KADIR HAS UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

# Dynamic Load Balancing in Distributed Systems:

# “Hands of God” IN PARALLEL PROGRAMMING with MPI

**Aliakbar Sadeghi Khameneh Tabrizi**

June, 2014

Aliakbar Sadeghi Khameneh Tabrizi

Master Thesis

2014

Dynamic Load Balancing in Distributed Systems:

"Hands of God" IN PARALLEL PROGRAMMING with MPI

Aliakbar Sadeghi Khameneh Tabrizi
B.S., Computer Engineering, Tehran Azad University Central Branch, 2005
M.S., Computer Engineering, KADIR HAS University, 2014

Submitted to the Graduate School of Science and Engineering

In partial fulfilment of the requirements for the degree of

Master of Science

In

Computer Engineering

KADIR HAS UNIVERSITY
June, 2014

Dynamic Load Balancing in Distributed Systems:

"Hands of God" IN PARALLEL PROGRAMMINGwith MPI

Dynamic Load Balancing of MPI+MPICH Applications:

HoG IN PARALLEL PROGRAMMING

Aliakbar Sadeghi Khameneh Tabrizi

Assist.ProfDr. Taner Arsan (Kadir Has University)

(Thesis Supervisor)

Assoc.Dr. D.Turgay Altılar (Istanbul Technical University)

(Thesis Co-Advisor)

Prof.Dr. Nadia Erdoğan (Istanbul Technical University)

Assist.Prof.Dr. A. Selçuk Öğrenci (Kadir Has University)

APPROVAL DATE:

“Ben, Aliakbar Sadeghi Khameneh Tabrizi, bu Bilgisayar Mühendisliği Yüksek Lisans Tezinde sunulan çalışmanın şahsıma ait olduğunu ve başka çalışmalardan yaptığım alıntıların kaynaklarını kurallara uygun biçimde tez içerisinde belirttiğimi onaylıyorum.”

aliakbar Sadeghi Khameneh Tabrizi

## Dynamic Load Balancing in Distributed Systems:

## "Hands of God" IN PARALLEL PROGRAMMING with MPI

## Abstract

In today's technology, what really is missing in computer systems is more artificial intelligence, and in the same time implanting lots of intelligence in computer systems, is not as easy as it sounds, but even one step ahead to make computer software to act more efficient and intelligent is noteworthy.

*MPICH* is a *message passing interface framework* designed to be the host for parallel programs, but like too many other great programming frameworks, development of *MPICH* is ongoing and once a while we are witnessing new updates, which mostly these updates are, in order to support more functionality, and performance improvement updates, are more rare.

One of the issues that happened while we were working on *Snake in the box* problem (more details on this problem could be found in Wikipedia) with my parallel programming Professor *Dr Turgay Altilar* in *Kadir Has* University, was lack of intelligence in the parallel version of the algorithm of *Snake in the box* in *MPICH* framework.

In general, when you are designing a parallel algorithm you should focus more about splitting the jobs, somehow equal in order to keep all the processes working during the execution to achieve the best performance but in so many cases it's more close to a wish, actually there was not any clear method to act smart in the middle of execution in order to balance the load intelligently.

So I came to the point to tune the *MPICH* execution process by adding a new function to the list of *MPI* commands by the name of *MPI_HoG()*, which one single call of this function in the user program in the initialization section and after *MPI_Init(),* will add some intelligence behaviour in the run time of any *MPI* program which executed under *MPICH* framework.

HoG stands for *Hands of God,* which is related to the behaviour of HoG, Hands of God has root in an old belief, which explaining a situation which in that status, in the process of completion of a task or a phenomenon something

extraordinary happened, that it wasn`t supposed to occur, and also that phenomenal occurrence fulfilled an extenuation.

*MPI_HoG()* or in short *HoG*`s duty is to monitor all the *MPI* processes during the execution and surveys their behaviours, and after a specified number of iterations, based on some predefined scenarios, it will make an important decision to change the node of the process which needs help, and which stuck in the heavy load of tasks. HoG will handle it by migrating the related MPI processes from the busy node to another machine which we call it *super node* and it will be explained in detail later in chapter five, but in order to have an idea about the definition of super node, the abstract definition will be like this: A super node is a machine which have better performance mark in the cluster.

In today`s clusters, mostly infrastructure designers are trying to set up clusters somehow to meet equality in hardware specification and performance, which gave the software developer a relaxation in order to focus only on splitting the tasks equal and parallel as possible between nodes in the software, and trust the cluster to provide same performance strength for each node. But what is unpredictable is the fact that *no developer can predict 100% that everything will continue with equal load on every nodes in a parallel cluster*.

However, the HoG` idea was to designing a normal cluster which mostly all the machines have same hardware specification and same performance rank and also add some *Super Nodes* which in the case of performance issue, *HoG* will understand it and migrate the performance intense task from the busy node(s) to the Super Node(s) and help the busy node(s) when they really need it.

Eventually, when the step one of design and implementation of the *HoG* finished, HoG achieved a fast, secure and reliable method of dynamic load balancing during the execution of any MPI program, in another word, HoG is the first checkpointing implementation in order to approach faster execution and better performance.

## Acknowledgements

I would like to state that it was a great honour to meet and work with *Dr Taner Arsan*, and I would like to appreciate all his supports and professional advices.

I would like to state that it was a great honour to meet and work with *Dr Turgay Altilar*. He gave me such a great view, on parallel programming and also supported me on my challenging idea of the HoG project.

I would like to appreciate all supports and kind advices that I received from *Dr Feza Kerestecioglu* and *Dr Ayse Bilge*.

In addition I would like to appreciate all the supports and helps which I received from *Mr Taylan Ocakcioglu* during my studying in *Kadir Has University*.

Last year was a year of hard working and effort, which did not have any meaning if my wife and in the same time my best friend *Pelin Tabrizi* was not there for me, I would like to appreciate her support and the warm atmosphere that she made in the way of my achievement and in the other hand the love and supports which I received from my mother *Ladan Tahmasebpour* and my father *Shamseddin Tabrizi* and also my mother in law *Birsen Yalinkilic*.

# Table of Contents

**Abstract** ........ **i**

**Acknowledgements** ........ **iii**

**Table of Contents** ........ **iv**

**List of Tables** ........ **viii**

**List of Figures** ........ **x**

**List of Abbreviations** ........ **xi**

**1. Introduction** ........ **1**

1.1. Thesis Introduction ........ 1

1.2. Motivations ........ 1

1.3. Contribution ........ 4

1.4. Thesis Structure ........ 5

**2. Check Pointing** ........ **7**

2.1. Introduction ........ 7

2.1.1. Backgroud ........ 7

2.2. Application Checkpointing ........ 8

2.3. Different Checkpointing Implementations ........ 9

2.4. BLCR ........ 11

2.4.1. BLCR & HoG ........ 12

**3. MPICH** ........ **13**

3.1. MPICH ........................................ 13

3.1.1. MPICH2 vs. MPICH3 ........................................ 13

3.2. Hydra ........................................ 14

3.3. Integration with BLCR ........................................ 14

**4. System Environment ........................................ 17**

4.1. Virtual Box ........................................ 17

4.2. Ubuntu ........................................ 17

4.2.1. Ubuntu Customizations ........................................ 17

4.3. BLCR ........................................ 19

4.4. MPICH ........................................ 20

**5. HoG ........................................ 21**

5.1. Overview ........................................ 21

5.1.1. Actively waiting ........................................ 23

5.2. Structure ........................................ 23

5.2.1. Transparency ........................................ 23

5.2.2. Resource safety ........................................ 24

5.3. MPICH Workflow ........................................ 27

5.3.1. MPICH Workflow in Checkpointing ........................................ 28

5.3.2. MPICH Workflow in Restarting ........................................ 30

5.4. HoG Workflow ........................................ 32

5.4.1. HoG Workflow before Checkpointing ..... 33
5.4.2. HoG Workflow after Checkpointing ..... 34
5.4.3. HoG Server Application ..... 35
5.4.4. HoG Workflow after Restarting ..... 36
5.5. Single Node Switch Scenarios ..... 36
5.5.1. All Waiting for One ..... 37
5.5.2. All Waiting For Two ..... 39
**6. Tests and Experiments ..... 41**
6.1. Test Cases ..... 41
6.1.1. Test Cases Structure ..... 41
6.2. Performance Results ..... 47
6.3. Result Comparison ..... 51
**Related Works ..... 54**
**Conclusion ..... 55**
Future works ..... 55
Node Switch Mechanisms ..... 55
Scenarios ..... 56
Conclusion ..... 58
**References ..... 61**
**Curriculum Vitae ..... 63**

## List of Tables

Table 1- Assistants and tasks example in first 10 minutes.......................................................2

Table 2- Assistants and tasks example in first 20 minutes.......................................................2

Table 3- Assistants and tasks example in first 100 minutes ....................................................2

Table 4- Overview of different Checkpointing implementations.............................................9

Table 5- Virtual Box nodes description ...................................................................................18

Table 6- HoG example`s cluster specification.........................................................................21

Table 7- Before HoG do the migration ....................................................................................39

Table 8- After HoG did the migration......................................................................................39

Table 9- Before HoG do the migration ....................................................................................40

Table 10- After HoG did the first migration.............................................................................40

Table 11- After HoG did the second migration ........................................................................40

Table 123- Cluster structure of test case 1 ...............................................................................42

Table 13- Cluster structure of test case 2 .................................................................................43

Table 14- Cluster structure of test case 3 .................................................................................44

Table 15- Cluster structure of test case 4 .................................................................................45

Table 18- Cluster structure of test case 5 .................................................................................46

Table 19- Cluster structure of test case 6 .................................................................................47

Table 20- Related Works..........................................................................................................54

Table 21- Multiple nodes switch, node`s specification ........ 56

Table 22- Multiple nodes switch, node`s specification ........ 56

Table 23- Before HoG do the migration ........ 57

Table 24- After HoG did the migration ........ 57

# List of Figures

Figure 1- Assistants Performance ........ 3

Figure 2- Thesis Structure ........ 5

Figure 3- HoG Abstract Workflow ........ 26

Figure 4- MPICH Abstract Workflow ........ 28

Figure 5- Checkpointing Workflow ........ 29

Figure 6- Checkpoint Restart Workflow ........ 31

Figure 7- MPI HoG Hierarchy ........ 33

Figure 8- HoG Server Structure ........ 35

Figure 9- Scenario 1, All Waiting for One ........ 37

Figure 10 - Performance Comparison Based on the Execution Time ........ 52

Figure 11- Performance Comparison, Percentage of the Execution Time Improvement ........ 52

Figure 12- Processing Hierarchy ........ 58

Figure 13- Processor Schema ........ 59

## List of Abbreviations

| | |
|---|---|
| HoG | Hands OF God |
| MPI | Message Passing Interface |
| MPICH | Message Passing Interface Chameleon |
| BLCR | Berkely Lab CheckPoint / Restart |
| Open VZ | Open Virtuozzo |
| LXC | LinuX Countainers |
| DMTCP | Distributed Multi-Threaded Check Pointing |
| CRIU | Check point / Restart in User space |
| VEs | Virtual Environments |
| PVS | Private Virtual System |
| FD | File Descriptor |
| PID | Process ID |
| GPID | Group Process ID |
| PPID | Parent Process ID |
| LTS | Long Term Support |
| MAC | Media Access Control |
| DNS | Domain Name Service |
| SSH | Secure Shell |
| NFS | Network File Sharing |

# Dynamic Load Balancing in Distributed Systems:

# "Hands of God" IN PARALLEL PROGRAMMING with MPI

## Chapter 1

## 1. Introduction

### 1.1. **Thesis Introduction**

In today`s technological improvement human is trying to make things faster, more secure and in the same time more reliable, computer systems are one of the main subjects among the technological improvements, as we know, these Electronic machines are designed to do whatever we programed them to do, which exactly this approach became one of the challenging subjects in the science, because after facing hardware limitation, the only solution to make computer systems faster is implementing more intelligence into the software and hardware, in order to make smart and useful decisions, according to the situation.

### 1.2.**Motivations**

Importing more intelligence into computer systems were our goal in this project, to make parallel programming much easier and capable under the MPICH framework, and to use the infrastructure more efficiently and intelligently, a simple example can clarify the subject.

As we know parallelism is one of the solutions to performance dependent tasks, having this in mind, let`s suppose we hold an exam and after the exam, we want to correct the papers and give each one of them a mark, and let`s assume that we have 100 papers and 5 assistants and one supervisor to do the task.

One of the easiest and obvious methods that the supervisor will take ahead of the assistance is splitting the papers just base on alphabetical orders or even student number (we suppose all 100 papers are belong to a same course) in this distribution each assistant will receive 20 papers. All the assistance will begin, and after 10 minutes by having a little luck we suppose that everyone finished 2 papers and accordingly, the supervisor will predict the same rate for the next 10 minutes.

| ***Assistants*** | *A* | *B* | *C* | *D* | *E* |
|---|---|---|---|---|---|
| ***Done*** | 2 | 2 | 2 | 2 | 2 |

Table 1- Assistants and tasks example in first 10 minutes

And luckily we achieved the same rate like the first 10 minutes, as we can see in the *Table-2*.

| ***Assistants*** | *A* | *B* | *C* | *D* | *E* |
|---|---|---|---|---|---|
| ***Done*** | 4 | 4 | 4 | 4 | 4 |

Table 2- Assistants and tasks example in first 20 minutes

The reason is obvious and it`s because all the assistants are fresh and also papers were clear and easy to correct (let`s suppose we got lucky) therefore the supervisor will calculation based on this rate and he will predict that all the tasks should finish after 100 minutes or 1 hour and 40 minutes, but unfortunately it was just a lucky prediction and *Table-3* is showing the real performance that the group of assistants achieved in 100 minutes.

| ***Assistants*** | *A* | *B* | *C* | *D* | *E* |
|---|---|---|---|---|---|
| ***Done*** | 15 | 18 | 20 | 10 | 20 |

Table 3- Assistants and tasks example in first 100 minutes

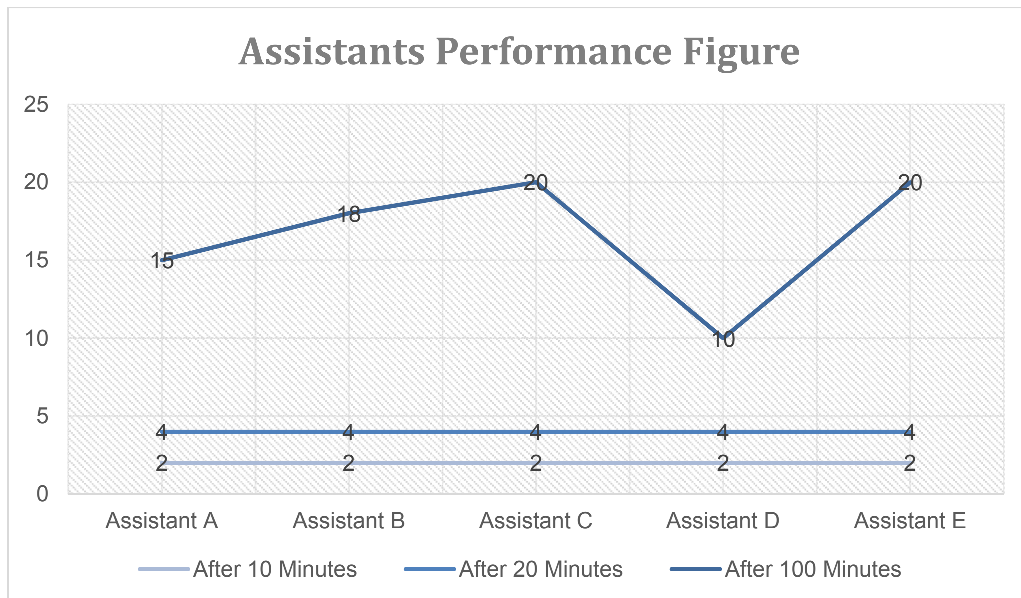


Figure 1- Assistants Performance

As it can be seen in *Figure 1* the assistants group could not continue with the same rate. This is the fact that all supervisors are always scare of, when you are working with human work force, you should always consider performance deficiency during the execution process of any task.

But let`s see what are the possible reasons for assistants A and D that their performance were considerably slower than others.

- Getting tired or other health issues.
- Difficulty with one paper which made that assistant to stuck with that paper.
- The number of hardly solved papers which assigned to them are not equal with other assistants.
- And so on …

Fortunately, human is learning from his experiences and always applying its knowledge to the future tasks (definition of intelligence), therefore supervisors are always monitoring the executers, to locate the performance deficiency in a very early stages, and in our example this knowledge will lead the supervisor to locate the assistant A`s performance issue by monitoring their outcomes regularly, and in the case of any faults or latency they can make the decision to help the assistant to get rid

of the issue or by replacing him/her with a higher capable assistant or other possible solutions.

This smart monitoring and making decisions when it`s needed, require high amount of experience and intelligence which made us able to finish big projects nearly or sometimes exactly on the deadline date. However the challenge is to implementing this as much as it`s possible into the computer systems in order to be able to learn from its mistakes and also to make harmless and on-time decisions.

### 1.3.**Contribution**

[HoG] or Hands of God project`s idea is also based on this issue, which it have been explained in the example above, in a MPI application`s execution, under MPICH framework, if you only replace the assistants with *processors*, and also the supervisor with the *Rank0* which will do the distribution of the tasks (in client server approach) interestingly the same issue will happen, and normally the termination time of the application is unknown and moreover balancing the load equally between the nodes in the cluster became a wish in some cases.

However, HoG will act like an intelligent supervisor by monitoring all the processes and whenever they need help it will find a node with better performance and switch them on the air through a live migration process which will take less than a second in a cluster of nodes, which they joined together via high speed network connections.

## 1.4. Thesis Structure

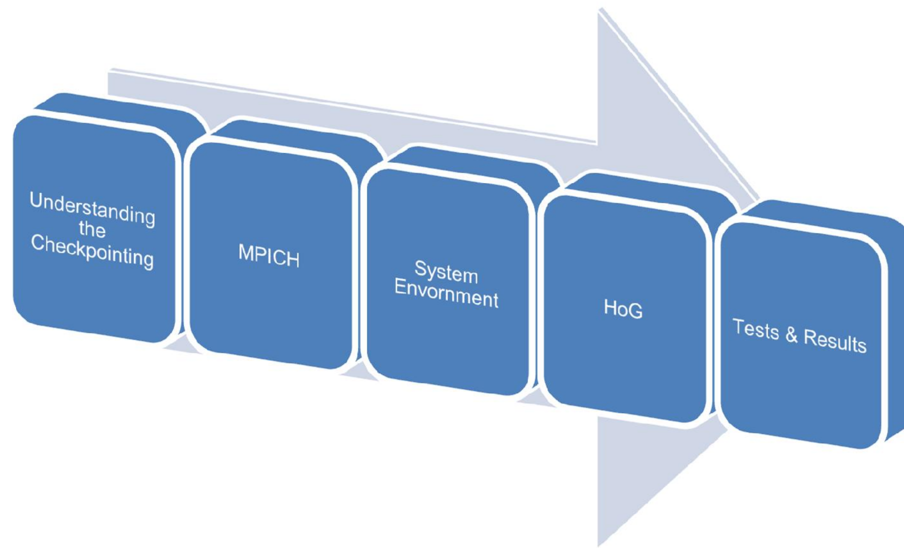


Figure 2- Thesis Structure

In this thesis there will a journey of the creation of *HoG* project, and as it can be seen in *Figure 2*, before everything, there will be a clarification of the Checkpointing definition which is the foundation of the *HoG* Structure. There will be a explanation of Checkpointing in depth in the *Chapter 2* and also there is a quick look on different implementation of Checkpointing that are available until today like *LXC*, *Open VZ*, *DMTCP*, *CRIU*, *CryoPID* and *BLCR*, furthermore there will be an explanation of the reasons that why implementation of *HoG* will be on the top of *BLCR* library.

Having a look at *MPICH* and understand how it works will be the content of *Chapter 3*, also getting familiar with the definitions like *Hydra,* and also how *BLCR* implemented inside *MPICH,* moreover in *Chapter 3,* there will be a brief review of some useful changes between *MPICH2* and *MPICH3* which is related to the *HoG* project.

After having a good understanding of fundamental concepts, in *Chapter 4* there will be an overview on setting our environments and a quick and abstract look into setting up a *HoG* friendly LAB which it will be useful for further improvement on *HoG*.

Through the first 4 chapters preparation has been done and it will be the time for HoG itself, and that will be the context of the *Chapter 5* which also will be the main chapter of this thesis, HoG and its structure can be found in detail in *Chapter 5* and also model of one complete runtime experience of the HoG during the monitoring and making decisions, and surely there will be a nice overview on scenarios which HoG make decisions based on them.

Eventually, HoG will get tested in real environment and also surveying the different tests with the available scenarios and at the end comparing the results of these tests to understand the power of the HoG and also to be aware that in what condition HoG will be beneficial and in which condition it will decrease the performance.

# Chapter 2

# 2. Check Pointing

## 2.1. Introduction

In general, application checkpointing is a mechanism to save the state of a process including it`s CPU and memory`s data, block I/O and also all the network connections (and other data if needed) in order to make a reffering point which it can be restarted that checkpointed application from a specific point. Checkpointing implemented in deferent levels which the following list is showing three major checkpointing levels:

1- Kernel Level CheckPointing
2- User Level CheckPointing
3- Kernel Level CheckPointing & User Level CheckPointing (combined)

### 2.1.1. Backgroud

Checkpointing is a complex mechanism that exist from the MainFrame era, MainFrames aslo were working on a different stages for a process which depend on the load of a process a process can changes it`s processor or changes the working clock of it`s processor, same thing that still it can be found on the normal PC and laptops, A normal PC can also for power saving reason or in the other hands for perfomance issues(*Over Clocking*) increase or decrease it`s own clock but usually, normal PCs and laptops have one physical CPU with one or more cores which they are always working in same CPU clock, therefore chaning processor would not be different for them and it`s only available in Servers infrastructure and if they support multi frequency feature.

After Mainframes, in order to provide an isolated environment in order to support features like resource isolation and alike, FreeBSD jail implemented, which the main reason of creation of FreeBSD jail was simply because of seperation between the private services and customer services by one R&D company considering that ChRoot doing something similar but ChRoot needs superuser access which FreeBSD jail dose not, moreover each jail could have different IP address and also separate configuration and ease of administration.

## 2.2. **Application Checkpointing**

However, checkpointing with the meaning that it have been used in *HoG* project introduced by the name of *Linux Containers* long time ago, which linux containers are kind of resource isolation mechanism, which in each container applications can access all the resources of the Host machine in a safe and private manner, it`s like, deviding a computer system to some smaller substances.

*CheckPointing* and *Restarting* and also *live Migration* were the definitions that introduced with *Linux Containers,* after making a isolated environment as a sunstance of a computer machine our modified and smaller version of the Operation System need to be insttalled, which it will be the OS of our countainer and it will initialize all the devices and hardwares that needed *(simple as a config file)*, like network cards and their type of conection to the outside-network *(because the container is located inside the host therefore other resources in the LAN for example, that that it can used simply from our host machine will be unreachable for the container until the proper connection type and configurations has been defined).*

For instance by defining a network adaptor for the container which using a NAT connection to the Host machine or even to outside networks, afetr setting up the container it can execute our application inside that container which actually what is happening is , a job schedular in the container engin, that working directly with the Host CPU and other resources on the host machine will execute the applications.

It`s look like that our application is having more parent processes which made it more flexible and in the same time more safe. Flexible in the way that it can change the maximum and minimum for memory and disk and network I/O without implementing in the application but only in a config file beside other hardware configurations for the container, and it will be more safe which the working area of our application will be isolated totally and no unpredictable resource interfering issue will happen.

Any time it needed it can pause the container and all the tasks that were running under that cotainer will pause with it, this is a bigger scale of the Checkpointing, actually the container process have been CheckPointed.

Depends on our needs, Checkpointing solution can be efficient for us, but if it wants to freez our application the system administrator needs to consider that, by freezing the container, actually it `s storing too many information that they are not only related to our application and moreover information about the OS of the container.

This problem will solve by using *User Level CheckPointing* which will only work on saving the data that blong to the specific application working area, which have been checkpoinited, this mechanism can be more efficient in order to having a snapshot of a state of an application and keeping the size of checkpointed file as less as possible, also there are some mechanisms which they combined both kernel level and user level checkpointing that this type will be explained more in detail in this chapter.

## 2.3.**Different Checkpointing Implementations**

Until here it have been explained briefly that what is the Checkpointing and the different levels of Checkpointing, now it`s the time to have a quick look on different implementation of Checkpointing, it needs to be mentioned that, there are some windows Checkpointing mechanisms available and also more than these six implementations for Linux, but for understanding the HoG`s mechanism, talking about these six implementations will be enough. These implementations have different usage and actually combined version of them used in the HoG project workflow, but anyway having a quick look on kernel level and user level virtualizations which they are the reason of existing of checkpoint and restart subjects will be useful to understand the combined version of that.

| ***Implementation*** | ***Level*** | ***MPI Support*** |
|---|---|---|
| *Open VZ* | Kernel Level Virtualization | Yes |
| *LXC* | Kernel Level Virtualization | Yes |
| *CryoPID* | Checkpointing Package | Not implemented yet |
| *CRIU* | Checkpointing Software Tool | Not implemented yet |
| *DMTCP* | Transparently Checkpointing Tool | Yes |
| *BLCR* | Hybrid Kernel / User Checkpointing | Yes |

Table 4- Overview of different Checkpointing implementations

### Open VZ

Open VZ is a kernel level virtualization which as same as other kernel level virtualization will offer VEs (Virtual Environments) which it`s coming from its ancestor FreeBSD jail and also look like FreeBSD jail it will create containers which they are substances of the host machine.

### LXC

Another kernel level virtualization implementation is LXC which same as other kernel level virtualizations will isolate resources in order to create PVS (Private Virtual System) and also will offer namespace isolation, therefore applications will have a private view on the process tree and file system and etc… LXC is a combination of the FreeBSD jail and Linux Containers implementation which is recommended for Ubuntu operation system.

**CryoPID**

One of the useful properties of the CryoPID over other Checkpointing implementation is the amount of Checkpointing data, which CryoPID will not only store the application data, CPU, Memory, I/O data but even it will go further and store all dynamic libraries, open files and open sockets which this feature of CryoPID will enable the user to migrate the Checkpointed application to a machine which does not have those libraries, this key feature it can be very useful for some cases that the application require some libraries that the destination system does not have them but obviously MPICH is a framework and supposing that the destination node is MPICH ready, therefore this implementation will only make overhead in the Checkpointing process for the HoG project.

**CRIU**

CRIU is a simple tool for freezing a running application into some files and store them, and then restart the frozen application from the Checkpointed point files, but CRIU is still under development.

**DMTCP**

A great transparent Checkpointing tool which support multi-thread applications and also it`s officially supporting MPI programs it was one of HoG project candidate but BLCR has some features which it could not be found in DMTCP, basically DMTCP is working based on a coordinator that will act as a server and it will listen on a specific port and you should run mpiexec as an argument to DMTCP which make the HoG workflow kind of messy.

For instance if coordinator is not listening or is running on another node there will be some issues for redundancy of too many coordinators plus it`s completely in user level which makes it unstable for enterprise environment.

## 2.4.**BLCR**

After all, it`s the time to go more in detail into the BLCR`s key features, which is the reason that it selected it for the execution of the HoG`s Checkpointing process:

BLCR has so many unique features and some of them listed below (more information about the features of BLCR it can be found in Berkeley LAB Computational research website):

- Fully SMP safe
- Rebuilds the virtual address space and restores registers
- Supports the NPTL implementation of POSIX threads (Linux Threads is no longer supported)
- Restores file descriptors, and state associated with an open file
- Restores signal handlers, signal mask, and pending signals.
- Restores the process ID (PID), thread group ID (TGID), parent process ID (PPID), and process tree to old state.
- Support save and restore of groups of related processes and the pipes that connect them.
- Should work with nearly any x86 or x86_64 Linux system that uses a 2.6 kernel.
- Experimental support is present for PPC, PPC64 and ARM architectures.

But between all these fantastic features one of them is very important for MPI programs which is the *Restores the file descriptors* feature, clearly all the messages are passing between MPI processes via Hydra which this communication method implemented in *HYDU_sock_read()* function which is working with FD or file descriptors, which it will explain the criticality of this feature for HoG.

Beside that the ability of restoring the PID and PPID and GPID which is necessary when your MPI application wants to continue from the Checkpoint file

these information should be exactly same in order to the MPI restore the state correctly.

### 2.4.1. BLCR & HoG

Among all these features some of them are essential for the HoG one of them is the *Architecture free* feature which means that, Checkpoint an MPI application can be done in X86 environment and restarted in X64 environment which is great for the X86 clusters which already have a up and running X86 architecture nodes and by adding some X64 HoG nodes to the cluster it can help the weak nodes in the cluster.

However, it`s crucial that the user didn`t write any code with static pointers and depends on the programming language, the other elements that any developer should consider while they are writing their codes in order to have a architecture free application.

HoG is a monitoring library and it`s working with signals so restoring the pending signals is the another necessary feature that BLCR is greatly supporting it.

The beauty of MPI is the connections between processes via their processors and in some topologies it`s a must that, the connection pipes should stay alive during the execution because MPI will kill the universe, if any connection or process would not respond in the specific timeout which BLCR carefully taking care of these connections.

It will be explained in detail that how MPICH nested BLCR inside itself and also how the BLCR library source code modification has been done to integrate perfectly without any performance loss with HoG, but before that understanding MPICH v2 and v3 and their differences and Hydra definition is necessary which it will be the next Chapter`s topic.

# Chapter 3

# 3. MPICH

## 3.1.MPICH

MPICH is one of the most popular distributed memory implementation of MPI which it`s major goal is to support most of the platforms and also flexibility in dealing with high speed network connections, MPICH offered all these features freely available to public along with its source code.

### 3.1.1. MPICH2 vs. MPICH3

Obviously, version 3 should be in some cases improved and in some cases new, comparing to the version 2, but there are some key features which are important for us in the HoG project that you can see it in bold in the following list:

- Non-blocking Collectives
- Neighbourhood Collectives
- New One-Sided Functions and Semantics
- **New Communicator Creation Functions**
- Fault Tolerance/Resiliency
- MPI Tool interface
- Matched Probe
- Language Bindings
- Large counts

Surely, all of them needs clarification but going to detail on all of them is out of the scope of this thesis, however one of them is highly important for HoG implementation.

**New Communicator Creation Functions**

For independency and also security issues HoG is using a reserved communication world by the name HoG_COMM_WORLD which MPICH3 improved HoG`s performance by implementing a new method for creation a new communication world, in version 3 of MPICH creating a new communication world will happen without involving all the processes in the parent communicator and it`s super-efficient for HoG alike libraries which working in load-balancing or also for fault –tolerance purposes which there is no need any more for global synchronization

and eventually thanks to this new feature, HoG`s impact on the overall performance of MPI program will be as less as possible by creating its new communication world.

### 3.2. **Hydra**

Hydra is a process manager which nested the MPICH and like a pre-processor will execute at the beginning of any MPICH application which called by mpiexec and will set the context ready for a balanced execution, it will begin its job by reading the *machine file* or host file which is a text file that each line of it includes the name of a host in the cluster and following by a colon and a number which is representing the number of processes that supposed to assign to that host.

host1:1
host2:2
host3:1

for instance in the example file above, if there will be *4* processes(mpiexec -n 4) therefore first process will assign to the host1 and then the next 2 processes will assign to the host2 and at the end the last process will assign to the host3.

This is kind of manual assignment but if any number has not been specified the Hydra engine will begin assigning based on round robin algorithm and in our case first process will assign to host1 and accordingly second process to host2 and third one to host3 and forth one to host1, therefore it`s better if static assignment implemented (it means in the user`s code it should be seen if (rank==0) and if (rank==1) and so on …) then there will be a supervised guess that which host should receive more process based on the load that they going to have, but in other cases for instance in client and server structure which rank0 is the server normally (it should be seen if (rank==0) and if (rank>0)) and if the execution flow is unknown, it`s better to leave the assignment to dynamic (round robin) algorithm that will arrange by Hydra by default.

Also Hydra will make it possible to route the traffic of MPICH through a specific network interface which can be a *bridge* or a *virtual interface* (for security or accuracy reason) there are other unique features like process-core binding, X-forwarding and debugging and most importantly Checkpointing and Restarting.

### 3.3. **Integration with BLCR**

MPICH offered Hydra for management of its resources and one of these tasks is Checkpointing and restarting which it have been mentioned in previous chapter, also BLCR has been discussed in detail, but as a reminder it was a hybrid

implementation of the user level and kernel level Checkpointing implementation, further in this chapter the integration between BLCR and MPICH will be discovered.

Hydra is the module which will call the Checkpoint functions that located in the BLCR libraries, as different types of Checkpointing implementation have been surveyed, it`s clear that BLCR is not the only implementation, but it is the only implementation which supported by MPICH, by default BLCR is a third party library which nested perfectly inside the MPICH, and Hydra is the resource manager which also has access to all the machines via SSH or any other launcher specified for Hydra to use, moreover Hydra will create number of proxies (depends on the number of process that defined in the mpiexec arguments) as children of the mpiexec process which everything else will be the children of those proxies, to be more simple if the proxies, are the parent of all processes, mpiexec will be the grand parent of them, anyway the point is that the Hydra can access everything via the proxies therefore it will manage the Checkpointing process by requesting from specified node to checkpoint, and then wait for the result (checkpoint completed) which also it will happen simultaneously with every other things else running in the cluster.

Hydra will organize the Checkpointing process and it has different approaches to do this, which knowing them is necessary. One of them is scheduling the Checkpoint to happen every x seconds which will get done by passing the argument *–ckpoint-interval* to mpiexec and follow that a number of seconds which you wish Hydra checkpoint the whole processes every x seconds that you specified.

Another approach is to checkpoint the MPI application by sending *SIGALRM* to the mpiexec process which signal handler in MPICH will receive it and send a checkpoint request to each node, you should be aware that Checkpointing a process needs to freeze a process for an amount of time despite that this amount of time is less than a second but if you set the interval carelessly it can cause serious impact on the performance, however it crucial to understand that each time that you call the Checkpointing function it will have a storage cost, therefore according to our needs Checkpointing feature should be used, for instance for fault tolerance, when the result is very important and also the execution time predicted to be for a long time, therefore any failure based on OS issue or technical ones like network failure or any other causes may not be acceptable, so in those cases maybe it needed to set the checkpointing interval close to each other or call the Checkpointing function frequently, but the point is have been done deliberately, therefore the consequences are obvious and have been accepted which mostly these consequences are performance related.

However in the source of MPICH under process management folder (*pm) and* in the *hydra* directory *ckpoint* folder can be found which it include two file that listed below:

1- src\pm\hydra\tools\ckpoint\blcr\ckpoint_blcr.c
2- src\pm\hydra\tools\ckpoint\blcr\ckpoint_blcr.h

As the relation between Hydra and BLCR have been explained previously it can be seen that MPICH follow same folder structure. This folder will be the home folder for other Checkpointing implementation in the future but right now it`s include and supports only BLCR which *chapter 5* of this thesis will explain more in detail about the usage method of this library, but before going any further our environments needs to be prepared for HoG`s development, which is the subject of the next chapter.

# Chapter 4

## 4. System Environment

In this chapter there will an abstract tour on how to setup the right environments for developing MPICH and also for future developments on HoG project, it's important to know that MPICH is alternatively available on windows platforms and if you want to develop MPICH in that kind of platforms you should refer to other sources for guide and instructions, but it`s not recommended to use windows for MPICH development purpose because there will be some performance difference and also portability issues later on your project.

**Setting up the Environments**

For HoG project Ubuntu 10.04.4 LTS release 10.04 codename lucid have been used with BLCR version 0.8.2 and MPICH version 3.0.4 released on 24th of April 2013, which further in this chapter it will be explained more in detail about each one of them and how to set them up correctly in your LAB.

### 4.1. Virtual Box

It`s not really important if you choose any virtualization software but it strongly recommended to use the Virtual Box which currently the latest version that is available is Oracle VM Virtual Box 4.3.10 (the latest version of VB is always available in its website) but you can check the website noted in the footnote for the latest version.

### 4.2. Ubuntu

As it mentioned Ubuntu 10.04.4 have been used in the HoG project, which there are some modification and customizations which it should be applied before going any further.

#### 4.2.1. Ubuntu Customizations

*(Step1)* Download and install the latest version of Virtual Box and then create four virtual machines which all the machines except the last one have same configuration as follows:

| *Machine Name* | *CPU* | *Memory* | *HDD* | *LAN* |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| *Node 1* | 1 core | 1024 | 8 GB | Bridged |
| *Node 2* | 1 core | 1024 | 8 GB | Bridged |
| *Node 3* | 1 core | 1024 | 8 GB | Bridged |
| *Node 4* | 4 core | 2048 | 8 GB | Bridged |

Table 5- Virtual Box nodes description

*(Step2)* Download the Ubuntu 10.04.4 ISO file and install on each virtual machine, there is also a better and time saver method to make this cluster which it can be done by creating one virtual machine in Virtual Box and install the Ubuntu 10.04.4 on it and then right click on the created virtual machine and select clone option which will ask you the new name for the future clone version that Virtual Box going to make, and also you should check the select box for re-initialize all the network MAC addresses in order to not facing any network IP conflicts, also be aware that when you clone the forth node before staring the virtual machine you should apply the CPU and memory alterations to it, latter on it will be explained that how is the usage of this node as a super node. Also you should be aware that the following configuration should be done before cloning. In order to clone a configured node the only changes you need to apply after cloning finished is the network setting and changing the host names and resetting the DNS names in the etc/hosts and that will be something like this in Ubuntu:

```
127.0.0.1    localhost
IP Address   node1
IP Address   node2
IP Address   node3
IP Address   node4
```

The next step will be installing SSH and configuring it in order to login from node1 to other nodes with public and private keys and also from node2 to other nodes and so on, for enabling this feature in Ubuntu you should do the following commands in every host:

*ssh-keygen* (it will make RSA public key file)

Then you should transfer the key file to the other nodes:

*ssh-copy-id node1* (it will copy the public key file to node1 and also add node1`s public key to the known host list file in the host which this command will executed from)

After setting up the SSH in all the nodes which make it possible to login from one node to others without any password or other prompts, it`s the time to configure NFS and basically you should assign the role of the server to one of your nodes which logically it`s best practice to assign this role to node1 which will prevent any confusion later in reading the monitoring logs and also in rank assignments, Anyhow after making your decision on that *(step3)*you should install NFS as server on the server node that it will be the node1 in our cluster, and it will be as following:

Server Side:

```
sudo apt-get install nfs-server
sudo mkdir /mirror
echo "/mirror *(rw,sync)" | sudo tee -a /etc/exports
```

Client Side (other nodes except node1):

```
sudo apt-get install nfs-client
sudo mkdir /mirror
sudo mount node1:/mirror /mirror
```

Note: if you wish to mount the mirror permanently which will be valid after reboot add this line to /etc/fstab

```
ub0:/mirror    /mirror    nfs
```

At the end of Ubuntu configuration our OS should be ready for compiling c and C++ codes therefore GCC needs to be installed on it, and its dependencies which in Ubuntu is to install *build-essential* package. Please be aware that there are some other details which if you are interested, it can be found in Ubuntu community by the name MpiClutser.

## 4.3.**BLCR**

There are three ways for preparing BLCR for being operational by MPICH which are:

- Using apt-get and install *blcr-dkms* directly and automatically
- Downloading the source file and compile it locally on each node
- Download the Debian package and install from it

In any method that you wish to install the BLCR you should be aware that while you are installing the BLCR there are some kernel modules which they should install properly otherwise you won`t be able to compile MPICH with BLCR enable option. Installing BLCR is sometimes tricky therefore you can find all the details you going to need in Berkeley Lab website under admin guide section, moreover please do not forget that BLCR version 0.8.2 have been used in HoG project.

## 4.4. **MPICH**

Finally MPICH framework installation time arrived which is very important part of our installation process in this chapter, after downloading MPICH 3.0.4, extract it, and go to the extracted directory which will be by default *mpi* following by the version that you downloaded, there, you can find the *configure script* file which this file will do all the localization that it needed before compiling the MPICH, because C programming language have been used for our project therefore Fortran language should be disabled and obviously blcr should be enabled, then *configure script* will do the rest of it for us, and this will happen by executing *configure script* with these parameters:

./configure --enable-checkpointing --with-hydra-ckpointlib=blcr --disable-fc --disable-f77

After executing the command above, all the necessary actions for localization will get done and simply running *make && make install* installation will be finished, and please remember to test which everything went ok by running *mpiexec –version* which beside giving you the version information of the MPICH and configure options that you used there should be a line like this:

Checkpointing libraries available: blcr

If you found this line, it means that everything is in order and MPICH is ready to run any MPI application with checkpointing feature enabled. And also means that our LAB is ready, you can use any IDE which you prefer to create a project out of the MPICH source file, but please note that any changes on the source file needs *make && make install* on the source of MPICH on each node moreover you need to compile your application with mpicc as well.

# Chapter 5

# 5. HoG

In this chapter HoG library will be explained in detail, in the beginning there will be an overview about HoG and how it`s work by demonstrating an example to understand the idea of the HoG and follow by the details about the HoG`s implementation and its algorithm and the possible scenarios.

## 5.1. Overview

Before going any further, it should be mentioned that a recursive function which will be used in all the examples in order to simulate an intensive and in the same time simple CPU jobs, therefore the source code for this function is as follow:

```
unsigned long long int recursiveFunc(unsigned long long int number)
  {
    if(number==0) return 0;
    else if(number==1) return 1;
    return recursiveFunc(number-1)* recursiveFunc (number-2)+1;
  }
```

This function simply is a recursive function with some calculation which as you can guess it`s a modified version of a Fibonacci function, anyhow a CPU exhausting operation is needed which in the same time it should be very simple in order to trace the performance.

In the first chapter of this thesis, an example has been made, so related to that example but not exactly same, the example will be continued, let's suppose that there is a cluster which in that cluster there are 3 nodes which their hardware specifications are as listed below:

| *Machine Name* | *CPU* | *Memory* | *HDD* | *LAN* |
|---|---|---|---|---|
| *Node 1* | 1 core | 1024 | 8 GB | Bridged |
| *Node 2* | 1 core | 1024 | 8 GB | Bridged |
| *Node 3* | 1 core | 1024 | 8 GB | Bridged |

Table 6- HoG example`s cluster specification

And a MPI application which going to do these steps:

1- In rank 0, the result of recursive function from other ranks will be received.

2- In other ranks, calculation the result of the recursive function

- In rank 1
    - Calculation the recursive function value for number 43
    - Send the result
    - Calculation the recursive function value for number 50
    - Send the result
- In rank 2
    - Calculation the recursive function value for number 43
    - Send the result

This simple MPI program will take about 350 seconds, in our cluster to execute, which by CPU monitoring it can be seen that at the beginning all the nodes are working and after 11 seconds rank 1 will print out the result for the recursive function of 43, and nearly at the same time rank 2 will come up with the same result, but as it programmed, the next step for rank 1 is to calculate recursive function value for 50 which approximately it will take 340 seconds, the reason of this difference is the exponential nature of the recursive function, anyhow our MPI application will finish after 350 seconds which ranks 0 and 2 were waiting 340 seconds for the rank 1 to finish and that`s what exactly should happen because it programmed to do so.

This behaviour could happen also without our demand, like *Snake in the Box* algorithm which it`s unknown that, after how long and also which nodes will reach the termination faster and which nodes will stuck in recursive calculation, so in previous example simulated situation just happened, which in that situation, some processes stuck in the recursive calculations while others are terminated or waiting for an answer from the busy nodes to continue. In such an unpredictable situation, there is no any other choices else than waiting for those busy nodes to finish.

Another issue is if blocking methods have been used like normal MPI send and MPI receive, from observer view, that node is working but actually is actively waiting.

### 5.1.1. Actively waiting

Actively waiting is a condition which in that, a node does not have any logical calculation to do and is waiting for next operation but the issue is blocking method have been used which it will block the process and put it in a loop of checking the status, it`s like those nodes are asking every nanosecond: *what should I do?* So asking this frequent will make the process so busy which is a fake busy status, but this fact is hidden from the eyes of the observer until you trace the system calls of that process, then you can see that that process is really working or only waiting for an event.

Eventually, for solving this issue a method for monitoring MPI application have been designed which is working transparently, whenever HoG detects that, MPI application stuck in a situation which it needs help, like the example have been discussed previously, HoG will look for a super node and switch the busy node with the supper node, therefore super node will resume the task of the busy node and like that, increase in overall performance will be reached, up to 40% and surprisingly all the migration tasks will take place in less than a second.

Too many researches get done by other groups on the checkpointing mechanisms related projects but because of the nature of the checkpointing which designed for keeping the state of the processes, approximately all of the researches have been done in fault tolerance area and HoG is the first implementation of checkpointing & restarting in the dynamic load balancing that by achieving, up to 40% performance improvement it can be claimed that it was a success for load balancing. Moreover it`s quite assuring that HoG will be a milestone in the designing of the parallel clusters and MPI programming.

## 5.2.**Structure**

Monitoring is a tricky business, which if the developer design it carelessly it can impact the performance hugely and also it can cause locks on some resources, therefore all the performance effective factors are carefully considered as well as the locking aspects in order to design a fully independent and transparent monitoring library which it has the least possible performance impact, which is approximately less than 1% and to be exact 0.2 % in the overall time of the execution.

### 5.2.1. Transparency

After all process created, in all proxies by mpiexec, it`s time to execute the user code which in the MPICH program you have to begin like this:

```
int rank,p;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Status status;
```

Which means that the user will define the *rank* and *processes number (p)* variables and then assign their values by using *MPI_Comm_rank* and *MPI_Comm_size* functions and all these assignments must happen after *MPI_Init.* What will happen is each node will run the same code and therefore they will assign their own values to the *rank* and because *p* is same in all of them, same value will assign to the *p*, but in different machine and different memory location. Anyhow this is the way user will begin a MPI program but if you wonder to activate HoG, to monitor your application you need only to call it once after MPI default declaration like this:

```
int rank,p;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Status status
MPI_HoG();
```

This single line is enough to wakes up the HoG library to join to the cluster and to begin it works.

Like other MPI commands HoG will create a new private thread as a child of MPI application on each node and the only common thing that it does have is it`s dead time, which whenever the MPI application dies HoG thread will die with it as well.

### 5.2.2. Resource safety

Woking with MPI application while it`s working will be performance consuming which it have been avoided by defining unique communication world by the name *HoG_COMM_WORLD* in the MPI universe, this communication world is unique because it defined just after *MPI_Init* which will prevent any duplication in the user area, in another word if the user wonder to create same name for communication world name, MPICH will return an error which this

communication world already exist moreover any joining to this communication world have been prevented, therefore this world will be exclusively dedicated to HoG library and no other world can join and also no user can make a communication name by this name.

The *HoG_COMM_WORLD* is the copy of the *COMM_WORLD* which is the parent of all other worlds in the MPI universe, thanks to the MPICH3 which by creating a new or a copy of an existing communication world does not needs a global synchronization, therefore MPICH3 guaranty that creating HoG communication world would not have any impact on the overall MPI application performance.

Therefore, HoG will begin its job parallel with MPI application and within a secure and private communication world, but this feature will only guaranty the memory and process area to be completely private and dealing with the monitoring issue is still remaining which is quite tricky, after doing too many tests and developments it have been discovered that the best way of monitoring a process even when HoG is a child of that process is to work with the kernel and watch the system calls and events and monitoring statics which kernel will produce by default, therefore the structure will be like this:

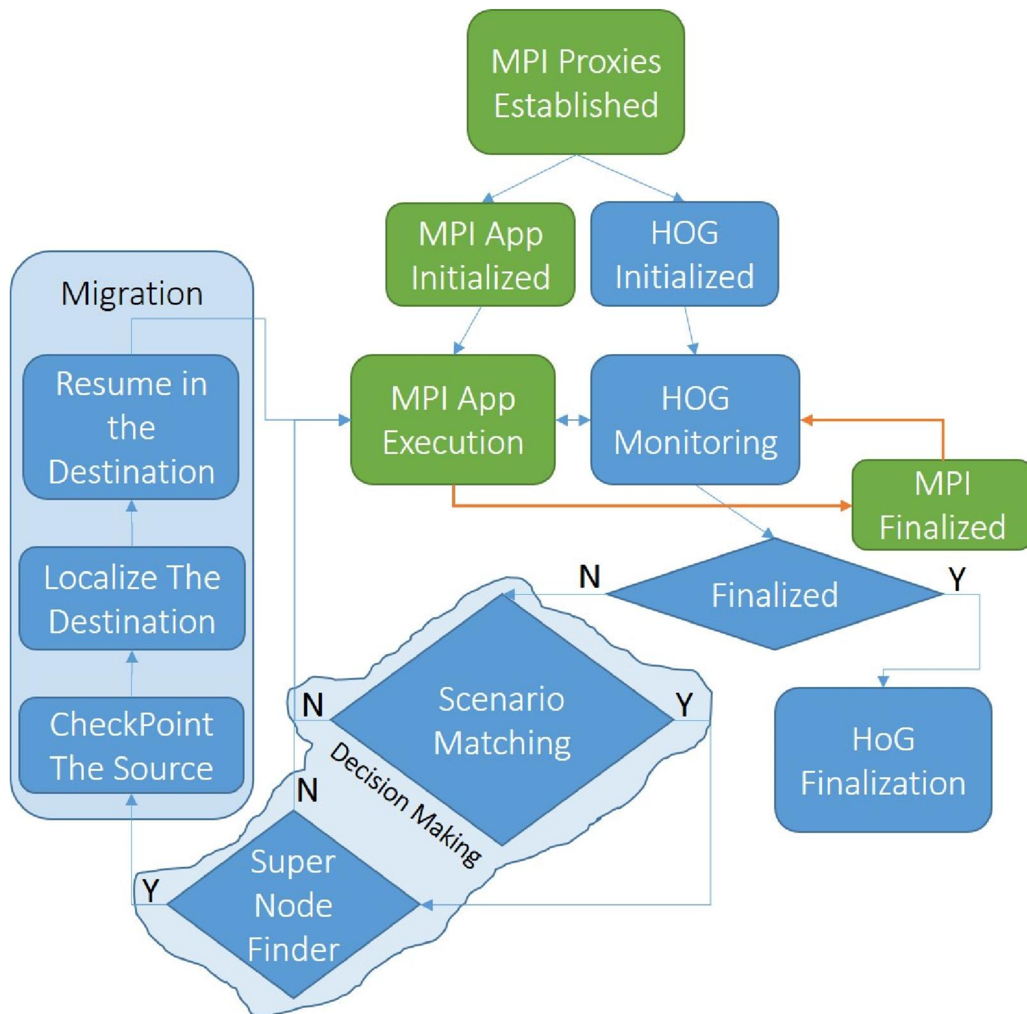

Figure 3- HoG Abstract Workflow

As *Figure 3* shows, the steps in a HoG execution will be as listed below:

1- MPI proxies will establish.
2- MPI application will initialize.
3- HoG will initialize.
4- MPI application will begin executing under the OS kernel.
5- HoG will receive all the statics, events and system calls which MPI application will generate except the ones it`s belong to HoG itself from OS kernel and also accumulating those statics for making the decision.

6- In the meantime HoG will make the decisions based on its matched scenarios and it will look for the super node.
7- In the case that the HoG find a super node, it will migrate the busy process from the weak node to the super node, and if not just will skip migration.
8- Go to step 5.

And that was an abstract and simplified version of the structure of the HoG library, undoubtedly there will be in detail discovering on each step which mentioned above further in this chapter in the HoG Workflow`s section. But right now a tour to the workflow of MPICH is needed, in order to have a better understanding of how the MPICH nested HoG inside it, which is the subject of our next section.

## 5.3. MPICH Workflow

MPICH is the nest of HoG library and also the MPI user application, therefore knowing that how does it works is crucial, and please consider this matter that what will be explained is an abstract of the MPI execution steps and in other word the MPICH workflow and going too much in detail for these steps is out of the subject of this thesis but if you are interested to learn more in detail it will be recommended to have a look at MPICH source code which well commented and by an IDE you can discover the steps in details. But for now an abstract list of MPICH workflow can be found as follow:

1- Initializing the demux engine
2- Initializing the bootstrap server
3- Creating the node list
4- Create a pipe connection
5- Launch the processes
6- Wait for their completion
7- Wait for exit status of all processes
8- Finalize and free the resources

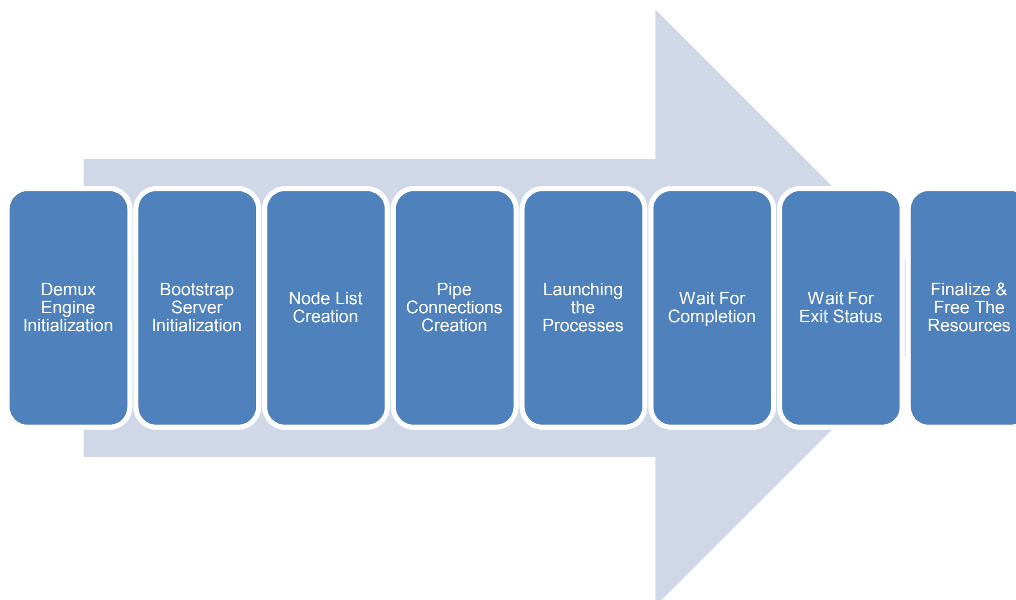


Figure 4- MPICH Abstract Workflow

So also as it can be seen in *Figure 4*, it`s not that complex that it may sounds, but this is the most abstract version of MPICH workflow, anyhow the important step for us is the step 6 which the MPICH is waiting for the processes to return their completion status, and that is the time that HoG will come to the story and begin its monitoring iterations. The next step is what will happen when an MPI application will be checkpointed.

### 5.3.1. MPICH Workflow in Checkpointing

There are two different approaches for checkpointing a MPI application, which not both of them are our concern in this project but according to the checkpointing mechanism which is same for both approaches and only the way of calling the checkpointing function will differ, however both will be discussed in abstract.

Basically and as been mentioned before, there are two approaches for initializing checkpointing process, the first one called checkpointing by intervals which does not need any modification in the user code and it will initialize at the beginning of the execution by passing an argument *–ckpoint-interval*, surely you need to pass the library name that you wish to use for checkpointing and also the

checkpointing prefix plus the address, which you want to store the Checkpoint files, the mechanism is simple, for fault tolerance you will set the arguments for mpiexec to checkpoint the MPI application every x seconds by using the specified library which is by default is the BLCR library, and also you should clarify the path that you wish to save the files, and the only important issue you should consider is the intervals, which if you set it very close to each either that they can interfere with each other, you will receive an error that *previous checkpoint did not finished* which in order to get rid of this issue and if you really need to checkpoint your application very often you should set *MPICH_ASYNC_PROGRESS* variable *equal to 1* in your environment variable which will get done by *export* command in Ubuntu.

In the second method you should send the *SIGALRM* signal to the mpiexec process, which will get done by *kill –s SIGALRM shell command* to the PID of the mpiexec in Ubuntu, but anyway in both cases same thing will happen in the MPICH, when you use the interval mechanism or when you send the signal manually, same function will initialize and that function is *HYDT_ckpoint_blcr_checkpoint* in BLCR library, now let`s see that what this function is doing.

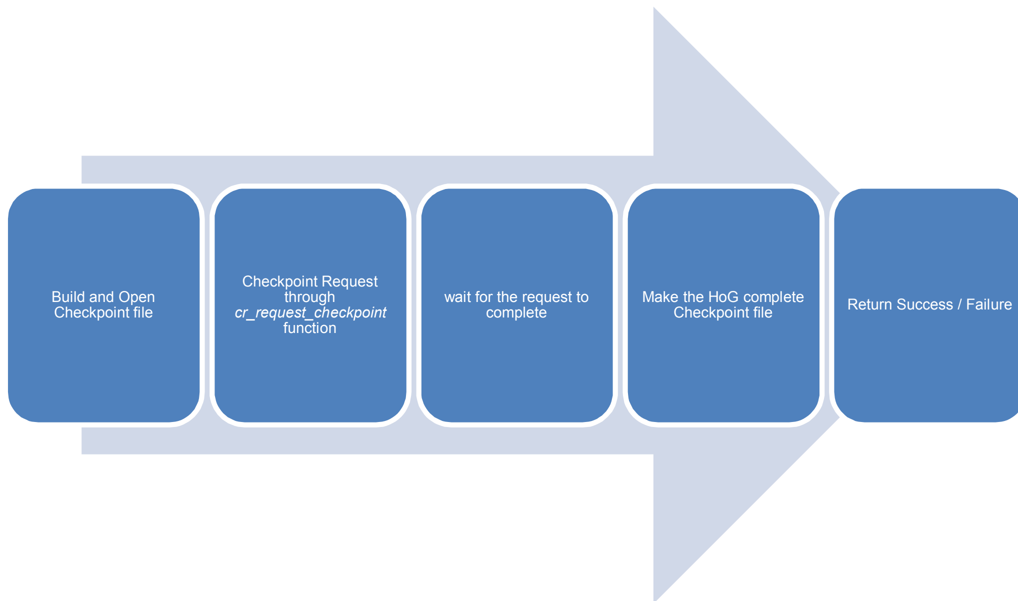


Figure 5- Checkpointing Workflow

As *Figure 5* also shows, the steps which will be done if you initialize checkpointing are as follows:

1- build the checkpoint filename and open it
2- issue the checkpoint request by calling *cr_request_checkpoint* function
3- wait for the request to complete
4- make the HoG complete Checkpoint file (HoGckpoint-theRankNumber)

Step one and three are clear enough but the second step, actually will happen out of the MPICH, as been explained previously about BLCR, it clear that the BLCR is hybrid implementation of Checkpointing which means that, part of it, will happen in the kernel which means it can be called everywhere in the code and by calling *cr_request_checkpoint* the processes will be paused for a fraction of a second and store all the states and CPU data, memory data, socket data and also all the open file descriptors which are related to our parent process which here is mpiexec and store it in a binary file as the Checkpoint files, which for each node there will have one file which by default named by this format: *context-num-(ckpoint number)-(pgid)-(rank)*

### 5.3.2. MPICH Workflow in Restarting

In a successful, restart everything is same with a normal execution of the MPI application except the first step which in the mpiexec.c every execution MPICH is checking if the execution is from a point zero or from a checkpointed point, which this check will lead the MPI application to recreate all the process and their connections and all other related resources deep inside the kernel as you can see in the step three of the following steps:

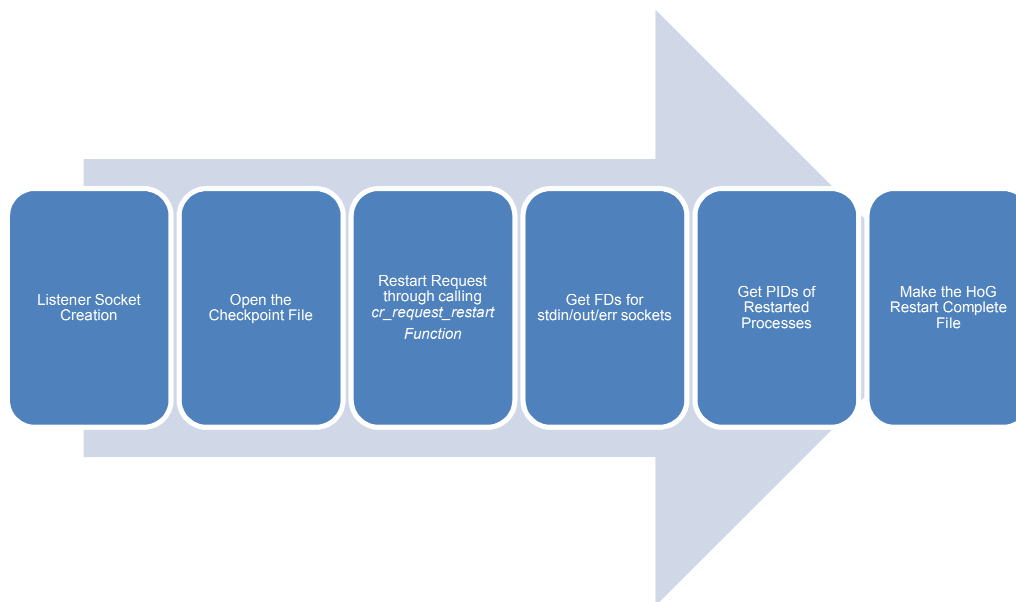


Figure 6- Checkpoint Restart Workflow

As Figure 6 also shows, the steps which will be done in a restart process are as follow:

1- create listener socket for stdin/out/err
2- open the checkpoint file
3- issue the request by calling *cr_request_restart* function
4- get FDs for stdin/out/err sockets, and get PIDs of restarted processes
5- make the HoG restart complete file (HoGckpoint-theRankNumber)

As it can be seen there are two extra steps comparing to checkpointing process, step1 and step4 which it`s because of localization issues, it`s clear that each time a process will be run, the kernel will assign a new PID and if file descriptors have been used, each time that our application have been run there will a new numbers for our file descriptors, this phenomenon must happen for processes and resources privacy to make sure that only one process with a specific PID can be found which makes BLCR to do the step1 and step4, when it`s waking up the frozen application, the kernel anyhow will see it as a new process which accordingly will assign a new PID to it and same for the other unique recourses therefore the mpiexec process will continue its work with new identity from the kernel view but same identity from the MPI user.

## 5.4.**HoG Workflow**

As promised previously in this chapter there will a detailed explanation of the workflow of HoG library, as has been mentioned previously:

1- MPI proxies will establish.
2- MPI application will initialize.
3- HoG will initialize.
4- MPI application will begin executing under the OS kernel.
5- HoG will receive all the statics, events and system calls which MPI application will generate except the ones it`s belong to HoG itself from OS kernel and also accumulating those statics for making the decision.
6- In the meantime HoG will make the decisions based on its matched scenarios and it will look for the super node.
7- In the case that the HoG find a super node, it will migrate the busy process from the weak node to the super node, and if not just will skip migration.
8- Go to step 4.

So in this section it will be discovered each step more in detail and also there are some more steps which it will be added accordingly. It`s clear that how MPICH will begin an MPI application then obviously the step1 and two will be skipped.

**HoG initialization**

After defining the parent function which is *MPI_HoG()* it will create an instance of our monitoring thread which have been named *processInfoThread* which is the parent of all other functions after *MPI_HoG()* method.

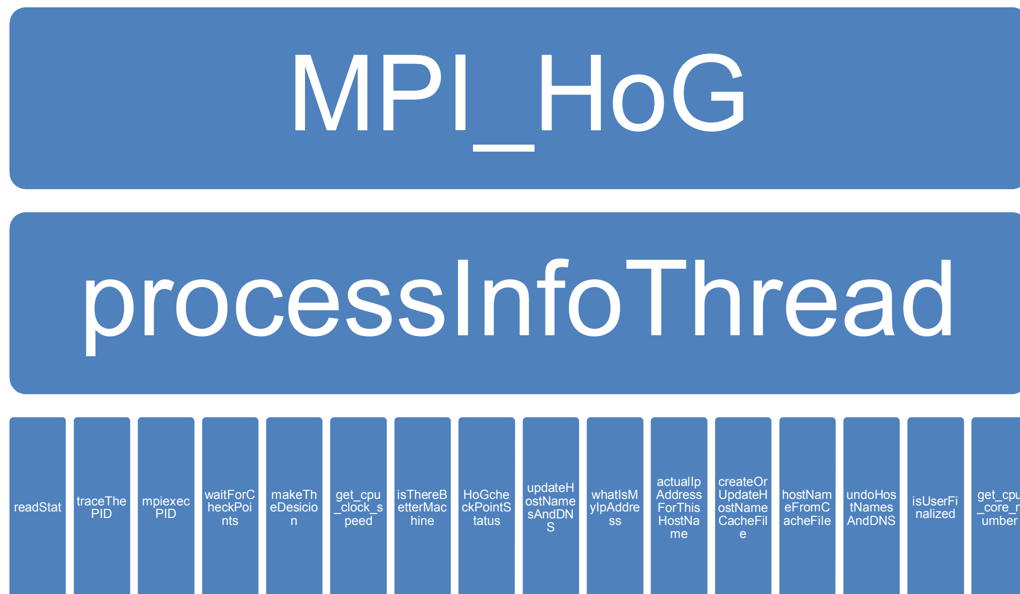


Figure 7- MPI HoG Hierarchy

As *Figure 7* shows the hierarchy of HoG`s methods. This pyramid view will be more straightforward to understand the relationships between the functions, and in the same time it will be a good viewpoint for debugging aims.

For initialization, the communication world will be duplicated into the unique *HoG_COMM_WORLD* and it will find out and store the hostnames and also IP addresses which would not change during the monitoring, and it will be done by requesting these information via a SSH socket which established dedicatedly for HoG.

### 5.4.1. HoG Workflow before Checkpointing

After our communication world initialized and constant data has been discovered from all the nodes in the running cluster, each node will begin to monitor itself for 10 seconds and at the end of 10 seconds it will send the monitored data to the server node, server node which mostly is rank0 will accumulate these data and after 20 seconds will examine the data by taking the average of received data and send these data with other info about nodes like their rank and IP addresses to decision maker method, this method will begin to analyse the data and match them with defined scenarios if possible, in the case of valid match it will take the proper

action accordingly which it can be continue monitoring for more time or migrating the process node to another node.

### 5.4.2. HoG Workflow after Checkpointing

There is an important part of the HoG functionality which is the line that the application has been checkpointed, the reason that made this line interesting is the behaviour of the application after it checkpointed and restarted, for clarification it will be explained with an example, let`s assume you developed an application which on line 20 the process will exit and when you run the application again it will continue from line 21, therefore you should be prepare to recognize that the execution is from the beginning or it a resume version, for this issue that exactly will happen when checkpointing decision has been made, in the MPI application a mechanism for recognizing this phenomenon have been designed which immediately after checkpointing line checking will be done to find out that application is still in the same process or it`s a restarted version of the same process that was running before, and this will happen by calling *HoGcheckPointStatus* function which will return 2 on restarted application and 1 on just checkpointed application.

In the case that the application just checkpointed or the *HoGcheckPointStatus* function returned 1 there will be a chance for BLCR checkpoint library to finish its job by calling the function *waitForCheckPoints method* and then the nodes identity will be changed by calling *updateHostNamesAndDNS* function according to the scenarios decisions and at the application will be killed when the restart preparation have been completed.

### 5.4.3. HoG Server Application

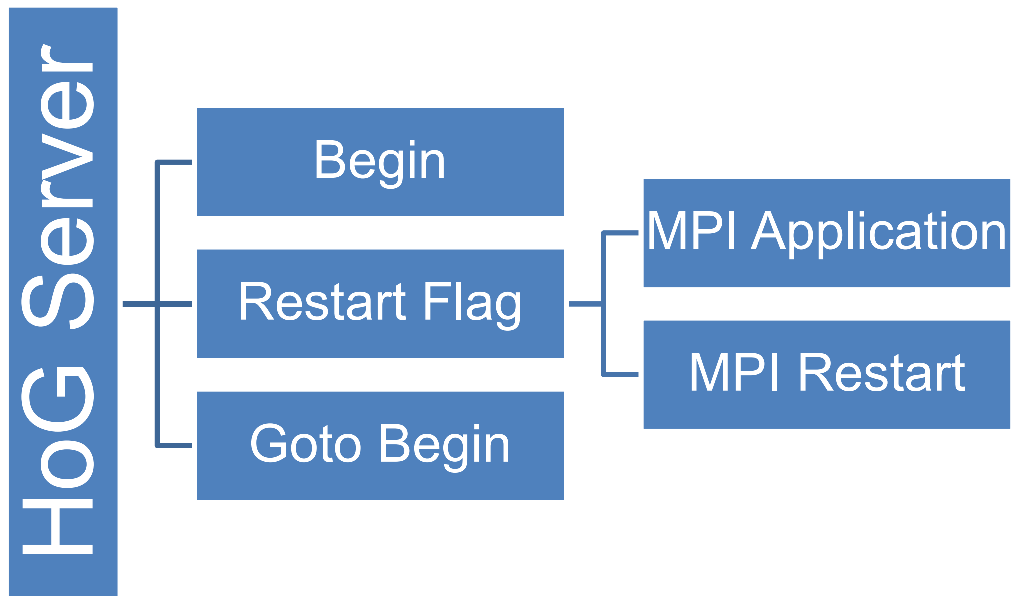


Figure 8- HoG Server Structure

According to the decision that has been made, and also by having in mind that the application has been killed at the end of previous section you should have this question that who wants to wake up us again from the checkpointed point. As it can be seen in the *Figure 8*, this is the duty of HoG Server application which is a simple caller application written in C, which will execute the mpiexec with its required argument and after that it will wait for the checkpoint files and if they exist it will restart and continue restarting until there is no checkpoint file in the specified path, anyhow for clarification the steps shown also in the figure-8 as well as the list below:

- Step1-Start the timer
- Step2-Running the fresh mpiexec command
- Step3-Look for the checkpoint files related to executed mpiexec
    - Found: Restart the MPI application with founded checkpoint file
        - Return to step3

- Not found: Terminate the MPI application with all their children and record the time.

Abstractly, the HoG server will execute the main application and then if there is any checkpoint file when the main application killed, therefore it will restart it from that checkpointed point and then continue restarting until main application exit without any checkpoint files, the point that should be known about the timing that logged in the HoG server is this, despite of all the timings which happening in the HoG itself, there is a different timing mechanism which will record the net time that the executable began executing in the kernel until the time which the same executable`s process have been killed.

#### 5.4.4. HoG Workflow after Restarting

After HoG server restart MPI application from the checkpointed files, execution pointer will come back to the exact line after checkpointing line in the HoG application and this time same *HoGcheckPointStatus* function will execute which this time it will return 2, which will translate as a restarted application, therefore the counters will be restarted and monitoring will be continued, because same behaviour should happen if a match with our scenarios have been found again.

### 5.5. Single Node Switch Scenarios

Scenarios are some patterns which defined in the HoG application that each one of them will describes a specific situation of nodes and accordingly in that situation there will a decision making, our first scenario called *single node switch* which as the name explains it`s a situation which in that condition node A `s MPI process will transfer to node B but there are some condition which will specify eligibility to make this decision which it will be discovered in the next two section.

### 5.5.1. All Waiting for One

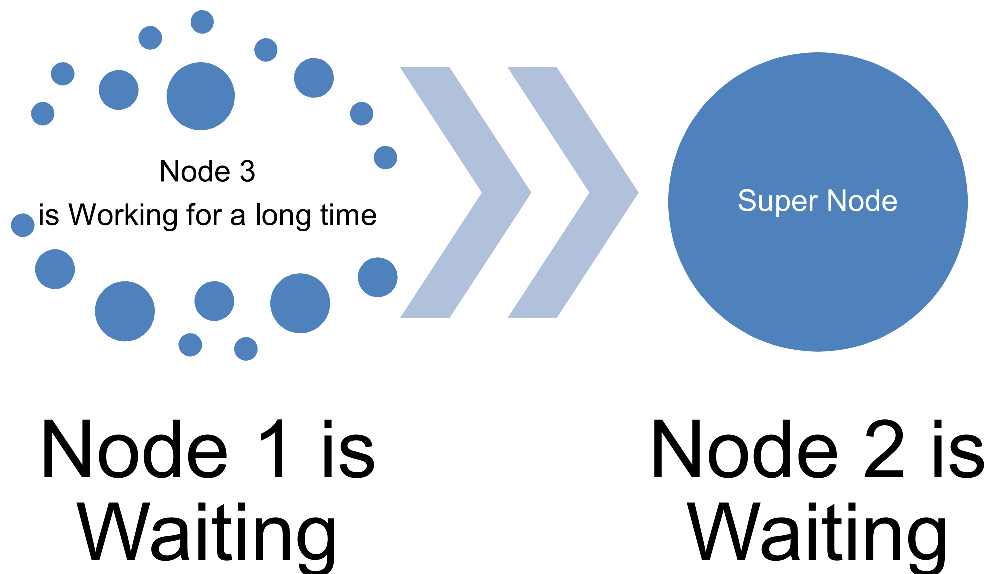


Figure 9- Scenario 1, All Waiting for One

*Figure 9* is showing an abstract view of *All Waiting for One* scenario, and the following will be discovering All Waiting for One scenario in detail.

In the HoG implementation there is a function that have been called *traceThePID* which its job is to receive a PID as an argument and trace the system calls between that PID`s process and the kernel and return 0 if the process that own that PID according to MPI tasks is working or return 1 which means that process is waiting, the important note here is the mechanism of this function which it would not only count on CPU usage or CPU activity of that process but also analyse the deep conversation of that process with kernel which will result to an interesting status for each process which is *MPI_Running* or *MPI_Waiting* which even if the user used blocking mechanism of send or receive which will keep the CPU in 100% usage level or non-blocking mechanism, in both conditions the real status of that process will be determined.

Having this in mind, first scenario is now ready be explained, which is a*ll waiting for one*, As the name explain, it`s the condition which all the nodes are waiting for one node, lets clarify it a little, there are situations which all the nodes

already done their tasks or they need an answer from one node, let`s suppose that there are three nodes and two of them are already finished their tasks and they are actively waiting (blocking method) for the node three and node three has lots of work to do, in this situation a matching pattern have been detected for scenario number one which is *all waiting for one* therefore HoG will store this state and wait for 40 seconds (it can be less or greater that this value but in the first release 40 seconds has been set) which is 2 cycle 4 iteration of monitoring and in the third iteration HoG will looks for a super node.

**Super Node**

Super node is a node which has a considerable better performance comparing to the other nodes of the cluster and it can be part of MPI application or just stay as passive node, but in the case of being passive you should define it in the machine file anyway, there are some key features in marking a node as a super node which listed below:

- Better CPU clock
    - In equal CPU clocks CPU with better cache will win
- If majority of cluster are single core the node with more cores will win

There are more features which it can be added to the *isThereBetterMachine* function which decide which node is a super node at a specific moment.

In the case that HoG find better machine according to the super node rules, it will prepare the busy node and super node for the migration, this operation can happen with different mechanism as follow:

- Nat the source and destination busy and super node
- Modifying the DNS and hostnames

The first method is under development right now, but the second method which is working by modifying the DNS and hostnames of the busy and super node already developed and tested completely, the only issue with this method is if other application are executing on the same time on these two machine and also if they are reading the hostname or using DNS and not caching it at the beginning of the execution will have a confusion therefore this method is not the best approach and working on the first approach is in progress to reach the safest and most secure method to changing the identity of these nodes temporary. Also changing the information of the hosts in the checkpoint file is an option which could be an alternative to the first mechanism.

However, in the case of finding the super node, the process on the busy node will migrate to the super node but otherwise if HoG would not find the super node the MPI application will continue and HoG will continue monitoring and looking for a super node.

The process will be as follow:

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Waiting |
| *Node 3* | 2 | Running |
| *Super Node* | | |

Table 7- Before HoG do the migration

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Waiting |
| *Super Node* | 2 | Running |
| *Node 3* | | |

Table 8- After HoG did the migration

The reason that HoG would not finish the monitoring when all the nodes are equal in the CPU clock is fluctuations in CPU clock of different nodes which depending on their load it can change and there is a possibility which in further iterations HoG would find a super node to migrate.

### 5.5.2. All Waiting For Two

In the previous section the situation that how to deal with all the nodes that are waiting for one node in the cluster have been discussed and also an example about 3 nodes which two of them finished their jobs and all the other nodes except node 3, were waiting for the third node has been explained, nearly same scenario will happen here but with a little difference which make the HoG even more useful.

let`s suppose that there are 4 nodes which two of them finished their jobs and waiting for other two nodes, it means node1 and node 2 finished their jobs or they are in a state which are waiting for a value from node 3 or node 4 or both, therefore there will be 2 busy and 2 waiting nodes, also let`s assume that both busy nodes are equally consuming the CPU and other resources, therefore HoG will look for a super node in the cluster, and in the case that it found the super node it will

prepare the busy node which in this case based on simple sorting is node 3 (because they are same in load in this example) and super node for migration and continue the execution as follow:

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Waiting |
| *Node 3* | 2 | Running |
| *Node 4* | 3 | Running |
| *Super Node* | | |

Table 9- Before HoG do the migration

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Waiting |
| *Super Node* | 2 | Running |
| *Node 4* | 3 | Running |
| *Node 3* | | |

Table 10- After HoG did the first migration

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Waiting |
| *Node 3* | 2 | Waiting |
| *Super Node* | 3 | Running |
| *Node 4* | | |

Table 11- After HoG did the second migration

As it can be seen in the table 9, that the situation that have been explained just happened and HoG made the migration in the Table-10 for node 3 and the owner of the rank 2 is the super node, but after this node also finished its task and stepped into *Waiting* state HoG will detect the first scenario which is *All Waiting for One* and it will give the rank 2 (which is waiting right now) back to its original owner and switch the rank 3 `s owner with a super node until all the tasks finish.

# Chapter 6

# 6. Tests and Experiments

Finally, everything about HoG and MPICH framework have been thought and also their workflows have been discovered enough in the last 5 chapters, now it`s the result time and to discovering the performance that it can be achieved with the HoG, in the first section there will be a definition of the test cases which going to be used in testing the HoG, afterward performance results will be revealed and at the end the result compassion will be performed, eventually the useful and useless usage of the HoG will be discovered.

## 6.1. **Test Cases**

However, it necessary to mention that in all the tests the same function that have been explained in chapter 5 section 5.1 the *recursiveFunc* will be used, which is perfect for our purpose in order to be simple as possible and also CPU intensive, by its recursive nature and simple calculation.

Our test cases will cover only the first scenario which was *all waiting for one*, and the reason is that, the other scenarios are under development right now therefore you can contact us anytime later for further test results on other scenarios.

### 6.1.1. Test Cases Structure

The last step, before defining our test cases is to discuss more about the structure of our test cases, which will be defined in the following list. Please be noted that all of the test cases will have the same structure as it have been listed below:

1- MPI application: which will be the user code that it will be used and actually will run inside the MPI processes
2- HoG status
3- Scenario Name
4- Cluster Structure: which will be the hardware configuration of the nodes in the cluster.
5- Description

**Test Case 1**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
    - In rank1: calculate the recursive function of 43 and send the result
    - In rank1: calculate the recursive function of 50 and send the result
    - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| *Node* | *#CPU* | *MHz* | *L1d* | *RAM* | *OS* |
|---|---|---|---|---|---|
| *Node 1* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3** | 4 | 3099 | 16 K | 2048 | Ubuntu 10.04 lucid |

Table 123- Cluster structure of test case 1

5-Description

**Test Case 2**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
    - In rank1: calculate the recursive function of 43 and send the result
    - In rank1: calculate the recursive function of 50 and send the result
    - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Not-Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| ***Node*** | ***#CPU*** | ***MHz*** | ***L1d*** | ***RAM*** | ***OS*** |
|---|---|---|---|---|---|
| *Node 1* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |

Table 13- Cluster structure of test case 2

5-Description

**Test Case 3**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
    - In rank1: calculate the recursive function of 43 and send the result
    - In rank1: calculate the recursive function of 50 and send the result
    - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| ***Node*** | ***#CPU*** | ***MHz*** | ***L1d*** | ***RAM*** | ***OS*** |
|---|---|---|---|---|---|
| *Node 1* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3** | 4 | 1589 | 32 K | 2048 | Ubuntu 10.04 lucid |

Table 14- Cluster structure of test case 3

5-Description

**Test Case 4**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
  - In rank1: calculate the recursive function of 43 and send the result
  - In rank1: calculate the recursive function of 50 and send the result
  - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Not-Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| ***Node*** | ***#CPU*** | ***MHz*** | ***L1d*** | ***RAM*** | ***OS*** |
|---|---|---|---|---|---|
| *Node 1* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 1 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3** | 4 | 1589 | 32 K | 2048 | Ubuntu 10.04 lucid |

Table 15- Cluster structure of test case 4

5-Description

**Test Case 5**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
    - In rank1: calculate the recursive function of 43 and send the result
    - In rank1: calculate the recursive function of 50 and send the result
    - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| ***Node*** | ***#CPU*** | ***MHz*** | ***L1d*** | ***RAM*** | ***OS*** |
|---|---|---|---|---|---|
| *Node 1* | 2 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 2 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3** | 4 | 3099 | 16 K | 2048 | Ubuntu 10.04 lucid |

Table 16- Cluster structure of test case 5

5-Description

While there is another MPI program is running

**Test Case 6**

1-MPI application:

- MPI initialization
- Receive the value of recursive function in rank0 from rank1 and rank2
- Parallel:
    - In rank1: calculate the recursive function of 43 and send the result
    - In rank1: calculate the recursive function of 50 and send the result
    - In rank2: calculate the recursive function of 43 and send the result

2-HoG status

Not-Active

3-Scenario Name

All Waiting For One

4-Cluster Structure:

| ***Node*** | ***#CPU*** | ***MHz*** | ***L1d*** | ***RAM*** | ***OS*** |
|---|---|---|---|---|---|
| *Node 1* | 2 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 2* | 2 | 2272 | 32 K | 1024 | Ubuntu 10.04 lucid |
| *Node 3** | 4 | 3099 | 16 K | 2048 | Ubuntu 10.04 lucid |

Table 17- Cluster structure of test case 6

5-Description

While there is another MPI program is running

## 6.2. **Performance Results**

After defining the test cases, it`s the time to run our test cases one by one and see their results, therefore in this section the execution result of the test cases will be presented, and for the beginning the test case 1 will be executed.

Please note that, the first two cases `s results are nearly complete, but after that because of the similarity in most lines writing the complete result text after test 2 have been exempted and you can find only the execution time for test 3 and after that.

**Result for Test Case 1**

HoGServer-> HoG Server started.

User Output-> Rank 1 : fibo for 43 is : 5364436878204150527

HoG-> Warning! this is 1 th time that All the processes are waiting for Rank 1

HoG-> Warning! this is 2 th time that All the processes are waiting for Rank 1

HoG-> Warning! this is 3 th time that All the processes are waiting for Rank 1

Making decision ...

Looking for Faster node to transfer ...

HoG-> Found! Rank 2 `s node has CPU clock: 3099 which is better than Rank 1 `s node with clock 2284

HoG-> Note: in the virtual environment it`s not possible to have different CPU clock thats why they can be equal but still we suppose one of them is better!

HoG-> Until checkpoint took about 68 seconds.

[proxy:0:0@n5] requesting checkpoint

[proxy:0:0@n5] checkpoint completed

[proxy:0:1@n6] requesting checkpoint

[proxy:0:1@n6] checkpoint completed

[proxy:0:2@n8] requesting checkpoint

[proxy:0:2@n8] checkpoint completed

HoG-> CheckPoint Num: 1 Has been Completed

HoG-> Preparing rank 1 for live migration from 192.168.0.34 to 192.168.0.41 ...

HoG-> Preparing Done!

HoG-> Restarting ...

Killed

HoGServer-> Original Running finished in about 76 seconds.

HoG-> CheckPoint Num: 1 Restarted

HoG-> Warning! this is 1 th time that All the processes are waiting for Rank 1

HoG-> Warning! this is 2 th time that All the processes are waiting for Rank 1

HoG-> Warning! this is 3 th time that All the processes are waiting for Rank 1

Making decision ...

Looking for Faster node to transfer ...

HoG-> NOT Found! Rank 1 `s node has CPU clock: 3099 which is best clock in the cluster!

HoG-> Pause the monitoring! (no better node)

HoG-> Resume the monitoring!

HoG-> Warning! this is 3 th time that All the processes are waiting for Rank 1

Making decision ...

Looking for Faster node to transfer ...

HoG-> NOT Found! Rank 1 `s node has CPU clock: 3099 which is best clock in the cluster!

HoG-> Pause the monitoring! (no better node)

HoG-> Resume the monitoring!

HoG-> Warning! this is 3 th time that All the processes are waiting for Rank 1

Making decision ...

Looking for Faster node to transfer ...

HoG-> NOT Found! Rank 1 `s node has CPU clock: 3099 which is best clock in the cluster!

HoG-> Pause the monitoring! (no better node)

HoG-> Resume the monitoring!

HoG-> Warning! this is 3 th time that All the processes are waiting for Rank 1

Making decision ...

Looking for Faster node to transfer ...

HoG-> NOT Found! Rank 1 `s node has CPU clock: 3099 which is best clock in the cluster!

HoG-> Pause the monitoring! (no better node)

User Output-> Again Rank 1 : fibo for 50 is : 5442052724786215423

User Output-> Rank 2 : fibo for 43 is : 5364436878204150527

Rank 0 finished in about 251 seconds.

HoG-> Resume the monitoring!

HoG-> Warning! All the processes are waiting for HoG to exit!

Killing The Universe ...

HoG-> from last restart until get killed took about 252 seconds.

Killed

HoGServer-> Restart Number: 0 finished in about 252 seconds.

HoGServer-> No available checkpoint file found ...

HoGServer-> Totally finished in about 253 seconds.

HoGServer-> Killing HoGServer ...

Killed

**Result for Test Case 2**

User Output-> Rank 1 : fibo for 43 is : 5364436878204150527

User Output-> Again Rank 1 : fibo for 50 is : 5442052724786215423

User Output-> Rank 2 : fibo for 43 is : 5364436878204150527

Rank 0 finished in about 362 seconds.

Execution without HoG took 363 seconds

**Result for Test Case 3**

HoGServer-> Totally finished in about 310 seconds.

**Result for Test Case 4**

Execution without HoG took 325 seconds

**Result for Test Case 5**

HoGServer-> Totally finished in about 266 seconds.

**Result for Test Case 6**

Execution without HoG took 460 seconds

## 6.3.**Result Comparison**

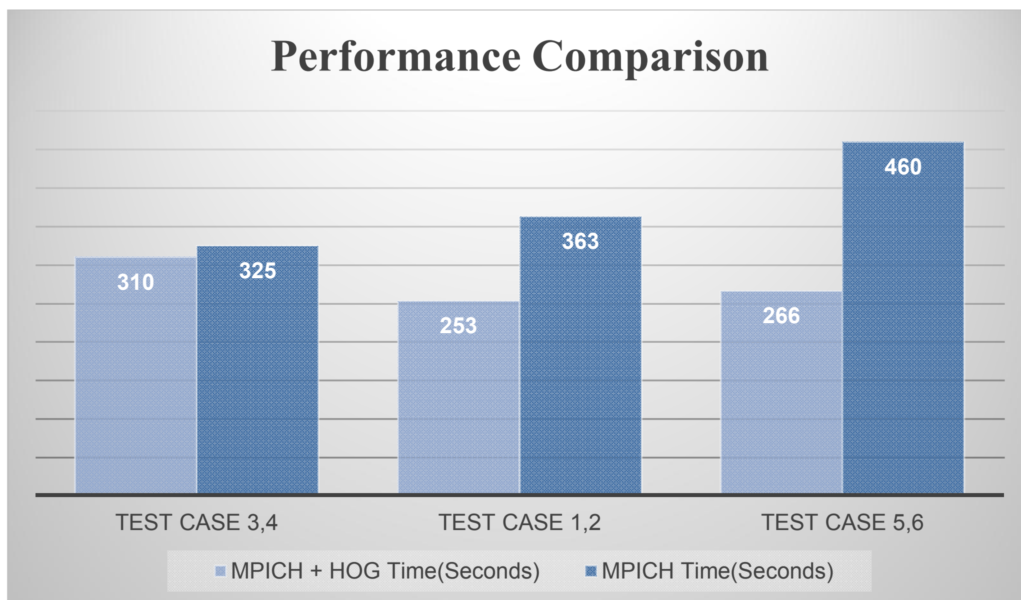


Figure 10 - Performance Comparison Based on the Execution Time

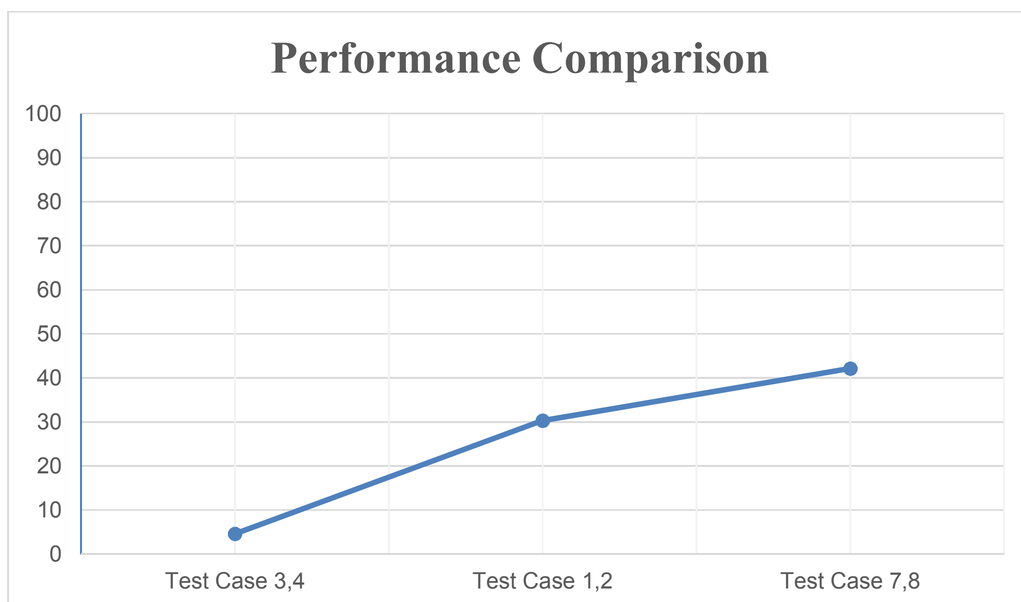


Figure 11- Performance Comparison, Percentage of the Execution Time Improvement

The following will be an analytical results which *Figure 10* and *Figure 11* have been shown above.

The test case 3 and 4 have been executed which test case 3 was using the HoG and with a slide difference HoG finished about 10 seconds faster than normal MPICH execution, but a huge performance improvement cannot be seen which the reason is using a super node which has CPU lower clock than normal nodes, the reason of this experiment is to showing that, if you do not measure the super node`s performance rank carefully therefore you will not receive a huge deduction in performance comparing to the normal execution.

Interestingly, 30% performance improvement can be seen in the execution of test case 1 comparing to test case 2 which as the figure shows, test case 1 have been using the help of the HoG, hence in these two test cases the reason of the difference is that, the normal nodes has only 1 CPU with 1 logical core which the super node has 4 cores with 1000 MHz difference in CPU clock which when the job belong to the heavily loaded node will migrate to the super node it will experience 30% improvement in the performance.

Surprisingly, in the test case 5 even more than 30% improvement can be seen, 42% execution time improvement is comparing the result of the test cases 5 and 6, which is a huge difference and it means that if a job wants to finish in a year, it can finish in 5 month and this is the true usage of the HoG library in MPICH, let us discuss, situations that HoG is extremely useful which it will outcomes the result like 42% execution improvement.

- If your cluster is busy with other MPI processes
- If you do not know when and which node will be super busy than others
- If you have or you can add a node which have better performance

If so, do not wait and begin using HoG right now, because currently there is no other possible way to dynamically balance the load of your cluster after compiling, simply and without any stress write your parallel code and try your best to split the jobs equally, but if you could not do not be worry just call the MPI_HoG() and Hands Of God Will take care of the rest.

## Related Works

So many researches in the area of dynamic load balancing have been done till now, which some of them will be mentioned abstractly in the *Table 20*.

| ***Research Title*** | ***Description*** | ***Method*** | ***Purpose*** |
|---|---|---|---|
| *FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world* [3] | Handling the MPI node crash, normally when one node crash all process will halt but FT-MPI will try to recover and halt prevention if possible. | Checkpointing | Fault Tolerance |
| *Dynamic Load Balancing in MPI Jobs* [4] | A User-level CPU Job scheduler for shared-memory MPI applications which is working with the queue manger which schedule the tasks with affinity and their timestamp | CPUM<br><br>(CPU Manager) | Load Balancing |
| *Dynamic Load Balancing of MPI +OpenMP applications* [5] | A dynamic processor balancing approach which will measure the load of MPI + OpenMP shared-memory application and redistribute the tasks between the processors in run time | DPB (Dynamic Processor Balancing) | Load Balancing |
| *Dynamic Load Balancing of Unbalanced Computations Using Message Passing* [6] | A shared memory load balancing approach, the idle nodes will search for busy nodes and share their jobs or steal it from them | UTS (Unbalanced Tree Search)<br><br>Work Sharing & Work Steeling | Load Balancing |

Table 18- Related Works

# Conclusion

## Future works

As it has been mentioned before, development in the HoG project is ongoing but also there are some sections that the design is complete and only implementation is in progress, therefore they will be discussed in this section as follows.

### Node Switch Mechanisms

#### Nat The source and Destination for Busy and Super Node

Currently, HoG is using the HOST name and DNS modification method for changing the identity of a node in the cluster, but there is another method also available which is not tested yet but theoretically it should have a better performance than DNS method, currently HoG`s switch mechanism is not extremely fast which In the early future it should replace with a faster and more simple mechanism which is the NAT-ing should replace with it.

#### Modify the Node Information in the Checkpoint File

Beside the technical solutions in OS level, there are some solutions programmatically as well, which one of them is to modifying the Checkpoint file while creating it is in progress by altering the data which BLCR is reading in the Checkpointing process.

#### Modify the Node Information in the MPICH

Another node switch mechanism is, to add some flexibility into MPICH source related to the hostnames resolving and the way which MPICH resolve the address of node in order to establish a SSH connection.

## Scenarios

### Multiple Node Switch Scenarios

Until now it have been mentioned some situations which the busy node will be detected and its tasks will be migrated to a super node in order that super node do the task for it, but let`s step forward and discover one of the future features of the HoG which is under development right now.

Important note: obviously for multiple migration there should be more than one super nodes in our cluster which in the example that it will be discussed, two super nodes are needed.

### All Waiting For Two

Migrating multiple nodes together will be a little complex in implementation but fortunately it`s not very difficult to explain, let`s assume another example, let`s say that there are 3 nodes in our cluster which nodes 2 and 3 are working and node 1 is waiting, therefore after HoG cycled its iterations to make sure that node 2 and 3 are working continually and node 1 is waiting or actively waiting, it will begin to look for two super nodes proper for nodes 2 and 3, but a little more detail about the *proper super node* concept is required.

Assume that, nodes 2 and 3 are both busy but their hardware configurations or the amount of the CPU consumption until now are not completely same and is as follow:

| ***Node Name*** | ***Rank*** | ***CPU Clock*** | ***L2 Cache*** | ***CPU Usage*** |
|---|---|---|---|---|
| *Node 2* | 1 | 3323 | 6144 K | 75 % |
| *Node 3* | 2 | 3323 | 6144 K | 100 % |

Table 19- Multiple nodes switch, node`s specification

OR

| ***Node Name*** | ***Rank*** | ***CPU Clock*** | ***L2 Cache*** | ***CPU Usage*** |
|---|---|---|---|---|
| *Node 2* | 1 | 3323 | 6144 K | 100 % |
| *Node 3* | 2 | 3313 | 2048 K | 100 % |

Table 20- Multiple nodes switch, node`s specification

Therefore the node which needs more help in both tables, 12 and 13 is node 3 that should receive a better super node if and only if, there is any difference in the performance mark of super nodes, the reason that in table 12, node 3 needs more help could be the CPU consumption which is 100% for three iterations but it should be kept in mind that more consumption is not always a good factor for our decision to find out the better node, because more consumption could be happened and because of the user`s arguments that have been passed to the mpiexec which maybe the user assigned more tasks, or tasks with heavier load to the node 3 but anyhow this is only an example of a perfect situation which it can be supposed that the user split the task perfectly equal between the nodes, in *Table 22* it can be seen a slide difference between L2 cache of the CPU which this difference can be a factor in some cases.

However, the meaning of proper super node will return to this phenomenon that node 3 was acting weaker so it will deserve a better super node for balancing as perfect as possible in the future migrations, and the steps will be as follow:

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Node 2* | 1 | Running |
| *Node 3* | 2 | Running |
| *Super Node 1* | | |
| *Super Node 2* | | |

Table 21- Before HoG do the migration

And it have been assumed that supper node 1 is completely same as super node 2 and also node 2 is completely same as node 3 in the aspect of configuration and performance mark and the switch will happen based on a simple sorting.

| *Node Name* | *Rank* | *State* |
|---|---|---|
| *Node 1* | 0 | Waiting |
| *Super Node 1* | 1 | Running |
| *Super Node 2* | 2 | Running |
| *Node 2* | | |
| *Node 3* | | |

Table 22- After HoG did the migration

However, these are not the only scenarios which HoG can match with them and as it have been mentioned before HoG`s development is ongoing, but these scenarios are the ones which until now designed and planned to implement so it will be expected in the early future to have the most scenarios as possible to be able to support more performance critical situation in order to result the best performance out of a normal cluster and also to provide the most balanced one.

**Conclusion**

There are so many different Dynamic load balancing approaches out there which it have been mentioned some of them in the in the related works section, but mostly all of their focus were in job scheduling or job stealing.

**The Fact**

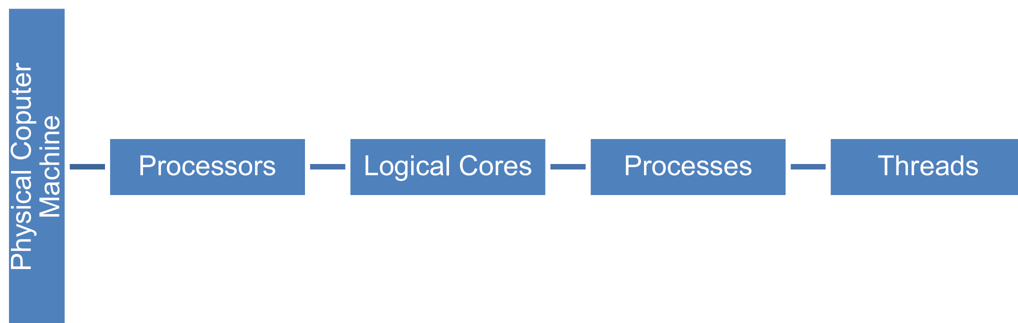


Figure 12- Processing Hierarchy

As the *Figure 12* shows, it can been seen the hierarchy of the processing system in today`s computer machines, which the most granular level is thread level, the fact is all the researches that have been done in the dynamic load balancing area implemented before assigning the task or stealing the task before the assignment because after the assignment the tail of the process stack and it memory area will be

isolated and no one can access it, and also in the message passing interface implementation, according to this fact the new requests are happening through sending and receiving messages between processes in a cluster but no process can access other process`s memory or registers they can only request to do so, therefore when a job received be processor it will outcomes with some processes and threads.

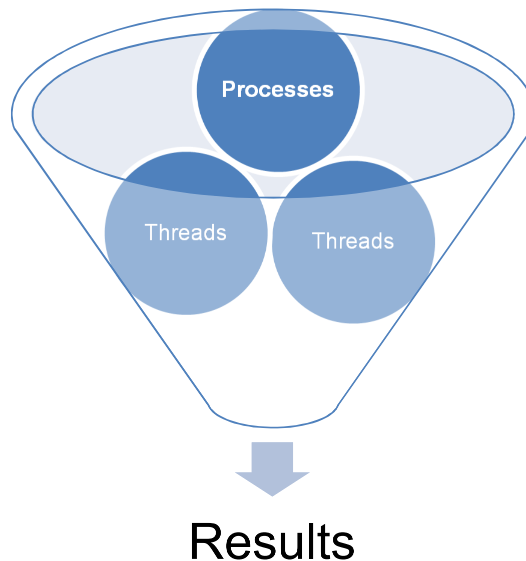


Figure 13- Processor Schema

As *Figure 13* shows the most granular state of the processing hierarchy, Therefore, even if there will be a best design and the best job scheduler or task splitting engine, in the best scenario, it can assign the new jobs to the idle processors or even more complex than that, it can change the assigned job by stealing it from one processor to another one, but in any condition after it assigned and the processor begin to execute the task, it needs to wait for the completion of the assigned task.

Another fact is, the job scheduler`s algorithm, which clearly it`s impossible to predict the absolute duration of the execution which make the job scheduler`s algorithm so week, that without the help of the programmer it can only have a guess that which task will be heavy or which one will be not.

**State of the art**

HoG `s algorithm is based on analyzing the performance and the load of each one the nodes in a cluster, by reverse engineering the analyze data and otherwise other approaches it will let the cluster work and the job schedulers assign the jobs and then it will supervise node`s load and performance, like a monitoring staff in a data center, and when a node (depend on the situation it can be more than one node for more information please visit Chapter 5 section 5) stuck in the execution of a scheduled job even though that job assigned to that node by the most intelligence job scheduler, it will begin to look for a stronger node in the cluster, and if there is a stronger node in the cluster, HoG will migrate that heavily loaded process to a stronger node which it can has a better CPU clock, more logical cores or even better cache, in overall that node can offer a better performance to that specific task and that`s why it has been called a super node, this operation will take less than a second and it can have up to 42% improvement in the execution time and performance result, which to understand this rate better let`s suppose there is a task that last time that it have been executed, it took one year to finish and now it can be said that the same job can finish in  only 7 month, which is a great success for the HoG project.

# References

[1] **E**.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[2] **E**.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", *ACM Comput. Surv.* , vol. 34, no. 3, pp. 375–408, 2002.

[3] **G**raham E. Fagg and Jack J. Dongarra.2012.FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world

[4] **G**ladys Utrera and Julita Corbalán and Jesús Labarta.Dynamic Load Balancing in MPI Jobs

[5] **J**ulita Corbalan and Alejandro Duran and Jesus Labarta.Dynamic Load Balancing of MPI+OpenMP applications

[6] **J**ames Dinan and Stephen Olivier and Gerald Sabin and Jan Prins and P. Sadayappan and Chau-Wen Tseng.Dynamic Load Balancing of Unbalanced Computations Using Message Passing

[7] **M**essage Passing Interface Forum. MPI: A Message-Passing Interface standard. The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3

[8] **M**essage Passing Interface Forum. MPI: A Message-Passing Interface standard (version1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3

[9] **M**enezes, Alfred; van Oorschot, Paul C.; Vanstone, Scott A. (October 1996). Handbook of Applied Cryptography. CRC Press. ISBN 0-8493-8523-7

[10] **R**.E. Ahmed, R.C. Frazier, and P.N. Marinos, " Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems", IEEE 20th International Symposium on Fault-Tolerant Computing (FTCS-20), Newcastle upon Tyne, UK, June 26–28, 1990, pp. 82–88.

[11] **T**. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In Recent Advantages in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI User's Group Meeting, Proceedings, LNCS 4192, pages 374–382. Springer, 9 2006

[12] **Y**ibei Ling, Jie Mi, Xiaola Lin: A Variational Calculus Approach to Optimal Checkpoint Placement. IEEE Trans. Computers 50(7): 699-708 (2001)

## Curriculum Vitae

Aliakbar Sadeghi Khameneh Tabrizi was born in March 21st, 1983, in Tehran, Iran. He received his BS in Computer Engineering in 2005 from Tehran Azad University Central Branch. From 2004 to 2005, he worked as a technical support member in EDP Co. Then since 2005 to 2011 he continued his career by working in TOSAN Co. as a technical support supervisor and from 2011 to 2012 he worked as IT manager and IT consultant in SEATRAVEL Ltd, and also from 2012 to now he is working in Kadir Has University as *Research Assistant* in institute of science and technology.