 35446
9 39907
10 39511
11 40978
12 38441
13 40884
14 40194
15 41104

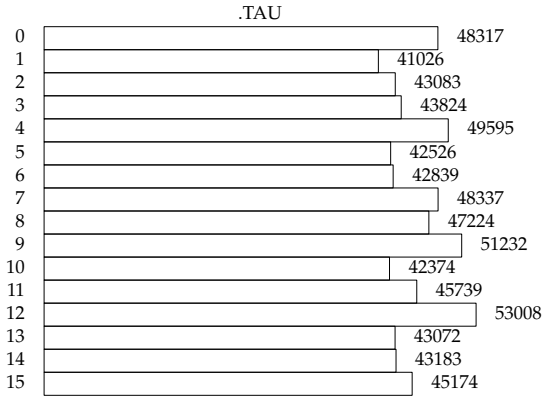

MPI_Gatherv()
0 25432
1 62.7
2 66.2
3 64.5
4 69.5
5 69.2
6 70.9
7 69.2
8 61.7
9 65.1
10 67.7
11 69.8
12 69.9
13 67.1
14 65.2
15 68.1

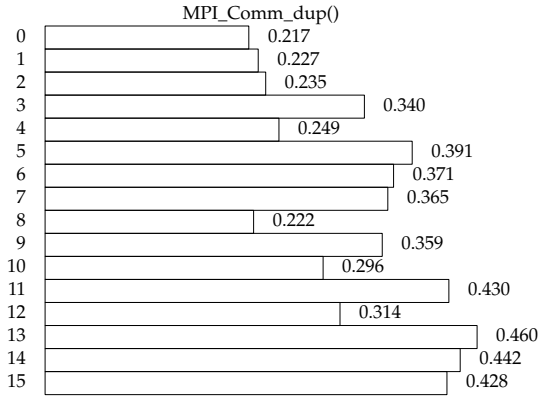

MPI_Allreduce()
0 47.7
1 66.3
2 43.4
3 47.8
4 0.571
5 31.7
6 33.9
7 0.621
8 38.2
9 43.2
10 68.4
11 38.2
12 24.2
13 83.5
14 50.6
15 24.3

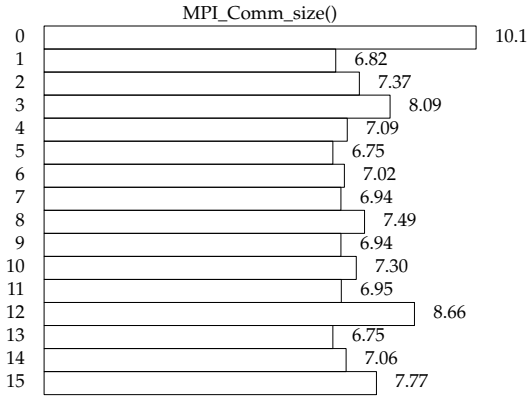

MPI_Wait()
0 880364
1 883038
2 880839
3 880038
4 874123
5 881045
6 880824
7 874463
8 876272
9 872244
10 881005
11 877251
12 871105
13 881033
14 881105
15 878529

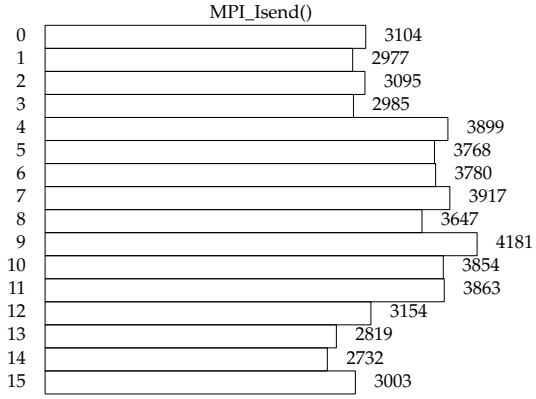

MPI_Type_size()
0 0.005
1 0.005
2 0.006
3 0.004
4 0.004
5 0.005
6 0.004
7 0.003
8 0.004
9 0.023
10 0.006
11 0.005
12 0.005
13 0.004
14 0.005
15 0.005

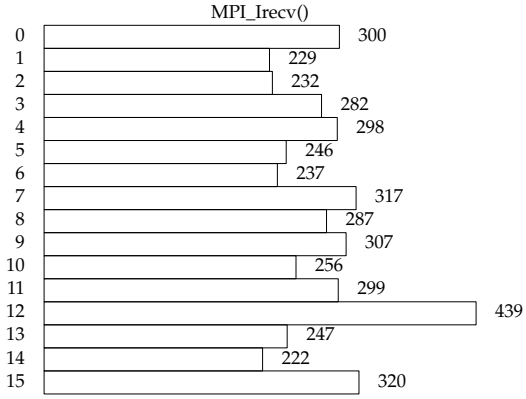

MPI_Comm_split()
0 1.31
1 1.34
2 1.34
3 1.24
4 1.31
5 1.28
6 1.29
7 1.35
8 1.35
9 1.33
10 1.38
11 1.24
12 1.26
13 1.22
14 1.25
15 1.29


.TAU
0 48317
1 41026
2 43083
3 43824
4 49595
5 42526
6 42839
7 48337
8 47224
9 51232
10 42374
11 45739
12 53008
13 43072
14 43183
15 45174


MPI_Comm_dup()
0 0.217
1 0.227
2 0.235
3 0.340
4 0.249
5 0.391
6 0.371
7 0.365
8 0.222
9 0.359
10 0.296
11 0.430
12 0.314
13 0.460
14 0.442
15 0.428


MPI_Comm_size()
0 10.1
1 6.82
2 7.37
3 8.09
4 7.09
5 6.75
6 7.02
7 6.94
8 7.49
9 6.94
10 7.30
11 6.95
12 8.66
13 6.75
14 7.06
15 7.77


MPI_Isend()
0 3104
1 2977
2 3095
3 2985
4 3899
5 3768
6 3780
7 3917
8 3647
9 4181
10 3854
11 3863
12 3154
13 2819
14 2732
15 3003


MPI_Irecv()
0 300
1 229
2 232
3 282
4 298
5 246
6 237
7 317
8 287
9 307
10 256
11 299
12 439
13 247
14 222
15 320

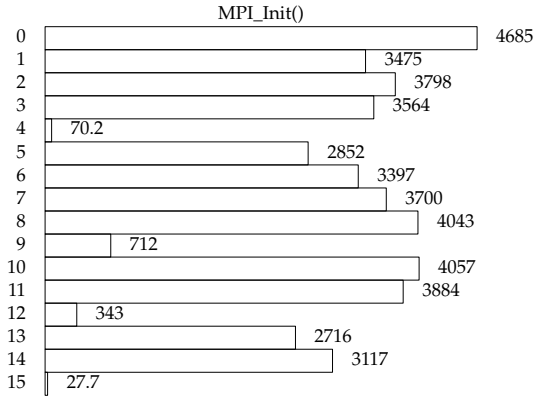



**Figure 35.** V-D Time distribution graph

*3. Physical environment*

The TAU profile for this experiment series was:

```
CPU Cores                    : 4
CPU MHz                      : 2500.000
CPU Type                     : Intel(R) Xeon(R) CPU L5420 @ 2.50GHz
CPU Vendor                   : GenuineIntel
CWD                          : /N/u/gdiaz/hpc/exp/P-A
Cache Size                   : 6144 KB
Command Line                 : ./wrf.exe
Executable                   : /N/u/gdiaz/hpc/src/WRFV3/main/wrf.exe
File Type Index              : 1
File Type Name               : Tau profiles
Memory Size                  : 32958160 kB
OS Machine                   : x86_64
OS Name                      : Linux
OS Release                   : 2.6.18-308.4.1.el5
OS Version                   : #1 SMP Wed Mar 28 01:54:56 EDT 2012
TAU Architecture             : x86_64
TAU Config                   : -mpi -MPITRACE -pthread -papi=/N/u/gdiaz/hpc/bin/papi
TAU Makefile                 : /N/u/gdiaz/hpc/src/tau-2.21.2/x86_64/
                                                    lib/Makefile.tau-papi-mpi-pthread-mpitrace
TAU MetaData Merge Time      : 9.1E-05 seconds
TAU Version                  : 2.21.2
TAU_CALLPATH                 : off
TAU_CALLPATH_DEPTH           : 2
TAU_CALLSITE_LIMIT           : 1
TAU_COMM_MATRIX              : on
TAU_COMPENSATE               : off
TAU_CUPTI_API                : runtime
TAU_EBS_KEEP_UNRESOLVED_ADDR: off
TAU_IBM_BG_HWP_COUNTERS      : off
TAU_PROFILE                  : on
TAU_PROFILE_FORMAT           : profile
TAU_SAMPLING                 : off
TAU_SIGNALS_GDB              : off
TAU_SUMMARY                  : off
TAU_THROTTLE                 : on
TAU_THROTTLE_NUMCALLS        : 100000
TAU_THROTTLE_PERCALL         : 10
TAU_TRACE                    : off
TAU_TRACK_HEADROOM           : off
TAU_TRACK_HEAP               : off
```

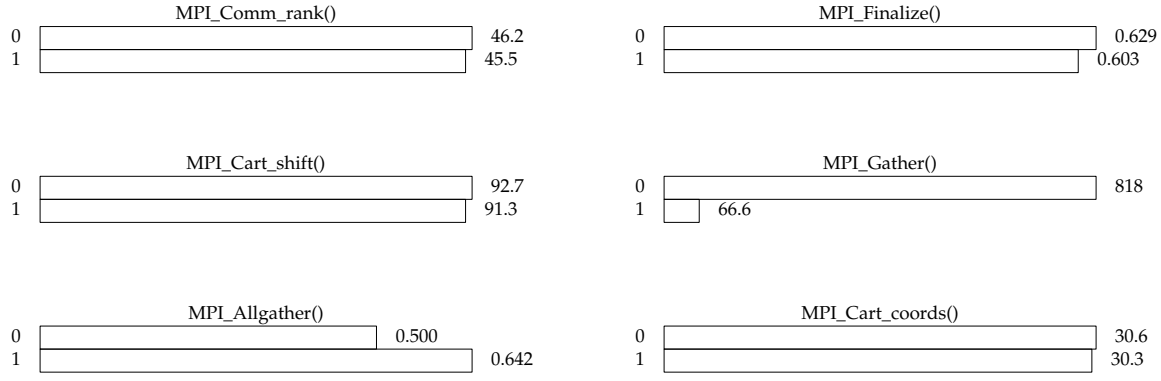
```
TAU_TRACK_IO_PARAMS         : off
TAU_TRACK_MEMORY_LEAKS      : off
TAU_TRACK_MESSAGE           : on
TAU_TRACK_SIGNALS           : off
username                    : gdiaz
```

Although the CPU has 4 cores, only one was used for the execution of this experiments.

### *3.1 Two physical machines (P-A)*

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 100.0 | 666888 | 751563 | 2 | 708290 | 375781626 | .TAU |
| 8.2 | 61257 | 61257 | 126760 | 0 | 483 | MPI_Wait() |
| 1.4 | 10513 | 10513 | 21764 | 0 | 483 | MPI_Bcast() |
| 0.8 | 5752 | 5752 | 63380 | 0 | 91 | MPI_Isend() |
| 0.5 | 3569 | 3569 | 13230 | 0 | 270 | MPI_Gatherv() |
| 0.2 | 1789 | 1789 | 2 | 0 | 894251 | MPI_Init() |
| 0.1 | 885 | 885 | 13372 | 0 | 66 | MPI_Gather() |
| 0.1 | 448 | 448 | 63380 | 0 | 7 | MPI_Irecv() |
| 0.0 | 184 | 184 | 200002 | 0 | 1 | MPI_Cart_shift() |
| 0.0 | 116 | 116 | 142 | 0 | 816 | MPI_Scatterv() |
| 0.0 | 91.7 | 91.7 | 126584 | 0 | 1 | MPI_Comm_rank() |
| 0.0 | 61.0 | 61.0 | 66264 | 0 | 1 | MPI_Cart_coords() |
| 0.0 | 5.47 | 5.47 | 13382 | 0 | 0 | MPI_Comm_size() |
| 0.0 | 1.23 | 1.23 | 2 | 0 | 616 | MPI_Finalize() |
| 0.0 | 1.14 | 1.14 | 8 | 0 | 143 | MPI_Allgather() |
| 0.0 | 0.781 | 0.781 | 4 | 0 | 195 | MPI_Allreduce() |
| 0.0 | 0.697 | 0.697 | 4 | 0 | 174 | MPI_Comm_split() |
| 0.0 | 0.439 | 0.439 | 2 | 0 | 220 | MPI_Comm_dup() |
| 0.0 | 0.005 | 0.005 | 8 | 0 | 1 | MPI_Type_size() |

**TABLE 10.** P-A Totals in miliseconds



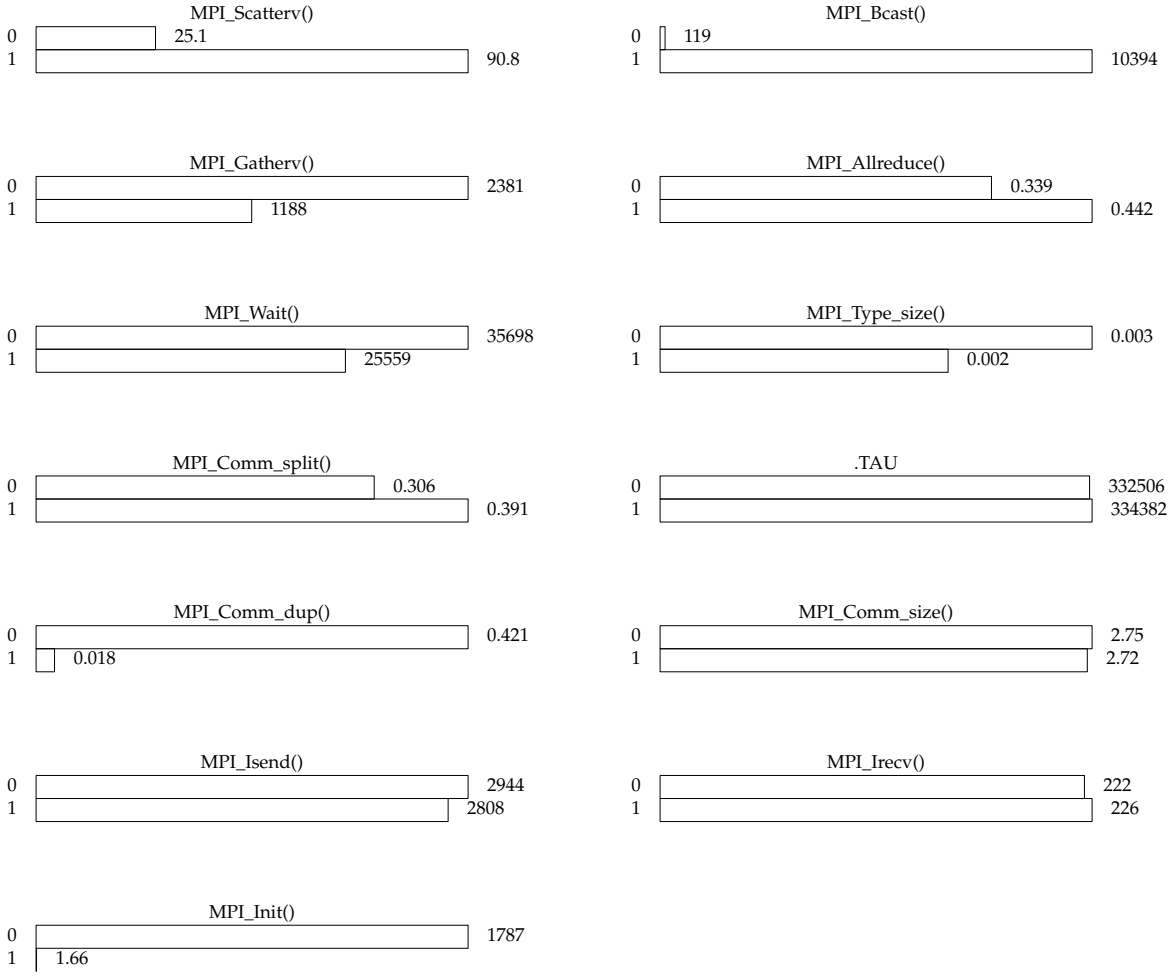
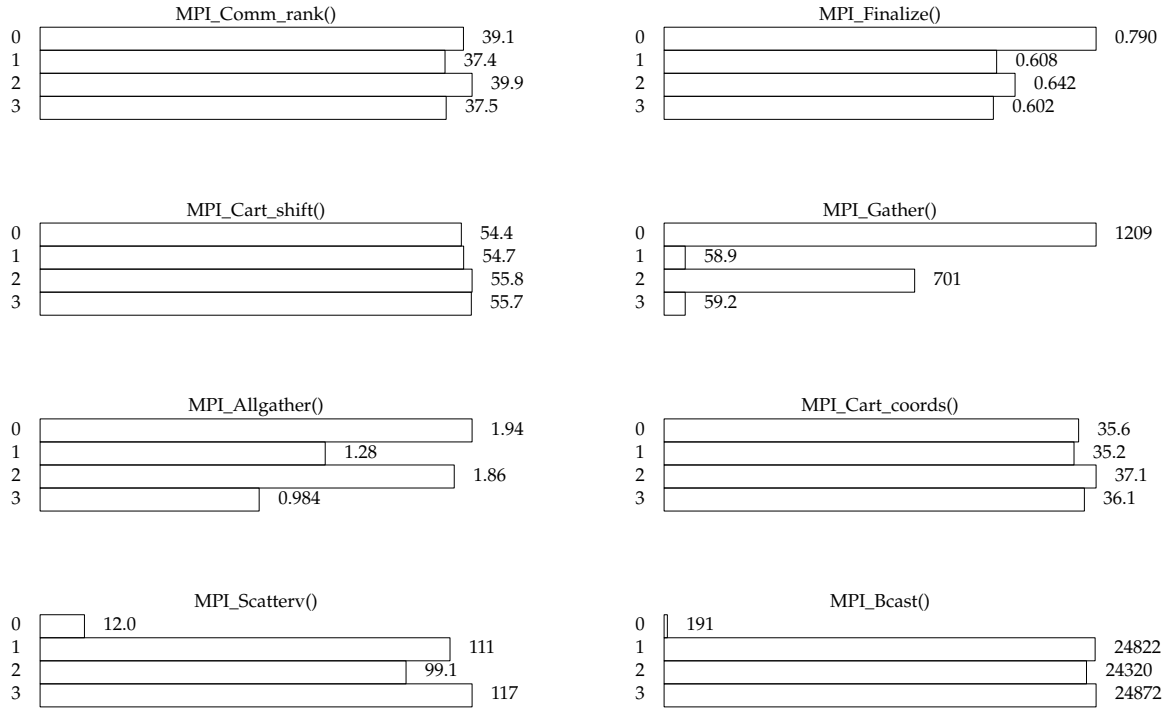



**Figure 36.** P-A Time distribution graph

### *3.2 Four physical machines (P-B)*

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 100.0 | 903022 | 1731586 | 4 | 2.86801E+06 | 432896399 | .TAU |
| 41.0 | 709272 | 709272 | 772080 | 0 | 919 | MPI_Wait() |
| 4.3 | 74205 | 74205 | 43528 | 0 | 1705 | MPI_Bcast() |
| 1.1 | 19873 | 19873 | 386040 | 0 | 51 | MPI_Isend() |
| 1.0 | 17138 | 17138 | 26460 | 0 | 648 | MPI_Gatherv() |
| 0.2 | 3425 | 3425 | 4 | 0 | 856340 | MPI_Init() |
| 0.1 | 2029 | 2029 | 26744 | 0 | 76 | MPI_Gather() |
| 0.1 | 1724 | 1724 | 386040 | 0 | 4 | MPI_Irecv() |
| 0.0 | 339 | 339 | 284 | 0 | 1195 | MPI_Scatterv() |
| 0.0 | 221 | 221 | 400004 | 0 | 1 | MPI_Cart_shift() |
| 0.0 | 154 | 154 | 400004 | 0 | 0 | MPI_Comm_rank() |
| 0.0 | 144 | 144 | 400004 | 0 | 0 | MPI_Cart_coords() |
| 0.0 | 23.0 | 23.0 | 26764 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 6.07 | 6.07 | 16 | 0 | 379 | MPI_Allgather() |
| 0.0 | 4.21 | 4.21 | 8 | 0 | 526 | MPI_Allreduce() |
| 0.0 | 3.06 | 3.06 | 8 | 0 | 382 | MPI_Comm_split() |
| 0.0 | 2.64 | 2.64 | 4 | 0 | 660 | MPI_Finalize() |
| 0.0 | 0.828 | 0.828 | 4 | 0 | 207 | MPI_Comm_dup() |
| 0.0 | 0.008 | 0.008 | 16 | 0 | 0 | MPI_Type_size() |

**TABLE 11.** P-B Totals in miliseconds



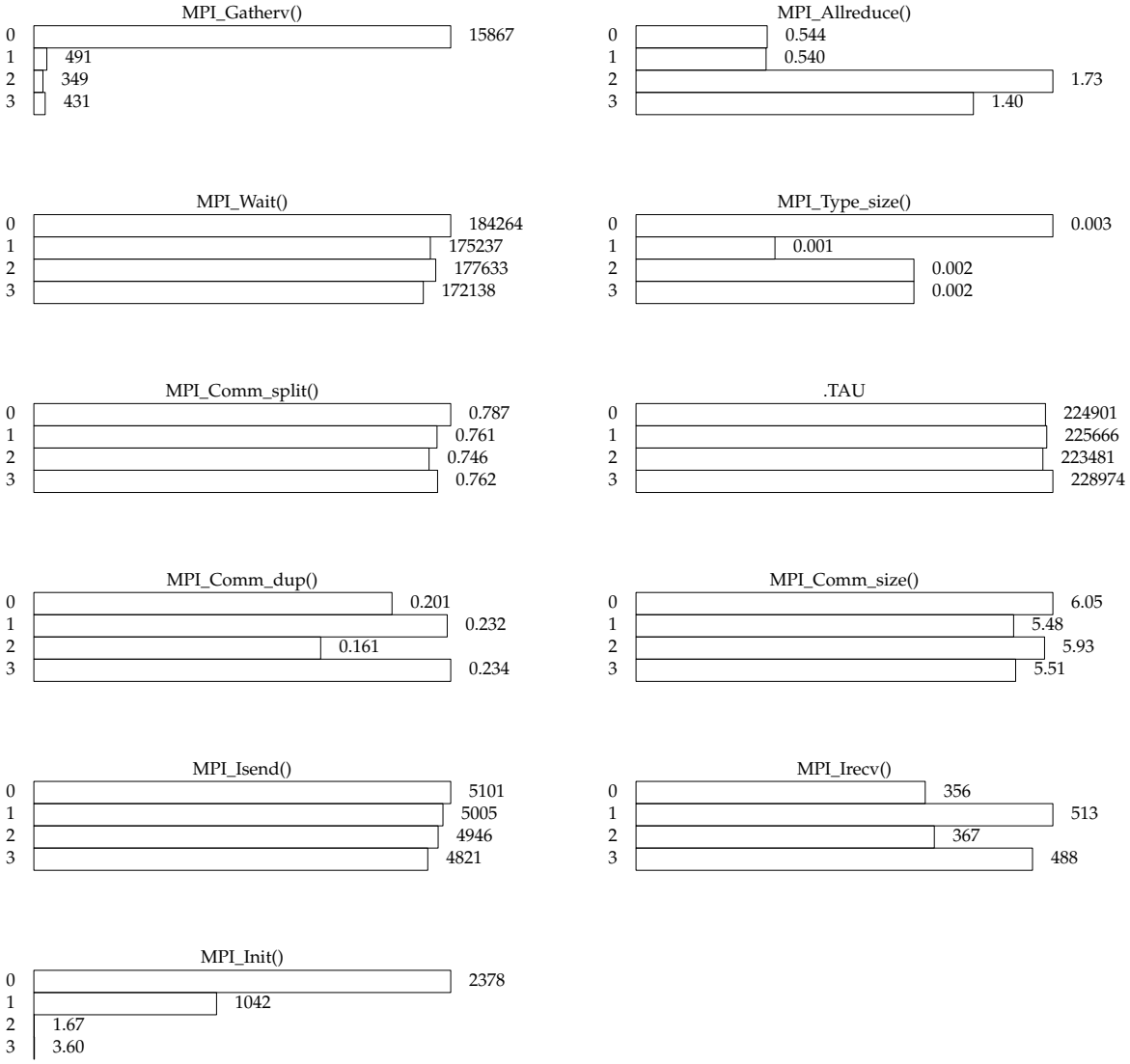
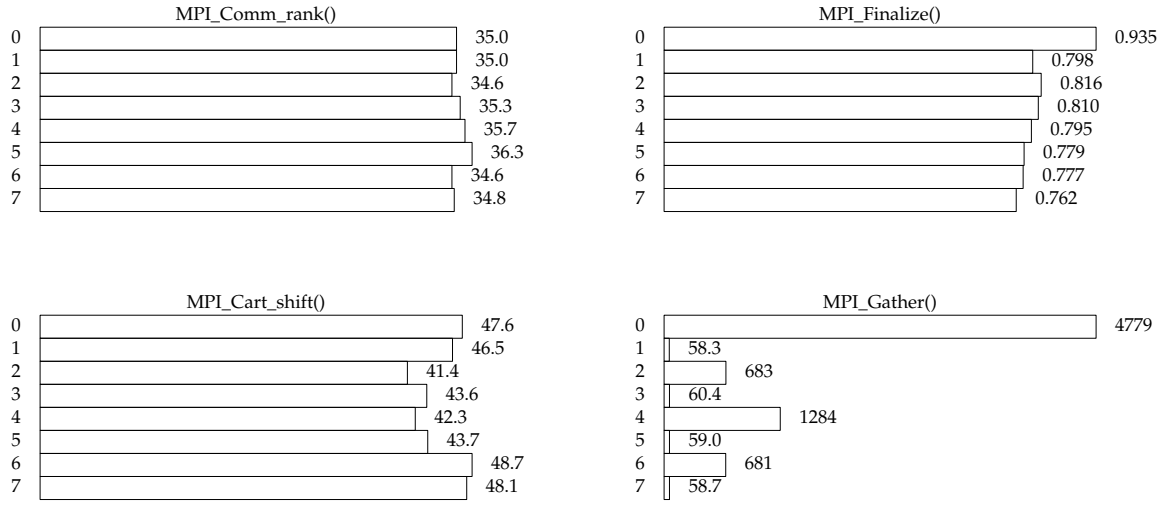


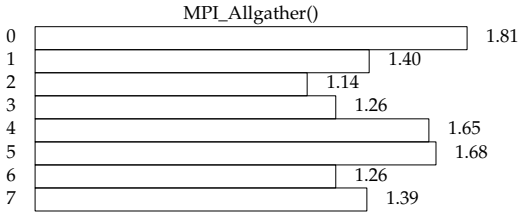

**Figure 37.** P-B Time distribution graph

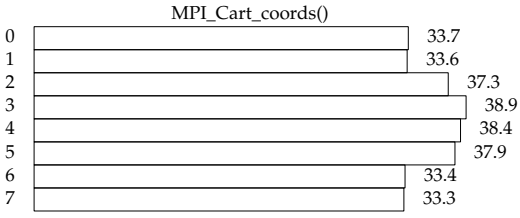
### *3.3 Eight physical machines (P-C)*

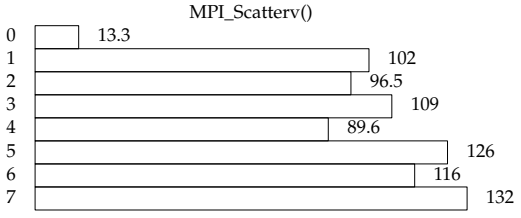
| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 49.7 | 1906213 | 1906213 | 1.79768E+06 | 0 | 1060 | MPI_Wait() |
| 24.4 | 935746 | 935746 | 87056 | 0 | 10749 | MPI_Bcast() |
| 100.0 | 811788 | 3833212 | 8 | 6.13027E+06 | 479151472 | .TAU |
| 3.2 | 122018 | 122018 | 52920 | 0 | 2306 | MPI_Gatherv() |
| 0.9 | 36237 | 36237 | 898840 | 0 | 40 | MPI_Isend() |
| 0.3 | 9908 | 9908 | 8 | 0 | 1238486 | MPI_Init() |
| 0.2 | 7664 | 7664 | 53488 | 0 | 143 | MPI_Gather() |
| 0.0 | 928 | 928 | 400004 | 0 | 2 | MPI_Irecv() |
| 0.0 | 917 | 917 | 386040 | 0 | 2 | MPI_Irecv() |
| 0.0 | 784 | 784 | 568 | 0 | 1381 | MPI_Scatterv() |
| 0.0 | 362 | 362 | 800008 | 0 | 0 | MPI_Cart_shift() |
| 0.0 | 287 | 287 | 800008 | 0 | 0 | MPI_Cart_coords() |
| 0.0 | 281 | 281 | 800008 | 0 | 0 | MPI_Comm_rank() |
| 0.0 | 38.5 | 38.5 | 53528 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 11.6 | 11.6 | 32 | 0 | 362 | MPI_Allgather() |
| 0.0 | 9.23 | 9.23 | 16 | 0 | 577 | MPI_Allreduce() |
| 0.0 | 8.61 | 8.61 | 16 | 0 | 538 | MPI_Comm_split() |
| 0.0 | 6.47 | 6.47 | 8 | 0 | 809 | MPI_Finalize() |
| 0.0 | 2.38 | 2.38 | 8 | 0 | 298 | MPI_Comm_dup() |
| 0.0 | 0.014 | 0.014 | 32 | 0 | 0 | MPI_Type_size() |

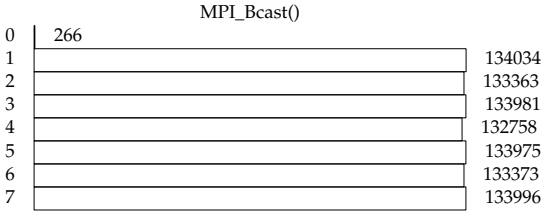
**TABLE 12.** P-C Totals in miliseconds

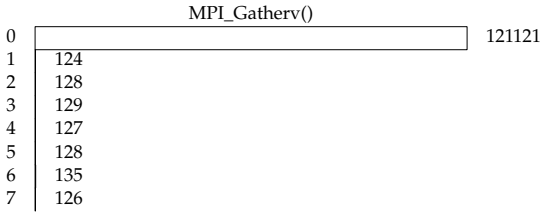


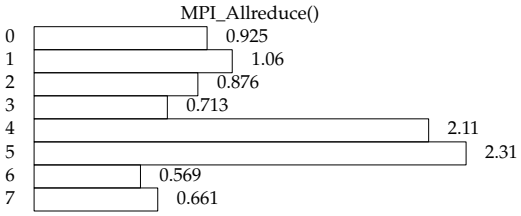

MPI_Allgather()
0 1.81
1 1.40
2 1.14
3 1.26
4 1.65
5 1.68
6 1.26
7 1.39

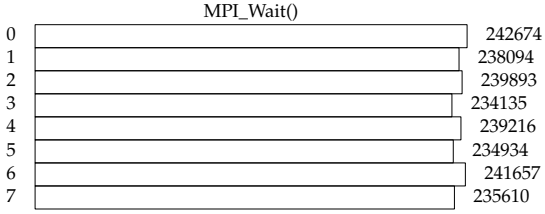

MPI_Cart_coords()
0 33.7
1 33.6
2 37.3
3 38.9
4 38.4
5 37.9
6 33.4
7 33.3

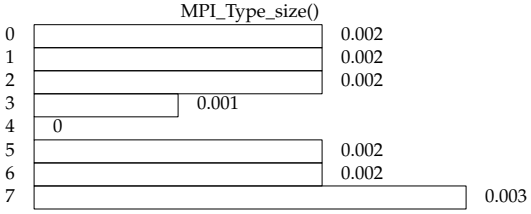

MPI_Scatterv()
0 13.3
1 102
2 96.5
3 109
4 89.6
5 126
6 116
7 132

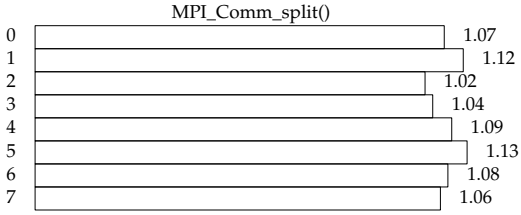

MPI_Bcast()
0 266
1 134034
2 133363
3 133981
4 132758
5 133975
6 133373
7 133996

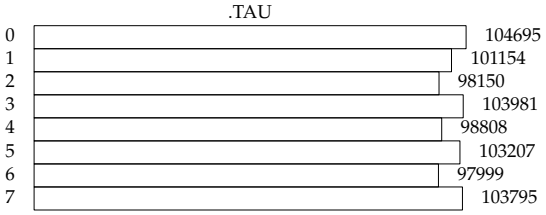

MPI_Gatherv()
0 121121
1 124
2 128
3 129
4 127
5 128
6 135
7 126

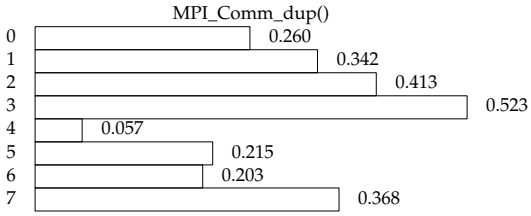

MPI_Allreduce()
0 0.925
1 1.06
2 0.876
3 0.713
4 2.11
5 2.31
6 0.569
7 0.661

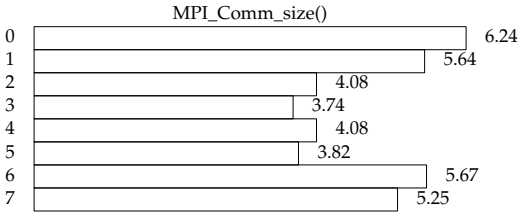

MPI_Wait()
0 242674
1 238094
2 239893
3 234135
4 239216
5 234934
6 241657
7 235610


MPI_Type_size()
0 0.002
1 0.002
2 0.002
3 0.001
4 0
5 0.002
6 0.002
7 0.003


MPI_Comm_split()
0 1.07
1 1.12
2 1.02
3 1.04
4 1.09
5 1.13
6 1.08
7 1.06


.TAU
0 104695
1 101154
2 98150
3 103981
4 98808
5 103207
6 97999
7 103795


MPI_Comm_dup()
0 0.260
1 0.342
2 0.413
3 0.523
4 0.057
5 0.215
6 0.203
7 0.368


MPI_Comm_size()
0 6.24
1 5.64
2 4.08
3 3.74
4 4.08
5 3.82
6 5.67
7 5.25

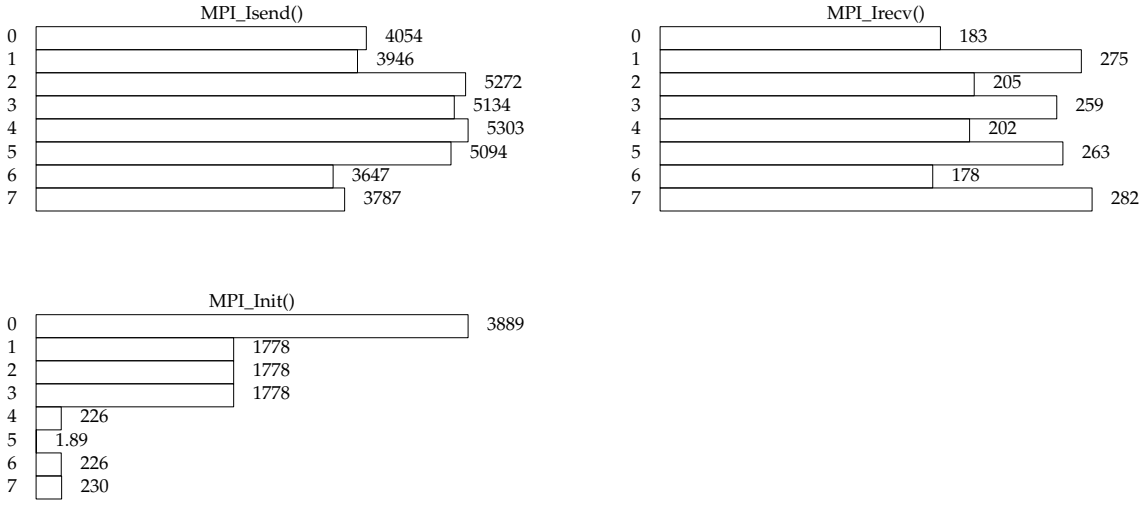
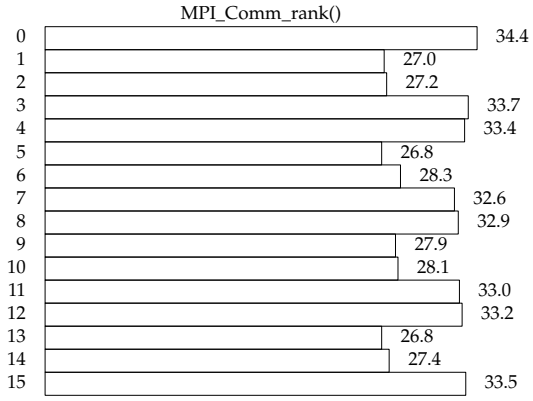


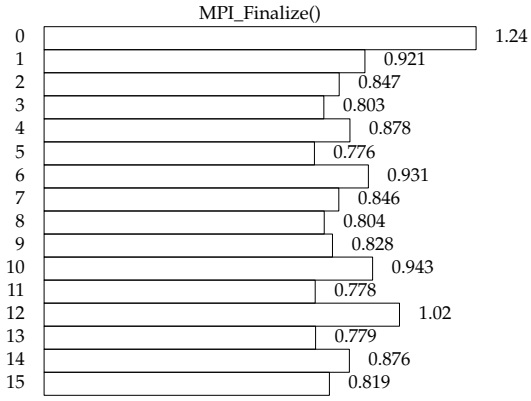

**Figure 38.** P-C Time distribution graph

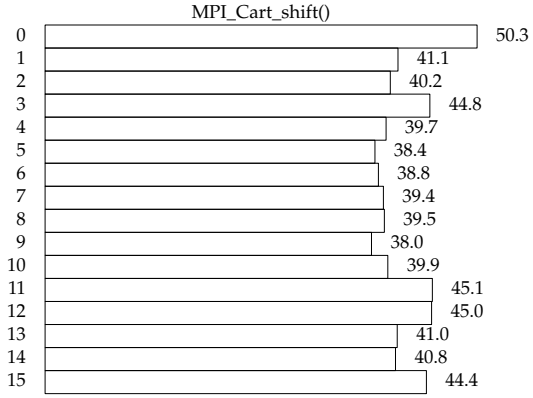
*3.4 Sixteen physical machines (P-D)*

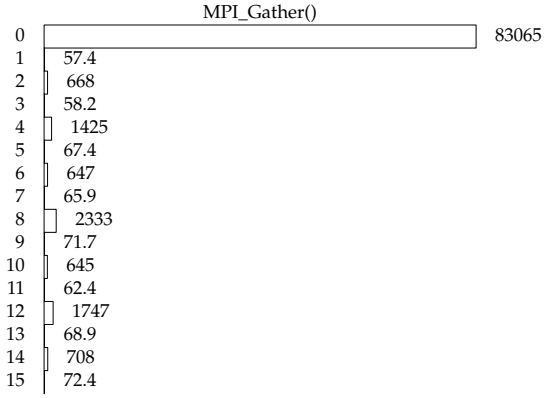
| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Name |
|---|---|---|---|---|---|---|
| 58.8 | 5189372 | 5189372 | 3.57232E+06 | 0 | 1453 | MPI_Wait() |
| 27.1 | 2393166 | 2393166 | 174112 | 0 | 13745 | MPI_Bcast() |
| 100.0 | 997951 | 8827063 | 16 | 1.22202E+07 | 551691429 | .TAU |
| 1.0 | 91764 | 91764 | 106976 | 0 | 858 | MPI_Gather() |
| 0.8 | 69955 | 69955 | 105840 | 0 | 661 | MPI_Gatherv() |
| 0.7 | 59498 | 59498 | 1.78616E+06 | 0 | 33 | MPI_Isend() |
| 0.2 | 18315 | 18315 | 16 | 0 | 1144710 | MPI_Init() |
| 0.0 | 1731 | 1731 | 766320 | 0 | 2 | MPI_Irecv() |
| 0.0 | 1719 | 1719 | 1136 | 0 | 1513 | MPI_Scatterv() |
| 0.0 | 1664 | 1664 | 800008 | 0 | 2 | MPI_Irecv() |
| 0.0 | 666 | 666 | 1.60002E+06 | 0 | 0 | MPI_Cart_shift() |
| 0.0 | 521 | 521 | 1.60002E+06 | 0 | 0 | MPI_Cart_coords() |
| 0.0 | 486 | 486 | 1.60002E+06 | 0 | 0 | MPI_Comm_rank() |
| 0.0 | 102 | 102 | 32 | 0 | 3186 | MPI_Allreduce() |
| 0.0 | 68.7 | 68.7 | 107056 | 0 | 1 | MPI_Comm_size() |
| 0.0 | 41.7 | 41.7 | 64 | 0 | 652 | MPI_Allgather() |
| 0.0 | 21.7 | 21.7 | 32 | 0 | 679 | MPI_Comm_split() |
| 0.0 | 14.1 | 14.1 | 16 | 0 | 880 | MPI_Finalize() |
| 0.0 | 6.66 | 6.66 | 16 | 0 | 416 | MPI_Comm_dup() |
| 0.0 | 0.020 | 0.020 | 64 | 0 | 0 | MPI_Type_size() |

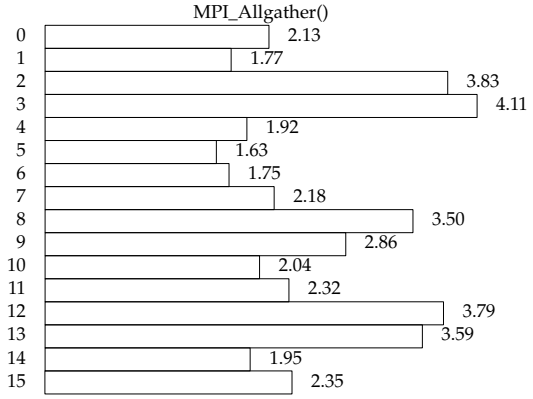
**TABLE 13.** P-D Totals in miliseconds

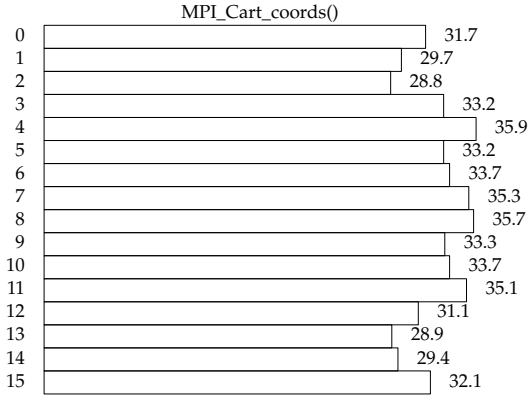

MPI_Comm_rank()
0 34.4
1 27.0
2 27.2
3 33.7
4 33.4
5 26.8
6 28.3
7 32.6
8 32.9
9 27.9
10 28.1
11 33.0
12 33.2
13 26.8
14 27.4
15 33.5

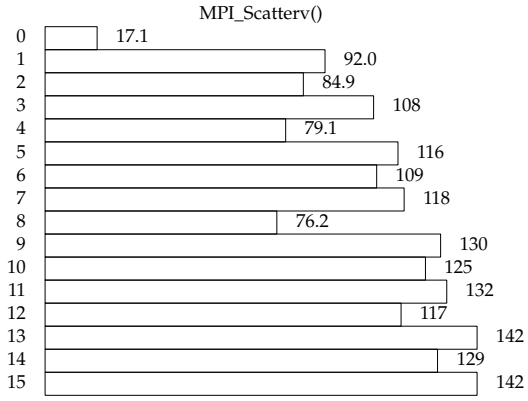

MPI_Finalize()
0 1.24
1 0.921
2 0.847
3 0.803
4 0.878
5 0.776
6 0.931
7 0.846
8 0.804
9 0.828
10 0.943
11 0.778
12 1.02
13 0.779
14 0.876
15 0.819

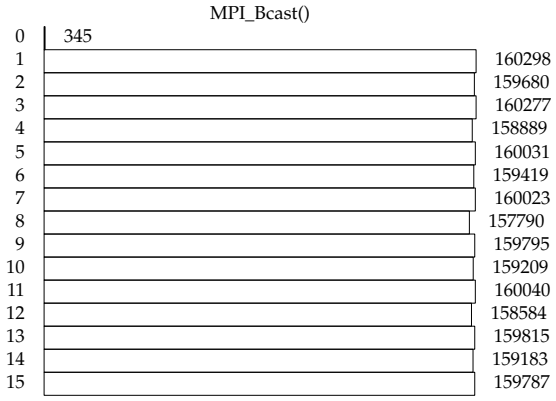

MPI_Cart_shift()
0 50.3
1 41.1
2 40.2
3 44.8
4 39.7
5 38.4
6 38.8
7 39.4
8 39.5
9 38.0
10 39.9
11 45.1
12 45.0
13 41.0
14 40.8
15 44.4

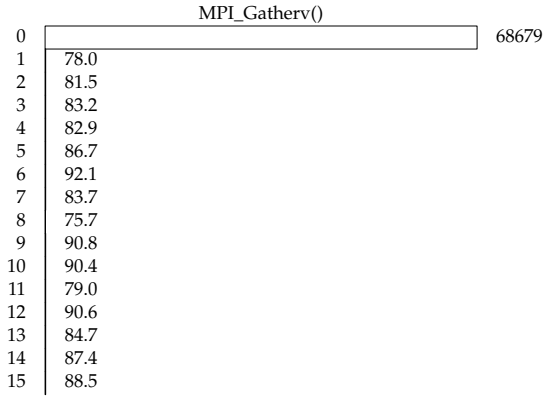

MPI_Gather()
0 83065
1 57.4
2 668
3 58.2
4 1425
5 67.4
6 647
7 65.9
8 2333
9 71.7
10 645
11 62.4
12 1747
13 68.9
14 708
15 72.4

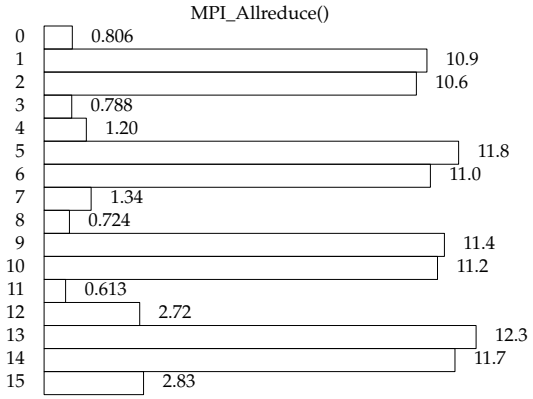

MPI_Allgather()
0 2.13
1 1.77
2 3.83
3 4.11
4 1.92
5 1.63
6 1.75
7 2.18
8 3.50
9 2.86
10 2.04
11 2.32
12 3.79
13 3.59
14 1.95
15 2.35

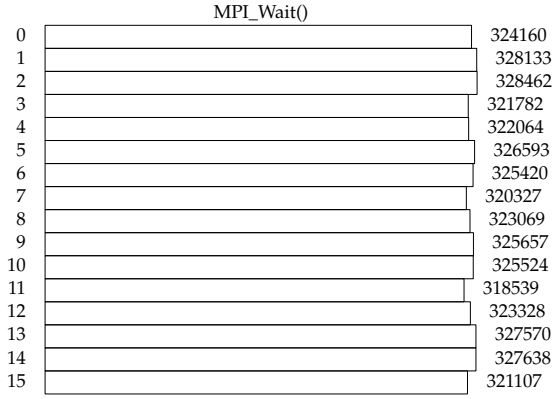

MPI_Cart_coords()
0 31.7
1 29.7
2 28.8
3 33.2
4 35.9
5 33.2
6 33.7
7 35.3
8 35.7
9 33.3
10 33.7
11 35.1
12 31.1
13 28.9
14 29.4
15 32.1

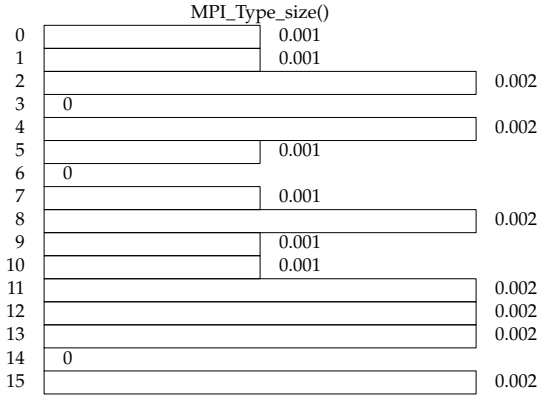

MPI_Scatterv()
0 17.1
1 92.0
2 84.9
3 108
4 79.1
5 116
6 109
7 118
8 76.2
9 130
10 125
11 132
12 117
13 142
14 129
15 142

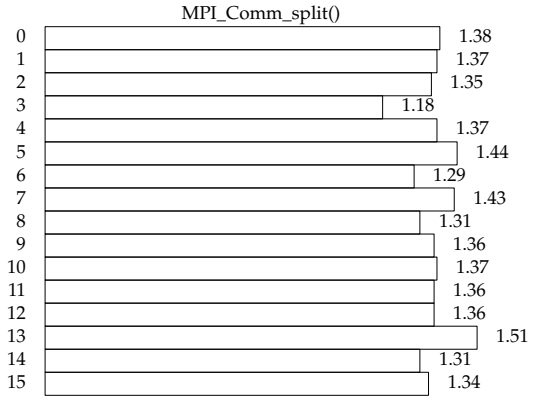

MPI_Bcast()
0 345
1 160298
2 159680
3 160277
4 158889
5 160031
6 159419
7 160023
8 157790
9 159795
10 159209
11 160040
12 158584
13 159815
14 159183
15 159787

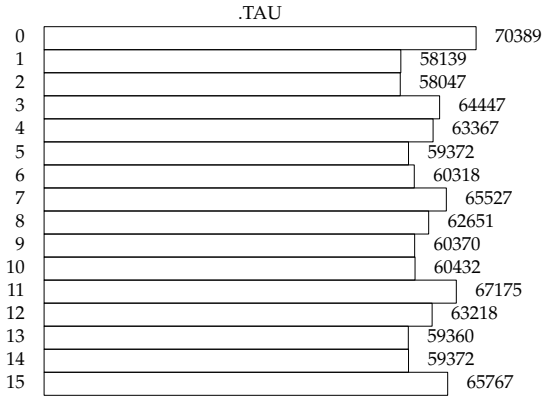

MPI_Gatherv()
0 68679
1 78.0
2 81.5
3 83.2
4 82.9
5 86.7
6 92.1
7 83.7
8 75.7
9 90.8
10 90.4
11 79.0
12 90.6
13 84.7
14 87.4
15 88.5

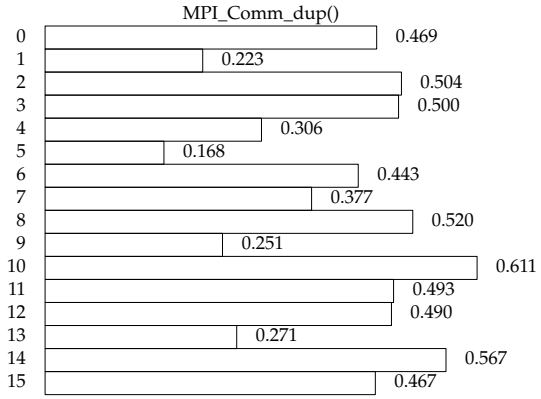

MPI_Allreduce()
0 0.806
1 10.9
2 10.6
3 0.788
4 1.20
5 11.8
6 11.0
7 1.34
8 0.724
9 11.4
10 11.2
11 0.613
12 2.72
13 12.3
14 11.7
15 2.83

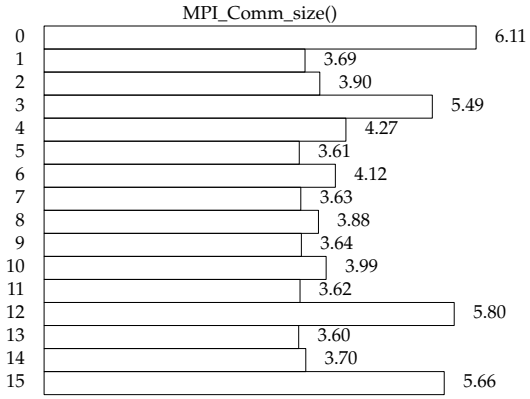

MPI_Wait()
0 324160
1 328133
2 328462
3 321782
4 322064
5 326593
6 325420
7 320327
8 323069
9 325657
10 325524
11 318539
12 323328
13 327570
14 327638
15 321107

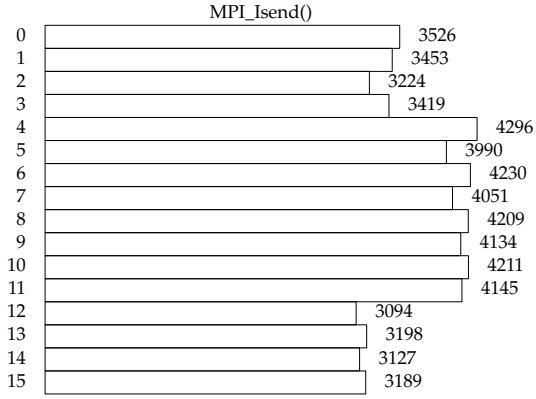

MPI_Type_size()
0 0.001
1 0.001
2 0.002
3 0
4 0.002
5 0.001
6 0
7 0.001
8 0.002
9 0.001
10 0.001
11 0.002
12 0.002
13 0.002
14 0
15 0.002

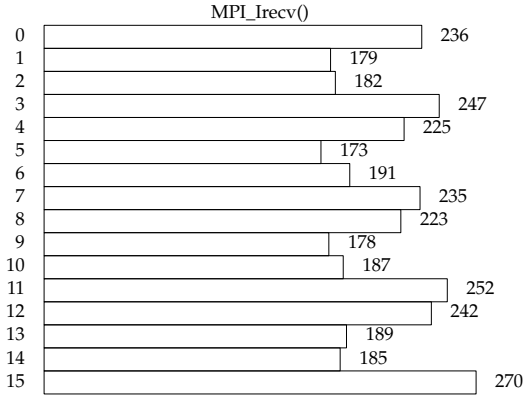

MPI_Comm_split()
0 1.38
1 1.37
2 1.35
3 1.18
4 1.37
5 1.44
6 1.29
7 1.43
8 1.31
9 1.36
10 1.37
11 1.36
12 1.36
13 1.51
14 1.31
15 1.34


.TAU
0 70389
1 58139
2 58047
3 64447
4 63367
5 59372
6 60318
7 65527
8 62651
9 60370
10 60432
11 67175
12 63218
13 59360
14 59372
15 65767


MPI_Comm_dup()
0 0.469
1 0.223
2 0.504
3 0.500
4 0.306
5 0.168
6 0.443
7 0.377
8 0.520
9 0.251
10 0.611
11 0.493
12 0.490
13 0.271
14 0.567
15 0.467


MPI_Comm_size()
0 6.11
1 3.69
2 3.90
3 5.49
4 4.27
5 3.61
6 4.12
7 3.63
8 3.88
9 3.64
10 3.99
11 3.62
12 5.80
13 3.60
14 3.70
15 5.66


MPI_Isend()
0 3526
1 3453
2 3224
3 3419
4 4296
5 3990
6 4230
7 4051
8 4209
9 4134
10 4211
11 4145
12 3094
13 3198
14 3127
15 3189


MPI_Irecv()
0 236
1 179
2 182
3 247
4 225
5 173
6 191
7 235
8 223
9 178
10 187
11 252
12 242
13 189
14 185
15 270

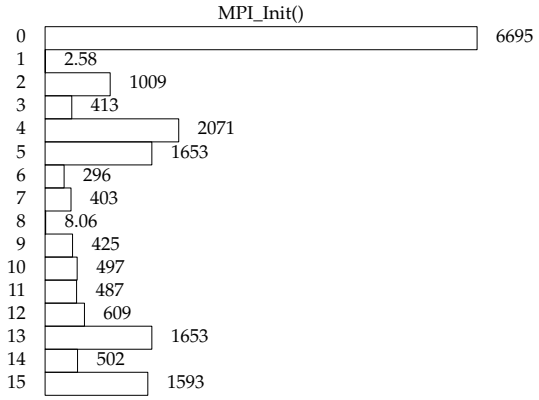



**Figure 39.** P-D Time distribution graph

# 7
# Conclusion

Our aim with this experiments is to test if MPI collective operations behave and scale the same way in physical and virtual environments. We prepared two environments, physical and virtual, running in the same hardware and the same software. We measured four simulations with two, four, eight and sixteen nodes in both environments.

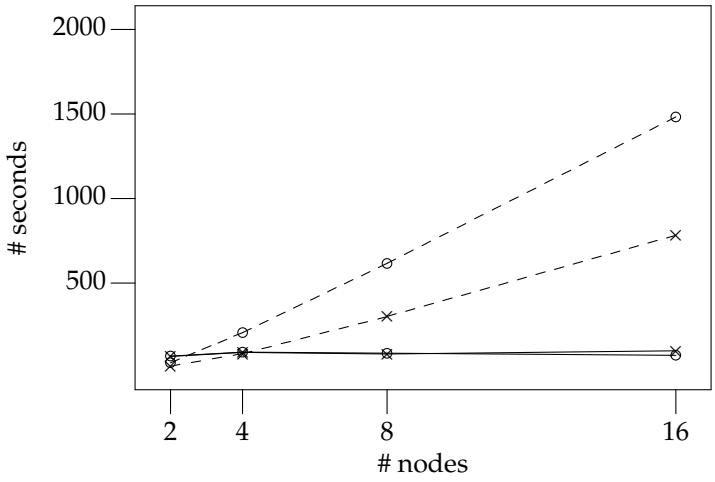



**Figure 40.** Virtual(o)/Physical(x) MPI(dashed) vs Application(solid) times

In figure [40] we compare the total time employed running MPI functions with the time employed running the application itself.

The experiment suggest that the scalability of MPI is inferior compared to the physical environment, while the application scales the same way in both environments. Also the tendency is to increase the difference as the number of nodes increase.

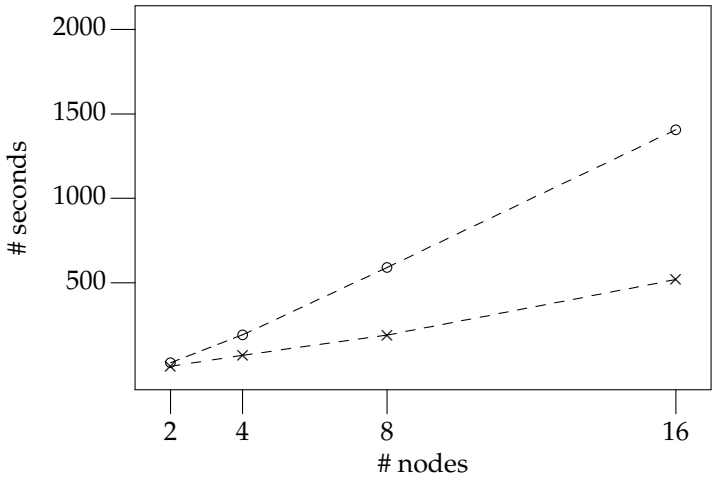



**Figure 41.** Virtual(o)/Physical(x) MPI_Wait() times

If we consider the time spent in `MPI_Wait()` calls in figure [41], the time used for waiting the completion MPI operations in virtual environment is bigger than in physical environments. Also, there is a huge magnitude difference between the time spent waiting in comparison with the time spent calling other MPI functions.

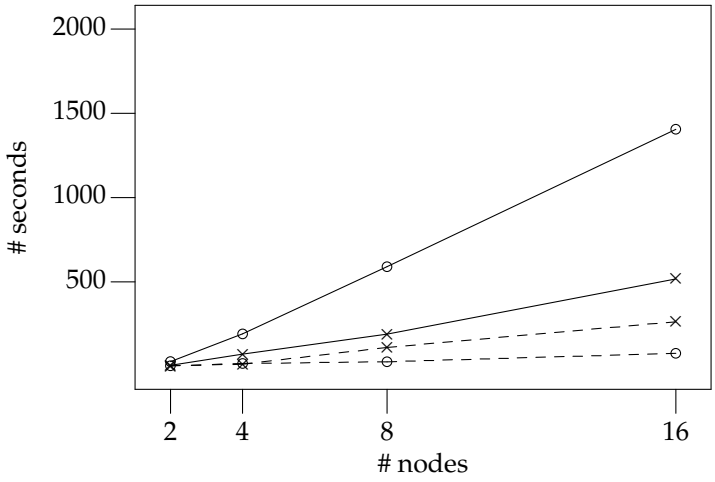



**Figure 42.** Virtual(o)/Physical(x) MPI_Wait() (solid) vs other MPI calls(dashed)

In the figure [42] we observe that the MPI collective operations behave similarly in physical and virtual environments, also the convergence between the time spent in `MPI_Wait()` and in the other MPI function calls in the physical environment is bigger than in the virtual environment, where the `MPI_Wait()` spends much more time than the other calls.

The MPI standard v2.0 states that `MPI_Wait()` is called to wait for a given operation to finish,

for example, after a `MPI_Recv` call, to make sure we have received all we need, a developer can call `MPI_Wait` to block until a given object is available. Thus, the observed behavior could be an effect of the virtual low performance in comparison with the physical environments. For example, if the transport of a given message takes longer in a virtual environment, the time spent by `MPI_Wait()` will be bigger too.

This leads to a series of new questions about why this could happen, including questions about the hardware infrastructure details, the software environment and the testing methodology. Still, the importance of measuring this behavior is fundamental in order to save computing power which at big scales would mean a lot of wasted resources.

The results prove that its important to do more experimentation, not only with real applications but with conceptualized tests, in order to be able to measure the impact of virtualization in the MPI collective operations.

The simplest simulation test of the WRF have a noticeable performance penalty in MPI collective communications under virtualized environment. To verify if this behavior is due to this current simulation or to the whole WRF approach of MPI usage, we propose to experiment with other real case applications and benchmarks, such as the HPL.

# 8
# Bibliography

*REFERENCES*


1. S. Ostermann, A. Iosup, N. Yigibasi, R. Prodan, T. Fahringer, and D. Epema, An early performance analysis of cloud computing services for scientific computing, Delft University of Technology, (December 2008)

2. Younge, A.J., R. Henschel,, J. Brown,, G. von Laszewski, J. Qiu, and G. C. Fox, Analysis of Virtualization Technologies for High Performance Computing Environments, The 4th International Conference on Cloud Computing (IEEE CLOUD 2011), Washington, DC, IEEE, (July 2011)

3. L. Youseff, R. Wolski, B. C. Gorda, and C. Krintz. Paravirtualization for hpc systems. In ISPA Workshops, volume 4331 of Lecture Notes in Comput. Sci., pages 474 486. Springer-Verlag, 2006

4. E. Deelman, G. Singh, M. Livny, J. B. Berriman, and J. Good. The cost of doing science on the cloud: the Montage example. In ACM/IEEE SuperComputing Conference on High Performance Networking and Computing (SC), page 50. IEEE/ACM, 2008.

5. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. 2008. UCSD Tech.Rep. 2008-10. http://eucalyptus.cs.ucsb.edu/

6. B. Quétier, V. Néri, and F. Cappello. Scalability comparison of four host virtualization tools. J. Grid Comput., 5(1):8398, 2007.

8. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, Parallel Computing, 22 (6), pp. 789-828., Department of Computer Science & NSF Engineering Research Center for CFS, Mississippi State University, http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/papers/mpichimpl.pdf (1996)

8. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, www.mpi-forum.org/docs/mpi-10.ps (May 5, 1994).

9. Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan, Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library, Technical Report MSU-940722,, Mississippi State University Dept. of Computer Science, http://www.erc.msstate.edu/mpi/mpix.html (April 1994).

10. Decker, Steven G., Nonlinear Balance in Terrain-Following Coordinates, on. Wea. Rev., 138, 605 624., Rutgers, The State University of New Jersey, New Brunswick, New Jersey, http://journals.ametsoc.org/doi/abs/10.1175/2009MWR2971.1 (2010).

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Partners in the FutureGrid project include U. Chicago, U. Florida, San Diego Supercomputer Center - UC San Diego, U. Southern California, U. Texas at Austin, U. Tennessee at Knoxville, U. of Virginia, Purdue U., and T-U. Dresden.