Process Placement Optimizations and Heterogeneity Extensions to the Slurm Resource Manager

Ioannis Vardas

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

Thesis Co-Advisor: Dr. *Manolis Marazakis*

The work has been performed at and supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Process Placement Optimizations and Heterogeneity Extensions to the Slurm Resource Manager

Thesis submitted by
Ioannis Vardas
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Ioannis Vardas

Committee approvals:

Manolis Marazakis
Doctor, Thesis Co-Advisor

Manolis Katevenis
Professor, Thesis Supervisor

Angelos Bilas
Professor, Committee Member

Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval:

Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, November 2019

Process Placement Optimizations and Heterogeneity Extensions to the Slurm Resource Manager

Abstract

HPC systems keep growing in size to meet the ever-increasing demand for performance and computational resources. Additionally, by utilizing diverse resources such as accelerators to further improve their computational power the HPC systems become more heterogeneous. The resources of such systems are shared among many users, whose number can reach up to a few thousands, and execute a broad spectrum of applications from all scientific fields. This brings up two issues, the first is the diversity of the software stack required by the various applications and the complete platform isolation among the users. Another issue is the managing of the many resources and the various jobs issued by the users. Additionally, applications that seek to lower their completion time rely on advanced models of parallelism for exploiting the system's resources, which leads to increased pressure on system interconnects.

The problem of managing the resources of a complex and large HPC system is tackled by the use of a special middle-ware often called Resource and Job Managing System. In order to deal with the issue of platform isolation and provide a more flexible software stack the Virtual Machine or container technology is often employed. A way to reduce the pressure on system interconnects and improve the performance of the parallel applications is by increasing their communication locality. Apart from the communication cost, the completion time of an application can be further improved by reducing the overhead of MPI job abortions due to node failures.

The author of this thesis implemented three extensions to Slurm which a Resource and Job Managing System that is widely adopted in HPC systems. Also the third extension is evaluated via simulation. The first extension enables Slurm to support FPGA-based accelerators. More precisely, it enables the users to select nodes that have FPGA capability. This extension makes Slurm better suited for heterogeneous platform. The second extension enables Slurm to run workloads in Virtual Machines. Compared to other similar approaches, this extension to Slurm maintains a simple user interface and also integrates the VM management into Slurm.

The final extension implemented a new topology and fault aware process placement approach which is the main focus of this thesis. With this extension, the author of this thesis approaches the communication locality issue via the topology mapping problem, as it also mentioned in the existing bibliography. The additional feature of the new approach is that it takes into account transient node failures. This approach models the topology mapping problem as a graph mapping problem which when solved produces the mapping of the parallel processes of an application to the processing elements of the system. Additionally, this extension implemented a new mechanism in Slurm in order to monitor the system's node

and discover transient failures. as topology mapping problem. Thus, attempting to avoid transient node failures and the paths that include such nodes. The topology and fault aware process placement approach contributes to the overall reduction of the completion time of a batch job by reducing two factors. First, it reduces the completion time of an individual job and secondly it reduces the percentage of job abortions due to node failures.

Finally, we evaluate the performance of the new topology and fault aware approach by simulating real MPI applications in two parts. For the first part of the evaluation, no node failures whereas in the second part node failures are emulated and taken into account. The results of our evaluation show a notable decrease in overall completion time of MPI jobs in both environments. Compared to the default process placement of Slurm, the topology and fault aware approach reduces significantly both the MPI job abortions and the overall completion time from 11% up to 31% for different MPI applications.

Βελτιστοποίηση τοποθέτησης διεργασιών και επεκτάσεις για ετερογενή συστήματα στο λογισμικό διαχειρισμού πόρων Slurm

Περίληψη

Τα υπολογιστικά συστήματα υψηλών επιδόσεων (HPC), στην προσπάθεια τους να ικανοποιήσουν τις συνεχώς αυξανόμενες ανάγκες για περισσότερη απόδοση και υπολογιστικούς πόρους, αναπτύσσονται όλο και περισσότερο σε μέγεθος. Επιπλέον, αξιοποιόντας διαφορετικούς υπολογιστούς πόρους όπως οι "Accelerators"για περαιτέρω αύξηση της υπολογιστικής ισχύος τους γίνονται πιο ετερογενή. Οι υπολογιστικοί πόροι αυτών των συστημάτων διαμοιράζονται μεταξύ πολλών χρηστών οι οποίοι σε πολλές περιπτώσεις μπορεί να ανέρχονται σε χιλιάδες και να εκτελούν εφαρμογές διαφόρων επιστημονικών πεδίων. Αυτό εγείρει δυο θέματα, το πρώτο είναι η παροχή μιας ποικιλόμορφης της στοίβας λογισμικού που είναι απαραίτητη για τις διαφορετικές εφαρμογές και το δεύτερο είναι η εγγυημένη απομόνωση μεταξύ των διαφορετικών χρηστών. Ένα επιπλέον θέμα είναι και διαχείρηση των πολλων διεργασιών των χρηστών καθώς και διανομή των υπολογιστών πόρων. Επίσης, οι εφαρμογές που επιζητούν περαιτέρω απόδοση βασίζονται σε προηγμένες τεχνικές παραλληλίας για να εκμεταλευτούν τους πόρους τους συστήματως αυξάνοντας την πίεση στο δίκτυο του συστήματως.

Το πρόβλημα κατανομής των πόρων στα πολύπλοκα και ογκώδη συστήματα HPC σε διαφόρους (πολλούς) χρήστες αντιμετωπίζεται με την χρήση ενώς ειδικού ενδιάμεσου λογισμικού που συχνά καλείται Σύστημα Διαχείρησης Πόρων και Εργασιών. Το θέμα της εγγυημένης απομόνωσης των διαφορετικών χρηστών καθώς και παροχής μίας πιο ειδικά διαμορφομένης στοίβας λογισμικού συνήθως επιλύεται με την χρήση Εικονικών Μηχανών. Το κόστος επικοινωνίας των παράλληλων διεργασιών μπορεί να μειωθεί αυξάνοντας την επικοινωνική τοπικότητα. Επίσης, η αυξημένη επικοινωνιακή τοπικότητα των εφαρμογών μπορεί να ελαχιστοποιήσει το κόστος της επικοινωνίας με αποτέλεσμα την μείωση του χρόνου εκτέλεσης των εφαρμογών καθώς και την μείωση της πίεσης που δέχεται το δίκτυο. Εκτός από το κόστος επικοινωνίας, ο χρόνος εκτέλεσης μπορεί να βελτιωθεί περαιτέρω μέσω της μείωσης του κόστους της διακοπής των MPI εφαρμογών των χρηστών λόγω σφαλμάτων στους κόμβους του συστήματως.

Ο συγγραφέας αυτής της μεταπτυχιακής εργασίας υλοποιεί τρεις επεκτάσεις στο λογισμικό διαχειρισμού πόρων "Slurm" το οποίο χρησιμοποιείται ευρέως από τα HPC συστήματα. Η πρώτη επέκταση προσφέρει στο "Slurm"την δυνατότητα να υποστηρίζει"FPGA-based accelerators"καθιστώντας το "Slurm"καταλληλότερο για ετερογενή συστήματα. Πιο συγκεκριμένα, δίνει την δυνατότητα στον χρήστη να επιλέξει κόμβους που έχουν επιπλέον λογική υλοποιημένη σε FPGA-based Accelerators.Η επόμενη επέκταση που παρουσιάζεται προσδίδει την δυνατότητα στο "Slurm" να εκτελεί εικονικές μηχανές και μέσα στα εικονικά περιβάλλοντα αυτά να εκτελεί τις διεργασίες των χρηστών. Σε σύγκριση με παρόμοιες δουλείες η εν λόγω επέκταση προσφέρει ένα πιο απλό περιβάλλον για τον χρήστη καθώς και την δυνατότητα διαχείρησης των εικονικών μηχανών από το "Slurm".

Η τελευταία επέκταση υλοποιεί μια νέα προσέγγιση για βελτιστοποίηση τοποθέτησης των διεργασιών λαμβάνοντας υπόψιν το μοτίβο της εφαρμογής, την τοπολογία και επιπλέον τους κόμβους που παρουσιάζουν σφάλματα. Η εν λόγω επέκταση αποτελεί το μεγαλύτερο μέρος της εργασίας. Μέ αυτή την επέκταση, ο συγγραφέας της εργασίας προσεγγίζει το πρόβλημα της επικοινωνιακής τοπικότητας των διεργασιών μέσω του "topology mapping" προβλήματως όπως αναφέρεται στην υπάρχουσα βιβλιογραφία. Το επιπλέον χαρακτηριστικό της νεας προσέγγισης σε σχέση με όλες τις δουλειές που πραγματεύονται το "topology mapping" πρόβλημα είναι η δυνατότητα να λαμβάνει υπόψιν παρωδικά σφάλματα στους κόμβους του συστήματος. Η εν λόγω επέκταση μοντελοποιεί το "topology mapping" πρόβλημα ως ενα γραφοθεωρητικό πρόβλημα και λύνοντας το παράγει τη αντιστοιχία των διεργασιών στους υπολογιστικούς πόρους του συστήματος. Επιπλέον, με έναν νεο μηχανισμό, που υλοιποιεί η επέκταση στο "Slurm" , γίνονται γνωστοί οι κόμβοι που παρουσιάζουν παρωδικά σφάλματα ώστε να αποφευχθεί η χρήση τους καθώς και τα μονοπάτια που τους συμπεριλαμβάνουν. Αυτή η προσέγγιση συνεισφέρει στο να μειωθεί ο συνολικός χρόνων ενώς συνόλου από δουλειές μειώνοντας δυο διαφορετικούς παράγοντες. Πρώτον μειώνει τον χρόνος εκτέλεσης κάθε μιας απο τις δουλειές του συνόλου και δεύτερον μειώνει το ποσοστό των απορριπτόμενων δουλειών λόγω σφαλμάτων.

Τέλος, παραγματοποείται αξιολογήση της νεας προσέγγισης τοποθέτησης διεργασιών σε δυο μέρη προσομοιώντας πραγματικές MPI εφαρμογές. Στο πρώτο μέρος δεν εξομοιώνονται κόμβοι με παρωδικά σφάλματα ενώ στο δεύτερο γίνεται και εξομοίωση των παρωδικών σφαλμάτων στους κόμβους. Τα αποτελέσματα δείχνουν συμαντική μείωση του χρόνου εκτέλεσης των εφαρμογών και στα δύο εικονικά περιβάλλοντα. Η νεα προσέγγιση καταφέρνει να μειώσει σημαντικά τον αριθμό των απορριπτόμενων MPI δουλειών λόγω σφαλμάτων καθώς τον χρόνο εκτέλεσης σε σχέση με την βασική προσέγγιση του "Slurm" από 11% έως 31%

Acknowledgments

I would like to thank Dr. Manolis Marazakis for his support and guidance as well his encouragement that aided me in overcoming the obstacles for implementing this thesis. My deepest appreciations and thanks to Dr. Manolis Ploumids for his invaluable guidance, his tireless support and insight that were crucial contributions to the current thesis. My deepest thanks to Prof. Manolis Katevenis, my Supervisor, who encouraged me in applying for the Master in Science degree, his teachings as well as introducing me to the Computer Architecture and VLSI (CARV) systems laboratory of the Institute of Computer Science (ICS) in the Research and Technology - Hellas(FORTH), where this thesis was performed and supported. I also wish to thank Professor Angelos Bilas, who through his teachings inspired and help me attain a higher level of knowledge required for this level of work.

It was an honour working with my colleagues of CARV laboratory whom I would also like to thank. Many thanks to Dr. Nikos Chrysos who provided with knowledge on topologies and networks. My thanks to Dr. Fabien Chaix for helping me use and SimGrid and particularly the tracing utility, which resulting in the two pie diagramms. Furthermore, I would like to thank Marios Asiminakis for his support and technical help in debugging that was vital for the implementation part of this thesis. I would also like to thank Antonios Psistakis for collaborating with me in many other projects during our period of studying in the University of Crete. Many thanks to Vasilis Flouris and Manolis Skordalakes who provided me with knowledge on software tools that assisted me in the completion of this thesis.

Finally, I would like to thank the Foundation for Research and Technology - Hellas(FORTH), Institute of Computer Science (ICS), for funding and providing the necessary infrastructure for the implementation of this thesis. Funding for this thesis was provided by the ExaNest research project supported by the European Commission under the “Horizon 2020 Framework Programme” with grant agreement id 671553.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past 60 years, computing technology has undergone a series of platform and architecture changes. Interest has shifted from single-node systems to parallel and distributed computing systems. *A parallel computing system is a collection of processing elements that cooperate and communicate to solve large problems fast* (Culler et al., 1998)[24]. Modern high performance computing (HPC) systems adopt the paradigm of parallel computing systems for delivering high performance. *HPC is a field of endeavour that relates to all facets of technology, methodology and application associated with achieving the greatest capability at any point in time and technology* (Sterling et al., 2017)[53]. The same definition can be given to HTC however the goal HPC is to deliver enormous amounts of computing power in a short period of time while HTC employs large amounts of computing power for quite lengthy periods of time.

The increasing demand in computational power and resources has led HPC and HTC systems to grow in size. Additionally, it has led to the integration of heterogeneous resources such as accelerators. A common resource for achieving some form of acceleration is the Graphics Processing Unit (GPS) which is integrated in most HPC systems. An indicative example is Summit [12], ranked as 1st in the well known top500 list [15] which consists of 4608 nodes and a total of 27,648 GPUs (6 per node). Another source of acceleration relies on Field Programmable Gate Array (FPGAs) which are commonly used to offload part of the computation to FPGA seeking for faster execution. Authors in [49] present an implementation of OpenMP offloading for FPGA-based accelerators. The work in [17] presents a framework for offloading MPI collective operations to FPGA-based accelerators. Existing HPC systems that utilize FPGAs include Catapult Academic Project [40] which is run in collaboration with TACC (TEXAS ADVANCED COMPUTING CENTER) [13]. HPC systems thus, grow not only in terms of size but also in terms of resource diversity, resulting in heterogeneous platforms.

Besides the huge number of resources of such a system, the number of users accessing it is also increasing. As HPC and HTC are becoming more popular among the various scientific fields it attracts an increasing number of platform

users. Users of such systems can easily reach a few thousands [53] where each of them can potentially execute multiple applications with different properties and requirements. Thus numerous applications can run concurrently, sharing the many resources of the system among the users. At such a scale of resources and users, management of HPC and HTC systems on a manual manner is prohibitive

The problem of allocating the resources to users and generally managing such complex and large parallel systems (as HPC and HTC) is solved by the use of special software often called Resource and Job managing System (RJMS). A RJMS performs two main tasks, The first is to allocate system resources to jobs and the second is to schedule different job executions. A job can be defined as user workload encapsulated into self-contained unit. A job may specify among others the following: binary to be run, input and output data sets or parameters, the resources required, and finally, the maximum time for running to completion. Each job can consist of several parallel tasks or job steps. Typical examples of resources allocated by a RJMS are to jobs are:

- Compute Nodes
- Processing Cores
- Storage and I/O
- Accelerators

Some popular RJMS include PBS - Portable Batch System [7] the recipient of 2014 Annual HPCwire Readers' Choice Award [1]. Another is HTCondor mainly used for HTC. Moab Cluster Suite is another popular RJMS which is not open source and requires licensing.

The focus of this thesis is extending *Slurm*, the resource manager and job scheduler for about 60% of the top 10 systems in the list of top500 [15]. *Slurm*, historically, is the acronym of Simple Linux Utility and Resource Manager. It is an open source project developed by Lawrence Livermore National Laboratory [48] in C programming language with a size of about 500,000 lines of code. It is popular among HPC clusters including Tianhe-2 and Sunway TaihuLight supercomputers. Slurm was chosen over the other RJMS mainly due to two reasons. The first one is that it is open source. The second reason is that a significant part of Slurm's functionality is offered through plugins that are separate from the source code tree. In this way it is easy to extend Slurm's functionality along different directions without modifying the main code tree. This results in portability across different Slurm versions and also across different clusters where modification of the main Slurm is not allowed.

1.1 Problem Statement and Motivation

Although Slurm includes support for several types of resources, platforms and topologies, it still lacks support for several important features and aspects of HPC systems.

Slurm offers support for the following generic resources, Graphics Processing Units (GPUs), CUDA Multi-Process Service (MPS), Intel® Many Integrated Core (MIC) processors. However, it does not include support for FPGA-based resources, which means that in the default version of Slurm, users are not able to select logic that is implemented through FPGAs. Utilizing FPGAs can offer significant speedup by accelerating specific parts of the code or by supporting certain MPI primitives as it demonstrated in [17]. One of the extensions contributed in this thesis extends Slurm functionality so as to support the selection of FPGA-based resources. The corresponding contribution is extensively presented in Chapter 3.

The use of Virtual Machines offers a variety of features such as, complete platform isolation between the users, fully privileged user access to a flexible software stack, as opposed to the restrictive user access in the physical Machine. Virtual Machines can also be redeployed and replicated and also offer the ability to capture snapshots for increased reliability. In this thesis, an extension of Slurm is contributed and fully described in chapter 4, that allows to run jobs using virtual machines as resources, apart from physical ones.

The main part of the work described in this thesis is related to the core functionality of Slurm, that is, the logic and mechanism for allocating resources to jobs. When a new job requests resources, the default policy of Slurm is to review the available resources and find a set of consecutive nodes, based on the internal list of nodes. Slurm can be configured to support topology aware resource allocation. However, it is ignorant to jobs' communication profile. This means that there is no provision for placing processes with a heavier communication profile on processing elements that are, topologically, less distant. In this way, the communication cost could be minimizes resulting in a lower application execution time. Additionally, several studies have outlined the importance of energy efficiency that can be achieved by reducing the interconnect resources involved for message delivery. Such a claim is made by the authors of [29]. Another important characteristic of Slurm is that although it is able to infer when a node is down, allocating resources to jobs while taking into account transient node failures is not supported. Trying to avoid nodes that have exhibited failures is a proactive means of dealing with node failures aiming at reducing the probability of a whole job being aborted. Recall that, according to the MPI standard, the default behaviour when a node fails is to call to an MPI primitive returning an error. The default handling provisioned by the MPI standard for such cases is abortion of the job. The TOFA-Slurm extension is described in 5.

1.2 Contributions

The author of this thesis implements three different extensions to Slurm and provides with an evaluation of the third extension which is the main work of this thesis. Each extension is covered through a separate Chapter, aiming at addressing the corresponding limitation to the default Slurm's functionality, as discussed in the previous section.

The first extension implements a plugin in Slurm plugin that enables Slurm to support FPGA-based accelerators. This plugin allows the user to select nodes based on the availability of a specific type and count of accelerators for a job. This extension uses the plugin API of Slurm, which means that it requires no modification of the source code tree.

The second extension, implemented in Slurm, enables it to launch, manage and run workloads into Virtual Machines. We call it *VIrtual MAchine(s) in Slurm - VIMA-Slurm*. This extension allows the user to run workloads using either virtual or physical resources. Compared to similar approaches this approach provides a simpler user interface without the need of additional scripts to run the VMs. This implementation required modifications deep in the Slurm code, particularly the RPC-based protocol for control and coordination.

The third extension of Slurm allocates resources to a MPI job with the dual goal of reducing that job's completion time and the job abort ratio caused by node failures. This extension focuses in improving the performance of MPI [36] applications. A wide range of scientific applications, such as, Computational Fluid Dynamics [6], Molecular Dynamics [33]. and Bioinformatics [4] rely on the MPI library for exploiting parallelism. MPI is very prominent in HPC systems, as the performance of an HPC system in the top500 list [15] is measured by two MPI applications, namely HPL [30] and HPCG [2].

In the third extension, the author of this work implements a topology and fault aware (TOFA) process placement approach in Slurm. This is the main contribution of this thesis. The topology and fault aware approach improves two aspects of a job's completion time, namely, communication cost and the overhead of a job abort due to node failures. Note that a node failure may result in a call to an MPI primitive returning an error. The default handling provisioned by the MPI standard for such cases is abortion of the job. For reducing the communication cost of an MPI job, this approach aims at placing ranks with a heavy communication profile at nearby nodes (in respect to topological distance). This assumes that a training run of the MPI job is performed, that has allowed us to extract that job's communication profile. For extracting the topological distance between any two nodes of the cluster, topology information is also assumed to be available. As far as job abortion is concerned, the TOFA approach estimates the node failure probability using heartbeats and uses this information to post-process the data structure resembling the topology graph. Several related studies have outlined the importance of allocating resources on an application and topology aware manner, for reducing congestion on the interconnect as well as overall energy consumption.

The authors of [29,] demonstrate that their efficient mappings show significant reduction of congestion in both torus and fat-tree network topologies. However, none of the studies address the issue of the node failures in contrast with this thesis. Additionally, while Slurm can support topology information for allocating resources it does not take into account the application's communication pattern and the transient node failures. For the implementation of this extension in our work, five different plugins were implemented in Slurm, together with the necessary interface for utilizing the external *Scotch* graph mapping library.

The author of this thesis implemented all the extensions in a real three node system; however, for the evaluation of the third extension a larger system was necessary. For this reason, the author of this thesis used a simulated environment, based on the SimGrid simulator, and more specifically using the SMPI interface. The applications simulated are unmodified real MPI applications used in HPC systems, such as NAS Parallel Benchmarks, High Performance Linpack and LAMMPS. This evaluation also provides with an analysis of the traffic patterns of applications as well as other characteristics of these applications. The evaluation is divided in two parts, one that uses a platform without node failures and another that considers node failures.

The rest of the thesis is organized as follows: Chapter 2 presents the architecture of Slurm while Chapter 3 discusses the extension of Slurm that allows node selection based on availability of accelerators. Chapter 4 presents the implementation of a Slurm extension that allows running workloads of jobs within virtual machines. Chapter 5 discusses the TOFA approach and presents its implementation in Slurm, the TOFA-Slurm. Additionally, chapter 6 presents an evaluation of the TOFA approach and finally, Chapter 7 discusses ongoing and future work for the last two Slurm extensions.

Chapter 2

Slurm Architecture

2.1 Introduction

For all Slurm extensions discussed in this thesis, either modification of Slurm source code was required or the corresponding functionality was implemented through plugins. For this reason, an overview of the architecture and main components of Slurm would ease the understanding of the technical aspects for each Slurm extension contributed.

2.2 Architecture

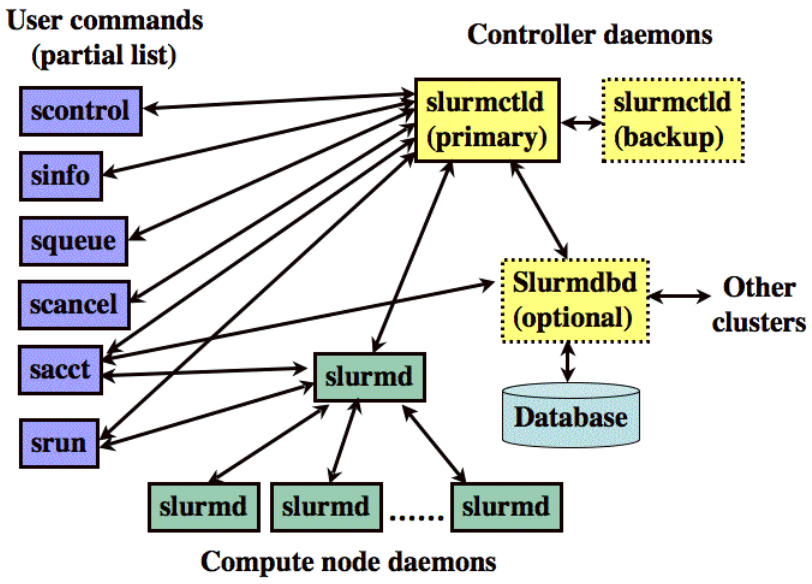


Figure 2.1: Slurm Components

Slurm components are divided into two categories, the daemons that run in the system's node and the tools offered to users and administrators. Both of them are depicted in Figure 2.1. The two main daemons run in the nodes of the HPC system are *slurmctld* and *slurmd*. *Slurmctld* is the daemon that runs only in the "Controller Node" and is responsible for allocating resource to different jobs and also scheduling them. *Slurmd* is the daemon that runs on the compute nodes and waits to execute work issued by *slurmctld*. To be more precise the *slurmd* spawns another daemon called *slurmstepd* to set the environment and launch the various job steps or tasks. This daemon is bound to a job step and exists only for the duration of job's execution. This component is not depicted in the figure and it is not initiated by the users or the administrators. Slurm assumes a single "Controller Node" with a single instance of *slurmctld* running and, optionally, another backup node as depicted in 2.1. Whereas it assumes multiple "Compute Nodes" with at least one *slurmd* on each node. *Slurmdbd* is the Slurm database daemon which is optional and runs only on a single node. It is responsible for account recording information for the multiple Slurm-managed clusters in single database.

Several user and administrative tools are offered in order to monitor overall system's state information as well as job information, they are depicted as "User commands" in Figure 2.1. *Srun* and *sbatch* are are used to initiate jobs. The difference between the two is that *srun* is used to run jobs in real time while *sbatch* issues a job for later execution. For job and system status *squeue* and *sinfo* are used respectively while *sacct* provides information about the completed jobs. *Sview* and *smap* offer a graphical representation of the system. Two powerful administrative tools offered are *scontrol* and *sacctmgr*. *Scontrol* can be used to monitor and modify the state of the system while *sacctmgr* is responsible for managing database information.

2.3 Slurm Plugins

Slurm extends its functionality and support of different platforms through plugins. It comes with many different plugins in its standard version while it also offers an API for developing additional plugins. The additional plugins can add various features without modifying the main source code tree. Effectively, this plugin-based functionality can be easily ported to newer Slurm versions since it leaves the main source code tree intact.

There are several categories of plugins including topology, resource selection, generic resources (GRES), job launch and many others. The topology plugins are used to provide explicit knowledge of the system topology to Slurm enabling optimized resource selection. The existing topology plugins offered by Slurm include tree and 3d Torus. The tree topology plugin represents a hierarchical topology based on switches, such as, a Fat Tree. As shown in Figure 2.2 this topology has 3 different tiers (or ranks) of switches.

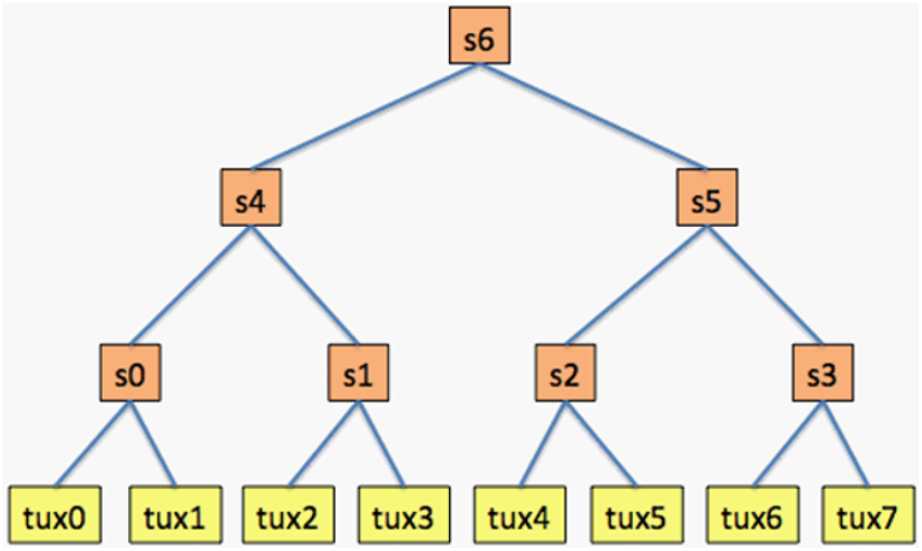


Figure 2.2: Fat Tree Topology

Using the tree topology plugin and a topology configuration file a Slurm administrator can represent a Fat Tree topology in Slurm. The configuration file of the topology shown in Figure 2.2 is presented below:

```
# Switch Configuration
SwitchName=s0 Nodes=tux[0-1]
SwitchName=s1 Nodes=tux[2-3]
SwitchName=s2 Nodes=tux[4-5]
SwitchName=s3 Nodes=tux[6-7]
SwitchName=s4 Switches=s[0-1]
SwitchName=s5 Switches=s[2-3]
SwitchName=s6 Switches=s[4-5]
```

Listing 2.1: Topology configuration file

The resource selection plugins determine how the resources are allocated for a job, while the generic resources plugin (GRES) can be used to manage special type resources, such as, GPUs or as we will see any other kind of accelerator. Two resource selection plugin offered by default Slurm are the “linear” and “cons_res”. The “linear” selection plugin allocates whole nodes using an one dimensional list of nodes whereas the “cons_res”(consumable resources) allows the allocation of individual processors, memory, etc. within nodes. The “cons_res” selection plugin is used for the Slurm extension presented in chapter 4.

Additionally, there are plugins that can dynamically modify job launch code, called SPANK plugins. SPANK, Slurm Plugin Architecture for Node and job (K)control, is a generic interface for plugins which can be used to dynamically modify the job launch process. SPANK plugins can be built without access to Slurm source code as opposed to the normal plugins which they need full access in

order to be built. This means that SPANK plugins do not require to be compiled with the rest of the Slurm source code. Thus, while Slurm is running in a system, a SPANK plugin can be compiled separately using the SPANK API without restarting *slurmctld*. Then, the SPANK plugin can be loaded to modify the job launch code. There are five different contexts in which the SPANK plugins can be loaded:

1. local context - *srun*
2. remote context - *slurmstepd*
3. *slurmd* context
4. *job_script*, job prologue and epilogue

The *slurmd* context refers only to the initialization and termination of the *slurmd* daemon. This means that SPANK plugin code will be run once during the initialization and once during the termination of the *slurmd* daemon and not every time a job is launched. On the contrary, the remote context means that every time a *slurmstepd* is spawned by the *slurmd*, as mentioned before, the plugin code will run. This happens prior to task launch and also during the termination. In job_script context besides the SPANK plugins, user scripts can also be invoked, which can also modify several parameters of the job launch, allocation or termination process. Specifically, prologue and epilogue scripts can be invoked and executed at a different time. To invoke a prologue and/or epilogue script the user can either input it as an extra argument in *srun* command when submitting the job. SPANK plugins can also be loaded during job prologue and/or epilogue. More details about the different contexts, for prologue and epilogue, and the time executed are shown in the Table 2.1 below:

Prolog/Epilog	Location	Invoked By	When Executed
Prologue	Compute Node	srun user	first job initiation
Prologue	Control Node	slurmctld	job allocation
Prologue	Compute Node	slurmd	prior to task launch
Prologue	Compute Node	slurmstepd	prior to task launch
Epilogue	Compute Node	slurmstepd	task completion
Epilogue	Compute Node	srun user	job step completion
Epilogue	Compute Node	slurmd	job termination
Epilogue	Control Node	slurmctld	job termination

Table 2.1: Different contexts where SPANK plugins can be loaded

In section 2.5, that discusses the job launch process we will again refer to the different contexts where the SPANK plugins are loaded.

2.4 Slurm Configuration File

Slurm also uses a configuration file to represent information regarding the system. Among others, the configuration file includes the names of the compute nodes, the controller node, the resources such as CPU or other features. The configuration file is provided and maintained by the system administrator. Additionally, the configuration file contains information about the different plugins used, such as which Topology plugin or Node select plugin is used and also the type of scheduling. An indicative example of Slurm configuration file is presented:

```
# Sample /etc/slurm.conf
SlurmctldHost=linux1  #Controller Node
AuthType=auth/munge
PluginDir=/usr/local/slurm/lib
SlurmdTimeout=120
MessageTimeout=60
StateSaveLocation=/usr/local/slurm/slurm.state
SelectType=select/linear
TopologyPlugin=topology/tree
TmpFS=/tmp
# Node Configurations
NodeName=linux1 CPUs=8 Gres=fpga:1,vm:c2m512:3
NodeName=linux2 CPUs=2 Gres=vm:c1m1024:1,vm:c2m512:1
NodeName=linux3 CPUS=8 Gres=gpu:fx:1,RealMemory=16384
PartitionName=debug Nodes=tux[1-3] Default=YES State=UP
PartitionName=vm_capable Nodes=tux[1-2] State=UP
```

Listing 2.2: Slurm configuration file

The above example illustrates a small cluster of 3 nodes where one of them is the controller node. The node selection is done by the linear select plugin, which is the default option of Slurm. For the topology of the system the tree plugin is used which assumes a hierarchical network topology. For each node the number of CPUs must also be provided in the configuration file. Additionally, for each node, the configuration file may list resources such as, generic ones based on the generic resource plugin. For two of the Slurm extensions contributed in this thesis, two generic resources have been introduced. One to denote an FPGA-based accelerator, and one to denote a Virtual Machine. The last line of the configuration file of presented above is the name of the Partition. A partition is another entity defined in Slurm which is a logical set of nodes. It can be explicitly defined to include nodes with specific characteristics. In the above example the partition named "debug" contains all the nodes, while the partition name "vm_capable" contains only those with the capability to run a Virtual Machine. A job can be initialized on any of the available partition. The limitation is that it can used more than one partitions at the same time.

2.5 Job launch process in Slurm

To further understand the methodology and technical/implementation details of our Slurm, an overview of the job launch in Slurm is presented. Generally, jobs can be considered as allocations of compute resources assigned to the user for a specific amount of time. A job can consist of many parallel tasks or job steps which require at least one CPU to run. The user can ask for more CPUs for each task when submitting a job. Figure 2.3 is an overview of a job launch in Slurm.

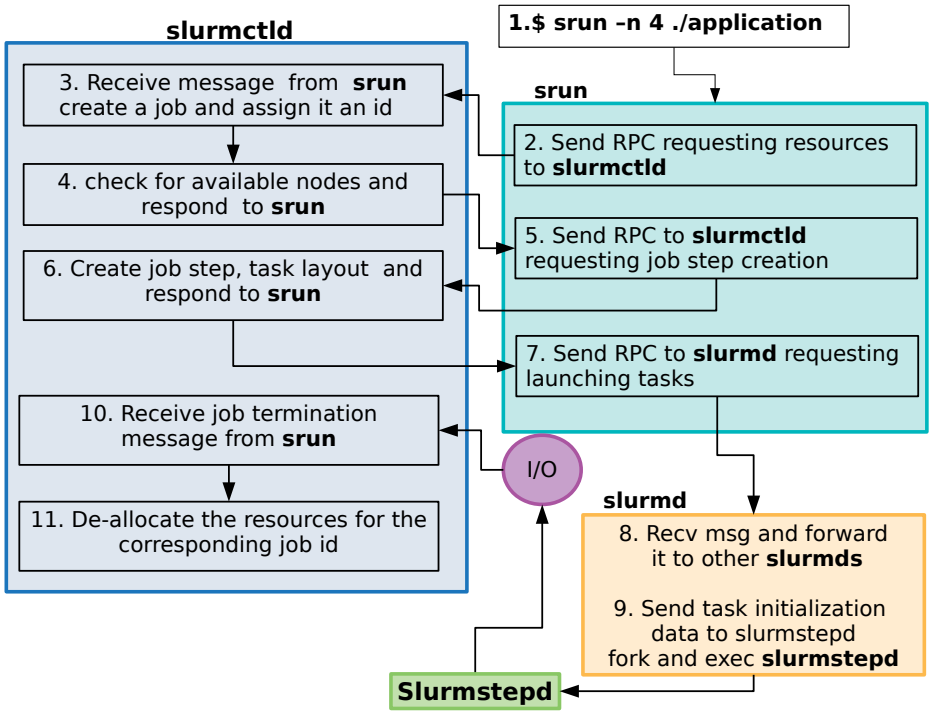


Figure 2.3: Job launch process overview

For starting a job in Slurm the user issues an *srun* command as shown in step 1 of Figure 2.3. The "-n" argument in the *srun* command refers to the number of tasks required for this job. The *srun* command can be invoked in any compute node by the user. At this point a SPANK plugin can be loaded to modify the job request (i.e. adding extra arguments to srun). This is the "local context" also mentioned in 2.3 for the SPANK plugins. *Srun* creates a data structure that contains all the job specifications (e.g node count, number of tasks, etc.). Then, sends a RPC for a resource allocation to *slurmcltd* as shown in Figure 2.3 (step 2).

When the allocation request is received by *slurmcltd* it will create a job structure that contains all information associated with the specific job (e.g. node count, task count, etc.) and assign an job id (step 4). Then *slurmctld* will check if there is

an allocation of nodes available for the job request (step 4). At this point the select plugin is called which will consider the network topology based on the topology plugin to select the resources for the job. Although, the resources have been allocated the actual layout of the tasks is not yet defined. The layout of tasks means which task is distributed to which node. If there a allocation of nodes is available *slurmctld* responds back to *srun* (step 5). When *srun* receives the response from *slurmcltd* it will send an RPC back to *slurmcltd* requesting for job step creation. The *slurmcltd* receives that request and creates the job step and the task layout then responds back to *srun* (step 6).After receiving the response *srun* will create a request to *slurmd* to launch the tasks (step 7). *Slurmd* is responsible for distributing this request to the *slurmd* daemons that are running in the rest of the nodes allocated for this job (step 8). The degree of fan-out in this message forwarding is configurable using the "TreeWidth" parameter in the slurm configuration file. *Slurmd* then, uses the **fork** and **execvp** system calls forks to execute the code of *slurmstepd* (step 9). The steps of *slurmstepd* are depicted in Figure 2.4.

Slurmstepd forks as many times as the number of tasks to be run in this node. As also seen in Figure 2.4, for the case of a job requiring n tasks, slurmstepd creates n images of itself. Each *slurmstepd* can call SPANK plugins at several points in time, this is the "remote context" referred to in 2.3. For example, the SPANK plugin at this point, could modify the tasks to run in a container environment. Additionally, *slurmstepd* performs the following actions:

1. Configures I/O for the tasks either using files or a socket connection back to the *srun* command.

2. Sets up environment variables for the tasks

3. execute the task using **execve** system call

what regards I/O configuration for tasks , it should be mentioned that the socket connection to *srun* command is created to make the task's output visible to the user. This is because the *srun* command can be running on a different node from the task. The user running the *srun* command expects to see the output of the tasks in the output of the *srun*, thus the output of the task is redirected to *srun*.

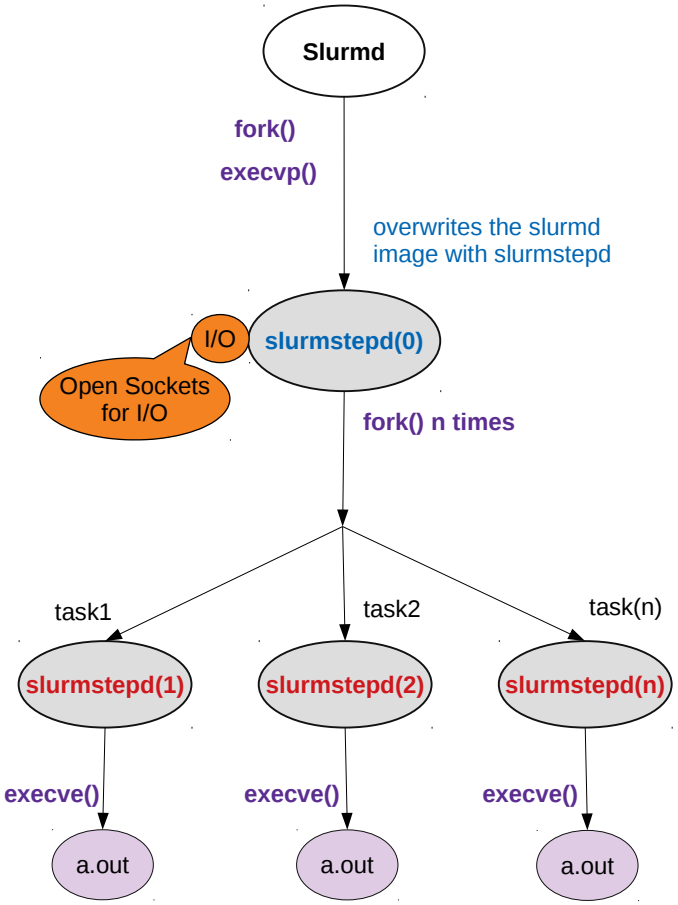


Figure 2.4: Slurmstepd Job Launch in a single node

The job termination, in the simplest case, occurs when all the tasks run to completion. *Srun* waits for each task to finish, for each finished task it sends a RPC to *slurmctld* for task completion. Remember, if multiple nodes were used for the job, there are *slurmstepd* daemons that run on different nodes from *srun*. *Srun* waits for all of the nodes to finish and it is responsible for notifying the *slurmctld*. Thus, the node where the job was started is responsible for monitoring the termination.

When *slurmctld* is notified for the termination of all the tasks involved, it will notify the *slurmd* on each node. *Slurmd* and reaches the job epilogue point where the a SPANK plugin can be loaded. When all *slurmds* are notified *slurmctld* also reaches the job epilogue point and the job is considered completed.

Chapter 3

Accelerator-Capable Nodes in Slurm

3.1 Introduction

The use of accelerators is becoming more prominent in the HPC as they can have a significant impact on the execution time of the applications. Two basic examples of accelerators, integrated in most HPC systems, are the Graphics Processing Units (GPUs) and the FPGAs. The integration of accelerators in HPC systems increases the number of resources available to nodes along with system heterogeneity. As authors of [49] claim, future HPC systems will include multiple specialized accelerators. Slurm categorizes the resources of a system into two categories, the standard resources and the generic resources (GRES). The standard resources include nodes, processors and memory. The generic resources include Graphics Processing Units (GPUs), CUDA Multi-Process Service (MPS), Intel® Many Integrated Core (MIC) processors. Support for the aforementioned generic resources is provided by plugins which are included in the default version of Slurm.

Although Slurm provides support for GPUs (a common type of accelerator), it does not provide support for FPGA-based accelerators. FPGA-based accelerators are an emerging accelerator technology and are becoming more popular in HPC. They are used to accelerate specific parts of the code, such as collective communication primitives used frequently by MPI applications. For example the authors of [41] propose the offloading of MPI collectives to FPGA-based accelerators and, as they demonstrate, it has a impact of 8% to 35% speedup on the application's execution time. Authors in [49] are extending the OpenMP [14] standard, widely used in HPC along with MPI, to support offloading for FPGA-based accelerators.

In this chapter we present Acn-Slurm (Accelerator-Capable Nodes in Slurm) which extends Slurm's functionality so as to support the selection of FPGA-based accelerators.

3.2 Related Work

The use of the generic resource plugin offered by Slurm has also been utilized in the M2DC project (Modular Microserver Data Centre) [20]. More precisely, as also described in the corresponding deliverable [3], the generic resource plugin can be used to allocate resources other than CPUs/nodes. In this project, such resources are referred to as *System Efficiency Enhancement (SEEs)* with two indicative examples being a *SEE* for Deep Neural Networks and another one for encryption.

3.3 Implementation and Usage

The functionality of Acn-Slurm is implemented through a generic resources (GRES) plugin. The generic resource plugin can be used to describe different accelerators, in this case the FPGA-based accelerator. This way Slurm is made aware of the nodes which offer an FPGA-based accelerator. Additionally, every generic resource can have different sub-types which are called GresTypes. The different GresTypes can be used to denote different configurations of the FPGA-based accelerator. The user can issue jobs that request the specific type of FPGA-based accelerator using *srun* without having explicit knowledge of the cluster. This means that the user is not required to know which node offers the specific FPGA capability and explicitly choose it when issuing a job.

To better understand the implementation and usage of Acn-Slurm let us consider the following example. In HPC systems, FPGAs can be programmed to accelerate and support specific MPI primitives as mentioned in 1.1. Consider a HPC system where every FPGAs are programmed to support either, the MPI_Allreduce or the MPI_Reduce. In Acn-Slurm the FPGA-based accelerator is denoted as a generic resource (GRES). The different configurations of the FPGA-based accelerators are described as different GresTypes. The Slurm configuration file of the above system could be as following:

```
NodeName=tux0 Gres=fpga:allreduce:1,fpga:reduce:2
NodeName=tux1 Gres=fpga:allreduce:2
```

Listing 3.1: FPGA as a generic resource in slurm.conf

In the above configuration example, the “fpga” is the generic resource that describes the FPGA-based accelerators. Also there are two different configurations of FPGA -based accelerator, the “allreduce” and “reduce”. The “allreduce” FPGA-based accelerator provides support for the MPI_Allreduce primitive while the “reduce” provides support for the MPI_Reduce primitive. The“allreduce” and “reduce” are two different GresTypes. The node “tux0” offers one FPGA-based accelerator that supports the MPI_Allreduce primitive and two (FPGA-based accelerators) that support the MPI_Reduce primitive. The node “tux1” offers two FPGA-based accelerator that supports the MPI_Allreduce.

If the user requires to run an MPI application that benefits from accelerating the MPI_Allreduce primitive the srun command will be:

srun -n 4 -gres:fpga:allreduce:1 ./application. This way Slurm will assign the job to a node with 4 CPUs and one FPGA-based accelerator that supports the MPI_Allreduce primitive. It is not required by the user to know explicitly which node is capable, thus it offers a simple interface. Moreover, if the user application requires support for both MPI_Allreduce and MPI_Reduce primitive the srun command will be :
srun -n 4 -gres:fpga:allreduce:1 -gres:fpga:reduce:1 ./application. In this case, Slurm will allocate one node that provides FPGA support for both MPI_Allreduce and MPI_Reduce. An important detail to be noted for the last example is that both FPGA-based accelerators must be on the same node, this is a limitation of the GRES scheduling mechanism which is not yet handled. Each node with generic resources also is required to contain a "gres.conf" file in the above example 3.1, the "tux0" node could contain a gres.conf file that would look like:

```
Name=fpga Type=allreduce Count=1 Cores=0,1
Name=fpga Type=reduce    Count=2 Cores=2,3
```

Listing 3.2: Sample gres.conf file

The "gres.conf" file the CPU cores that are preferable when using the specific generic resource. For example in NUMA node there can be preferable CPU cores as the distance between them and the generic resource can be shorter. "Count" variable shows the number of the specific GresType. There are many other options to configure a generic resource in Slurm we only provide some simple examples for the purposes of our work.

Another approach of implementing a similar functionality to Slurm is to create partitions that will isolate the nodes equipped with FPGA-based accelerators. A partition in Slurm can be defined through the configuration file, and any job can be issued to a single partition only. The partitions can be overlapping meaning a node with two different types of FPGA can belong to two different partitions. Thus in the above example where the user requests two different types of FPGA, it is required that the system must have one partition for the "allreduce" fpgas, one partition for the "reduce" and one partition that will contain both. In this scenario the job that will require the two different types of FPGA will run in the partition that contains both. However, if the cluster has many different types of generic resources it will require many different types of partitions, which will make the configuration much more complicated. If another generic resource, lets say GPU is added, this will increase the number of partitions from 3 to 7, "allreduce+reduce", "reduce+gpu", "gpu", "allreduce+reduce+gpu". As the number of resources increases the complexity of this approach increases impacting both the administrators and users of the system. Finally, if the FPGA-based accelerator is not a generic resource, the information of "gres.conf" aforementioned cannot be utilized.

Chapter 4

Virtual Machines in Slurm

In modern HPC systems there is a large variety of different workloads that may request computational resources including databases, I/O related applications and parallel applications solving scientific problems. This results in an increasing range of workloads requiring diverse software stacks and flexibility in terms of runtime environment, while also demanding complete platform isolation.

Virtual Machine (VM) is a mature technology that could meet the aforementioned requirements. VMs can deliver complete platform isolation among the different users. Furthermore, a more flexible software stack and different runtime environments can also be provided. Additionally, in a Virtual Machine the user can have fully privileged access whereas in the physical machine granting fully privileged access to the huge number of different users is prohibitive. Moreover, the environment of a Virtual Machine offers increased control and research reproducibility. This is achieved by the use of snapshots as well as replication and redeployment of the virtual environment.

In this chapter we discuss VIMA-Slurm, an extension of Slurm that enables running workloads using either virtual or physical resources through Slurm. It extends Slurm making it capable of managing the Virtual Machines. The implementation of VIMA-Slurm required modifications in the main source code of Slurm. The main advantages of VIMA-Slurm are:

- jobs can request either physical or virtual resources using the same command set with different arguments depending on the request
- spawning, networking setup and VM tear down are transparent - no prologue scripts needed. This is also a major difference from similar approaches

4.1 Related work

Similar work that integrates the use of VMs in Slurm is limited though there some notable mentions. A very similar work is [21] where Slurm is also extended to run Virtual Machines by including a VM scheduler which starts a new VM as an

exclusive job. This work requires prologue and epilogue scripts for launching the VMs. This means that the management of Virtual Machines is not done by *slurmctld*. The user is responsible for selecting available images for every job. Another work is Slurm-V presented in [11] which enables the VMs to be assigned SR-IOV and Ivshmem resources, which allow a memory region to be shared amount the Guests. Another technology related to Virtual Machines are the Linux containers. The Linux containers is a technology lightweight alternative to VM. The Singularity [10] container platform offers a SPANK plugin [18] that enables the integration of Linux containers in Slurm. Shifter [9] is another container technology designed for HPC and developed by NERSC [5]. Shifter also provides a SPANK plugin for integration with Slurm.

4.2 VIMA-Slurm Architecture

Slurm's functionality is extended to two directions. Firstly, Slurm is extended so as to enable jobs to be run using either physical or virtual resources. Secondly, it is made capable of managing the virtual resources. To enable workloads to execute through Slurm using virtual resources, or Virtual Machines it was necessary to make Slurm aware of the VMs. This was achieved by the development of a new generic resource (GRES) called "vm", which also enables the user to request for this resource when issuing a job. Furthermore, Slurm was extended to manage the different Virtual Machine images that are pre-installed in every node. Moreover, the core functionality of launching the Virtual Machines while also setting communication between them and running workloads using them instead of the physical machine. Finally, powering off the VMs when the job runs to completion or is canceled by the user.

4.2.1 Generic Resource to denote a Virtual Machine

The GRES plugins as discussed in Section 2.3, can be used to represent certain resources of a node. Using this API we developed a new generic resource plugin that denotes the capability of a node to host a VM. Besides denoting the capability of a node to host a VM, the GRES plugin also denotes the type of VM which includes the number of cores and memory of the Virtual Machine. Assume that a user requires to initiate a job using 4 Virtual Machines with 2 cores each and 512MB of memory, the *srun* command to do so is **srun -gres=vm:c2m512:4**. The number of cores and memory required for each VM are "encoded" in the generic resource type. This *srun* command asks for the generic resource of "vm" which a GRES Type of "c2m512" and the number of those resource instances is 4. For this plugin to work, the select plugin must be set to "cons_res" as seen in the Slurm configuration file in Section 2.4. This will allow the allocation of CPUs and Memory instead of whole nodes as it is done by the "linear" select plugin.

4.2.2 Management of Virtual Machines Images

We assume for simplicity and speed that every node which is capable of hosting a VM has pre-installed VM images and the root filesystems. Those VM images and root filesystems are pre-installed by the system administrator in each node. Thus it is not required for the image and the root filesystem to be copied to every node during the launch of such a job. The *slurmctld* keeps logs of these images that are in use. So additional modifications had to be implemented in the *slurmcltd* that reserve VM images during job submission and mark them free them at termination.

Two structures were added in *slurmcltd* for the management of VM images, the "vm_images" and "vmlist" structures. Any node can host up to k VMs simultaneously, where k is the number of CPUs. Thus, we assume that every node has at least k number of VM images. The "vm_images" structures is a 2d array that shows which VM image is allocated to which job. The "vm_images" array is initialized during *slurmcltd's* initialization. The size of its first dimension is the number of nodes and the size of its second dimension is the number of VM images of each node. The "vmlist" structure is a 2d array created every time a job requests for resources. It is sent from *slurmctld* to the nodes allocated to the job and contains which images will be used. The size of its first dimension is the number of nodes allocated for the job whereas the size of its second dimension is the VM images allocated on each one of the nodes. The use of both of these structures is demonstrated in Section 4.3.

4.2.3 Launching Virtual Machines

As mentioned in the previous section after making Slurm aware of the VMs the next step is to launch and manage the VMs. In our approach this is handled by VLITO, Virtual Load Injection TOol. VLITO is a lightweight tool developed in FORTH that spawns VMs and allows communication among them on different host machines (HMs) of the same cluster. VLITO's input consists of the following:

1. template VM images that are cloned for each new VM installed on a HM
2. configuration file with minimal description for each HM
3. configuration file for all installed/known VMs

The first two items from above, are pre-installed in every node capable of running VMs, denoted by the GRES plugin. The last configuration file is created by *slurmcltd* and is different for every job.

4.3 Running Workloads in Virtual Machines

After discussing the components developed for VIMA-Slurm the actual procedure of running workloads in the Virtual Machines is also presented. The main steps followed by VIMA-Slurm to run workloads in Virtual Machines are:

- Job and Resource Allocation
- Powering on the VMs
- Launching tasks in the VMs using SSH
- Job Completion and Powering off the VMs

4.3.1 Job and Resource Allocation

As also seen in Section 2.5, before running a job the resources must be allocated by *slurmctld*. First, the user interacts with *srun* to issue a job using the "vm" generic resource as depicted in Figure 4.1. The new arguments of *srun* should be parsed accordingly and a new field is added to the job information structure. The information of a job contains different job parameters such as number of nodes, numbers of CPUs. In our case it will contain an additional field that shows whether the job requires a VM to run. Thus *srun* was modified to create the new information for the job and also send it to *slurmcltd*.

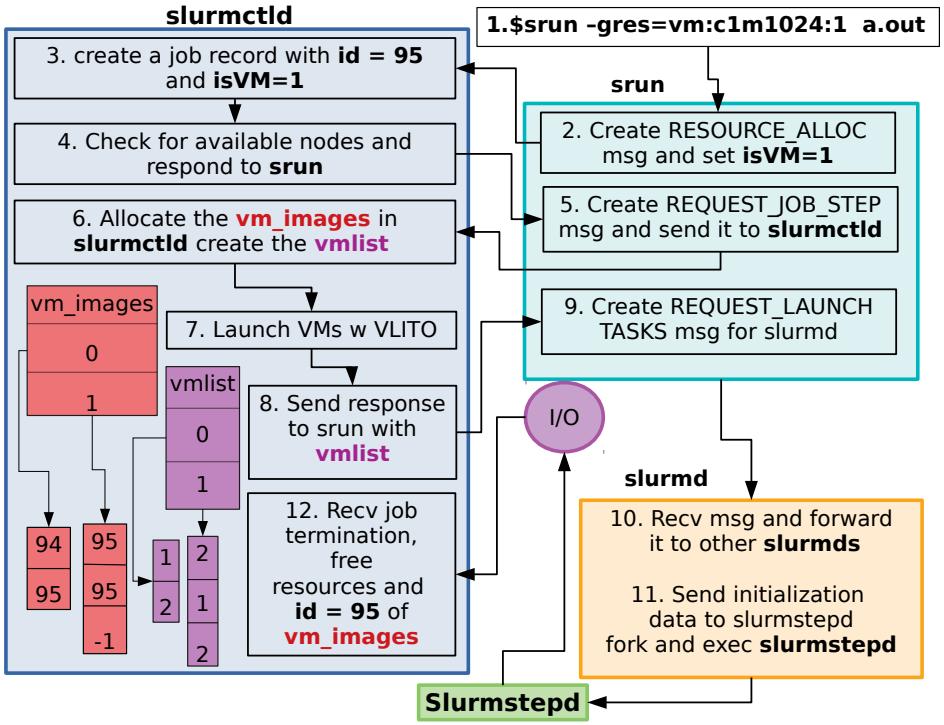


Figure 4.1: Launching a job using VMs in VIMA-Slurm

As seen in 2.5 Slurm components use RPCs to make requests. Each different RPC uses different types of messages. A message type is defined by the header. For example the RPC for resource allocation uses a message with a header

"REQUEST_RESOURCE_ALLOCATION". Besides the header a message has also body. The body of the message has different fields depending on the header. For example, a message with header "REQUEST_RESOURCE_ALLOCATION" will have fields concerning the number of CPUs allocated, memory, etc. Whereas a "SRUN_JOB_COMPLETE" will only contain the job id. In VIMA-Slurm it was required to add an extra field in the body to denote whether the job requires a VM. The header field was not modified. Slurm creates and reads the messages using special pack and unpack functions. Since an extra field was added, for VIMA-Slurm, it was required to modify those functions accordingly.

After receiving the message from *srun* (step 1), *slurmcltd* validates the availability of the resources and creates a job record with a specific id (step 2). First the job allocation is granted as shown in step 3 and then the task layout is determined (step 5). Remember, the extra resources handled by VIMA-Slurm are the images of the VMs. The management of VM images can only take place after the task layout is determined (step 5). This is because it is necessary to know which tasks are distributed to which nodes in order assign the corresponding VM images to the tasks on each node. The management of VM images is done by the use of a 2D array called "vm_images" where the first dimension is the node id and the second dimension is the vm image id in the node.

The example of Fig. 4.1, step 6, shows that the "vm_image" 1 of node 0 is allocated to a job with id 94 while image 2 of node 0 and images 1 and 2 of node 1 are allocated to job with id 95. The length of the second dimension is the same as the number of CPUs in the node. This array is initialized by setting every cell to -1 and when an image is allocated the cell is set to $jobid$.

Finally, an additional structure also created the, "vmlist", Fig. 4.1 step 6, which will be sent to all nodes allocated to the job. This structure is also a 2D array that contains the Virtual Image IDs required for each task of the job. Each index (except 0) of "vmlist" refers to a node and image ids that will be used by that node, index 0 is reserved for the number of images. In Fig. 4.1, the "vmlist" shows that node 0 will run one VM and will use the image with ID 2, while node 1 will run 2 VMs using images with ID 1 and 2.

4.3.1.1 Powering on the Virtual Machines

The main tool used for powering on the VMs and setting communication among them is VLITO. As mentioned earlier, it requires two configuration files. The first configuration file contains the special subnet that assigned to each node, this file is called "hm_map". It must be pre-installed in every node that is capable of running a VM. The second file called "vm_config" contains the ip of nodes that will launch the VMs. It also includes the number of VMs, the CPU and memory of each VM. In VIMA-Slurm, *slurmctld* is modified to supply the "vm_config" configuration file to VLITO. Next, *slurmctld* calls VLITO to launch the VMs and setup the communication between them. VLITO uses the QEMU system emulator to run the VMs [8]. Once the VMs are up and running *slurmctld* will respond back to

srun (step 7). It should be noted that at this point the VMs will be running but no workload runs in them yet.

4.3.1.2 Launching tasks in Virtual Machines

The response sent from *slurmstepd* at step 8 in Figure 4.1 is received by *srun*. This response also contains the "vmlist" structure which will be forwarded to *slurmd* and *slurmstepd* in the following steps. It then proceeds to create a request to *slurmd*, as it discussed in Section 2.5. *Slurmd* forwards the message to the *slurmds* that are running on the nodes allocated for this job and also forks and executes the *slurmstepd*. *Slurmstepd* will manage to run the tasks in the Virtual Machine(s) using the **ssh** in Virtual Machine of the local node. A representation of the execution of *slurmstepd* is provided in Figure 4.2.

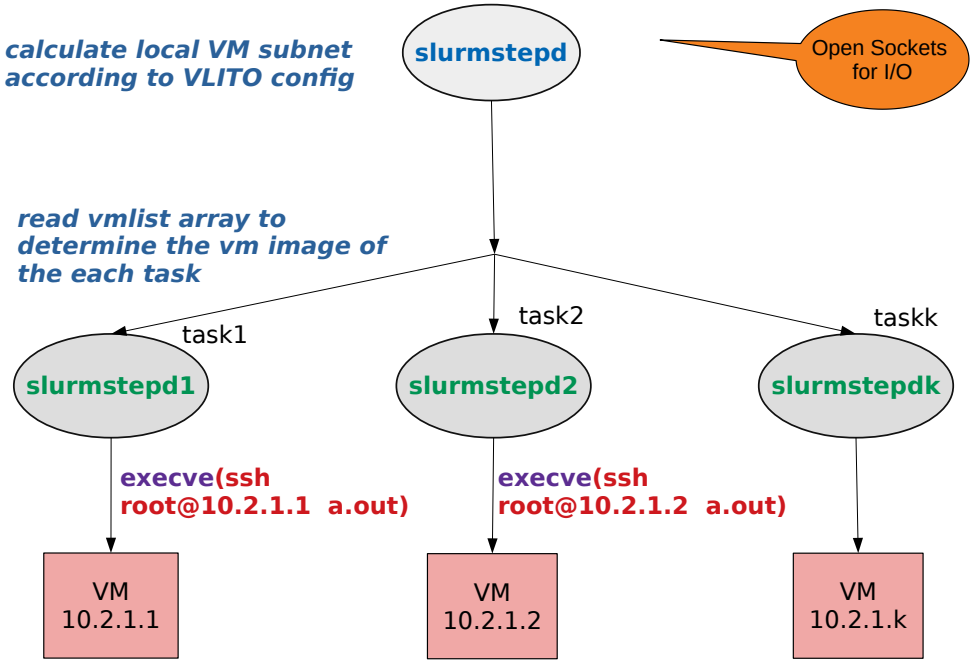


Figure 4.2: Task launch in Virtual Machine with VIMA-Slurm

Instead of using **execve** to execute a task on the physical machine, as seen in Section 2.5, *slurmstepd*, will use **execve** with **ssh** to run the task in the VM. The **execve** is used instead of bare **ssh** in order to have the same flow with the standard Slurm which simplifies the job termination. Remember for each task a different *slurmstepd* is created, thus for each VM a different *slurmstepd* is responsible. The *slurmstepd* is aware of the subnet used by the VMs that run on the node, using the same configuration file VLITO uses. It is also aware of which images are allocated for this job by using the information of the "vmlist". Each *slurmstepd* will read

from different index of the "vmlist" structure. Thus, using the subnet and the image id, *slurmstepd*, knows the ip of the VM allocated for this task and uses **ssh** to execute the task in the corresponding VM.

The procedure here is the same as described in a previous Section 2.5. The additional task for *slurmcltd* is to mark the the corresponding VM image IDs from the "vm_image" list as free. This is simple since the job completion message sent from *srun* carries the job id which as we saw in 4.3.1.1 was used to mark a VM image as allocated. Using the job id the corresponding VM images from the "vm_image" structure are de-allocated (set to -1). To power off the Virtual Machines VLITO is called again by *slurmctld*.

4.4 Summary

In this chapter we present VIMA-Slurm, a Slurm extension that enables running workloads in VMs. VIMA-Slurm allows the user to run a job using either physical or virtual resources. Besides launching VMs and running tasks in them, VIMA-Slurm also manages the VM images. This differentiates VIMA-Slurm from other similar efforts that integrate the VM technology to Slurm. The user interface provided by VIMA-Slurm is simple since the user can explicitly ask for virtual resources and it requires not further scripts.

For the implementation of VIMA-Slurm, several modifications deep in the main code tree of Slurm were made. About 800 additional lines of code were added in the main code tree. The Slurm components modified are:

- *slurmcltd*, has the bulk of the modifications for managing the VMs.
- *srun*, modifications were made to overcome the limitation of GRES scheduling
- *slurmstepd*, using **ssh** and "vmlist" structure

The rest of the modifications were done in the communication protocol of Slurm. The VM GRES plugin is about 100 lines of code and VLITO is 700 lines of code.

Chapter 5

TOpology and Fault Aware (TOFA)-Slurm

A common expectation for jobs or batches of jobs submitted to an HPC cluster is having as low completion time as possible. Generally, a batch job consists of multiple individual jobs. Lower completion time for a batch job benefits both the provider and the user of an HPC system. As already discussed in Section 1.2, a common standard used by parallel applications in HPC is the MPI. Thus, a substantial number of jobs that would execute in an HPC system use the MPI standard. An MPI job can consist of many parallel MPI processes, where each of them can run on a processing unit. Those parallel processes may spend their time in three main parts; I/O, computation and communication. Lowering the completion time of an MPI job can be accomplished by speeding up any of the three parts of aforementioned. The parallel processes exchange data by communicating directly with each other. However, the MPI applications and the standard MPI libraries are platform agnostic in order to be portable. Thus, at large system scales, the communication cost between the parallel processes can have a significant impact on the performance of the application. Also, additional pressure in the interconnect is imposed which results in higher energy consumption. A common way of addressing the issue is by improving the *Data Locality* of parallel applications. The *Data Locality* refers to the way that data are placed, accessed and moved by the processing elements of the underlying architecture.

Additional overhead may arise in the case of failures where part or the whole jobs needs to be restarted. Transient link faults may require several retransmissions for example from the underlying interconnect to recover from the error. Node failures during and MPI job will result in the whole job being aborted and restarted. As also referred in the MPI standard [36], the default behavior upon an error in an operation other than file-related ones is to abort the parallel program. The study of [45] demonstrates that, hardware is responsible for more than 50% of failures in a system. Furthermore, it is shown that specific nodes exhibit much higher failure rate than others, hence there is a certain pattern of failures. The probability of a

running job to be aborted depends on the execution time and the time between failures in a system. The authors of [45] find that the average failure rates might differ wildly across systems, ranging for 20 to 1000 for a year. Also, the authors of [27] state that *Faults are becoming a normal "feature" of the current parallel computers. Computing mappings that are able to cope with failures is therefore of high interest*

In this chapter we present the TOpology and Fault Aware (TOFA) process placement approach and its integration in Slurm. This is our third extension of Slurm called TOFA-Slurm. TOFA-Slurm aims to reduce the batch job completion time in two different ways: The first way is to reduce the cost of communication of an MPI job. The second way is the reduction of job abort ratio due to node failures. avoid faulty nodes which would result in individual job abortion, increasing the overall time of a batch job. As mentioned earlier, an approach to reduce the communication cost is by improving the data locality of the parallel application. An efficient way to achieve this to determine a matching or mapping of the application's communication pattern to the target system topology. This is also referred to as the topology mapping problem. To reduce the the job abort ratio, TOFA-Slurm monitors the nodes of the system and attempts to avoids allocating those who are considered as faulty.

5.1 Contributions

In this chapter we discuss a new process placement approach for MPI jobs with the dual goal of reducing job abort rate while also reducing the communication cost between the parallel processes. The process placement approach is called *TOpology and Fault Aware* and is the first contribution of this chapter. The second contribution is a Slurm extension that implements the TOFA process placement approach called *TOFA-Slurm*.

The *TOpology and Fault Aware* process placement models the underlying topology of the system as well as the communication pattern of the application as two separate graphs. The communication pattern is obtained via application profiling by using the MPIAPT profiling tool which is an external tool. The mapping of the communication graph to the topology graph is created by an external graph mapping library called *Scotch*. This approach to the topology mapping problem is commonly used by other related work. However, unlike other related work, TOFA also takes into account node failures and produces mappings that cope with node failures. This way, the TOFA approach also reduces the job abort rate, of MPI jobs, due to node failures. We achieve this by monitoring the system's nodes and keep a record that shows past failures. This information is incorporated into the topology graph so that the graph mapping library will not prefer allocating the faulty nodes.

The second contribution is the *TOFA-Slurm* extension. *TOFA-Slurm* incorporates the TOFA process placement approach. Although standard Slurm is aware

of the nodes that are out of service it does not keep track of past failures for them. Also, standard Slurm can be topology aware but it is not aware of the application's traffic pattern. For implementing *TOFA-Slurm* we have developed five different plugins. Those plugins model the underlying topology of the system, monitor the system's nodes, keep a record of them and more importantly call the *Scotch* graph mapping library which solves the graph mapping problem and returns the corresponding mapping. Finally, we have implemented a heuristic that attempts to find a set of consecutive nodes without failures to further assist the graph mapping library in avoiding the faulty nodes.

5.2 Related Work

There is a wide range of different approaches, concerning different stack layers, for improving the performance achieved by MPI applications. There are studies related to hardware, systems software like resource managers , the MPI library and the application itself. A taxonomy of related studies can be derived by considering the type of feedback they use, that is, feedback related to the system architecture, or topology and information regarding the communication pattern or profile of a specific job.

5.2.1 Approaches using no topology- or application-related feedback

Several studies aim at optimizing the MPI implementation as a means for improving application performance. Work in this direction does not rely on topology or application-related information. Those approaches optimize different aspects of the MPI implementation. Such as aspects are; optimizing point-to-point communication, tune the MPI implementation for HPC or RDMA-capable interconnects or improve the performance of collectives. Work on optimizing point-to-point implementations; Authors of [23] attempt to increase the MPI communication-computation overlap by separating the meta-data exchange from the application data exchange. This is done with the implementation of communication library that replaces only key data exchange calls in MPI applications. The authors of [42] propose a speculative MPI Rendezvous protocol that could increase the communication to computation overlap. The work of [51] improves the communication latency of MPI applications implementing one-sided point to point communication primitives. Additional work to take advantage and tune over high performance or RDMA-capable interconnects; The authors of [55] implement a point-to-point methodology that can support concurrently multiple network types (e.g. Myrinet, Infiniband, GigE) and can utilize multiple NICs. Also the work of [34] presents an RDMA-based MPI implementation over Infiniband. The authors of this RDMA-based MPI implementation claim that it benefits not only large but also small and control messages. Another work of [19] seek to improve MPI performance for the case of derived data types. The authors of [19] improve the performance of

derived data types by using packing algorithms that are optimized for the current memory architecture. Several studies target at deriving algorithms for collective operations that improve their performance. The authors of [41] proposed five new algorithms for the Reduce and Allreduce operations of MPI depending on the vector size and number of processes. The work of [52] attempts to minimize the latency of short messages and the bandwidth use of large messages by optimizing the allgather, broadcast, all-to-all, reduce-scatter, reduce and allreduce MPI collectives. The work of [43] presents a method to improve the Allreduce MPI collective using message pipelining hardware. Multiple studies seek to derive optimal implementation of collectives targeting a specific machine system such as [38]. There is also work for improving collectives exploiting machines features such as [28] where authors improve the performance of Broadcast collective utilizing the native Infiniband multicast. All the approaches mentioned here do not use topology or application related feedback. However, as we see in most cases they perform optimizations on specific systems by exploiting additional features offered by them.

5.2.2 Approaches using only topology-related feedback

There are approaches that utilize knowledge regarding the topology or system architecture. Such approaches focus on deriving topology aware or improving the implementation of collective primitives for specific platforms. Authors in [44] for example, suggest non-minimal algorithms for allgather and reduce-scatter primitives for the case of Clos, single-, and multi-ported torus networks. Additional work along this direction aims at improving the performance of the MPI runtime by deriving topology aware collectives. Such work is [16] where the authors optimize some MPI collectives (Barrier, Brodcast, Alltoall and Allreduce) for a BlueGene/L Systems. The authors of [32] developed a library to optimize collective communication by minimizing the ammount of data sent through slow links.

5.2.3 Approaches using topology and application related feedback

Approaches that use topology and application are more related to the TOFA process placement approach. Authors in [29] for example, address the problem of mapping arbitrary communication topologies to arbitrary-heterogeneous machine topologies with the goal of minimizing the maximum congestion and average dilation. To calculate the mapping the authors combine several techniques including greedy heuristic, recursive bisection mapping and graph similarity.

A similar approach to the presented is referred in [22] where a communication graph of the MPI application is derived by profiling the application. The edges of this graph represent the communication bandwidth of any given pair of processes. The communication graph is mapped to a topology graph that is similar to our approach to minimize the communication cost. However, this approach does not attempt to incorporate fault related information in the topology graph and it is

not integrated in Slurm

The work of [26] also follows the same principles using a different mapping technique called *Treematch*. Although they integrate their work in Slurm, the authors do not follow a Fault Aware approach and the evaluation they present is based on workload traces instead of MPI applications. The work presented in [56] incorporates the influences of the underlying communication protocols to the mapping problem. Thus, the profiling information used also takes into account the communication protocol.

The approaches discussed so far are profile guided, that is, they assume that the underlying communication pattern of a specific problem/application is available. There are approaches though that do not carry this dependence. Authors in [35] for example, explore four heuristics to perform rank reordering for realizing run-time topology awareness for the case of the MPI's Allgather primitive. The corresponding approach does not rely on an application's profile. Instead it is based on the communication pattern exhibited by each algorithm used in the allgather primitive. The work of [50] developed a framework to detect the topology and speed of the underlying Infiniband network. Also they modified the MPI library to query this topology detection service and redesigned the broadcast algorithm to take into account the topology related information and adapt dynamically.

5.3 The TOFA process placement approach

The Topology and Fault Aware process placement approach attempts to optimize the placement of parallel processes of an MPI job by exploiting both application and topology related information. In accordance with the most relevant studies, we formulate the process placement problem as a topology mapping problem. Two different graphs are required, one that models the topology of the system and another that models the application's communication pattern. Both are complete and undirected graphs. The topology mapping problem is NP-Complete [27]. Thus, in order to solve it several heuristic methods are employed. Our method consists of the following steps:

1. Gather the communication graph of the corresponding application
2. Create a topology graph that models the underlying system's architecture
3. Monitor the system's nodes and maintain a record with fault related information (i.e. which nodes exhibited failures)
4. Post process the topology graph taking into account the fault related information
5. Compute the mapping between the communication graph and the post-processed topology graph using *Scotch* graph mapping library

In order to gather the communication graph of the application our approach relies on profiling. The tool used for application profiling is MPIAPT which is an external tool and is discussed in Section 5.5. We model the communication graph as a undirected complete graph $G = (V_G, E_G)$. Each vertex $v_g \in V_G$ corresponds to an individual process of the MPI job while an edge $e \in E_G$ connecting vertices u_g and v_g denotes communication between the corresponding processes. More precisely, the edge weight represent the traffic volume between them. The edge weight could also represent the amount of messages but in this case we use the traffic volume. The output of the profiling tool is a $N \times N$, where N is the number of processes involved in the MPI program. The element i, j of this matrix represents the sum of bytes sent from MPI rank i to rank j and the bytes sent from j to i. This matrix can be used to build the communication graph G of the application.

It should be noted that the TOFA process placement approach is profile-guided. This means that for deriving an assignment of MPI processes to nodes, a training run should be carried first to derive the corresponding communication graph. However, this cost can be amortized by performing multiple runs of the same application using the same input or configuration. The main drawback of profiling is that if the number of processes or input data changes the profile of the application also changes. This means that if those two parameters change it is required to execute another profiling run for the application.

The underlying platform is modeled through host graph $H = (V_H, E_H)$. Each vertex $v_h \in V_H$ corresponds to one of the platform nodes. Vertices carry no weight. Let R denote the routing logic of the underlying platform. Having assumed a 3D torus topology with fixed routing, routing function $R(u, v)$ provides the list of links (along with source and destination of each link) that are traversed by a message sent from node u to v. The number of hops traversed to reach v starting at v is used to denote $w(e_{u,v})$ which corresponds to the weight of $e(u, v) \in E_H$.

As also described in the introduction, the main difference of the TOFA process placement approach from similar studies is that, assignment of processes to nodes also takes into account node failures. The fault model assumed in this study is the following. Nodes exhibit transient failures independently of each other. Such failures may be the result of a hardware fault or reboot. When a node enters the failed state, it is incapable of performing both computation and communication. This means that, it can not send, receive, or forward traffic on behalf of other nodes. Consequently, communication attempts initialized by the MPI library will result in an error and in turn, job abortion. Moreover, when a node is in the failed state, it does not respond to heartbeat messages. In this way, the corresponding Slurm module that collects heartbeats is able to infer node outage. More details regarding the heartbeat mechanism will be deferred until Section 5.4. Avoiding nodes that fail frequently is expected to decrease the probability of a job being aborted. The importance of node failures becomes even more prominent for applications that have large running times. However, a strict policy of avoiding failed nodes may force selecting a more sparse subset of available nodes. This in turn, may have

an adverse effect on communication cost. There is thus a trade-off, between the abort ratio tolerated and the communication cost. This means that, a more loose placement approach, with some tolerance for node failures, may strike a better balance between abort ratio and completion time for a batch of jobs.

TOFA process placement approach attempts to capture the effect of transient node failures and incorporate it to the topology graph. In the TOFA placement approach, we approximate the effect of node failures on the cost of traversing a path as follows: we assign larger weights to paths that include nodes that have a non-zero outage probability.

Assume that outage probability is available for each node. We employ routing function $R(u, v)$ mentioned above, to infer the list of links that are traversed by a message sent from $u \in V_H$ to $v \in V_H$, where H corresponds to the topology graph. For each link $l \in R(u, v)$, let l^s and l^d denote the origin and target of that link respectively. Using this information, a registry is created whose input is a node id and the output of it is the list of paths that this node serves as an intermediate hop for. Combining information provided by the routing function and node outage probabilities, we update edge weights in the topology graph through Equation 5.1.

$$w(e_{u,v}) = \sum_{l \in R(u,v)} c + c \times 100 \times \mathbb{1}[(p^f_{l^s} > 0) \vee (p^f_{l^d} > 0)], \tag{5.1}$$

In the above equation, $p^f_{l^s}$ and $p^f_{l^d}$ are the failure probabilities for nodes l^s and l^d respectively. Constant c denotes the cost in terms of number of hops. Moreover, $\mathbb{1}[p^f_{l^s} > 0]$ denotes an indicator function whose value becomes one if $p^f_{l^s} > 0$. The rational for the above equation is the following: if either of the two nodes involved in a link has an outage probability other than zero, the cost of that link is set to 100 instead of one. In this way, the cost of that path becomes significantly higher than the cost of traversing the longest path (in terms of number of hops) of the platform. Part of the future work related to this study is to better understand the sensitivity of TOFA regarding the way in which path costs are updated.

The final step is to create the mapping of the communication graph to the post processed topology graph. We solve this graph mapping problem by using an external tool called *Scotch*. *Scotch* is a graph mapping library and its details are discussed in 5.6. Before the two graphs (G and H) are fed to *Scotch* our heuristic attempts to find N consecutive faultless nodes. Where N is the number of nodes required by the application to run. *Scotch* maps the uses that list of consecutive nodes if the heuristic succeeds and maps the G graph to H. If the heuristic does not return a list of nodes *Scotch* will use all available nodes to create the mapping. A pseudo-code describing the steps followed by TOFA is described in the listing 5.1.

Listing 5.1: TOFA process placement approach

```
Input: G
Graph modeling an application's communication
pattern
Input: H
Graph resembling the topology. Edge weights
estimated through Equation 5.1
Output: T
    T = <Process Id, Node Id>

procedure TOFA(G, H):
    S=Find |V_G| consecutive nodes s.t. p_n^f = 0, ∀n ∈ V_H
    if S = {∅}: then
        T := ScotchMap(G, H)
    else
        H_S := ScotchExtract(H, S)
        T := ScotchMap(G, H_s)
    end if
    return T
```

As TOFA pseudo-code shows, the input to the TOFA process placement approach consists of the communication graph G and the topology graph H. Note that, edge weights in the topology graph are updated according to Equation 5.1 for capturing the effect of node failures. The output of TOFA is set T which has one entry per process. Each entry consists of two values, one corresponding to the process id and a second one denoting the id of the node where that process is assigned. In step 10, TOFA first searches for a set of consecutive nodes that have zero outage probability. If this set is non-empty then, the set of $|V_G|$ faultless nodes is passed as input to Scotch library along with the topology graph H. Scotch then produces a new topology modeled through graph H_s which is a subset of the original topology depicted through H. This functionality is denoted as *ScotchExtract* in step 14 of the above pseudo code. In step 15, Scotch is used then to map the communication graph G in H_s. If TOFA is unable to identify $|V_G|$ consecutive faultless nodes, Scotch is used to map the communication graph into the topology graph in step 12.

5.4 TOFA-Slurm Architecture

This section describes the technical details of the TOFA-Slurm implementation. For the implementation of TOFA- Slurm we have developed various Slurm plugins. This way, TOFA-Slurm is implemented without modifying with the main source code tree of Slurm, making it compatible with future Slurm versions. TOFA-Slurm also relies on two external tools, MPIAPT for application profiling and Scotch for solving the graph mapping problem. Those external tools are described in the following Sections 5.5 and 5.6.

For $TOFA$-Slurm we have developed five different plugins which are described henceforth. The *NodeState* and *LoadMatrix* SPANK plugins run in every compute node, whereas *FAT Topology*, *Fault Aware Slurmctld* and *Fault Aware Node Selection* (FANS) plugins run only in the controller node. A general picture of the TOFA architecture and the plugins developed is depicted in Fig. 5.1.

5.4.1 LoadMatrix SPANK plugin

The first step of our method requires the communication graph or *Guest* graph (G), which is obtained via profiling and is provided by the user as an extra argument during job submission. For this purpose we have developed a plugin the LoadMatrix SPANK plugin. The *LoadMatrix* SPANK plugin enables *srun* to have an extra argument which can be used to provide the file that contains the communication graph G of the desired application. Thus, the user can submit a job using *srun* and also provide the file path of the G as shown in Fig. 5.1. This information will be sent to *slurmctld* where the mapping will take place.

5.4.2 NodeState SPANK plugin

On each compute node this plugin is responsible for responding periodically to the heartbeats sent by *Fault Aware Slurmctld* plugin which runs in the controller node. This is depicted in Fig. 5.1. The *NodeState* plugin is SPANK plugin that runs in *slurmd* context, thus it runs only when the daemon is initialized or terminated. During the initialization a separate thread is created that is responsible for responding to the controller.

5.4.3 Fault Aware Slurmctld plugin

Fault Aware Slurmctld is responsible for periodically polling the compute nodes through heartbeats. As seen previously, *NodeState* SPANK plugin is responsible for responding to those heartbeats. The heartbeat of node i at interval t is denoted as $Hb(t, i)$ in 5.1. The responses of each node are gathered in a 2D array which contains the recent heartbeats of every node. Absence of a reply to a heartbeat is translated as node outage. This 2D array is is referred to as the Hb array in 5.1 and is used to defer the node outage probability. The Hb array is sent to *Fault Aware Node Selection* plugin for post processing the topology graph, at job submission time. Then, node outage probabilities can be combined with output from the routing function $R(u, v)$ according to Equation 5.1 to update edges weights in the topology graph.

5.4.4 Fault Aware Torus Topology (FATT) plugin

This plugin is responsible for creating the initial topology graph and the providing routing function $R(u, v)$. The creation of this graph takes place during *slurmctld* initialization. Although Slurm offers a topology plugin for 3D Torus our approach

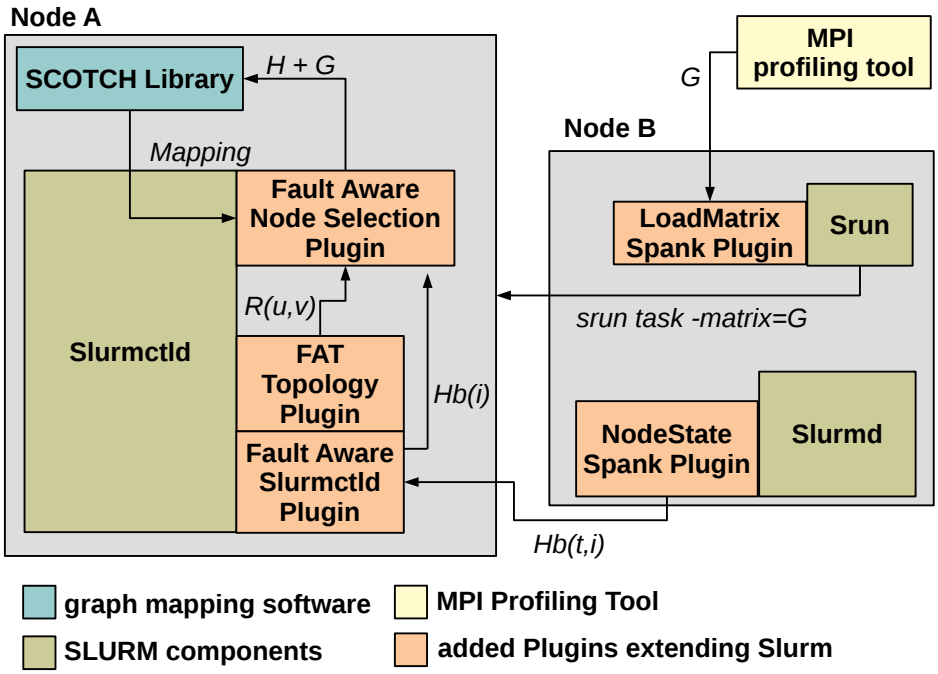


Figure 5.1: TOFA-Slurm Architecture

requires additional information which are not available. The default topology plugin does not have routing information which is included in our implementation. This plugin reads a topology file and creates the topology of the system based on the information provided in that file. This file contains a $N \times N$ matrix that represents the topology of system. N is equal to the number of nodes and each element of the matrix represent the communication cost between two nodes. This plugin assumes a 3D Torus with fixed xyz routing based on the routing function function $R(u, v)$. Hence, utilizing the information of the topology file and $R(u, v)$ the $FATT$ creates a list to represent the topology of the system. This list includes all the intermediate nodes traversed for routing traffic from a node u to a node v. topology file and the routing this plugin. The topology of the system is a list where every element is a node structure. A node structure contains a list with the distances to all other nodes. Additionally, it contains the id of the intermediate nodes of a path. This information is required by the plugin *Fault Aware Node Selection* along with the Hb array structure to decide which paths are affected when the quality of a node changes. *Slurmctld* reads the list of nodes and creates the "node record table" which is the main structure used to allocate nodes. This information is processed by the various *node selection* plugins when a job requests for resources. This plugin can be provided as a framework to create other topologies for TOFA-Slurm.

5.4.5 Fault Aware Node Selection plugin

The core functionality of $TOFA$-Slurm is implemented by this plugin:

- Receives the communication graph G of the application
- Collects the topology graph H
- Collects the Hb array with fault related information
- Post process the H graph with the Hb array
- Calls Scotch to create the mapping

At job submission time this plugin requests the Hb array that is maintained in *Fault Aware Slurmctld*. Then uses Hb array to post process the H graph from $FATT$ plugin which represents the topology of the system. The post processing of H updates the edges of according to the Equation 5.1. If a node is assumed as faulty using the information of Hb array it's edges will have an increased weight in the H graph. In order to decide which edges of the graph are affected when the quality of a node changes, this plugin uses the list of the intermediate nodes provided by the $FATT$ plugin. Otherwise, the intermediate nodes should be recalculated every time.

To further improve the quality of the mapping and increase the chance of avoiding faulty nodes we have developed a heuristic that attempts to find consecutive nodes that are not faulty. This heuristic will create a list of preferred vertexes which is a subset of the system's available nodes. This list of preferred vertexes is used by *Scotch* along with the G and the H to create the mapping. If the heuristic fails to find the consecutive nodes then *Scotch* will use every available node to create the mapping.

Scotch graph mapping library is called with the G, sent by *LoadMatrix* plugin, the H graph Fig. 5.1 and the list of preferred vertexes. *Scotch* produces an array that represents the mapping of the processes to the system's nodes. The *graph_map*() of *Scotch* that produces the mapping also requires a mapping Strategy. TOFA uses a combination of strategies provided by *Scotch* and we refer those strategies in the following section 5.6. The output of *Scotch* is a one dimensional array where the index is the MPI rank of the application and the value of each element is the node id. The final structure to be updated is the node selection bitmap. This is also a one dimensional array that is used by Slurm to denote which nodes are allocated to the specific job. The index of this array represents a node id and the contents can be either 1 or 0. 1 denotes that the corresponding node id is allocated to the specific job whereas 0 denotes the opposite.

5.5 Profiling Tool for MPI Applications

As already discussed in the previous section, the TOFA process placement approach relies on a communication graph that captures and represents the traffic

exchanged between each process pair. In this section an overview of this tool is provided along with details on how the corresponding communication graph is produced.

The profiling tool used, named *MPIAPT - MPI Application Profiling Tool*, is a dynamically linked library that intercepts all calls to MPI primitives that initiate traffic, such as, point-to-point, collective, and one-sided communications ones. MPIAPT is not sample based. Instead, it captures every call to MPI primitives that initiate traffic and keeps track of the amount of traffic exchanged for each pair of processes, both in terms of volume-bytes and number of messages. The output of MPIPATH consists of two complete graphs, namely, G_v and G_m. Each of these graphs is represent as a matrix of size $N \times N$, where N is the number of processes involved in the MPI program. Graph G_v captures the number of bytes exchanged for each pair of processes while graph G_m captures the corresponding number of messages. For the case of G_v for example, element $G_v(i, j)$ captures the total bytes sent from process with rank i to rank j. For the case of collective primitives MPIAPT is tuned to emulate the appropriate algorithm for each collective in order to accurately capture the traffic exchanged between each pair of processes during each phase of that collective's schedule. As it will also be discussed in Section 6, for the evaluation of *TOFA*, we used SimGrid [47]. This simulator assumes that each collective operation is implemented through a specific algorithm. For the case of *MPI_Bcast* for example it assumes a binomial tree based approach. Assuming a broadcast involving eight ranks that receive a single integer (4 bytes) from root zero. As Table 5.1 shows, broadcast proceeds in three phases, with different ranks active in each phase. In the second one for example, ranks 0 and 4 send to ranks 2 and 6, respectively.

Phase	Active Pairs
1	$(0, 4)$
2	$(0, 2)$, $(4, 6)$
3	$(0, 1)$, $(2, 3)$, $(4, 5)$, $(6, 7)$

Table 5.1: Schedule of a binomial tree based broadcast with 8 ranks and root=0

Assuming an MPI program with a single call to *MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD)* involving eight ranks, the corresponding G_v and G_m graphs will be the ones, assuming MPI_INT is 4 bytes, shown in Table. 5.2 and Table. 5.3 respectively.

MPI rank	0	1	2	3	4	5	6	7
0	0	4	4	0	4	0	0	0
1	4	0	0	0	0	0	0	0
2	4	0	0	4	0	0	0	0
3	0	0	4	0	0	0	0	0
4	4	0	0	0	0	4	4	0
5	0	0	0	0	4	0	0	0
6	0	0	0	0	4	0	0	4
7	0	0	0	0	0	0	4	0

Table 5.2: Communication Graph with bytes of MPI_Bcast

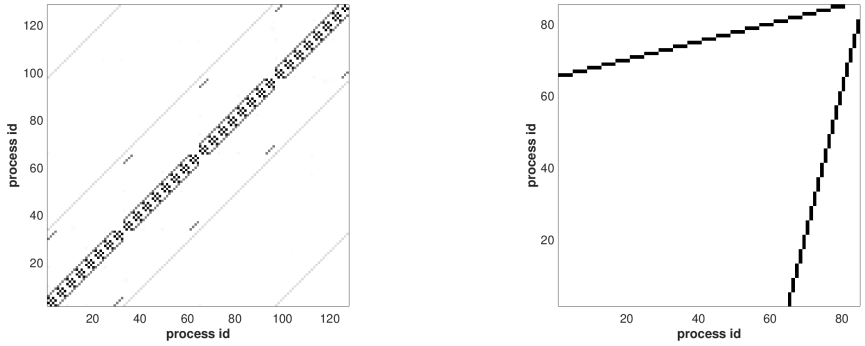
MPI rank	0	1	2	3	4	5	6	7
0	0	1	1	0	1	0	0	0
1	1	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	0	1	0	0	0	0	0
4	1	0	0	0	0	1	1	0
5	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	1
7	0	0	0	0	0	0	1	0

Table 5.3: Communication Graph with messages of MPI_Bcast

The above tables exclude the messages exchanged
during the initialization phase of the MPI. Note than the main diagonal is always 0 since it represents the communication of a process with itself.A key feature of MPIAPT is that it records traffic through communicators other than the default one, For correctly updating G_v and G_m, the rank of a process in communicator other than *MPI_COMM_WORLD* is transformed to the rank in *MPI_COMM_WORLD* (R_{comm_world}), and counters in G_v and G_m are updated based on R_{comm_world}.

Furthermore, we have developed an additional feature for *MPIAPT* to assist us in understanding the traffic pattern via visual inspection. This feature produces heatmaps of the application's communication pattern depicting the amount of bytes exchanged between parallel processes of the application. Using the communication graph produced by *MPIAPT* we create such heatmaps to better understand the general traffic pattern of an application and essentially characterizing it as regular or irregular. Two examples are shown in the figures below.

X and y axis denote the process id and the amount of bytes exchanged is is depicted by the gradation of color, the darker the color the higher the amount of bytes exchanged. The Fig. 5.2a shows the traffic pattern of LAMMPS. We categorize this traffic pattern as regular because most of the traffic is concentrated



(a) (b)

Figure 5.2: a)regular traffic pattern, b)irregular traffic pattern

in the main diagonal. In Fig. 5.2b Data Traffic benchmark from NPB suite (NAS Parallel Benchmarks). As we see from Figure 5.2b the main diagonal of the graph shows no traffic. Thus, we categorize the Data Traffic benchmark as an application with irregular traffic pattern. Using similar to the above heatmaps we are able to make a crude categorization of the applications between regular and irregular.

5.6 SCOTCH overview

In Section 5.1 it was mentioned that the topology mapping problem is solved using the *Scotch* software. As we already discussed in 5.3 the topology mapping problem is NP-Complete and thus it approached by various heuristics. Scotch is a library that provides methods for graph mapping and graph partitioning [46]. Fundamentally, graph mapping is the procedure of assigning each vertex of a source graph to every vertex of a target graph. In our case the source graph is the communication graph of the application and the target graph represents the topology of a system. This graph mapping method is also referred to as static since it is computed prior to the execution of the program.

To compute an efficient mapping *Scotch* uses the Dual Recursive Bi-partitioning algorithm, as it is referred in [39], the algorithm recursively allocates processes to processors. More specifically it starts with two different sets, a set of processors which represents all the processors of the system and a set of processes which represents all the processes of the application. At every step the set of processors is partitioned in two disjoints sets and maps the subset of processes associated with the two processor sets using a graph bi-partitioning method. The recursion stops when all processor sets contain one processor. The graph partitions method can be defined by user. Scotch offers various different graph bi-partitioning methods such as Fiduccia-Mattheyses algorithm, greedy and random algorithms. Scotch also allows for several methods to be used consecutively. Each method is able to

use the result of the previous resulting in multiple refinements of the graph.

There are several different graph mapping libraries. The most popular are mentioned in [31] where they are also compared. *Scotch* was our choice of software for a two reasons. The first reason is that from [31], it seems to have a low overhead to create the mappings. The second reason is that the mappings produced are also of better quality. This means that the mappings produced are more effective in reducing the communication overhead of the application, as demonstrated in [31].

5.6.1 Scotch time to map

Firstly, as it is illustrated in[31] *Scotch* seems to be performing reasonably well compared to other Graph mapping software, presented in this particular work. To measure the scalability of *Scotch* we have mapped a number of processes ranging from 512 to 16k to a 32x32x16 Torus which consists of 16K nodes. From figure 5.3 it takes 80ms to map 512 processes and 88 seconds to map 16k processes.

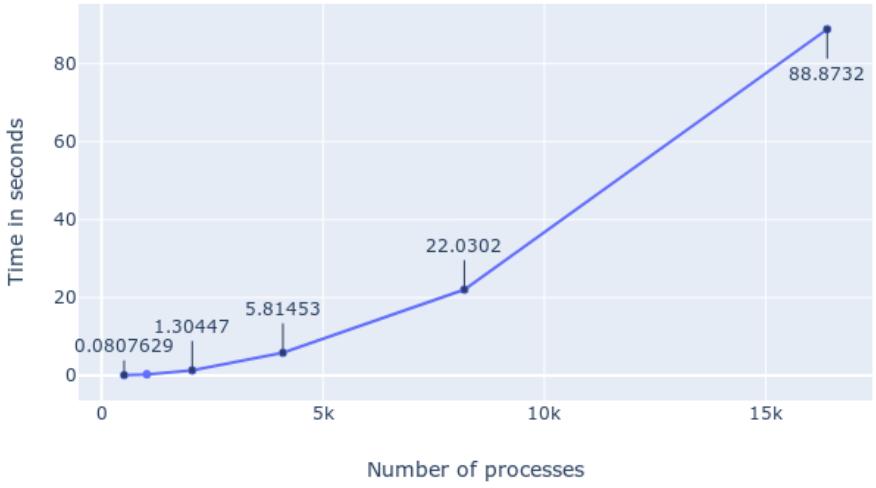


Figure 5.3: Scotch Time to map in a 32x32x16 Torus

In this example we observe that by doubling the number of processes the time to map is quadrupled. This rate seems to be constant at least for the number of processes we were able to test. For the applications we used in the following evaluation section 6 the overhead of mapping is much lower than the benefit gained from the reduced communication cost.

Two main routines of the above tool are used, the *amk_grf* routine which creates the decomposition defined architecture from the topology graph *H*. This

decomposition defined architecture describes the actual topology graph with additional parameters that are required by Scotch. The next routine that is used is *gmap*, which produces the actual mapping. *Gmap* uses the the decomposition defined architecture produced by *amk_grf* and the communication graph to create the mapping. Scotch, by default, uses a combination of methods which partition the communication and the topology graph. The main method used is called multi-level method which coarsens the graph multiple times in order to compute a partition. Coarsening the graph multiple times means that in every iteration the size of the graph is reduced until the graph is small enough to compute a partition. This is done by grouping the vertexes and edges into super-vertexes and super-edges. The partition calculated then can be projected to the original graph. Two other methods are also applied to the graphs in order calculate a partition are; the Fidducia-Mattheyses and the Greedy Graph growing heuristics. More information about the above methods can be obtained from the manual [46].

Chapter 6

Evaluation

The evaluation of TOFA process placement approach is divided in two parts and is done using a simulated environment. The first part evaluates Scotch, the graph mapping library to which TOFA heavily relies on for deriving the mappings of the processes to the corresponding topology. In this part, we measure the performance and study the various parameters that affect Scotch. Such parameters are, the 3D Torus topology arrangement, the algorithm of the MPI collectives and the traffic criteria. We also study the sensitivity of Scotch to the application's input. In the second part, TOFA is evaluated in an environment where node failures are emulated and as a result MPI jobs can abort. This part is considered as the complete evaluation of TOFA because it shows us whether TOFA has achieved both of its goals; reducing the communication cost and the abort ratio of MPI applications. We use real unmodified MPI applications and benchmarks which are simulated using the SimGrid simulator. Before presenting the simulation results we also perform a crude characterization of the MPI applications based on their traffic patterns.

6.1 SimGrid

For the evaluation of both *TOFA* and *Scotch* we rely on SimGrid [47], which is a simulator for Distributed Computer Systems. There are several reasons for using a simulated environment instead of a physical system. The first one is that it offers a convenient way of simulating node failures. Secondly, multiple scenarios can be run in parallel. Also, it enabled us to experiment with different topologies which is not trivial in a real system.

SimGrid, more specifically, is a framework for the simulation of applications that execute on distributed systems. Within the SimGrid framework, the SMPI [25] interface is capable of simulating unmodified real MPI applications. In SMPI, the communications of the application are intercepted by the simulator, thus simulated whereas the computations are executed on the host machine (i.e., emulated).

To run an MPI application in the simulated environment it is required to

provide the simulated platform, which describes the system to be simulated. This is also referred to as virtual platform. The main components of a simulated platform are the links, the hosts and routes. A host is a machine with a defined computing capability in flops, in our case it is 6 Gflops. Links have two main attributes, the bandwidth and latency which in our platform are 10 Gbps and 1 µs respectively. The route consists of links between two distinct hosts and the topology of the simulated environment is defined by the routing which, in our experiments, is a 8x8x8 Torus. The 8x8x8 Torus refers to a 3D Torus with 8 nodes on each dimension, it can also be referred as a 8-ary 3-cube topology [54]. Three different Torus topologies are shown in Figure 6.1

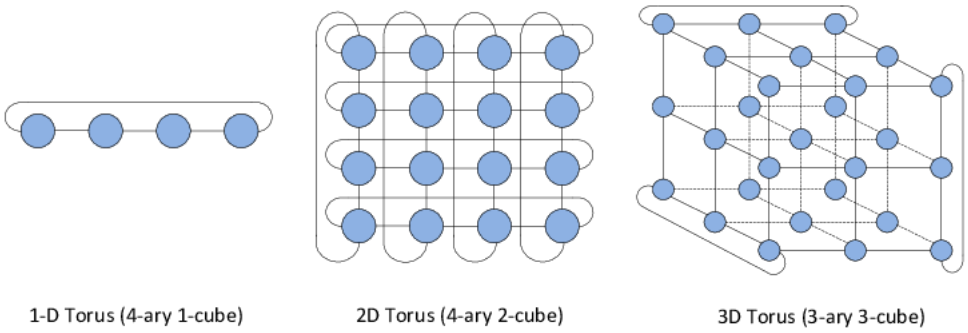


Figure 6.1: Torus topologies

Those parameters were chosen in order to allow us for meaningful simulations given the problem size but also reflect a realistic HPC system. The bandwidth is sufficiently low in order to derive meaningful simulation results. High bandwidth values would mask out the effect of communication cost on job completion time. The platform description that is fed to SimGrid also lists the route used for each pair of nodes. In this way we ensure that the topology simulated matches exactly the topology assumed for deriving the mapping of processes to platform nodes.

As stated before, we have chosen the simulation approach as it also enabled us to simulate faults. The simulated platform we described can provide this feature by assigning a trace to any link. The trace assigned to link contains the link's capacity at various simulation time stamps. By setting the capacity of any link to zero it enables us to simulate faults at any desired time in the system when an application is running on our simulated platform.

6.2 MPI Applications

The performance of the *Scotch* tool and *TOFA* (for the second part) is explored via the following MPI applications: HPL, LAMMPS, NPB-CG, NPB-DT. HPL[30] is the High Performance Linpack that solves a linear system in double precision

arithmetic and provides virtual broadcast topologies, bandwidth reducing swap-broadcast algorithm and various other features. LAMMPS[33] is an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator. It can model atomic, polymeric, biological, solid-state metals, granular or macroscopic systems using a variety of inter-atomic potentials and boundary conditions and it was designed to run efficiently on Massively Parallel systems. The simulation used in this context is the dynamics of rhodopsin protein in solvated lipid bilayer. NPB-CG is the Conjugate Gradient benchmark from the NAS Parallel Benchmarks (NPB) [37], which computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. It employs unstructured matrix vector multiplication and long distance communication. The other benchmark from the NPB suite, NPB-DT is the Data Traffic benchmark which falls in the subcategory of unstructured computation, parallel I/O, and data movement benchmarks. It basically operates with graphs and evaluates the communication throughput.

6.2.1 Traffic patterns of the applications

As mentioned in 5.5 we have extended the profiling tool to create heatmaps of the applications' traffic pattern. Visual inspection of the heatmap that depicts the traffic pattern of the applications allows us to perform a crude categorization as *regular* and *irregular* as mentioned in 5.5. This categorization is important as our TOFA process placement approach attempts to reduce the communication overhead by optimal placement of the processes. As we will demonstrate in the following section the benefit the TOFA process placement also depends on the degree of irregularity of traffic pattern of the applications. For our evaluation we chose applications with both regular and irregular traffic patterns in order to be complete. The first application to be presented is NPB-CG. NPB-CG exhibits irregular traffic as depicted in the Fig. 6.2. The x and y axis represent process id and the dots depict the amount of traffic exchanged between them. The darker the color of a dot the more the larger the volume of traffic exchanged between two processes.

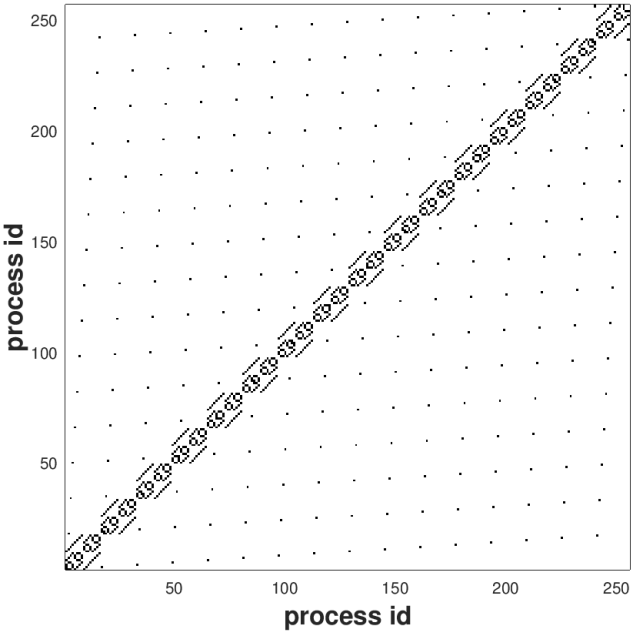


Figure 6.2: Traffic pattern of NPB CG 256 processes class D

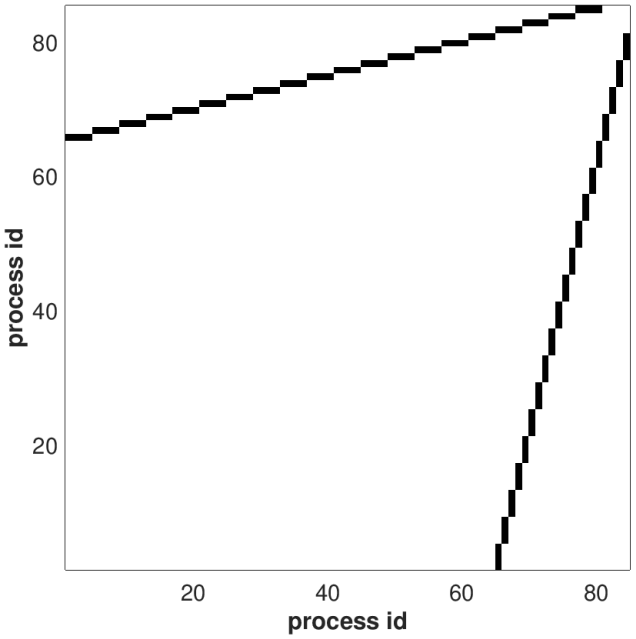


Figure 6.3: Traffic pattern of NPB DT 85 processes class C

As it is shown from Fig. 6.2 most the traffic is done in the main diagonal. However, a fair amount of traffic is not in the main diagonal which leads us to categorize this application irregular. A second application with irregular communication pattern is NPB-DT which is also mentioned in section 5.5. Figure 6.3 depicts its communication pattern where we can observe that the main diagonal has no traffic. Thus, we can derive NPB-DT benchmark has highly irregular communication pattern. The next two benchmarks exhibit a regular traffic pattern, the first presented is HPL, which, in addition to the regular traffic pattern, is also computation intensive. The heatmap depicting the traffic pattern of HPL benchmark with 256 processes is presented in Fig. 6.4. Another benchmark with regular traffic pattern is LAMMPS in Fig. 6.5. The traffic pattern of LAMMPS is presented in a previous section Figure 5.2a where only 128 processes are shown. Generally, the patterns presented do not change by increasing the problem size or the number of processes. We categorize both LAMMPS and HPL traffic patterns as regular due to the fact that most of the traffic takes place in the main diagonal. The main reason for using these benchmarks is to contradict the performance of Scotch and TOFA to the default round robin allocation policy of Slurm. Slurm's allocation policy iterates over the available nodes on a sequential manner. As a result, it is most probable for processes with ranks in some range $[i-k, i+k]$ to be placed on topologically nearby nodes. On the other hand, the default placement policy used by Slurm, is not expected to perform well on an irregular communication pattern like the one exhibited by NPB-DT.

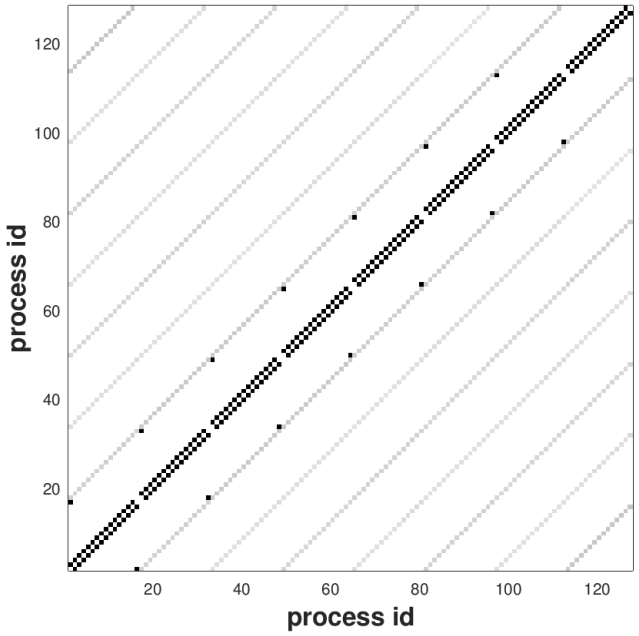


Figure 6.4: Traffic Pattern of HPL 128 processes

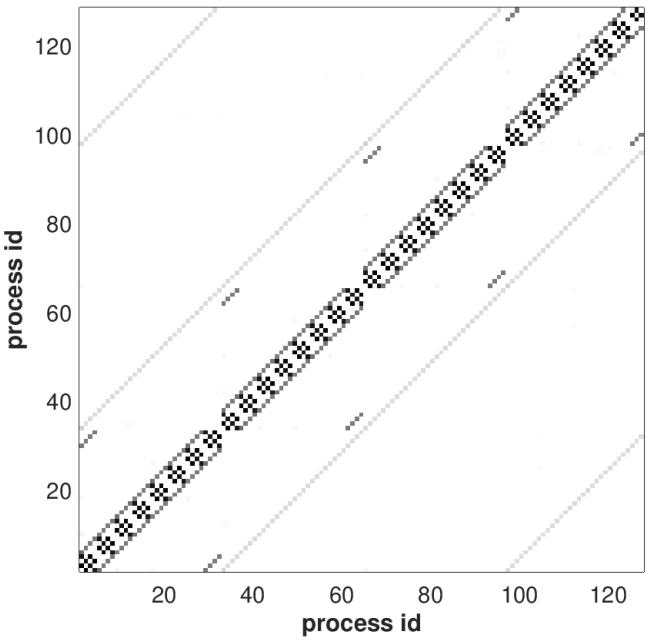


Figure 6.5: Traffic Pattern of LAMMPS 128 processes

6.3 Evaluation of Scotch

To measure the performance of *Scotch* graph mapping library we use it to map the communication graphs of the MPI applications to the simulated topology. Then, based on the mappings, the simulator will select specific hosts of the simulated platform to run the MPI applications using SMPI 6.1. We compared *Scotch* with the following process placement approaches:

- Default-Slurm
- Random
- Greedy

The *default* placement policy refers to the allocation of nodes to processes employed by the default version of Slurm. In the *random* approach, each node is assigned one rank, randomly selected. The *Greedy* placement approach, sorts all different process pairs in terms of total traffic exchanged. Then, it iterates over all pairs, starting from the one with the higher volume. The goal is to place the processes involved as close as possible starting from a distance of one hop.

Finally, for the first part of this evaluation, we explore the effect of various parameters on the mappings produced by *Scotch*. Those parameters are; the 3D Torus topology arrangement, the MPI collective algorithms, the traffic criteria

and the effect application's input on the its traffic profile. The main metric we use for performance is the output of the applications. For the NAS Parallel Benchmarks we use the execution time as a performance metric. Whereas for the HPL benchmark we use the Gflops that is reported by HPL and for LAMMPS we use the timesteps/s. The last two metrics are more accurate due to the fact that LAMMPS and HPL have very low completion time. A secondary metric that we also present is the *Average Hops per Byte* [26]. This metric provides us with an insight regarding the mapping quality before performing a full simulation of the application.

6.3.1 Average Hops per Byte metric

The average Hops per Byte is the average number of links every byte will traverse to reach its destination. It can be calculated using the topology graph H, the communication graph G and the corresponding mapping of G to H. The authors of [26] use the *Hop-Byte* metric which is similar to the *Average Hops per Byte* metric. The average hops per byte can be calculated as:

$$averageHopsPerByte = \sum_{0 \leq i,j < n} \frac{v(i,j) \times d(i,j)}{\sum_n v(i,j)} \tag{6.1}$$

where n is the number of processes, $v(i,j)$ is the volume exchanged between two processes and $d(i,j)$ is the number of hops between the computing resources $p(i)$ and $p(j)$ where processes i and j are mapped to. Low value for this metric suggests that the network traffic uses less hops. Which results in lower communication cost and lower energy consumption [27].

Benchmark	Processes	Default	Scotch	Greedy	Random
NPB-CG (class D)	256	2.68	2.07	2.48	5.01
NPB-CG (class C)	64	2.83	2.0	2.37	3.91
NPB-CG (class C)	32	2.73	1.66	1.73	4.37
NPB-DT	85	4.79	1.75	1.76	3.16
HPL	256	2.33	2.18	4.09	4.85
HPL	128	1.73	2.20	3.59	5.21
HPL	64	1.65	1.65	3.81	4.18
HPL	32	1.21	1.21	3.46	3.94
LAMMPS	256	1.19	1.47	3.69	3.22
LAMMPS	128	2.83	1.78	4.0	3.65
LAMMPS	64	2.83	1.78	3.9	3.78
LAMMPS	32	2.44	1.70	3.56	3.75

Table 6.1: Average Hops Per Byte for the mappings of different applications

From the above table 6.1 a rough impression of the mapping quality for every application can be obtained. In most cases *Scotch* reduces the average Hops

per Byte compared to all other approaches. However, as we see in the cases of HPL the reduction is marginal. This is due to the regularity of the application's communication pattern and as it is evident the Hops per Byte are already low (between 1 and 2). In the case of LAMMPS with 256 processes we observe that *Scotch* fails to reduce the average Hops Byte. This is also shown in the simulation of this experiment and this behaviour is explained in later sections.

6.3.2 Results of the experiments comparing different approaches

In this section we present the results of the simulations for the different process placement approaches compared to Scotch. For the NAS Parallel Benchmarks we use the execution time as a performance metric. Whereas for the HPL benchmark we use the Gflops that is reported by HPL and for LAMMPS we use the timesteps/s. We observed marginal variability in the simulation results reported by SimGrid. Thus, it was not needed to perform many simulations for every application in order to obtain the results. The only exception to this is the *random* approach where we performed 10 runs with 10 random mappings and present their average.

Fig. 6.6 depicts the execution time of the NPB-CG class D benchmark with 256 processes using the four different approaches. As seen in figure 6.6 *Scotch* provides

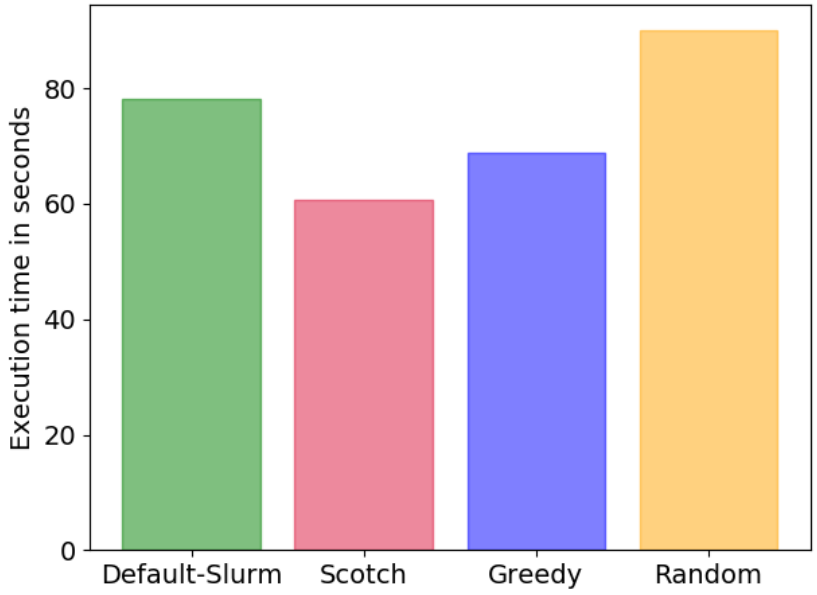


Figure 6.6: Execution Time for NPB-CG class D with 256 processes

a 1.3x speedup over the Default mapping, this is attributed to the fact that the application exhibits an irregular communication pattern. Also, it is communication intensive as it is shown in table 6.2, where we demonstrate that it spends more

than 50% of the total execution time in communication. Our Greedy approach also shows a better execution time compared to default but worse than by *Scotch*. Random mapping seems to be the worst case in this benchmark.

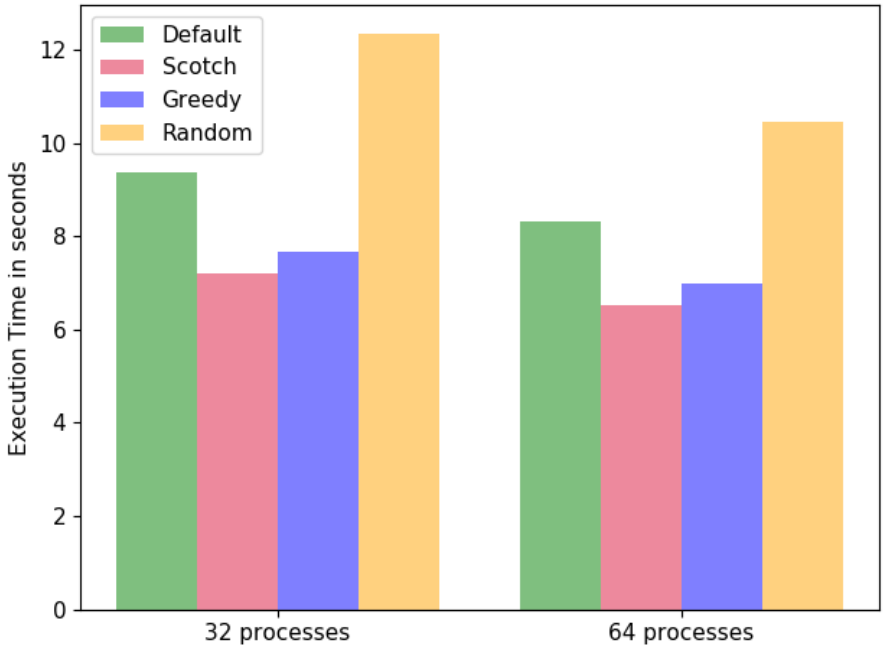


Figure 6.7: Execution Time for NPB-CG class C

We also present NPB-CG class C, which is a smaller problem size compared class D, with 64 processes and 32 processes in Figure 6.7. These results are similar to those of Class D. Thus, Scotch succeeds in reducing the communication cost of the application. The benefit is substantial even if the problem size is smaller and with less processes.

The next application is NPB-DT class C which also exhibits a more irregular pattern than CG as seen previously in figure 6.3. The speedup provided by *Scotch* over the Default-Slurm is 1.22x and all methods show a better execution time than the Default. Again our Greedy approach peforms almost as good as Scotch but as it is shown afterwards that is not always the case. According to table 6.1 Scotch improves the average Hops per Byte by 2.73x. Whereas the improvement for NPB-CG is 1.35x. However, NPB-DT shows less speedup in execution time than the NPB-CG benchmark. Both applications use only MPI_Send and MPI_Recv primitives for communication. Through SimGrid we are able to track the average time spent in MPI_Send primitive and report it in table 6.2. From table 6.2 and the execution time graphs presented for those two benchmarks Fig. 6.6 and Fig. 6.8 we can derive the following: For NPB-DT the execution time of the Default mapping is 30 seconds while the Time spend in MPI_Send primitive is

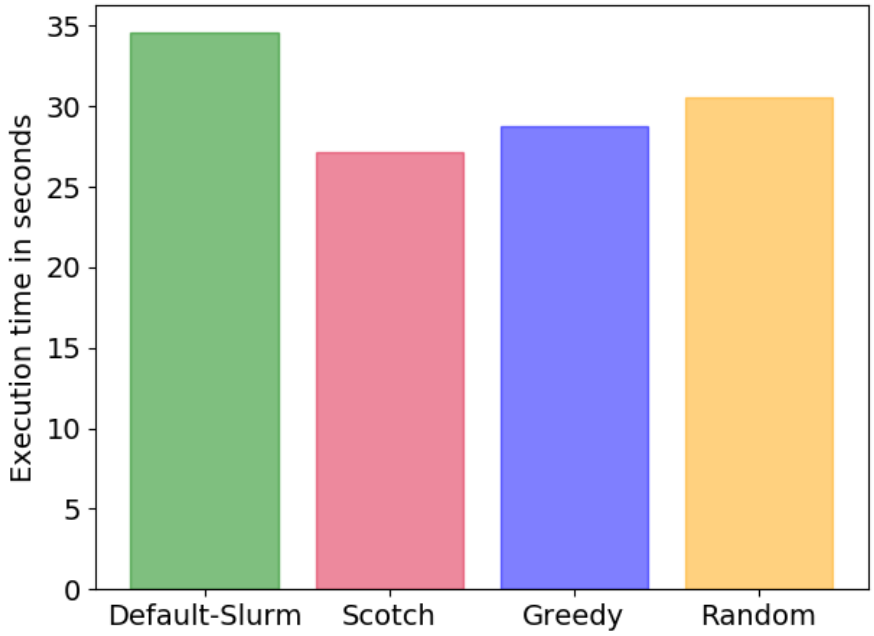


Figure 6.8: Execution Time for NPB-DT

Benchmark	Default	Scotch
NPB-CG	40	30
NPB-DT	7.5	5

Table 6.2: Seconds spent in MPI_Send primitive

7.5 seconds that is 25% of the total execution time. For NPB-CG the execution time is 78 seconds while the Time spend in MPI_Send is 40 seconds which is about 50% of the total execution time. By reducing 7.5 seconds to 5 seconds and 40 to 30 we have the same reduction in relative terms. However NPB-CG is more communication intensive thus the impact on the total execution time is larger. A formula to approximate the expect speedup can be described as:

$$Speedup = \frac{1}{1 - comRatio + \frac{comRatio}{comSpeedup}}$$

$$comSpeedup = \frac{average(Hops\ per\ Byte\ before\ placement)}{average(Hops\ per\ Byte\ after\ placement)}$$

Where the *comRatio* is the communication to computation ratio and the *comSpeedup* is the speedup of time spent in communication. We can derive that *the more communication intensive and irregular an application is the more it can benefit from process placement optimization.*

The next application is HPL which is run with 32, 64 and 128 processes.

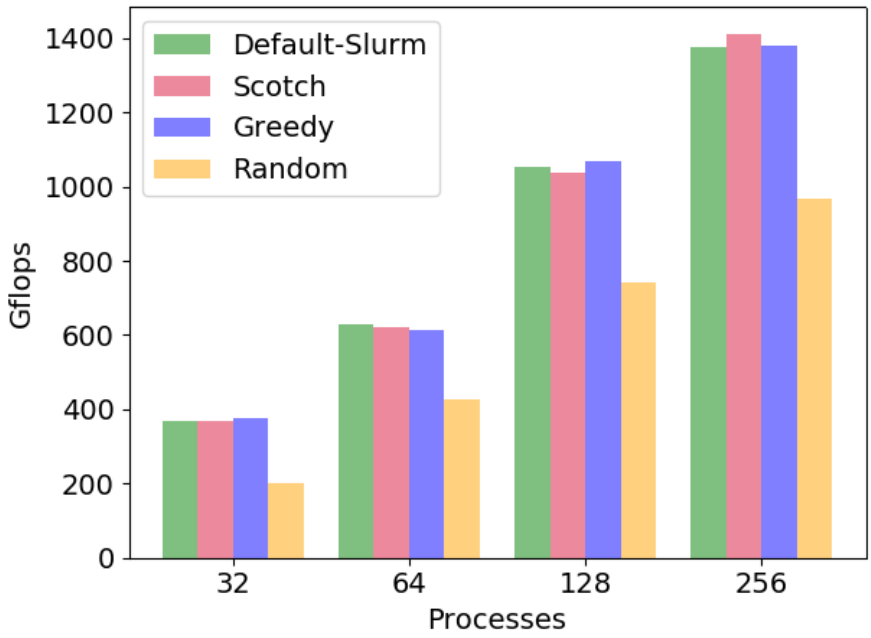


Figure 6.9: GFLOPS for HPL(higher is better)

As depicted in Figure 6.9 the Greedy and Scotch process placement methods have no impact on the performance of the HPL benchmark. The main reason that the process placement methods have no impact is that the communication of this specific benchmark contributes to small fraction of the application's total execution time. This is shown by the profiler which outputs the time spent in communication and computation. This profile was obtained by running HPL in a single node. HPL exhibits a regular communication pattern which benefits the round robin placement of Default-Slurm in the 3D Torus Topology. On the contrary, this does not benefit the Random policy because it scatters the processes to the processing elements and as a result the distances between them are longer

Random seems to have worse even though the fraction of communication is small, if the pairs with heavy communication, as we seen in Figure 6.4, are placed in longer distances it can have a significantly negative impact on the application's performance. Thus, even though process placement would not benefit this application it should be noted that a random process placement has a negative impact on its performance.

The application presented next is LAMMPS, which also has a regular communication pattern. Unlike HPL, LAMMPS uses MPI collectives and generally exhibits heavier communication as reported by the profiling tool. The performance is measured in timesteps per second (timesteps/s). LAMMPS can be configured

to run for several timesteps, the default configuration will run with 100 timesteps. Each timesteps LAMMPS concludes specific calculations, thus by decreasing the application's execution time, timesteps/s will be increased. Figure 6.10 shows the performance of LAMMPS with 32, 64, 128, and 256 processes for 1000 steps. We

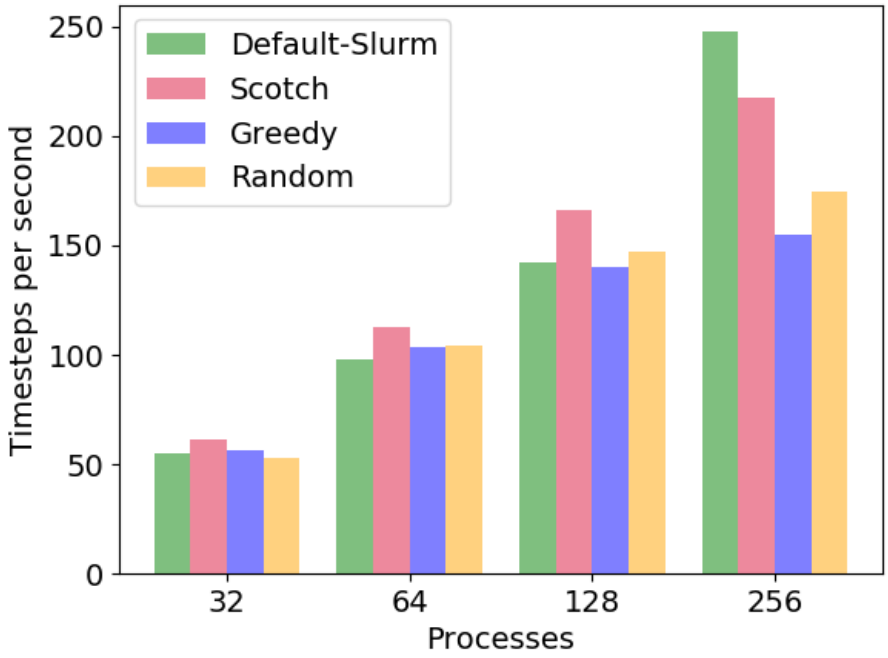


Figure 6.10: Timesteps per second for LAMMPS(higher is better)

notice, by these results, that *Scotch* performs better for every case except the 256 processes. Also as the number of processes increases the benefit from optimal placement seems to be increasing too. This is only true for *Scotch* as the rest of the methods seem to perform worse or about the same as the Default. Scotch, compared to Default Slurm process placement, reduces the completion time of LAMMPS with 32 processes by 10%, for 64 processes by 12.5% and for 128 the completion time is reduced by 16%. Since we know that traffic pattern does not change the reason for this benefit is the increase of communication to computation ratio as the number of processes increases. To explore the communication to computation ratio we used the SimGrid simulator and traced the time spent in computation and the various MPI primitives.

Figure 6.11 shows the time spent in computation and the MPI primitives. From Figure 6.11 that LAMMPS spends 70% of the execution time in computation. However, in Fig. 6.12 where more processes are involved,the computation ratio drops to 56%. The two charts Fig. 6.11 and 6.12 use only 100 steps of LAMMPS. To explain the behaviour of LAMMPS with 256 processes first we will

refer to the table 6.1 shown earlier where it is demonstrated that *Scotch* fails to reduce the average Hops per Byte of LAMMPS with 256 processes compared to Default, in fact it increases them. This results in a worse mapping than the Default which is also reflected in the actual performance. The average Hops per Byte for the Default-Slurm are already significantly low (1.19) as shown in the table. The Default-Slurm seems to be performing optimally in this case. From this benchmark we see that the mapping produced by Scotch are not flawless. We provide with more information regarding this this exceptional behaviour in the rest of this section.

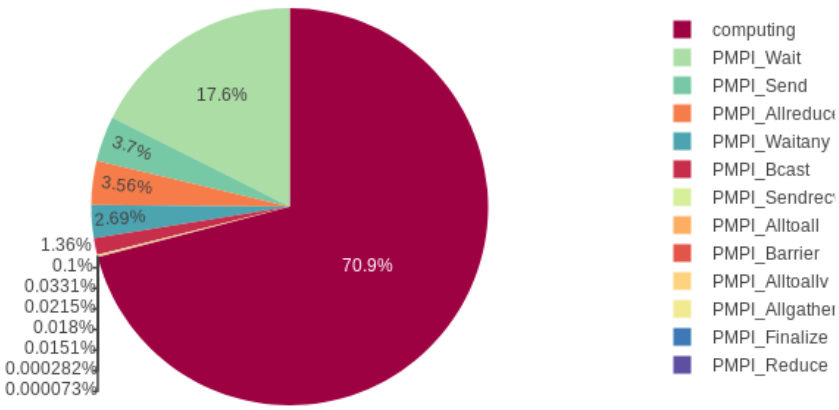


Figure 6.11: LAMMPS time spent in different primitives 32 processes

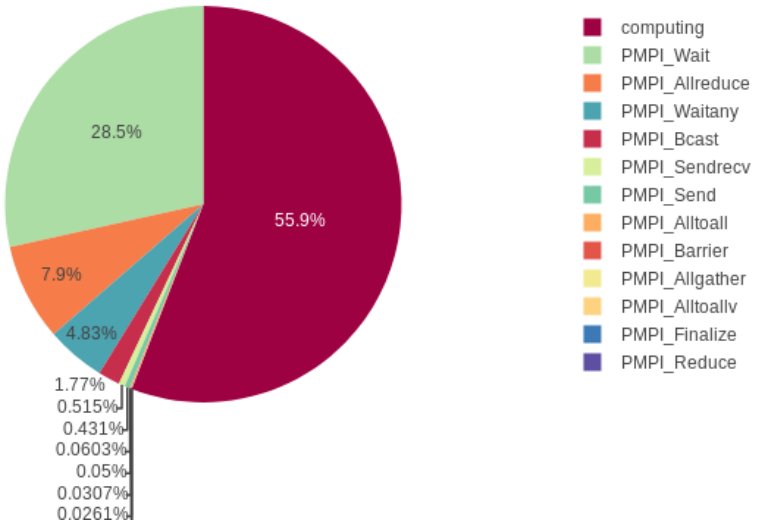


Figure 6.12: LAMMPS time spent in different primitives 128 processes

To this point we have explored the benefit, in terms of performance, for various categories of applications. As we saw, applications with irregular traffic pattern seem to benefit more from an optimized process placement. Whereas the benefit for the applications with regular traffic patterns seem to have less or no benefit. Another application characteristic that has to be taken into account is the communication to computation ratio. An example for this claim is the NPB-CG and NPB-DT applications whereas NPB-DT seems to have a highly irregular pattern but benefits less than NPB-CG. In the example of HPL and LAMMPS which both exhibit a regular communication pattern, HPL does not benefit from an optimal placement while LAMMPS does. This is mainly to the fact that LAMMPS is more communication heavy than HPL. However, for the case LAMMPS with 256 processes we saw that Scotch underperforms compared to the Default-Slurm. In the remaining of this section we will explore how several parameters affect Scotch and explain this exceptional behaviour of mapping LAMMPS.

6.3.3 Effect of torus topology

In this section we explore the effect of the underlying topology and its arrangement on the performance achieved by the resource selection of Default Slurm and the Scotch. As also mentioned in Section 6.1, the topology simulated is a 3D torus. Simulations of different scenarios though reveal that the arrangement of the underlying torus has a significant effect on the performance achieved by the

aforementioned resource allocation strategies. We present an analysis of the topological distances on some steps of the Broadcast collective, with the binomial tree, in table 6.3

Topology	Phase 1		Phase 2		Phase 3		Phase 4	
	Def.	Scotch	Def.	Scotch	Def.	Def.	Def.	Scotch
8x8x8 Torus	2	3	1	3	4	5	2	6
4x8x16 Torus	4	2	2	2	1	1	4	3

Table 6.3: Distances of communicating nodes between the Broadcast phases

The table 6.3 shows that for the 8x8x8 Torus topology *Scotch* places the nodes in sub-optimal distances compared to the Default. However, that is not the true when using a different Torus topology. The Default mapping fails to optimize the placement compared to *Scotch* for the 4x8x16 Torus. This is attributed to the fact that the Default approach has no information about the underlying topology. Figure 6.13 summarizes the performance reported by LAMMPS, with 256 processes in terms of timesteps per second, for five different 3D Torus topologies with 512 nodes. The first one is the 8x8x8 Torus topology used for all the experiments shown in this evaluation section. Although, all the topologies have the same number of nodes they have a different diameter. The 8x8x8 Torus has the shortest diameter, the maximum path is 12 hops whereas for the 4x32x4 Torus the diameter is 20 hops.

Figure 6.13: LAMMPS timesteps/s in different 3D Torus topologies

From Figure 6.13 we can derive that since the default approach has no notion of the underlying topology its performance is heavily depended on it. As seen, performance can be degraded by up to 36% as the diameter of the underlying topology increases.

To better understand the above results we provide an analysis of both the average and the maximum time spent in Broadcast and Allreduce collectives of LAMMPS presented in table. 6.4 and table 6.5

Topology	Broadcast		Allreduce	
	Default	Scotch	Default	Scotch
8x8x8 Torus	0.48	0.54	0.52	0.58
4x8x16 Torus	0.55	0.53	0.52	0.50

Table 6.4: Average Time of Broadcast and Allreduce

Topology	Broadcast		Allreduce	
	Default	Scotch	Default	Scotch
8x8x8 Torus	0.55	0.62	0.52	0.58
4x8x16 Torus	0.67	0.64	0.63	0.60

Table 6.5: Maximum Time of Broadcast and Allreduce

LAMMPS simulation with 256 processes and 1000 steps takes more than an hour to finish in wall clock time. The benchmark reports an execution time of around 4 seconds, thus the results we see in table 6.4 indicate that, on average, 15% of the execution time is spent in those two collectives. While the average time is a useful metric for the collectives, it does not demonstrate the actual impact in some cases. As Broadcast and Allreduce collectives are blocking, the maximum time spent would give us another perspective of our results. From table 6.5 we can see that compared to 8x8x8 Torus the maximum time for both collectives is higher for the 4x8x16 Torus. This is due to the fact that the maximum path is longer in 4x8x16 Torus than the 8x8x8 Torus. In the 4x8x16 the maximum number of hops in a path is 14 whereas in the 8x8x8 Torus is 12.

6.3.4 Effect of collective algorithms

In this section we explore the effect of the algorithms that implement certain collective operations. More precisely we focus on the algorithm used in SimGrid for implementing the broadcast primitive. Profiling LAMMPS with the tool described in Section 5.5 reveals that broadcast and allreduce consuming primitives are the most time consuming. The default algorithm that SimGrid offers for broadcast is the binomial tree which proceeds in different phases (also discussed in Section 5.5). As presented in the Tables 6.3, 6.4 and 6.5 for the 8x8x8 Torus topology the topological distances of the nodes between the phases of the collectives are longer by *Scotch* compared to Default, which result in higher time spent in collectives. To eliminate the effects of the algorithm's phases we have developed an alternative broadcast algorithm that it proceeds in a single phase. This means that one node is responsible for broadcasting the messages, without intermediate nodes and phases. This alternative broadcast algorithm is also used by Allreduce primitive, thus both of the collectives proceed in a linear manner, making their traffic pattern easier to profile. The main difference of the two algorithms is that in the binomial tree one, its completion time is affected by the completion of each step with step $k+1$ depending on step k. In the case of the broadcast call described in Section 5.5) for example, rank 6 will wait to receive traffic until the transmission from 0 to 4 is completed. The topological distance between the nodes where ranks 0 and 4 are placed thus, has a significant effect in the schedule and completion time of the whole broadcast operation. This effect becomes more prominent in the case of an MPI application that makes frequent use of the broadcast primitive as the

LAMMPS one. Table. 6.6 presents the results when running LAMMPS with 256 processes with both binomial tree and linear algorithms for the collectives.

Collective Algorithm	Default	Scotch
Binomial Tree	247.563	217.291
Linear	234.429	259.911

Table 6.6: LAMMPS timesteps/s with two different collective algorithms

Scotch performs better when using the linear algorithm while the performance of the default scheme is, marginally, lower for this case. These results are expected since due to the following reasons; the linear algorithm is less sensitive to distance between nodes and does not proceed in phases that depend on each other. Besides showing better performance, there are issues with the scalability of the linear algorithm that are not explored in this context.

6.3.5 Effect of traffic criteria

In the results present in the previous section, the matrix that represents an MPI application's communication graph captures the traffic volume exchanged between each pair of processes in term's of bytes. However, the profiling tool that is used to derive the communication graph can also report the corresponding traffic in terms of number of messages. The number of messages is expected to be a suitable metric for traffic in the case where processes exchange small messages. In the case however where processes also send large messages, it is expected to mask out the heavier communication profile that certain process pairs may exhibit. As a result, *Scotch* may result in a less optimal process placement.

To solidify this above claim we present the results of two different applications, NPB-IS (Integer Sort) and LAMMPS. Both applications are benchmarked with *Scotch* but with two distinct communication graphs, a communication graph that contains only the messages exchanged between processes and a communication graph with the traffic volume exchanged. NPB-IS is an application with small messages, based on our profiling, the maximum, minimum and average message sent/received is 4 bytes. We expect this application to yield the same benchmark results with both types of communication graph. On the contrary, LAMMPS has average message size of 1.8 KB, maximum of 83.8 KB and minimum of 4 bytes.

Communication Graph	NPB-IS(exec time)	LAMMPS(timesteps/s)
Traffic Volume	12.15s	217.291
Number of Messages	12.14s	184.384

Table 6.7: Effects of different types of Communication Graphs

For NPB-IS we measure the execution time whereas for LAMMPS we present the timesteps/s as before. From Table 6.7, the execution time of NPB-IS is the

same for both types of communication graph. Thus, the communication graph using only the number of messages exchanged can be a suitable metric traffic for NPB-IS, however this is not true for LAMMPS. LAMMPS has a variety of message sizes as opposed to NPB-IS thus the number of messages is not a suitable metric of traffic in this cause. It is not trivial to predefined the type of communication graph for each application. As we saw, it requires knowledge of certain characteristics which can vary dramatically across the spectrum of MPI applications in HPC.

6.3.6 Effect of the application's input

Scotch maps the communication graph to the graph representing the topology of the system. Thus, Scotch is tightly coupled with the specific MPI job's communication profile. This means that the mapping problem should be solved again when the corresponding communication profile changes. The communication profile of the MPI job may change in the following cases. First, if the application uses a specific input or configuration file. Rhodopsin of the LAMMPS suite, relies on a configuration file that among others specifies the number of timesteps to be simulated as referred in section 6.2 Another example is NAS-CG which uses a generator of pseudo-random numbers to initialize a sparse matrix as its input. The second reason for which an application's profile may change is the variation of number of processes involved. In this section we explore the effect of both these parameters on the resource allocation derived by the proposed approach.

For all applications used to evaluate Scotch, changing the number of processes involved incurs a significant change in the corresponding communication profile. In the case of LAMMPS, the traffic between a pair of processes changes when the number of process changes. Additionally, for LAMMPS, using a different configuration file, for example increasing the number of steps also affects the mappings produced by Scotch. This is due to the fact that the traffic pattern of LAMMPS depends on the input. Thus, for the case of LAMMPS, *Scotch* is sensitive to its input from the configuration file. For exploring the sensitivity of the Scotch to the application's input we also use NAS-CG application. NAS-CG uses a pseudo-random number generator that returns positive floating point numbers, written in FORTRAN. More precisely, for the case of NAS-CG and class C we replaced this generator with another one that generates random numbers using a seed. Then by perform ten different runs of the NAS CG benchmark of class C with 64 ranks, ten communication graphs were created using the profiling tool described in Section 5.5. Each communication graph is then used to create a mapping of processes to nodes. Then, for each such mapping, we simulate the benchmark's performance and compare it with the corresponding performance of the default resource allocation policy of Slurm. We first notice that varying CG's input has no effect on the corresponding mapping. For all ten different runs, the mapping remained constant, thus *Scotch* does not seem to have a certain sensitivity to the particular input.

Our approach to the process mapping problem is profile driven, however there

are different ways which do not require a profile of the application. One such approach, offered by the MPI standard [36], enables the creation of a "process topology". MPI supports three main categories of process topologies types, Cartesian, Graph and Distributed Graph. Torus and hypercubes are considered Cartesian topology types. The basic functions include the creation of a process topology using a special function, e.g. MPI_Cart_Create and the other functions that allow the user to get information about the distance between two processes or their rank using their coordinates. It also allows for the processes to be reordered to better match the topology of the machine. Additionally, this framework provides functions to for determining the "neighboring" processes and collectives that use only the neighbor processes. For example, MPI_Neighbour_Allgather where every process i sends and gathers data from a process j if there is an edge in process topology graph. In the Cartesian process topology this function will consider neighbors those with the smallest distance(1). However, this requires the MPI application to be rewritten using these primitives whereas the profile based approach requires no modifications on the application's code.

6.3.7 Summary

In the first part of our evaluation we explored the benefits and some of the limits of Scotch, which is the graph mapping tool used by the TOFA approach, in the given scenarios. As seen from our experiments, mapping the processes to the underlying topology can benefit applications that exhibit irregular traffic pattern and/or are communication intensive. We showed that *Scotch* mappings are not flawless in every case. The are cases such as the case of LAMMPS with 256 processes where it performs worse than the Default round robin approach of Slurm. However, as we also demonstrated that is exceptional case and when the topology was modified *Scotch* performed better than the Default due to the fact that it is topology aware. Finally, we explored the effect several factors such as the MPI collective algorithm and the traffic criteria on the mappings produced by Scotch. The first part also reveals that by using Scotch, TOFA can improve the performance of applications (that fulfill with the aforementioned criteria) even in the absence of transient node failures. Now we have a clearer view of the properties of the graph mapping tool used by TOFA and we can move to the second part of the evaluation.

6.4 Evaluation of TOFA emulating node failures

In this section we evaluate the performance of the TOFA approach in the presence of node failures. Using the SimGrid simulator we emulate node failures which as we already discussed are considered transient node failures. Instead of using single MPI jobs we use batches of jobs for this evaluation scenario. TOFA process placement approach has a dual goal. Firstly, it aims at reducing the communication cost by placing processes with heavy communication in closer nodes. This is formulated as topology mapping problem and is solved by using Scotch. The

evaluation of Scotch is presented in the previous section 6.3. Secondly, it aims at reducing the job abort ratio due to transient node failures. This part evaluates the effectiveness of TOFA in achieving both of its goals.

In this evaluation scenario we compare TOFA to Default-Slurm process placement policy only as the other two approaches proved to be inefficient in the previous evaluation scenario. The *Greedy* policy is performing well for irregular applications, however its performance for regular applications is degraded. Also we excluded the *random* process placement policy since it underperforms in most benchmarks as presented in the previous section.

6.4.1 Methodology and evaluation criteria

The evaluation of $TOFA$ in the presence of node failures was conducted by running batch jobs and measuring the total execution time for each batch job. Instead of simulating a single MPI job instance, we use job batches, each consisting of 100 instances of the same MPI application. Batches of jobs will be referred to as *batches* for the rest of the study. The criteria used to evaluate each process placement approach are two: a)batch completion time, b)abort ratio. The batch completion time refers to the total time required to complete the simulated run of all 100 instances. Each time a job is aborted, batch completion time is increased by a time interval equal to a successful run and the failed job is restarted. The abort ratio on the other hand, depicts the fraction of instances that were aborted due to one or more node outages.

For the evaluation purposes of this section, we will focus on the default placement approach of Slurm and the proposed placement approach, namely, TOFA. Different simulated scenarios are derived based on the following parameters:

- MPI application
- N: Number of MPI processes involved
- p_f: node outage probability
- n_f: number of nodes emulated in the failed state
- N_f: set of nodes emulated in the failed state

The MPI applications simulated are NPB-CG class C (64 processes), HPL (64 processes) LAMMPS (64 processes) and NPB-DT (85 processes) of class C. Similarly to the evaluation results presented in Section 6.3, the platform assumed consists of 512 nodes arranged in an 8x8x8 Torus. For each application, 10 different batches are created consisting of 100 instances each. The simulations we present in this part follow these steps:

1. Before starting a batch, the nodes to populate the set N_f are randomly selected. This set remains the same for the duration of the batch. The number of elements of this set, n_f, is fixed to either 8 or 16.

2. All the nodes $\in N_f$ have the same fixed outage probability. For our the experiments this is either 1%, 2% or 5%.

3. Before running an instance of a batch (an MPI application) choose a random subset of N_f which will contain the nodes that will actually exhibit failure for this instance. This subset $S_f \subseteq N_f$ can have from 0 up to n_f number of elements/nodes. To determine if a node will be added in this subset a roll depending on its outage probability is made.

4. The simulated (virtual) platform is modified to include the faulty nodes of subset S_f and the instance of the batch executes

5. If the instance of the batch job aborts add the execution time of the previous successful run to the batch timer, to denote the penalty of fault. Then go to step 3 and restart this instance of the batch. If the run is successful start executing to the next instance of the batch.

The batch job will be considered complete when 100 successful instances are complete. To simulate a faulty node the platform file is modified before the job's execution as described in step 4 above. The bandwidth of the links concerning the faulty node, in the platform file is set to 0. SimGrid allows the bandwidth to be expressed periodically. For example at simulated time 0 the bandwidth of the link is 10 Gbps, and at simulated time 2.0 the bandwidth changes to 0 Gbps. This allows the job to start normally but will abort at time 2.0 (or later) if a route that includes this link is used by the job.

6.4.1.1 Results of experiments when taking account failures

The completion time for 10 batches of jobs that consist of NPB-CG class C applications using 64 processes is shown in Figure 6.14. For each batch 8 nodes ($n_f = 8$) out of all 512 are randomly selected and are assigned an outage probability of 5%, $p_f = 5\%$. Since 8 nodes out of 512 exhibit transient failures it means that about 1.6% of nodes have a chance of exhibiting failure. As this figure shows, for all batches, TOFA achieves significantly lower completion time than the default placement policy of Slurm.

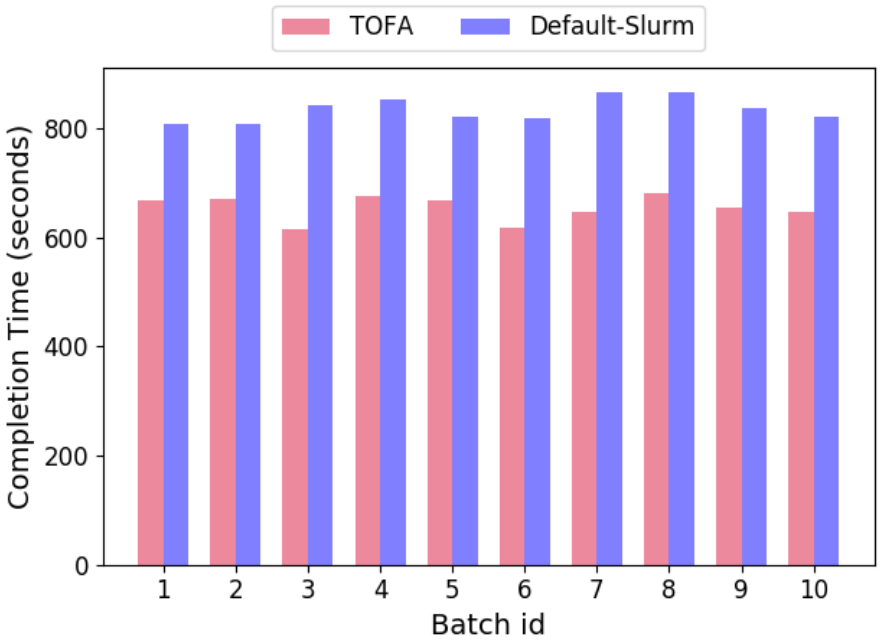


Figure 6.14: Batches of NPB-CG 64 processes, $n_f = 8, p_f = 5\%$

Remember that in the first part of evaluation, *Scotch* provided 1.25x speedup for NPB-CG with 64 processes over the Default-Slurm, figure 6.7. This means that every batch job of TOFA benefits by this speedup. In this scenario, TOFA achieves 30% lower completion times on average compared to the Default-Slurm. This is attributed to two factors; the lower completion time for every MPI job of the batch and also the lower abort rate. The average abort rate for the Default approach is 5% whereas for TOFA it is 0%. TOFA achieves 30% lower completion times on average compared to the Default-Slurm. Every job abort increases the overall completion time of the batch as it causes the job to be restarted.

The next application is HPL which as was discussed in 6.9 has the same execution time between *Scotch* and Default-Slurm approach. In the following experiments the difference between them can be only attributed to the fact that fault aware scheme exhibits less job aborts. The results are shown in figure 6.15 and we notice that $TOFA$ outperforms the *default* approach of Slurm in a scenario where failures also taken into account. This means that $TOFA$ succeeds in avoiding faulty nodes and as a result having a lower completion time. As opposed to NPB-CG where the lower execution time for a batch is attributed to the lower completion time for every instance of the batch in addition to the lower job abort rate . As seen in Figure 6.15, $TOFA$ performs better in most cases than Default with some exceptions. For example in batch no 1 where $TOFA$ performs significantly better, default scheme had a 20% abort rate whereas for jobs 2, 3 and 10 no jobs were aborted. $TOFA$ exhibited no errors in any of the job run. The average

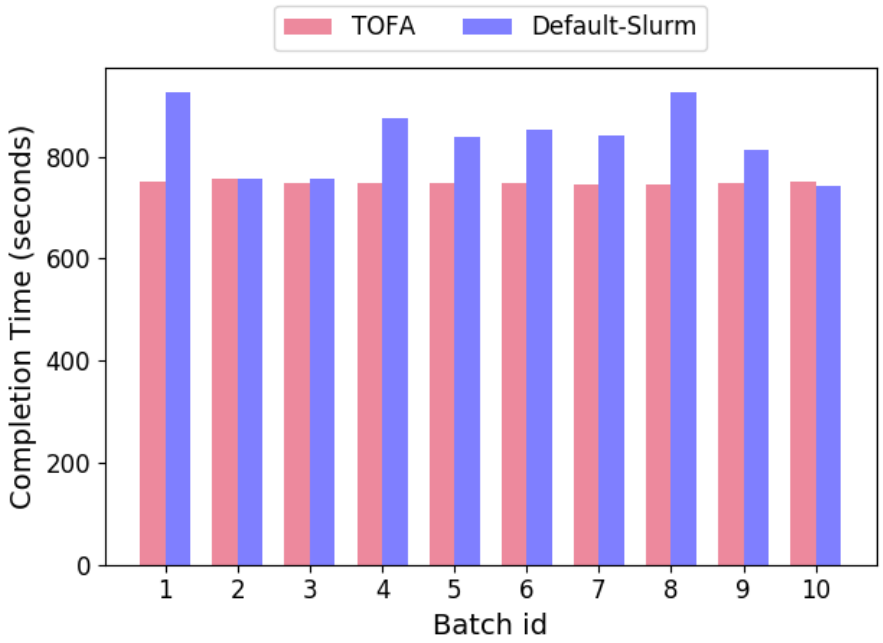


Figure 6.15: Batches of HPL 64 processes, $n_f = 8, p_f = 5\%$

abort rate for the Default-Slurm is 10.3% for this run and TOFA achieves 10% lower completion times on average.

The above two experiments demonstrated that TOFA approach benefits batches of jobs by reducing the communication overhead but also by reducing the number of job restarts due to job abortions. *TOFA* process placement has a 0% job abort rate in the above experiments. This is attributed to the fact that since only 8 nodes are characterized as faulty, our heuristic always succeeds finding 64 consecutive nodes (required by the applications) that are considered as faultless. In the next two experiments we increase the number of nodes that are considered faulty, from 8 to 16, but we decrease the outage probability for 5% to 2%. Also the first experiment uses 85 nodes instead of 64 which further increases the difficulty for the TOFA heuristic to find consecutive faultless nodes.

Figure 6.16 depicts the completion time for 10 different batches of NPB-DT jobs with 85 processes each. For each batch, 16 nodes out of all 512 ($n_f = 16$) are randomly selected and are assigned an outage probability of 2%, $p_f = 2\%$. 16 nodes out of 512 means that about 3.1% of the nodes of the system have a chance of exhibiting failure. As this figure shows, for all batches, TOFA achieves significantly lower batch completion time than the default placement policy of Slurm. More precisely, the average improvement in batch completion time over all 10 batches is 31%. This drop in batch completion time is attributed to two factors. The first one is the reduction in the communication cost which is the result of the

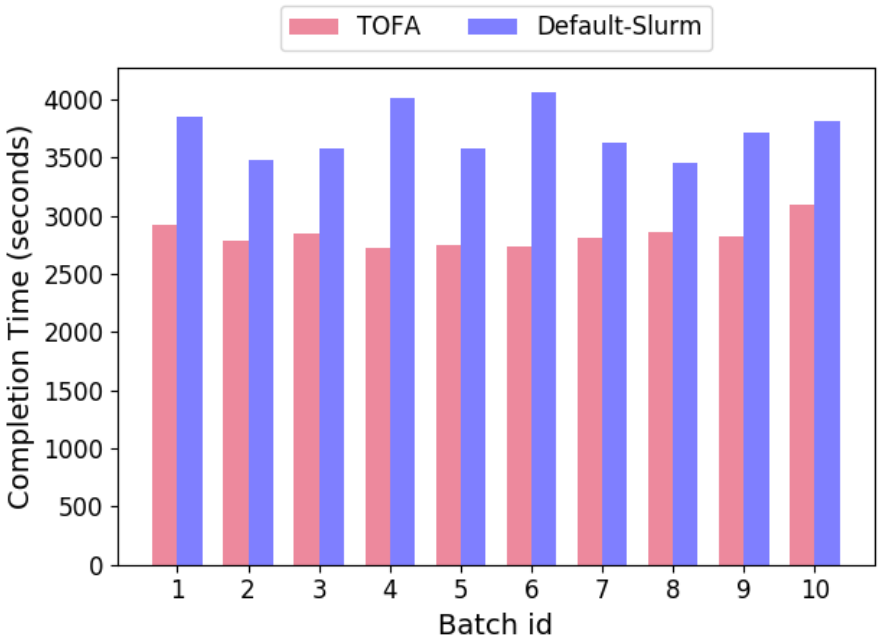


Figure 6.16: Batches of NPB-DT 85 processes, $n_f = 16, p_f = 2\%$

topology- and application profile-aware placement policy. The second reason is the drop in the jobs being aborted due to node outages. The average hob abort ratio (over 1000 simulated NPB-DT instances) is 2% for the case of TOFA and 7.4% for the default process placement policy of Slurm.

The simulation results that follow are derived from LAMMPS runs of the rhodopsin problem involving 64 processes. As discussed in the first part of the evaluation LAMMPS exhibits a regular traffic pattern and mappings created by Scotch marginally increased the performance of LAMMPS. The increase in performance for LAMMPS with 64 processes, in the first part of evaluation, is 12.5% as depicted in figure 6.10. Figure 6.17 depicts the completion time for 10 different batches of the LAMMPS application for the two aforementioned scenarios and two different process placement approaches, that is *TOFA* and the default placement policy of Slurm (*Default-Slurm*). As these figures show, for all 10 different batches, the completion time achieved by *TOFA* is lower. On average over all 10 batches, *TOFA* achieves 18.9% lower batch completion time than *Default-Slurm* As in the case of NPB-DT, the drop in the batch completion time is attributed to both the reduction of the communication cost and the overhead of failed jobs that need to be restarted.

The experiments conducted in the previous sections have higher failure rates, for the individual faulty nodes, than a realistic scenario. However, as stated before if we used realistic failure rates the benefit of fault aware approach would not be

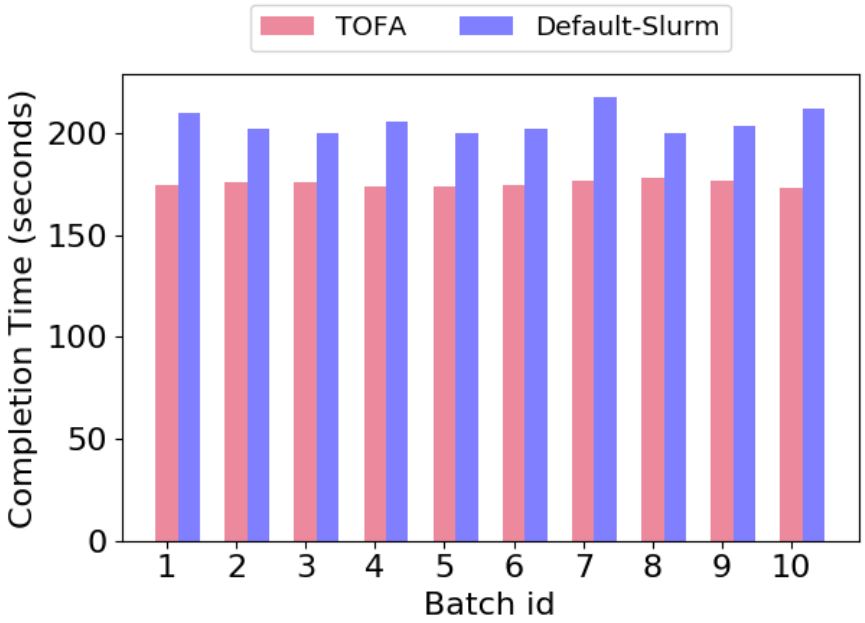


Figure 6.17: Batches of LAMMPS 64 processes, $n_f = 16, p_f = 2\%$

reflected because the runtime of a single job of a batch is only few seconds and the faults are very rare. To demonstrate the importance of the fault awareness approach we also present experiments with a more realistic fault model, and report the average abort rate for 10 batch jobs instead of the completion time. The fault model of our experiments is 8 faulty nodes ($n_f = 8$) which is about 1.6% of the 512-node cluster we are simulating, with a 1% chance of exhibiting failure ($p_f = 1\%$). As also stated before the application is more sensitive to node failures when more nodes allocated, thus in the following experiments the allocated nodes will range from 32 to 256.

Processes	Default	TOFA
32	0.4%	0%
64	1.4%	0%
128	2.4%	0.5%
256	5.6%	3.1%

Table 6.8: Average abort rate of 10 batch jobs of NPB-CG, $n_f = 8, p_f = 1\%$

From the results presented in Table 6.8 two different points can be derived. Firstly, our approach seems to be exhibiting about 2.0% less faults in all cases compared to the Default approach. Secondly, as we also stated in a previous section, by increasing the number of nodes allocated to a specific job the abort

rate also increases. Allocating 256 nodes out of 512 for a single job that is 1/2 of the cluster is not a very common practice. It should be noted that a job abortion might have a significant impact on cost due to the fact that HPC applications tend to run for a considerable amount of time.

6.5 Summary

In this chapter we evaluated the TOFA in two different parts. The first part focuses on the evaluation of Scotch which is the graph mapping library used by TOFA process placement approach. For evaluating both TOFA and Scotch we simulated real unmodified MPI applications. We analyzed their traffic patterns and categorized them as regular and irregular. The experiments showed that applications with irregular communication pattern and high communication to computation rate can benefit up to 30% from process placement using Scotch. The first part of the evaluation reveals that Scotch (which TOFA heavily relies for graph mapping) can improve the performance of applications even when there are no transient failures in the system's nodes. Moreover, we present the impact of various other factors on the mappings produced by Scotch. Such factors are, the topology, the MPI collective algorithm and the traffic criteria. Additionally, we determined the sensitivity of NPB-CG and LAMMPS traffic profile to their input. Moreover, Scotch, compared to the default round robin process placement exhibits a more stable behaviour as the system's topology changes. When the topology of the system is regular, such as a 8x8x8 Torus, the default process placement can produce better mappings compared to Scotch. However, using other 3D Torus topology arrangements (i.e. 4x8x16 Torus) the performance of the default process placement drops significantly whereas Scotch's performance is more stable. This is due to the fact that the default process placement policy is topology agnostic. Also, using the message size instead of traffic as traffic criterion can be sub optimal if the applications uses various messages sizes.

The second part of evaluation concerns the complete evaluation of the TOFA process placement approach. For this part of the evaluation, node failures are also simulated. As we demonstrate in an environment with potential node failures TOFA benefits both applications with irregular and regular traffic pattern. This is due to the fact that less number of jobs are aborted and thus by reducing the overhead of restarting them the overall completion time of a batch is reduced. TOFA is successful at reducing the job abort ratio in every case and also the communication overhead of individual jobs. TOFA can have almost 4 times lower job abortion ratio compared to Default-Slurm. Additionally, the job completion time due to optimal due placement is also reduced resulting in even lower completion time of a batch. Hence, these two factors reduce the overall completion time of a batch job from 11% up to 31% depending on the application.

Chapter 7

Conclusion

The author of this thesis implemented three different extension to Slurm; Acn-Slurm that extends Slurm to support FPGA-based accelerators, VIMA-Slurm that extends Slurm to support running workloads in Virtual Machines and TOFA-Slurm which is the main focus of this thesis. TOFA-Slurm improves the performance of MPI applications using a new TOpology and Fault Aware process (TOFA) placement approach. The TOFA process placement approach, reduces the communication overhead of the MPI applications and also reduces their abort rate due to node failures. This functionality is incorporated into Slurm via plugins.

In order to reduce the communication overhead, the TOFA process placement approach follows a common method that optimizes the placement of parallel process by creating a mapping of the processes to the system's processing elements. More precisely, the communication overhead is achieved by assigning processes to specific processing elements (i.e. nodes) that reduces the distance between them based on their traffic. By representing the topology of the system and the traffic of an application with graphs, the problem of assigning processes to processing elements becomes a graph mapping problem. The communication graph of the application is obtained via static profiling of the application. Unlike the common process placement approaches, the TOFA approach also attempts to reduce the abort rate of the MPI applications by avoiding faulty nodes. This is achieved by monitoring the nodes of the system and gathering fault related information. This fault related information is incorporated into the topology graph in order to avoid the faulty nodes. An additional heuristic was developed, to create more efficient mappings, that searches for a continuous block of nodes that are less likely to exhibit failure. The topology graph and the communication graph are fed to *Scotch* which is a graph mapping library and produces the mapping of the application to the underlying system topology. TOFA-Slurm implements the TOFA process placement and for this purpose we have developed five different plugins. This means that TOFA-Slurm can be adopted by any version of Slurm since it uses a pure plugin architecture without modifications in the main source code tree.

To evaluate both the reduction of communication overhead and the reduction of

job abort ratio of the TOFA approach, we use real HPC MPI benchmarks. We also perform a categorization of the different applications based on their traffic pattern by extending the profiling tool. In addition to performance of the benchmarks we also use the average Hops Per Byte to measure the quality of the mappings. We use the SMPI interface of SimGrid in order to run the applications in a simulated environment. For the first part of our evaluation we use various mapping methods and compare the results to the mappings produced by Scotch which is the graph mapping library used by TOFA. The mappings produced by Scotch reduce the completion time by up to 30% for communication intensive applications with an irregular traffic pattern. However, it shows no or marginal benefit for applications with more regular traffic patterns. Additionally, we perform an in depth analysis of the applications that enabled us to determine various factors which can have a profound effect on Scotch. Such factors are the topology, the algorithm used for implementing the MPI Collectives and the sensitivity to the communication graph. For the second part of our evaluation we emulated node failures in the SimGrid environment. We used batches of MPI jobs and compared the TOFA approach to the default Slurm approach. The TOFA approach achieved a notable reduce in the overall completion time of the applications from 10% to 31%. This is attributed to both reduced job abort ratio and the optimal process placement.

The rest of the extensions of Slurm are Acn-Slurm and VIMA-Slurm. Acn-Slurm provides Slurm with a generic resource (GRES) that enables the selection of nodes that are capable of a specific accelerator type. The user can submit a job and ask for a specific accelerator implemented in an FPGA, without explicit knowledge of the system. This extension is implemented without modifying the main source of Slurm which means that any Slurm version can be compiled with this plugin.

VIMA-Slurm, extends Slurm enabling it to run workloads in Virtual Machines. VIMA-Slurm enables the user to run applications using either virtual or physical resources in a transparent way. Additionally, VIMA-Slurm manages the virtual machine images that are installed in the nodes. Thus, it is not required by the user to have knowledge about the availability of the VM images. For the implementation of VIMA-Slurm we modified the main source code tree of Slurm. Due to this, VIMA-Slurm is not portable to other versions of Slurm. However, the usage of VM is completely transparent compared to the other approaches which require complex scripts from the user. Finally, VIMA-Slurm manages the VM images while for the other approaches the user to know which VM images are free to use.

7.1 Future Work

TOFA-Slurm is implemented in a three node system, however optimizing the placement of parallel processes is meaningful in large systems with thousands of nodes. Also, the performance of TOFA was measured using a simulated environment.

Thus, in order to have a realistic view of TOFA-Slurm it should be used in a real system where it will be handling real workloads. This would test the robustness of TOFA-Slurm as well as its efficiency in allocating resources using the TOFA process placement approach. Moreover, as discussed earlier, TOFA process placement approach tries to capture the effect of the node failures in the topology graph. At the moment this is done in a crude way by increasing the weight of the edges that contain faulty nodes. To capture this effect more precisely further tuning in the parameters of the graph is required. Furthermore, in this evaluation only 3D Torus topologies are used, mainly the 8x8x8 Torus, which is not a hierarchical topology as a Fat Tree. In order for the evaluation to be more complete, TOFA-Slurm should also be evaluated in other topologies such as Fat Tree or Dragonfly. For the topology and application graphs single files are used which contain $N \times N$ matrices. In larger scales that would represent a real HPC system or a parallel application with millions of processes this is not scalable. Thus, in larger scales those graphs are in the form of distributed graphs. Scotch offers a version called PT-Scotch that can handle such graphs. TOFA-Slurm can be extended to provide such graphs and use PT-Scotch in order to address the issue of scalability.

VIMA-Slurm could also be extended to enable the user to select images for running the virtual machine. Another extension could support containers along with virtual machines and thus provide a more uniform interface. Finally, measuring the overhead of VIMA-Slurm in a real HPC system is also a future work endeavour.

Extending Slurm proved to be a challenging but also highly educative task. It allowed me to decompose Slurm and learn more about its internals and functionality as well as the plugin APIs it offers. Furthermore, for implementing TOFA-Slurm it was required learn more about the topology mapping problem and the graph theory behind it. Since TOFA process placement relies on Scotch graph mapping library it was required to understand its use, capabilities and limits. Additionally, in order to set up the simulated environment for the evaluation I used SimGrid which was also modified to provide us with more information. Thus, for the completion of this thesis, a complex hierarchy of components was and developed and utilized. My hope for this thesis is to be a more detailed tutorial for anyone who seeks to extend Slurm in any way. I also wish for this work to be proven useful and provided insight to other projects that involve optimizations in process placement.

Bibliography

[1] Altair pbs works^{TM} named 'best hpc software or technology'. https://www.pbsworks.com/newsdetail.aspx?news_id=11069.

[2] High Performance Conjugate Gradients (HPCG) Benchmark. http://hpcg-benchmark.org.

[3] M2DC: Deliverable 4.3 - Report about integration into existing data-centres and cloud management systems. https://www.m2dc.eu/media/filer_public/30/7b/307bea09-8101-46df-9be8-2316c057894b/d43_integration_into_existing_data_centres.pdf.

[4] mpiBLAST is a freely available, open-source, parallel implementation of NCBI BLAST. https://wiki.rc.usf.edu/index.php/MpiBLAST#Description.

[5] NERSC, National Energy Research Scientifc computing Center. hhttps://www.nersc.gov/.

[6] OpenFOAM has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence and heat transfer, to acoustics, solid mechanics and electromagnetics. https://openfoam.com.

[7] Portable Batch System. https://www.pbspro.org/.

[8] QEMU, the FAST! processor emulator. https://www.qemu.org/.

[9] Shifter, Bringing Containers to HPC. https://docs.nersc.gov/programming/shifter/overview/.

[10] Singularity, the container platform for performance sensitive workloads. https://sylabs.io/.

[11] Slurm-v: Extending slurm for building efficient hpc cloud with sr-iov and ivshmem. Euro-Par 2016: Parallel Processing (P.-F. Dutot and D. Trystram, eds.),(Cham), pp. 349-362, Springer International Publishing, 2016.

[12] Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit.

[13] TEXAS ADVANCED COMPUTING CENTER. `https://www.tacc.utexas.edu`.

[14] The OpenMP API specification for parallel programming. `https://www.openmp.org/specifications`.

[15] top500 Supercomputer list. `https://www.top500.org/`.

[16] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[17] O. Arap and M. Swany. Offloading collective operations to programmable logic on a zynq cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 76–83, Aug 2016.

[18] Slurm Singularity Spank Plugin. `https://slurm.schedmd.com/SLUG17/SLUG_Bull_Singularity.pdf`.

[19] Byna, Gropp, Xian-He Sun, and Thakur. Improving the performance of mpi derived datatypes by optimizing memory-access cost. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 412–419, Dec 2003.

[20] Mariano Cecowski, Giovanni Agosta, Ariel Oleksiak, Michał Kierzynka, Micha vor dem Berge, Wolfgang Christmann, Stefan Krupop, Mario Porrmann, Jens Hagemeyer, Rene Griessl, Meysam Peykanu, Lennart Tigges, Sven Rosinger, Daniel Schlitt, Christian Pieper, Carlo Brandolese, William Fornaciari, Gerardo Pelosi, Robert Plestenjak, and Chris Adeniyi-Jones. The m2dc project: Modular microserver datacentre. pages 68–74, 08 2016.

[21] Running virtual machines in a slurm batch system. `https://slurm.schedmd.com/SLUG15/SlurmVM.pdf`.

[22] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. Mpipp: An automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 353–360, New York, NY, USA, 2006. ACM.

[23] Anthony Danalis, Aaron Brown, Lori L. Pollock, Martin Swany, and John Cavazos. Gravel: A communication library to fast path mpi. pages 111–119, 09 2008.

[24] David Culler, Jaswinder Pal Singh, Anoop Gupta . *Parallel Computer Architecture: A Hardware/Software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 1st edition, 1998.

[25] Augustin Degomme, Arnaud Legrand, Georges Markomanolis, Martin Quinson, Mark Lee Stillwell, and Frédéric Suter. Simulating MPI applications: the SMPI approach. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):14, August 2017.

[26] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. Topology-aware job mapping. *Int. J. High Perform. Comput. Appl.*, 32(1):14–27, January 2018.

[27] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In Emmanuel Jeannot and Julius Zilinskas, editors, *High Performance Computing on Complex Environments*, pages 75–94. Wiley, June 2014.

[28] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. pages 1–8, 01 2007.

[29] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 75–84, New York, NY, USA, 2011. ACM.

[30] HPL - a portable implementtion of the high-performance linkpack benchmark for distributed-memory computers. `https://www.netlib.org/benchmark/hpl`.

[31] Emmanuel Jeannot, Guillaume Mercier, and Francois Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):993–1002, April 2014.

[32] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 131–140, New York, NY, USA, 1999. ACM.

[33] Lammps - large-scale atomic/molecular massively parallel simulator. `https://lammps.sandia.gov`.

[34] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. In *In 17th Annual ACM International Conference on Supercomputing (ICS '03*, 2003.

[35] S. H. Mirsadeghi and A. Afsahi. Topology-aware rank reordering for mpi collectives. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1759–1768, May 2016.

[36] MPI:A Message-Passing Interface Standard. `https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[37] Nas parallel benchmarks. `https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf`.

[38] Pangfeng Liu and Da-Wei Wang. Reduction optimization in heterogeneous cluster environments. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 477–482, May 2000.

[39] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, pages 493–498, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, May 2015.

[41] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[42] M. J. Rashti and A. Afsahi. Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. In *2008 22nd International Symposium on High Performance Computing Systems and Applications*, pages 95–101, June 2008.

[43] Martin Ruefenacht, Mark Bull, and Stephen Booth. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing*, 69:24 – 44, 2017.

[44] Paul Sack and William Gropp. Faster topology-aware collective algorithms through non-minimal communication. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 45–54, New York, NY, USA, 2012. ACM.

[45] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[46] Scotch: Software package and libraries for sequential and parallel graph partitioning, static mapping and parallel sparse matrix block ordering. `http://www.labri.fr/perso/pelegrin/scotch/`.

[47] SimGrid, a Simulator for Distributed Computing systems. `https://simgrid.org/`.

[48] Slurm, an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small linux clusters. `https://slurm.schedmd.com/SLUG18/slurm_overview.pdf`.

[49] L. Sommer, J. Korinth, and A. Koch. Openmp device offloading to fpga accelerators. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 201–205, July 2017.

[50] H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. Mclay, K. Schulz, and D. K. Panda. Design and evaluation of network topology-/speed- aware broadcast algorithms for infiniband clusters. In *2011 IEEE International Conference on Cluster Computing*, pages 317–325, Sep. 2011.

[51] Osamu Tatebe, Yuetsu Kodama, Satoshi Sekiguchi, and Yoshinori Yamaguchi. Highly efficient implementation of mpi point-to-point communication using remote memory operations. 05 1998.

[52] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[53] Thomas Sterling, Matthew Anderson and Maciej Brodowicz. *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann, 2017.

[54] CS-534: Packet Switch Architecture, CSD University of Crete. `https://www.csd.uoc.gr/~hy534/15a/s8_NOCs_sl.pdf`.

[55] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Teg: A high-performance, scalable, multi-network point-to-point communications methodology. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 303–310, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[56] Sendren Sheng-Dong Xu, Xinhai, Zhong Wenbin, Yufei, Tomor Harnod, and Xuejun. Protocol-aware process placement for mpi programs. 2014.