Resource management techniques aware of interference among high-performance computing applications

a dissertation presented
by
Ana Jokanović
to
The Department of Computer Architecture

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Universitat Politècnica de Catalunya
Barcelona, Spain
November 2014

Author: Ana Jokanović
Thesis director: Professor Jesus Labarta
Thesis co-directors: Jose Carlos Sancho and German Rodriguez

Resource management techniques aware of interference among high-performance computing applications

Abstract

Network interference of nearby jobs has been recently identified as the dominant reason for the high performance variability of parallel applications running on High Performance Computing (HPC) systems. Typically, HPC systems are dynamic with multiple jobs coming and leaving in an unpredictable fashion, sharing simultaneously the system interconnection network. In such environment contention for network resources is causing random stalls in the progress of application execution degrading application's performance. Eliminating interactions between jobs is the key for guaranteeing both high performance and performance predictability of applications. These interactions are determined by the job location in the system. Upon arriving to the system, the job is allocated the computing and network resources by resource managers. Based on the job size requirements, the job scheduler finds a set of available computing nodes. In addition, the subnet manager determines the allocation of the network resources such as paths between nodes, virtual lanes, link bandwidth. Typically, resource managers are mainly focused on increasing utilization of the resources while neglecting job interactions. In this thesis, we propose techniques for both, job scheduler and subnet manager, able to mitigate job interactions: 1) a job scheduling policy that reduces the node fragmentation in the system, and 2) a quality-of-service (QoS) policy based on a characterization of job's network load; this policy is relaying on the virtual lanes mechanism provided by modern interconnection network (e.g. InfiniBand). In order to evaluate our job scheduling policy we use a simulator developed for this thesis that takes as an input the job scheduler log from a production HPC system. This simulator performs the node allocation for the jobs from the log. The proposed QoS policy is evaluated using a flit-level network simulator that is able to replay multiple traces from real executions of MPI applications. Experimental results show that the proposed job scheduling policy leads to few jobs sharing network resources and thus having fewer job's interactions while the QoS policy is able to effectively reduce the degradation from the remaining job's interactions. These two software techniques are complementary and could be used together without additional hardware.

Contents

Listing of figures

To my parents/ Mami i tati

Acknowledgments

This thesis would not have been achieved without my advisors.

First of all, I would like to thank Professor Jesus Labarta for being strict and honest, for teaching me to strive for simple and practical solutions and for making sure I never lose the sense of what the high quality research is.

I would like to thank Jose Carlos Sancho for his enormous patience, time and energy to listen to me and to discuss the ideas. His always positive attitude and support helped me a lot to become more confident in my research.

I would like to thank German Rodriguez for his tireless enthusiasm and dedication to work, and for being a great friend, especially at the beginning of my Ph.D. when it was very much needed.

I would like to thank Cyriel Minkenberg from IBM Research – Zurich for the collaboration on the papers from which I learned a great deal on how to write more comprehensively. I also thank Cyriel for the opportunity to do my four-months stay in the lab which was a rewarding experience in multiple ways.

I would like to thank my friends and colleagues from the reading group for the engaging discussions that helped me broaden my big picture on computer science and feel more comfortable doing my research.

Also, I am grateful to all my friends at BSC/UPC who made my stay in Barcelona a truly fulfilling experience.

Finally, this thesis is supported by unconditional love from my family and to them I owe the greatest gratitude of all.

0

Introduction

In today's high-performance computing (HPC) systems a large number of processors is interconnected in order to solve advanced scientific and non-scientific computation problems. Commonly, many parallel applications are being executed simultaneously, arriving to and leaving the system in an unpredictable, dynamic fashion and sharing systems' resources (e.g. interconnection network, I/O). One of the most critical shared resources for parallel application's performance is the interconnection network.

Non-blocking network topologies are becoming an expensive solution in the exascale era due to their large size. Therefore, the blocking networks will be applied to allow for affordable scalability. As

a consequence, we will face the increased criticality of network resources.

Typically, parallel applications communicate in a bursty fashion, and some of them send high load into the network causing the network links to be fully utilized.

There are several problems that arise in this sharing scenario, both from the user perspective and from the system perspective. On one side the users want high and stable applications' performance, as well as, low waiting time in the job scheduler's queue. On the other side, system administrators strive for high system utilization and system throughput. High-bandwidth demanding traffic creates a significant variability of user appplication's performance due to their interference with other applications in the network. Second, system throughput – number of applications executed in time – drops severely in the situation where the interconnection network is occupied by one or more applications with huge communication demands, slowing down and even stopping the progress of the rest of applications running simultaneously in the system. This causes processors that run these applications to stop as well wasting a significant compute power of the system.

A situation of inter-application contention is depicted in Figure 1. Jobs B and C are being blocked in the switch S9 due to congestion that happened in other part of the network (switch S11) caused by the traffic of job A. As a consequence there is an increase in the completion time of jobs that contended for network resources as shown in Figure 2.

This is an undesirable situation since the cost of the system and its energy consumption are so high nowadays that these systems can only be amortized by guaranteeing a high system throughput. An intuitive approach in solving this problem would be always allocating exclusive resources to each application (the case of job D in Figure 1). In that case there would be no interference between the jobs. However, this would increase the time application spends in the scheduler queue until the desired exclusive allocation becomes available. As a result, holding applications back in the queue will decrease both system utilization and system throughput.

The main objective of this thesis is to protect HPC applications performance from interference in the network by enabling the system with resource management techniques to either reduce or remove interference completely without impacting severely system performance - system utilization and sys-

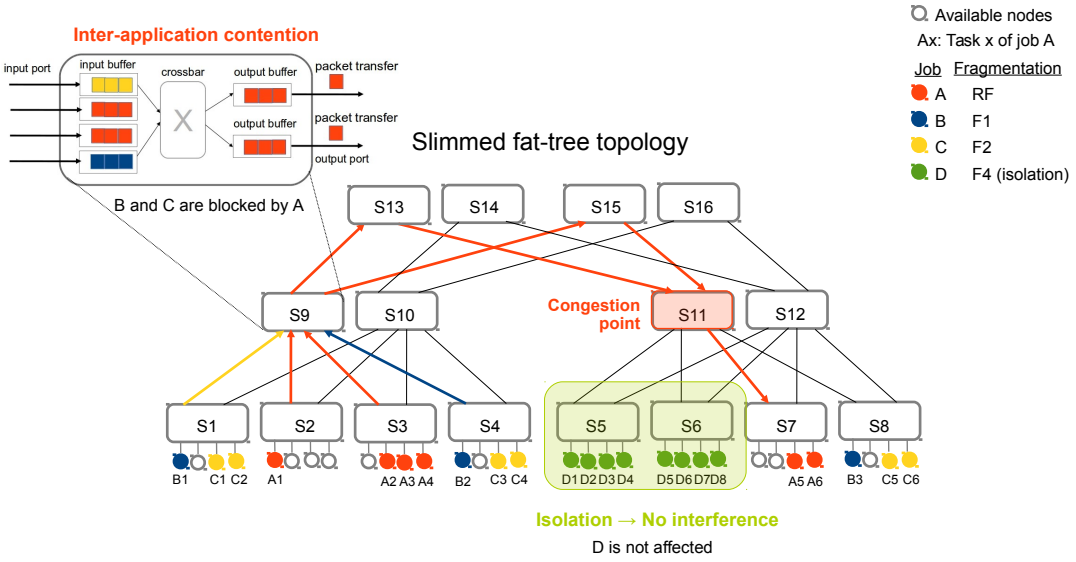

Figure 1: An illustration of inter-application contention.

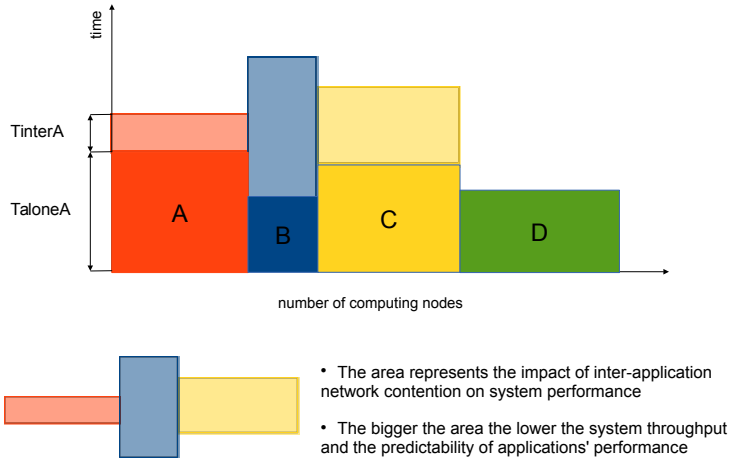

Figure 2: The impact of inter-application contention on the individual application performance and on the system performance.

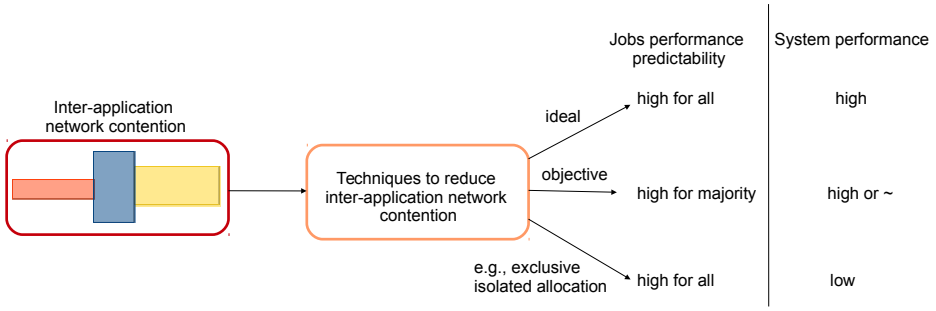

Figure 3: The objective of the thesis.

tem throughput (Figure 3).

The contributions of the thesis are the following:

- We propose a methodology for characterizing the applications that takes into account the applications communication behavior and its actual task placement. This methodology allows us to, at first place understand communication behavior of applications and to identify the potential of an application to create interference in terms of both how much it can impact the other applications and how many applications can be impacted.

- We show that the placement of the job's tasks is one of the most important resource management decisions for reducing inter-application contention and can lead to high variability in both application's performance and system throughput. Several computation node fragments might be available on arrival of a new application and the permeability of the network partition that connects each of the fragments might vary a lot. We explore the job scheduler strategies for the choice of the most suitable fragment based on the network permeability information and explain the trade-offs incurred in this choice. However, application communication behavior is typically not known a priori, i.e., before it is allocated, executed and profiled. Additionally, network permeability status can be learned, but it cannot be guaranteed for how long it will last. Due to the high cost of migration and unpredictability of the network state, we proposed an improved strategy for system-level resource management. Namely, taking into account net-

work topology and workload distribution, we propose creating virtual network topologies on top of the physical topology, which help to increase locality of application's tasks and thus reducing the number of applications sharing the same part of the network.

- We further reduce inter-application network contention applying other, more fine-grain resource management techniques at the link-level based on virtual channels arbitration enabled in modern interconnects (e.g. InfiniBand). We propose quality-of-service techniques based on identifying bandwidth sensitiveness of applications and separating their traffic across different virtual channels. The applications are either fully segregated or partially segregated depending on availability of the virtual channels. Finally, by tuning virtual channel quotas according to the information gathered from the identification process we are able to shape the global traffic demands to the system capabilities to achieve a fair progress for each application while improving system throughput.

Figure 4 summarizes our top-down approach in interference-aware resource management. It consists of several techniques that could be implemented in the system software. Upon arrival of new application to the system job scheduling policy (number 1) solely based on job size information decides which computing nodes to allocate for the job and that in turn determines the network resources that the job is going to use (switches, links, etc.). As the job is being run, it can be profiled and its network traffic can be characterized (number 2). Based on this information, proposed quality-of-service policy (number 3) decides how to share the network link using mechanism of virtual channels and weights.

The thesis is organized as follows. In the first chapter the backround on network performance issues, network topologies and resource management techniques is given. In the second chapter the tool chain used for experiments, as well as performance metrics are described. In the third chapter our work on applications characterization methodology as well as quantifying the variability of application performance under sharing resources scenario is discussed. In the forth chapter our work on system-level management of resources is discussed. In the fifth chapter we describe the work on quality-of-service policy and the link-level resource management. In the sixth chapter we give an overview of the previous related work. In the seventh chapter, we give the main conclusion of the thesis. Finally, in

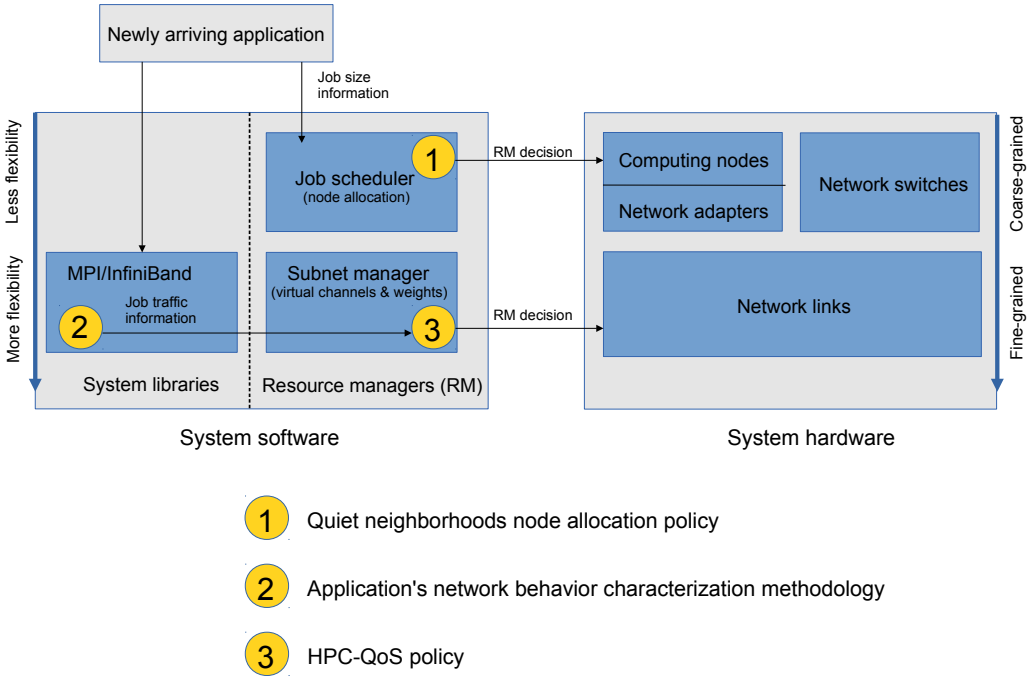

Figure 4: Interference-aware unified resource management.

the eight chapter we give the ideas we would like to explore as future work.

1

Background

Scientific applications solve complex problems by splitting the problem in several smaller parallel tasks each assigned to a single application's process. In order to exchange their intermediate results application processes communicate sending messages through the network. Depending on the algorithm applied to solve the problem, the tasks may communicate the messages following different patterns; a single pair of processes at a time, i.e., point-to-point communication or all processes at time i.e., colective communications. Further, depending on the problem size and scale, i.e., the number of processes involved in the computation, the message sizes vary, as well, making application's communication bandwidth or latency-sensitive.

Interconnection networks are designed with the objective to satisfy high throughput and low latency requirements. Typically, the network topologies are optimized for uniform random traffic; each node might send message to all other nodes with equal probability. The traffic of scientific applications, usually, does not follow uniform pattern, therefore the problems of competition for network resources from different flows, such as contention and congestion, may occur and lead to applications performance loss. Resource management techniques, node allocation, routing, virtual lanes arbitration and flow-control, are applied to provide better match between application traffic requirements and underlying network bandwidth availability.

1.1 Inter-application network contention

Typically, the performance of a parallel program is presented as its completion time consisting of the following components[23]:

$$T_{completion} = T_{computation} + T_{communication} - T_{overlap} \tag{1.1}$$

The communication time represents the sum of the transfers of all the messages on the critical path. The critical path represents the sequence of the program activities that take the longest time to execute[64]. A single message transfer would require the following time[15]:

$$T_{transfer\_of\_i^{th}\_message} = T_{message\_head\_latency} + \frac{Message\_size_i}{Bandwidth} + T_{contention} \tag{1.2}$$

Each message is segmented into packets and packets are further split into minimal network transfer units called flits. Time to transfer head flit of the message is bound to the flit processing time at each switch and at the source and destination adapters. The length of message in bits, i.e., message size together with network bandwidth defines another component of message transfer time called serialization latency. If the serialization latency is higher than the message head latency, we call such flows bandwidth-sensitive, otherwise we call them latency-sensitive. Additionally, every time there is a mes-

sage occupying the network link, the messages from all other traffic flows have to queue in the network buffers until the link becomes free and their turn to use the link comes. This competition of the messages from different flows for the same network resources is called network contention. For example, if we have three flows competing for a single link, we say the contention ratio is 3:1. In case the link bandwidth is not high enough to support all the traffic loaded onto the link (being it from a single or multiple traffic flows), than we face the problem of network congestion. Network congestion creates the congestion trees filling up all the buffers from the source of congestion until the traffic sources and basically, causes the stall in progress of all affected parallel programs. Thus, network congestion leads to higher $T_{contention}$ latency component.

Traffic flows may belong to the same application or to different ones. In the first case the competition of the flows for network resources we call intra-application contention. The second case, where the traffic flows from different applications are competing for the same network resources we call inter-application contention.

For illustration purposes, let us assume that there are two flows of data – A and B – which share the same link. We also assume that B is coming last to the network, and hence it is suffering the inter-application contention from A. The inter-application contention experienced by B can be expressed as

$$T^{B}_{contention} = T^{A}_{transfer} + T^{A}_{congestion} + T^{A}_{blocking} \tag{1.3}$$

where $T^{A}_{transfer}$ is the inevitable delay of waiting for one packet of A to finish transmitting; $T^{A}_{congestion}$ occurs when A transmits more data than the communication channel can tolerate, and thus more than one packet has been put into the output buffer in front of B. And finally, $T^{A}_{blocking}$ is an extreme case of the previous one that happens when the transmission of B packets is also being stopped because some congestion generated in another part of the network. Moreover, there is a finer-grained distinction to be made on blocking that separates the time where blocked packets of A prevent packets of B from proceeding to output ports that are not being blocked. In this case, B is said to suffer from Head-of-Line (HoL) blocking. Note that if all the output ports are blocked there is no HoL effect. In current switches, HoL effects is typically eliminated by using virtual output queues[39].

1.2 Interconnection network topologies

1.2.1 Fat-trees

Fat-trees are multi-stage tree-like topologies. Different kinds of fat trees have been proposed in the literature[37,49] all of which can be described under the parametric family of Extended Generalized Fat Trees (XGFT)[45].

The most common kind of fat trees found in supercomputers are k-ary n-trees[49]. k-ary n-trees are full-bisection fat trees that require $n \cdot k^{n-1}$ switches to connect k^n nodes that can be constructed using $2 \cdot k$-port switches.

It is possible to construct "slimmed" fat trees (Figure 1.1), which provide less than full bisection bandwidth (with the corresponding saving in cost and complexity) at the expense of reducing the available bisection bandwidth. This topology is used in RoadRunner [6], the world's first PETAFLOPS machine, and proposed under name "fit-tree" in the work of Kamil et al.[32].

Such topologies can be described with the XGFT notation. An XGFT$(h; m_1, ..., m_h; w_1, ..., w_h)$ of height h has $N = \prod_{i=1}^{h} m_i$ leaf nodes that can accommodate communicating tasks, while the inner nodes serve only as traffic routers. Each non-leaf node at level i has m_i child nodes, and each non-root has w_i parent nodes[45]. An XGFT of height h has $h + 1$ levels, level 1 being the leaf node level. XGFTs are constructed recursively, each sub-tree at level l being itself an XGFT.

1.2.2 Dragonflies

Dragonfly topologies[33] are highly scalable high-radix two-level direct networks with a good cost- performance ratio, used for example in the PERCS interconnect[4] and likely to make up the future Exaflop/s machines. A dragonfly is a two-level hierarchical network, where a number of cliques (fully-connected groups) of low-radix switches at the first level form a virtual high-radix switch. These virtual high-radix switches form another fully-connected graph of first-level groups[33]. The ports that the virtual high-radix switches use to connect to the other virtual switches are in fact distributed across the

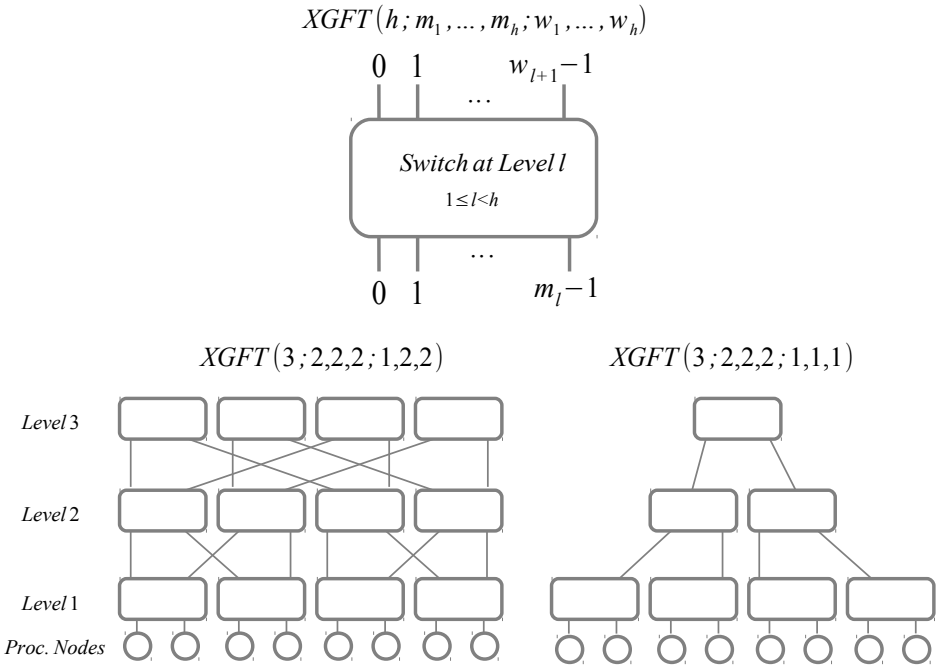

Figure 1.1: Switch ports at level l in XGFT(h;m1,...,mh;w1,...,wh) (top) and examples of a full-bisection fat-tree (left bootom) and its slimmed version (right bottom).

low-radix real switches that make up the virtual switch. Dragonflies can be described by means of three parameters: p, the number of nodes connected to each switch, a, the number of switches in each first level group, and h, the number of channels that each switch uses to connect to switches in other groups. For certain values for these parameters it can be shown that ideal throughput can be achieved for uniform traffic. The longest possible shortest path is made up of a traversal of a local, L link in the first level group to get to the switch that has a global, R link towards the destination group, a traversal of the R link and a second local link traversal in the destination group to get to the switch directly connected to the destination node.

1.2.3 Infiniband technology

The adapter and switch architecture parameters used throughout the thesis are based on the current Infiniband[1] adapter and switch architecture employed in many computing systems today. InfiniBand's Maximum Unit Transfer (MTU) is 256B-4KB. The typical link bandwidth supported in InfiniBand is 10-40Gb/s and the switch latency 100ns. The main advantage of InfiniBand technology for the topic addressed in this thesis is its support for quality-of-service (QoS) mechanism.

1.3 Resource management techniques

1.3.1 Computing nodes allocation

The computing nodes allocation is performed by the system scheduler (e.g., SLURM[66], LSF[70]). The scheduler manages the allocation of computing nodes to jobs that are being submitted by the users to the system. The choice of computing nodes determines the part of the network that is going to be used by the job. Thus, the decission on node allocation may directly contribute to the amount of inter-application contention.

By default, a computing node can be only used by one job (exclusive use of computing nodes). Users can change this default behavior allowing more jobs to be allocated per computing nodes but it is not common.

Once a set of computing nodes are found by job scheduler, the tasks of the parallel jobs are, by default, mapped sequentially to the allocated nodes. Optimization of mapping tasks to computing nodes, i.e., the order in which the tasks are placed to nodes based on communication pattern of the job is out of the scope of this work.

When there are not enough available computing nodes to allocate a job then the scheduler holds the job on a queue until enough computing nodes become available, i.e., some running jobs finish and their nodes are released.

As we see, job schedulers have to deal with the selection (which job) in a spatial (where to allocate jobs) and a temporal axis (when to allocate). Job schedulers usually incorporate policies that deal well with the temporal axis, prioritization, resource reservation, backfilling, etc. On the other hand, job schedulers generally have a poorer, if it all, view of the network topology and of how the placement decisions could impact the performance of applications. The discussion on prior work is done in Chapter 6.

1.3.2 Routing algorithms

The purpose of routing algorithms is to calculate a path between every pair of computing nodes.

The routing algorithms are classified as folows: 1) adaptive or oblivious, depending whether the network state is taken into account or not, 2) static or dynamic, depending whether a route for a pair of nodes is constant during the whole execution or not, 3) source-based or switch-based, depending at which place the decision on route is taken, 4) shortest-path or non-shortest path.

Knowing the traffic pattern of the parallel application, it is possible to deduce an optimal routing algorithm [69,54,53]. Optimal routing algorithm is the one that calculates paths between computing nodes pairs for a given set of these nodes such that the contention for the links on the path is minimal, i.e., the traffic is balanced. Optimization is normally done for the traffic patterns (pattern-aware [54]) that the application employs. In the ideal case we can know the traffic pattern a priori, or we can learn the pattern fast enough after the execution starts. However, a supercomputer is a dynamic system and optimizing routes for a single application may negatively impact another or even several other applications. In [29] we have shown that, from the system point of view, a routing algorithm can range from the best to the worst depending on the number and the behavior of other applications running simultaneously in the system.

Applying a dynamic routing algorithm may seem as a logical step for balancing traffic in a dynamic system. However, the decision on changing the routes is taken based on the previous measurement of network state for which we do not have any gurantees on how long it will last; there is multitude of applications with different traffic patterns coming and leaving the system at every moment. Thus, by

applying the route change we are not sure whether we are solving the problem or making it worse.

In order to make timely decision on route change an option would be to employ switch-based adaptive routing. In this way, an application's flow could be re-routed through another port every time the flow would be delayed by other applications that use the first-choice port. This approach would require re-calculating the route at the switch, thus additional complexity in the switch. More importantly, in this way the problem would be solved locally, but could create a problem at another point in the network.

The approach of re-routing based on feedback from the network is a good approach for balancing the traffic in the direct network topology such as Dragonfly. We have shown the impact of adaptive routing applied on various adversarial traffic patterns under sequential and random task placement[51]. Balancing the traffic in this way is rather application-interference oblivious for the reasons previously mentioned.

1.3.3 Virtual channels arbitration

InfiniBand provides the concept of Service Level (SL) which is used to identify different flows within an InfiniBand subnet. In this work we consider as a flow all packets sent by one application. An SL identifier is carried in the local route header of the packet.

Virtual lanes (VLs) provide a mean to implement multiple logical flows over a single physical link (Figure 1.2). In InfiniBand one VL (VL15) is reserved for subnet management traffic (A fabric interconnected with switches is called subnet). All other VLs are for regular data traffic. VL0 and VL15 are default lanes, while VLs 1-14 may be implemented to support additional traffic segregation. The number of data VLs configured has to be 1, 2, 4, 8 or 15. In current systems typically eight VLs are implemented (e.g., Mellanox).

For each VL independent buffering resources are provided. Link-level flow control is implemented on a per data VL basis. The sending port of an InfiniBand device identifies each packet with the VL to be used. The number of VLs used by a port is configured by the subnet manager. The port at the other end of the link may support a different number of VLs.

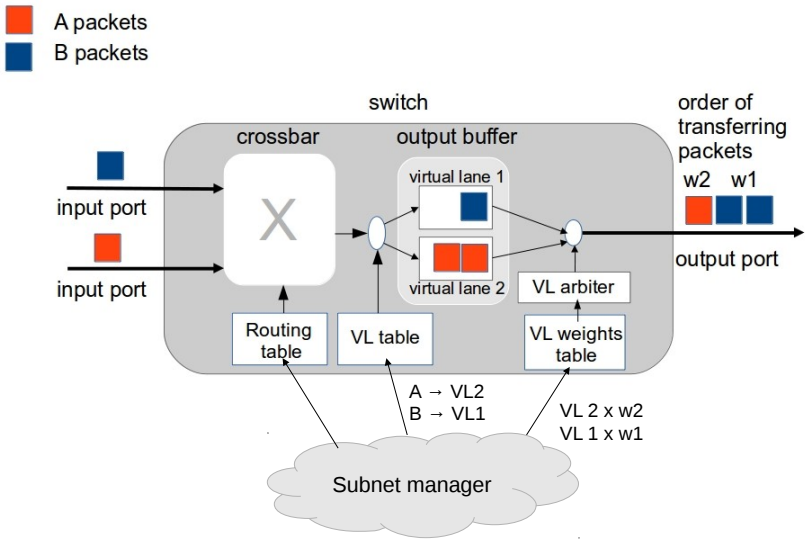

Figure 1.2: InfiniBand switch and its virtual lanes mechanism.

Also, two additional mechanisms that support appropriate forwarding behavior of each class of flows are specified. One is SL-to-VL mapping, the other is data VL arbitration.

SL-to-VL mapping is used to change VLs as a packet crosses a subnet. As a packet is routed across a subnet, it may be necessary for it to change VLs when it uses a given link in case the link does not support the VL used previously by the packet, or when two traffic streams arrive on different input ports of a switch heading towards the same outgoing link and use the same VL. By default SL_n goes to VL_n. SL is specified by the user when launching application. From now on, assigning an application to VL means to assign the same SL as well.

VL arbitration refers to the arbitration at an outgoing link on a switch, router or channel adapter. Each output port has a separate arbiter. The arbiter selects the next packet to transmit from the set of candidate packets available for transmission on that port.

The data VLs are at a lower priority than VL15 (highest) and flow control packets (the second highest). Devices implementing more than one data VL also implement an algorithm for arbitrating between packets on the data VLs.

VL arbitration is controlled by the VL arbitration table (The table should be initialized by the

subnet manager prior to use by data traffic.) which consist of three components, high-priority, low-priority and limit of high-priority. The high-priority and low-priority components are each represented by a list of VL/weight pairs. The weighting value indicates how many units may be transmitted from the VL when its turn in the arbitration occurs.

Arbitration between High and Low Priority VLs

The high-priority and low-priority components form a two level priority scheme. If the high-priority table has an available packet for transmission and the limit of high-priority is not reached then the high-priority is active and a packet may be sent from the high-priority table. If the high-priority table does not have an available packet for transmission, or if the limit of high-priority is reached, then the the low-priority table becomes active and a packet may be sent from the low-priority table.

Arbitration within High and Low Priority VLs

Within each high or low priority table, weighted fair arbitration is used, with the order of entries in each table specifying the order of VL scheduling, and the weighting value specifying the amount of bandwidth allocated to that entry. Each entry in the table is processed in order. Weighted fair arbitration between the VLs of the same priority provides a mechanism to allow more sophisticated differentiation of service classes.

Although InfiniBand provides mechanisms for QoS, it does not specify the policies for utilizing these mechanisms. It is necessary to develop QoS strategies to support a highly varied set of HPC applications.

2

Experimental methodology

Simulation tools are typically employed to evaluate application and system performance for different system parameters (e.g., network topologies, routing mechanisms, arbitration mechanisms, etc.). As this study requires analyzing HPC traffic in conjunction with the details of the network technology, we have used an MPI simulator driven by post-mortem traces of real MPI applications executions in conjunction with an event-driven network simulator. Besides, a scheduler simulator is developed for evaluating system-level resource management techniques. This chapter describes the set of simulation tools employed in the thesis, along with the workload used in evaluation.

Our experimental methodology consists of the following elements:

- HPC workload
- Extrae: Tracing tool
- Paraver: Visualization tool
- Dimemas: MPI simulator
- Venus: Network simulator
- Barrio: Scheduler simulator
- Performance metrics for quantifying the impact of applications' interference on the system performance

Each of these elements will be described as follows in a different section of this chapter.

2.1 HPC workload

In order to study inter-application contention it was important to choose a diverse set of HPC applications. A variety of scientific kernels from the NAS parallel benchmarks [5] set such as FT, CG, BT and MG are used in this study.

- FT (Fast Fourier Transform) solves a three-dimensional partial differential equation using Fast Fourier transform (FFT).
- CG (Conjugate Gradient) estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations.
- BT (Block Tridiagonal) is one of the algorithms used for solving a synthetic system of nonlinear partial differential equations.

- MG (MultiGrid) approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method.

Additionally, a set of real production scientific applications such as WRF, CGPOP, GROMACS and MILC has been used for this study.

- WRF (Weather Research and Forecast model)[41] is a numerical weather prediction system designed to serve the atmospheric research community. It is being used by an increasing community of researchers to implement and refine physical models.

- CGPOP[57] is a miniapp for the Parallel Ocean Program (POP) developed at Los Alamos National Laboratory, USA. POP is a global ocean modeling code and a component within the Community Earth System Model (CESM). CGPOP encapsulates the performance bottleneck of POP, which is the conjugate gradient solver.

- GROMACS[52] performs molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids, but also for research on non-biological systems, e.g. polymers.

- MILC[26] is MIMD (Multiple-Instruction Multiple-Data) Lattice Computation code used to study quantum chromodynamics(QCD).

The selected workload consists of a wide spectrum of scientific applications characterized by different communication traffic load. Figure 2.1 shows the aggregated number of bytes injected into network by an application's task during its execution. Since duration of a single Alltoall communication phase of FT application (4s) is longer than whole execution of some other applications from our set, we present all applications communication frequency at 4 seconds scale. Figure 2.2 in continuation shows the case of multiple iterations of FT application at larger time scale.

Figure 2.3 shows the output of the Equation 2.1 , i.e., average bytes in transit at any time of each application:

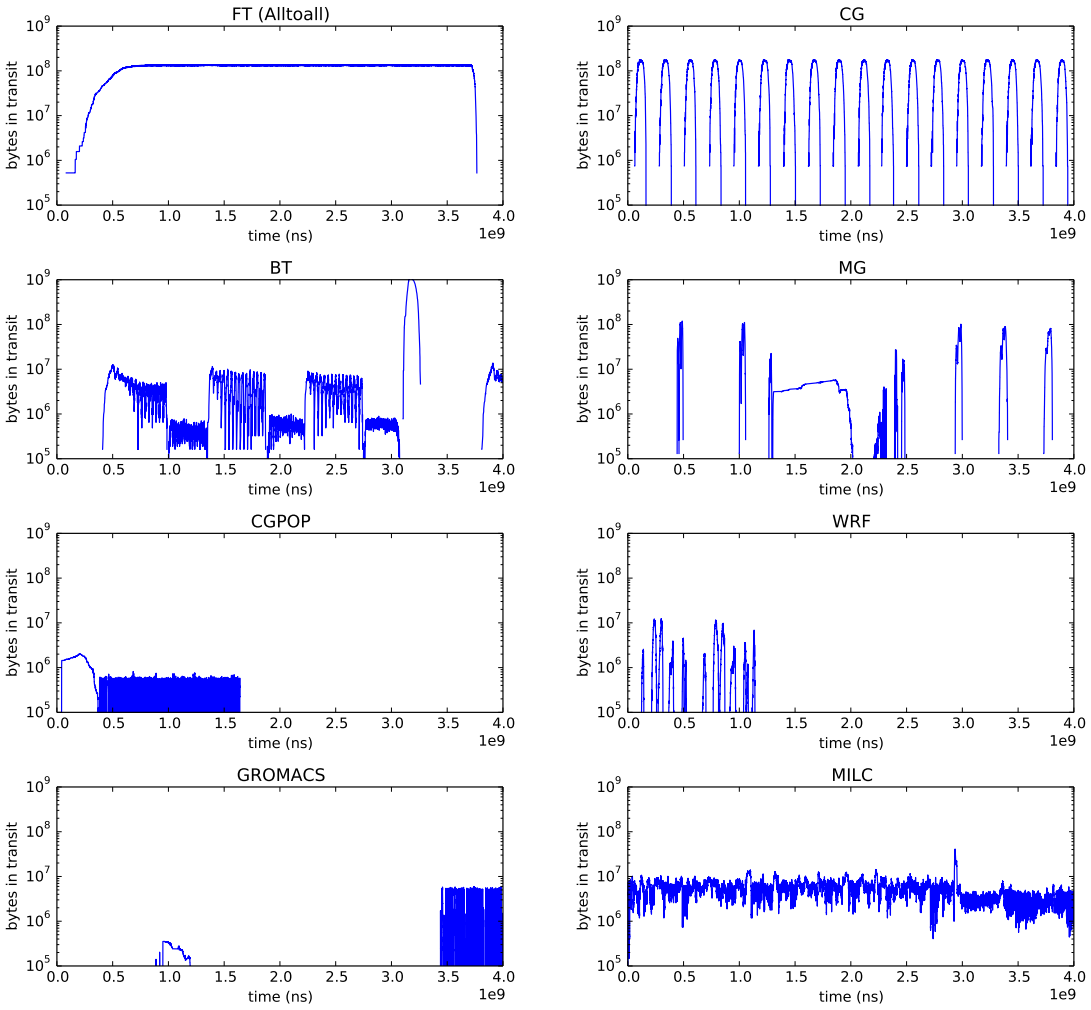

Figure 2.1: Bytes loaded into network by each of the studied applications.

$$\frac{\int_{0}^{\text{endTime}} \text{bytesInTransit}(t)\, dt}{\text{endTime}}. \tag{2.1}$$

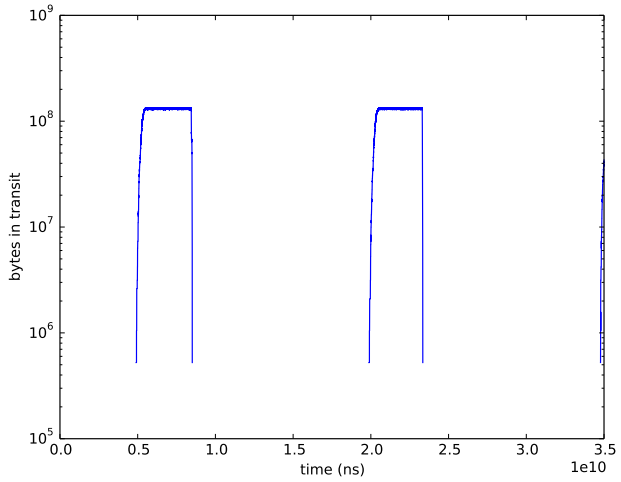

Figure 2.2: Bytes loaded into network by FT.

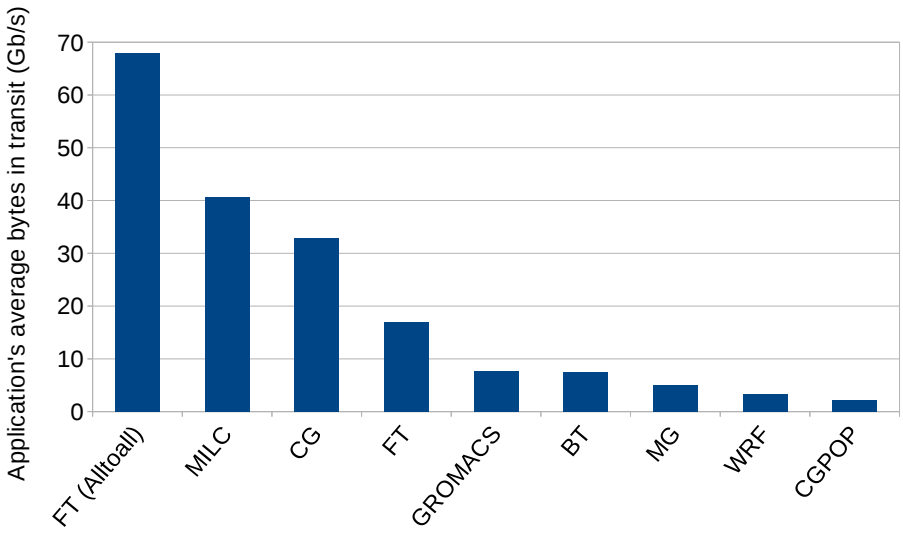

Figure 2.3: Applications' average average bytes in transit.

2.2 Toolchain

Two sets of toolchains will be used in this thesis. The toolchain from Figure 2.9 will be used for our characterization methodology, application's performance analysis and QoS policies. The scheduler

simulator in Figure 2.11, developed for the purpose of this thesis, will be used for the evaluation of our system-level resource management policies.

2.2.1 Extrae: Tracing tool

In order to get information about an application communication behavior and its performance in general it is necessary to insert instrumentation into it during its execution. An instrumented program generates a report that is called an application's trace. An application trace consists of a sequence of procedures called during application run, as well as information on how much time was spent in various parts of the application. The applications' traces were obtained using Extrae[11], tracing tool developed at Barcelona Supercomputing Center (BSC).

MPI standard includes a mechanism that enables users to profile MPI applications through MPI's Profiling interface layer. The idea behind profiling interface is to allow a secondary entry point to MPI library routines. Namely, while MPI function names have prefix MPI_, the secondary names for the same function will have prefix PMPI_. The application calls with MPI prefix can be intercepted and recorded, followed by the call of PMPI calls (Figure 2.4). PMPI calls are equivalent to MPI ones, the only difference in application performance is a small overhead incurred due to two calls.

Extrae intercepts the MPI calls that are coded with MPI prefix. However, the low-level operations of MPI collective calls are not coded with MPI prefix, but with MCA_PML_CALL macro, thus, they are not being recorded by Extrae. These low-level calls are either the actual sequence of point-to-point communications performed by MPI collective calls (e.g., MPI_Send, MPI_Recv), or the time an MPI task spent waiting for the message (MPI_Wait). For our study, these are very important pieces of information. Therefore, the OpenMPI[22] library is adapted to allow translation of the macros to MPI_-like names (Figure 2.5).

In order to instrument an MPI application at run time, Extrae uses the LD_PRELOAD mechanism to dynamically intercept calls to MPI library, as presented in the example scripts in Figure 2.6 This interposition is done by the runtime loader by substituting the original symbols ("MPI") by those provided by the instrumentation package ("PMPI"). Note that in the tracing script the MPI_INTERNALS

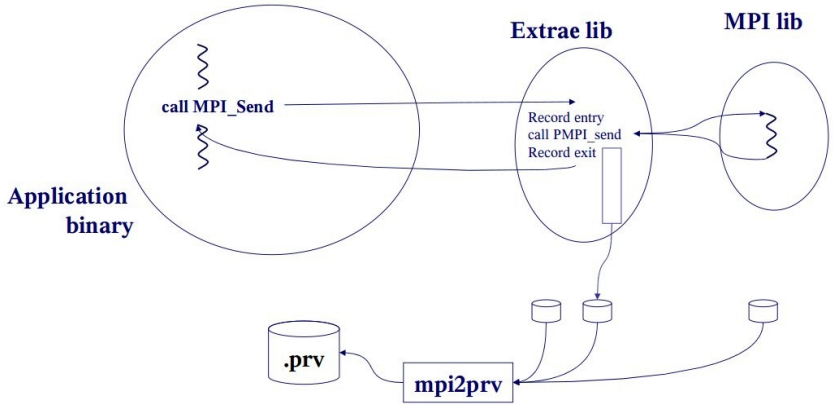

Figure 2.4: Dynamic library calls intrumentation

library is also loaded, to allow instrumentation of low-level communications of collective calls as previously explained.

We obtain the traces of instrumented applications from their runs on MareNostrum supercomputer.

The execution of a parallel application under Extrae generates a per-process record denoted as an mpit trace. The mpi2prv tool [11] developed at BSC can merge all the individual mpit traces into a single file. This merged trace file is suitable for the Paraver visualization tool which will be described in Section 2.2.2.

During instrumentation, each consecutive sequence of computation activities from the same process is translated into a trace record indicating a busy time for a specific CPU whereas the details of actual computation performed are not recorded. Communication operations are recorded as send, receive, or collective operation records, including the sender, receiver, message size, and type of operation.

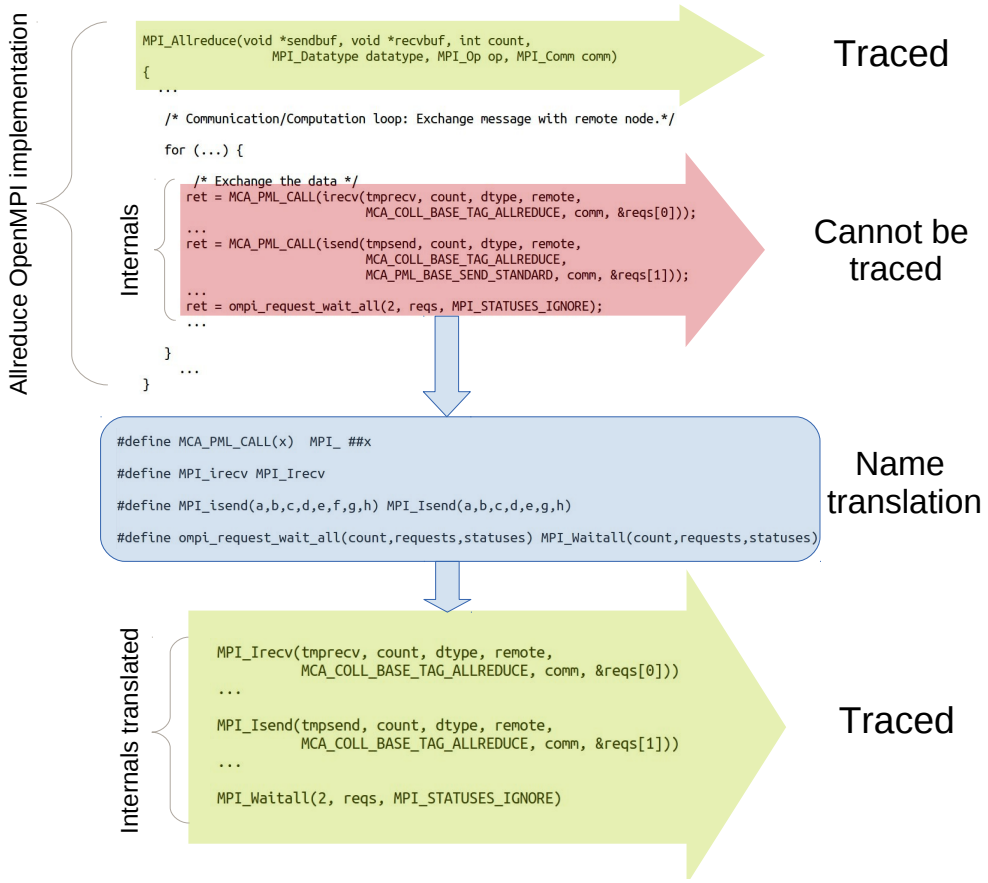

Figure 2.5: Tracing internals of collective communications. The OpenMPI library adaptation to allow for translation of MCA_PML_CALL macro to standard MPI call format.

The resulting traces normally consist of multiple iterations. Typically, each iteration has one computation and one communication phase. As the size of each trace can be several GBytes and therefore too big to be used with our simulators, a portion of each trace will be extracted, namely a cut of around 10-15% of the whole trace. The used trace cuts captures all the relevant characteristics of application's execution: communication patterns, computation/communication ratio, etc.

The records of high-level collectives and low-level internals of collectives cannot coexist in the trace since the MPI simulator will not understand them. To avoid this, we remove high level collectives' records before translating the Paraver trace to its corresponding Dimemas format.

Figure 2.6: An example of the tracing scripts

2.2.2 Paraver: Visualization tool

Paraver[12,50] is a visualization tool developed at BSC to show the insights of the execution of parallel applications, based on the traces obtained with Extrae. This visualization tool is very useful to detect load imbalance problems by simple inspection. Besides, we can visualize an application communication pattern, number of bytes exchanged between each pair of tasks, bytes in transit sent by a single task or total bytes sent by application at each point of its execution.

The mpi2prv tool described in Section 2.2.1 converts the multiple mpit individual trace files obtained with Extrae to a single file which can be understood by Paraver. However, the trace format of this tool differs from the trace format required for the Dimemas simulator described later in Section 2.2.3. The prv2dim tool converts the required fields from the Paraver trace to the format expected by Dimemas.

2.2.3 Dimemas: MPI simulator

Dimemas[10,36] is a tool developed at BSC to analyze the performance of message passing programs. It is an event-driven simulator that reconstructs the time behavior of message passing applications on a machine modeled by a set of performance parameters.

The input to Dimemas is a trace containing a sequence of operations for each thread of each task. Each operation can be classified as either computation or communication. Dimemas replays a trace using an architectural machine model consisting of a network of SMP nodes. The model is highly parameterizable (Figure 2.7), allowing the specification of parameters such as number of nodes, number of processors per node, relative CPU speed, number of communication buses, mapping tasks to nodes, etc.

The simulator is able to replay one or several MPI applications' traces simultaneously. Each of the traces contains a sequence of operations of each MPI application task as explained previously. Computation operations are not performed, but represented by the time the actual computation would last. Communication operations are send and receive point-to-point communications.

Dimemas allows for multiple replays of the traces within one simulation run. We can set number of restarts with an option "-r" followed by a number of replays. Setting "-r 0" means that Dimemas will restart traces as many times as needed until each trace has been played at least once. The difference in duration of different traces cuts might have some impact on the level of inter-application contention that the longer applications experienced. To solve this problem, we simply performed additional replay(s) of the trace of the application that finished earlier, immediately after its first execution. This way, applications were running concurrently at least until the end of their first execution. To calculate the impact of inter-application contention, we take the time of the first execution.

Task allocation is an important parameter in the context of network interference problem. In Dimemas, mapping MPI tasks to nodes is defined using a mapping vector. The mapping vector has a form {x,y,z,w}, where the lenght of the vector is the number of application's tasks, and x, y, z and w are the computing nodes. The mapping vector is interpreted as follows, a vector's index is an MPI task of

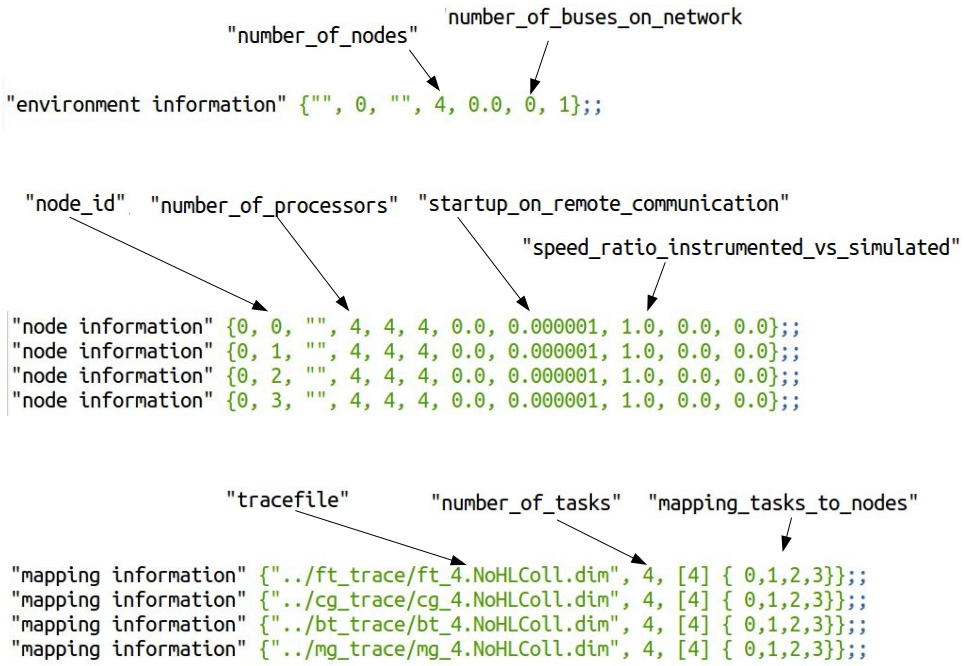

Figure 2.7: Dimemas parameters relevant for our study.

application mapped to the node at that index. In the example given in the Figure 2.7 {0,1,2,3}, we will have the following mapping for each application: task0 to node0, task1 to node1, task2 to node2 and task3 to node3. Note that number of processors per node is set to 4. Thus, with the given mapping we will have each node populated with four tasks each from different applications. On the other side, an alternative setting for ft mapping as {0,0,0,0}, cg as {1,1,1,1}, bt as {2,2,2,2} and mg as {3,3,3,3} would map all tasks of one application on a single node.

Dimemas outputs various statistics, such as execution time of each application, time spent in computation and communication, as well as output Paraver trace.

2.2.4 Venus: Network simulator

Venus[43,42] is an event-driven simulator based on OMNeT++[61] developed at IBM Zurich that is able to simulate up to the flit level any generic network topology of nodes and switches. It is able to provide a detailed simulation of the network topology and the processing inside the switches. It has two

main configurable basic components, the Adapter and the Switch, that are used to build arbitrary network topologies. The adapter and switch are based on InfiniBand technology specification and have implemented QoS mechanism.

Network topology, routes and mapping of applications to nodes are specified in corresponding topology, routing and mapping files. The creation of these files is explained in more detail as follows.

The applications are simulated using different network topologies: (i) fully connected one-hop network (crossbar), (ii) full bisection fat tree, and (iii) slimmed fat trees with different degrees of slimming [45,49] (Figure 1.1).

The topology file for fat tree topology is generated in two steps. First, by using Venus xgft tool with option -m and passing the parameters that define the desired fat tree topology, the intermediate topology file is generated. For instance, intermediate topology file xgft.map for a 3-level fat-tree with switch radix 4 will be generated in this way:

```
xgft -m 3:4,4,4:1,4,4 > xgft.map
```

and in case of a crossbar:

```
xgft -m 1:4:1 > xgft.map
```

where 4 is the radix of crossbar. Furter, we transform xgft.map file to actual topology files xgft.ini and xgft.ned using map2ned tool. Similarly, the route file is generated using xgft tool followed by -r option and a number that represents specific routing scheme (e.g., 3 for random routing, 1 for d-mod-k routing, etc.).

Venus task allocation file is a file with .scb extension and contains a list of computing nodes such that each line of the file contains only one node in the form hn where n takes values from 0 to total number of computing nodes in the system. Relation in between task mapping in Dimemas and Venus is shown in the Figure 2.8. Basically, if the task is placed on Dimemas N^{th} node, in Venus it will be placed on the node encountered in the N^{th} line of Venus mapping file.

The link bandwidth is set using two parameters unit_size and unit_time; unit_size is equivalent to flit size of the real network (it is possible to define min_unit_size, as well), where as unit_time is time needed to transfer amount of data defined by unit_size over the network link. The link bandwidth

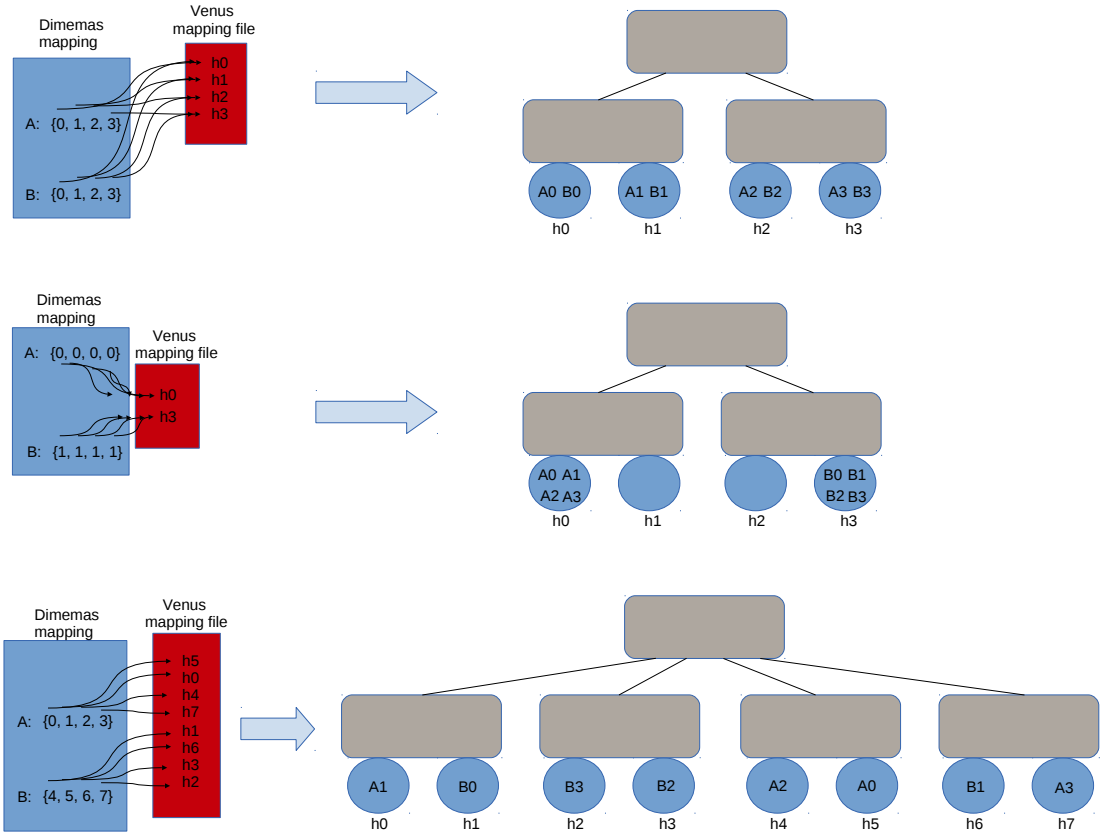

Figure 2.8: Relation between tasks-to-nodes mapping in Dimemas and Venus. Ax and Bx are the x^{th} task of applications A and B, respectively. Task B2 is placed on the node at the 2nd position of the B's Dimemas mapping vector, i.e., node 6; this node corresponds to Venus node on 6th line of Venus mapping file, i.e., node h3.

represents the ratio of the unit_size and the unit_time.

Each packet carries information on the application it belongs to. This allows us to apply different per-packet policies based on its application QoS requirements.

Venus allows for defining different QoS policies through InfiniBand[1] mechanism of virtual lanes. Basically, virtual lanes mechanism divides buffer physical space in several virtual partitions, where each partition is served according to a virtual lane arbitration policy. A different number of priorities, and virtual lanes can be defined, as well as virtual lane arbitration policies. Namely, we can define how many packets can be served from each virtual lane in one turn and in that way engineer and tune the

traffic differently based on its requirements. The buffer sizes are configurable, as well.

The Adapter and the Switch model collect information during the simulation that gets sent to a module that processes statistics. The output of Venus can either be a summarized collection of statistics or a detailed description of the point to point communications that took place during the simulation time.

We use Dimemas integrated with Venus. The complete tool chain of Dimemas-Venus co-simulation is given in the Figure 2.9. As previously described, Dimemas takes care of simulating the application at the MPI level, it feeds each individual communication to a proxy component built in Venus that instructs the Venus Adapter to inject the message. This proxy component in Venus also monitors the reception of messages and communicates this information to Dimemas. Venus and Dimemas together act as a discrete-event simulator synchronized using the Null message protocol/algorithm [62]. Whenever there are messages in flight, Dimemas and Venus exchange messages with the look-ahead time that they can continue the simulation until some event change the state arises.

One of the main advantages of the integration is the possibility of using Paraver to analyze and compare traces from actual runs with traces obtained from simulations. The flexibility of Venus allows for many topologies to be studied with relatively little extra developing effort. Another advantage of this model is that the MPI and the network level simulation are totally decoupled. This means that different MPI models can be implemented independently of the network simulator. An example of a script setup for running a Dimemas and Venus co-simulation is given in the Figure 2.10.

2.2.5 Barrio: Scheduler simulator

Barrio simulator has been built in order to evaluate the effectiveness of the proposed node allocation policies (Figure 2.11). The list of jobs to be simulated are taken from a trace recorded during normal operation of a supercomputer. The trace is generated by the Marenostrum's scheduler. For each job the scheduler recorded information such as the time when job arrived to the system, the duration of its execution, and the number of computing nodes used among other data. Figure 2.12 shows an example of a job record from the scheduler log. This information is enough to model the scheduling of jobs in

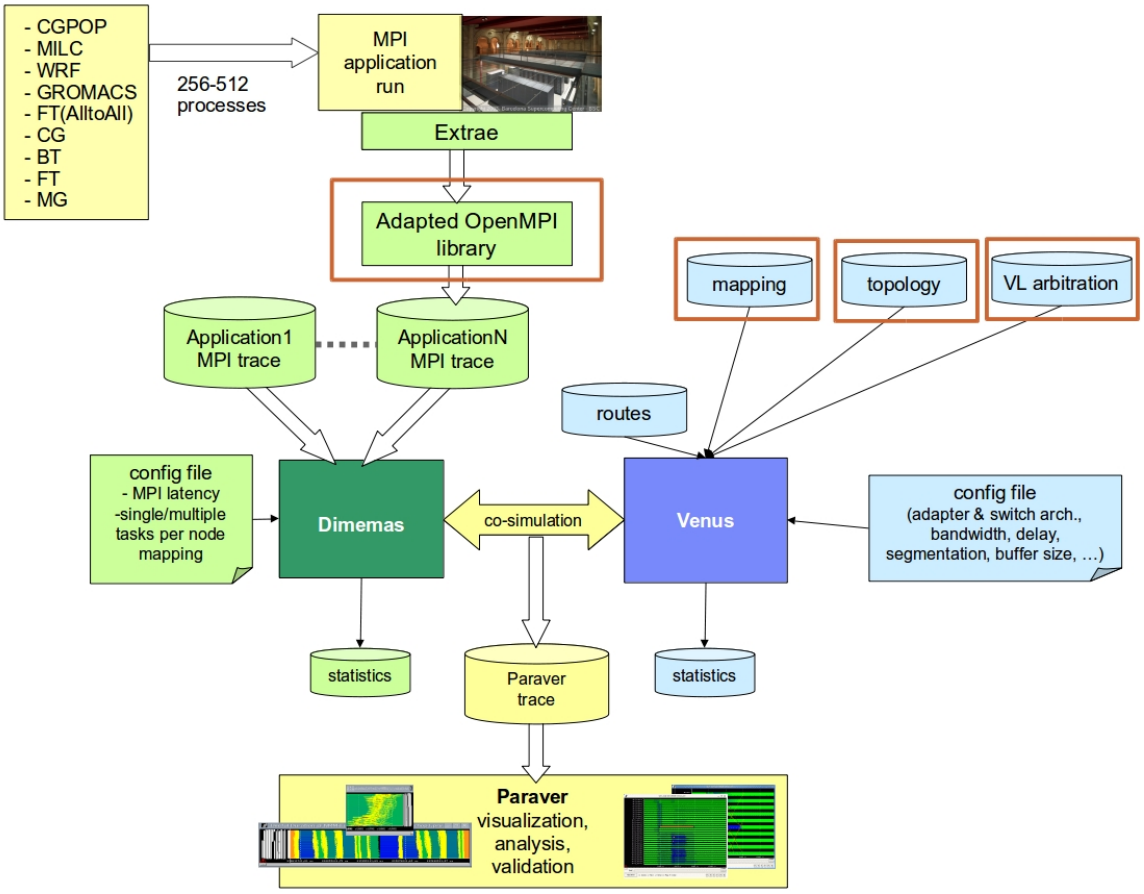

Figure 2.9: Dimemas & Venus co-simulation toolchain.

```
#! /bin/bash

./venus -f venus-config-file.ini &

sleep 10

./Dimemas -venus -pa output_trace.prv dimemas-config-file.cfg
```

Figure 2.10: An example of Dimemas & Venus co-simulation script.

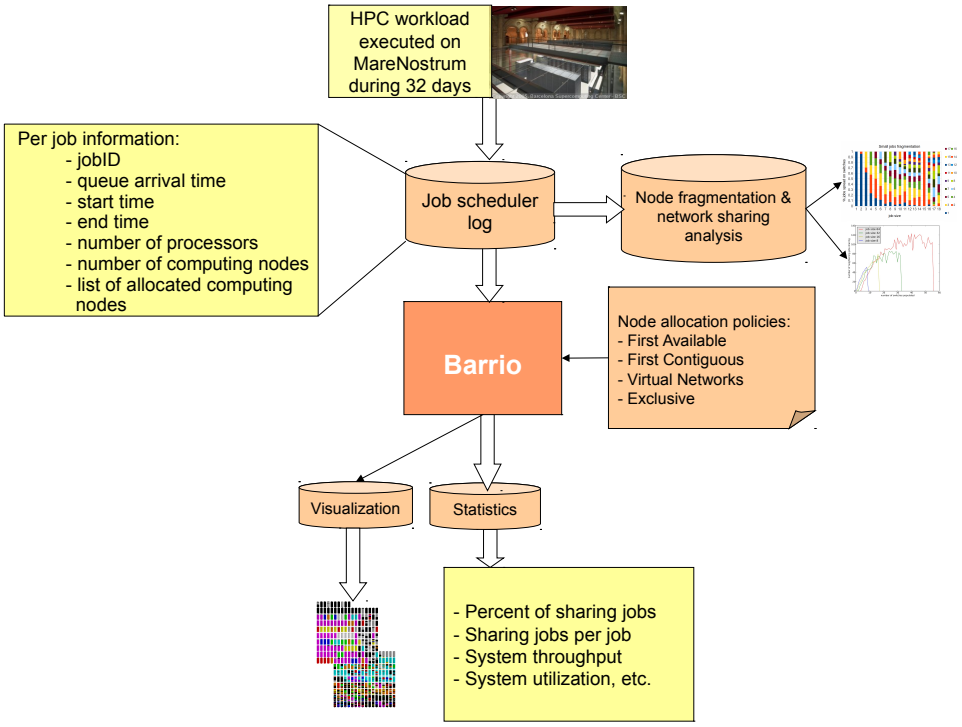

Figure 2.11: Toolchain for evaluation of system-level resource management policies.

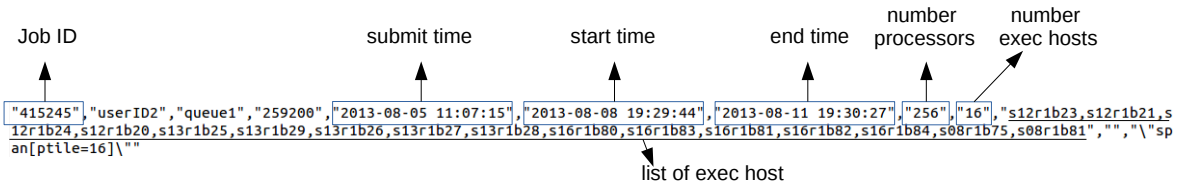

Figure 2.12: An example of per-job information from MareNostrum scheduler log used in our evaluation.

the simulator.

Note that the simulator does not replay the execution of jobs, it only accounts the time that every computing node has been using the system. In the simulator we are measuring different parameters during the execution of the job trace. A description of the key parameters reported are given as follows:

- Completion time. Reports the total elapsed time to process all the jobs in the input trace.

- System utilization. Reports the percentage of the system computing nodes that have an allocated job.
- Queue time. Reports the time that jobs are waiting for computing resources to become available.
- Per job sharing. Measures the average number of jobs that a job shared the system network with.
- Sharing jobs. Reports the percentage of total jobs that share system network with other jobs.
- Sharing network per level. Reports the total number of job pairs that are sharing the network at the second and at the third level.

2.3 Performance metrics for evaluating the interference impact on system performance

To quantify the impact for a particular workload, the application was simulated in the same system twice (Figure 2.13). First, it was executed alone, i.e., without any interference. The completion time of an application is T_{alone}. Then, it was run simultaneously with another application (or several applications) sharing the system and thus experiencing interference.

We will refer to the completion time of an application in the latter scenario as $T_{sharing}$. Both T_{alone} and $T_{sharing}$, we get from Dimemas output statistics at the end of simulation. Note that due to application trace cuts not being of the same duration, we perform a replay of the shorter one while the execution time of the first iteration only is considered. Therefore, the increase in completion time due to interference for an application i can be calculated as:

$$T^{i}_{inter} = T^{i}_{sharing} - T^{i}_{alone} \tag{2.2}$$

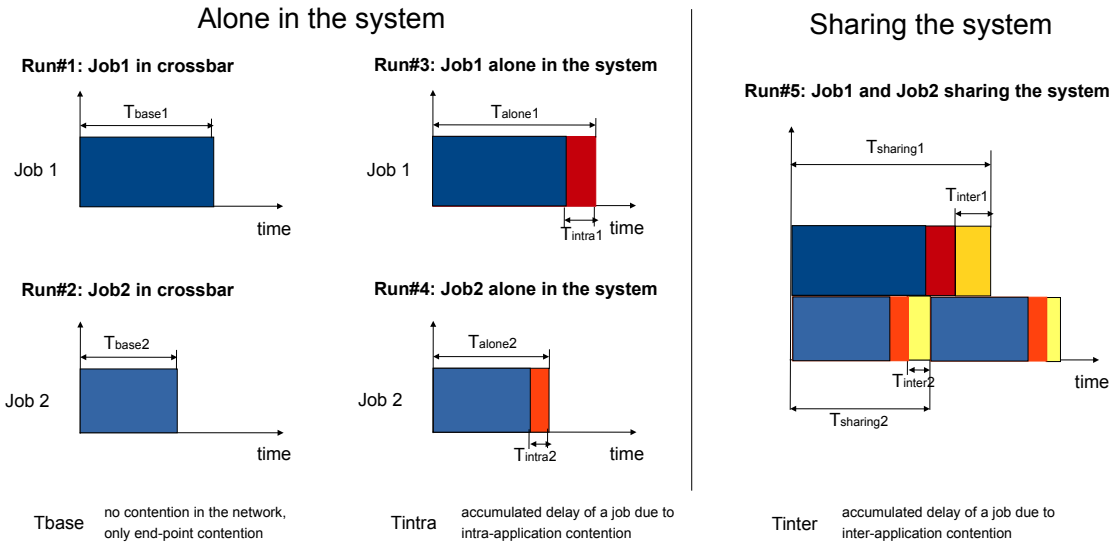

Figure 2.13: Simulation experiments needed for quantifying the impact of the network interference on the performance of each job. The case of two-applications workload. T_{base}, T_{alone} and $T_{sharing}$ are the outputs of the simulation runs.

Also, note that all system parameters and settings (e.g., size of the network, task allocation, routing, MTU size, etc.) have to be the same in both scenarios so that the increase in the execution time of the application can be attributed solely to the interference, and not to a coupled effect of interference and other factors.

To measure the impact of job's interference on system performance we will use the computing node time metrics defined in Figure 2.14.

C^i being the size of i^{th} job i.e. the number of computing nodes it is allocated to, we can calculate total waste of computing node time due to interference in the workload of n jobs as:

$$J_{inter} = \sum_{i=1}^{n} C^i \cdot T^i_{inter} \tag{2.3}$$

Finally, to quantify the effectiveness of our policy P in reducing the impact of interference, we use the following formula:

$$E(P) = \frac{J_{inter}}{J_{inter}(P)}. \tag{2.4}$$

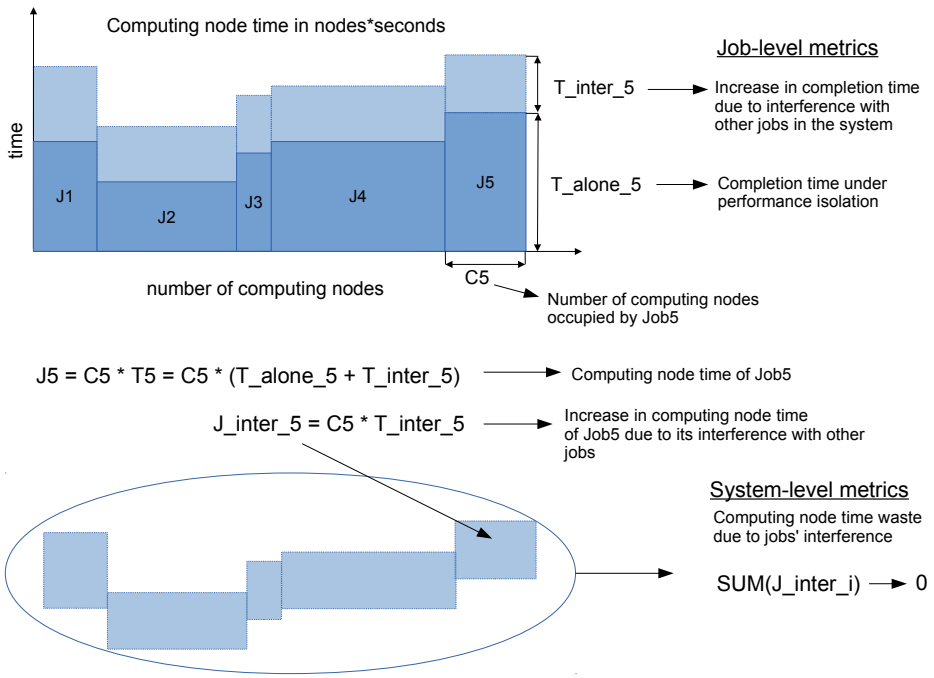

Figure 2.14: Quantifying the impact of the jobs' interference on the system performance.

3

Characterizing applications at network-level

Full bisection indirect topologies, such as fat trees, have been one of the preferred interconnection networks for high-performance computing (HPC). However, with increasing system size the cost of providing full bisection bandwidth accounts for an increasing portion of the total system cost. An underutilization of the communication network has been observed for some HPC workloads [32] triggering an effort to optimize the network in terms of cost and performance for the typical communication characteristics of HPC workloads.

A commonly adopted approach to improve this situation is to deploy a slimmed fat-tree topol-

ogy. Such a network reduces cost by eliminating some switches in each level of the traditional fat-tree topology at the expense of reducing the available bisection bandwidth. These topologies are prone to higher congestion.

Additionally, another factor that strongly impacts system throughput is job fragmentation. This occurs when multiple jobs running in the system require different number of nodes and have different execution times. In this scenario, it is very likely that the job scheduler is unable to assign a set of contiguous nodes (i.e., nodes that are topological neighbors) to the next job, and instead assigns nodes that are spread throughout the system and are not topological neighbors. Unfortunately, this effect degrades system throughput, as the performance of various jobs can simultaneously be degraded by the contention produced among each other. This type of contention is commonly called inter-application contention. In contrast, we denote contention suffered internally by a single application as being intra-application. Today, job schedulers such as Moab support various job allocation policies, including contiguous allocation, but to obtain a contiguous allocation the scheduler might have to hold jobs for a long time in the scheduler queue, which may also severely degrade system throughput. Several recent works have studied this relationship between task mapping and job scheduling policies[60,44]. However, only the impact of intra-application contention on system throughput was evaluated, not that of inter-application contention.

We have to be aware that the level of inter-application contention is impacted by several other factors, such as routing, topology, MPI tasks ordering and relative starting time between the applications. In[29] we evaluated the impact of several routing schemes on inter-aplication contention. The effect of interference between applications can be reduced using certain routing schemes. However, the communication characteristics of applications in the workload are the dominant factor regardless of the routing scheme.

Also, we should make distinction between task placement and task ordering. In this work we experiment with different task placements, both random and regular, but assuming sequential task ordering. This is because schedulers order tasks sequentially by default on a chosen node allocation. Some works have proposed topology and pattern-aware task orderings to reduce both intra- and inter-

application contention [44,68]. Further, the performance impact also depends on how much the communication phases of the applications overlap. The overlap could vary with every new execution of a mix of the application depending on the starting time offset between them. These are all additional factors that influence inter-application contention which makes it a very complex problem to analyze. In order to understand the impact of every single factor our approach is to make some of the factors constant while varying the others.

In this work we:

- Proposed the methodology for characterization of application's network traffic demands taking into account both applications nature and its actual task placement in the system,
- Classified applications based on the utilization metric obtained from the proposed methodology,
- Evaluated and understood the increase in intra-application contention of representative scientific applications as a function of slimming and fragmentation,
- Analyzed the performance of the applications in a shared environment,
- Classified the workloads based on the overall impact of sharing resources to the applications in the workload,
- Identified trade-offs that the capacity systems are dealing with in order to get the best performance when running multiple applications,
- Explored the strategies to mitigate the problem of inter-application network contention.

3.1 Simulation setup

The MPI simulator Dimemas is used driven by post-mortem traces of real applications executions in conjunction with an event-driven network simulator (Venus) as described in Chapter 2. The system parameters used in the simulations are given in Table 3.1.

Table 3.1: Simulated machines parameters.

Simulator	Dimemas+Venus
System size	2048 processing nodes
Topologies	Extended Generalized Fat Trees[45] and Crossbar
Connectivity	XGFT(3;16,16,8;1,W,W), W=1,2,4,8,16
Switch size	32 ports
Contention ratio	1:1, 2:1, 4:1, 8:1, 16:1
Switch Technology	InfiniBand
Input Buffer Size	4 KB
Network Bandwidth	10 Gbits/s
Segment size	4 KB
Flit size	256B
MPI Latency	1 µs
CPU Speedup	10x
Routing scheme	Random routing[24]

The CPU speedup factor of 10 was applied, meaning that the simulated computation time of the application is ten times faster than the machine on which the trace was originally collected (MareNostrum 2.5 GHz PowerPC 970 CPUs). This scaling is necessary to correlate results to the computation speed of today's processors. In addition, the simulated network (10 Gb/s) is five times faster than the original network (2 Gb/s Myrinet).

The applications are simulated in three different network topologies: 1) fully connected network (crossbar), 2) full bisection fat tree, and 3) slimmed fat trees with different degrees of slimming

A set of real production applications such as GROMACS, CGPOP, MILC, and WRF, and some scientific kernels from NAS parallel benchmarks such as FT, CG, and BT has been used. Table 3.2 gives the base execution time (simulated on a crossbar topology) for each of the application traces.

We have studied the sensitivity of the selected applications with respect to three aspects: i) network connectivity (degree of slimming) ii) task placement (fragmentation), and iii) with which other applications it is sharing network resources.

Table 3.2: Applications reference times.

Application	T_{base} (s)
FT (Alltoall)	0.211964
BT	0.685480
CG	0.086342
WRF	0.123653
CGPOP	0.127269
MILC	0.586126
GROMACS	0.425134

To present the results concisely we will introduce the following notation:

- S + number: represents the w parameter of an XGFT. The larger, the less slimmed a network is. S16 is a full bisection fat tree. S8 has half the links at each level, etc. Figure 3.1 shows the available bandwidth at each level of fat-tree for used slimming factors, i.e., number of links up from the switch.
- RF: Random Fragmentation - uniformly distributed tasks across the whole 2,048 nodes topology.
- F + number: Regular fragmentation. F8 indicates that 8 nodes per switch were used for each application. As we simulated a radix 16 network, i.e., 16 nodes per switch, only 2 applications can share a subtree. The smaller the number, the less nodes per switch that has been allocated to the application, and the more applications that share a particular switch.

One MPI application process is allocated per node to measure the effect of network contention without effects of intra-node contention.

Either the execution times or normalized execution times of application are reported. For example, normalized execution time of an application i given node allocation F_n is calculated as follows:

$$\frac{T^i_{base} + T^i_{intra}(F_n) + T^i_{inter}(F_n)}{T^i_{base}}. \tag{3.1}$$

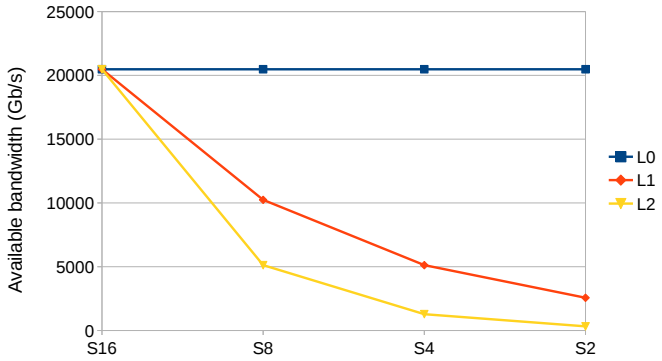

Figure 3.1: Total available bandwidth per level of a fat-tree network xgft(3;16,16,8;1,S,S) for different slimming factor S.

where T^{i}_{base} is the execution time of application i on a system with an "ideal" single-hop full bisection network (see Table 3.2). $T^{i}_{intra}(F_n)$ is the difference between the execution times obtained from application's run alone on a fat tree topology (on fragmentation F_n) and T^{i}_{base}. $T^{i}_{inter}(F_n)$ is difference between the execution times obtained from application's run on the same fat tree topology mixed with other applications (using the same fragmentation F_n) and the execution time of application's run alone. The presentation of the sensitivity of applications is separated in two sets of graphs. First we will analyze each application's sensitivity to the topology and fragmentation when running alone, without considering interference from other applications, followed by discussion on the reasons behind observed variability. Then, the cases of multiple applications sharing the system simultaneously will be introduced.

3.2 Characterization of the applications network behavior

In this section we will explain how we perform each step from our characterization methodology depicted in Figure 3.2. The applications injection rate Figure 3.3 we obtain as follows. Using Paraver traces we have extracted the total injected load in bytes of each application per fat-tree level for a given task placement. The load was averaged by reference times (crossbar execution times) from Table 3.2 giving us the average injection rate that an application would communicate per level of a fat tree for each of the task placements. Dividing by crossbar time gives us the injection rate an application would

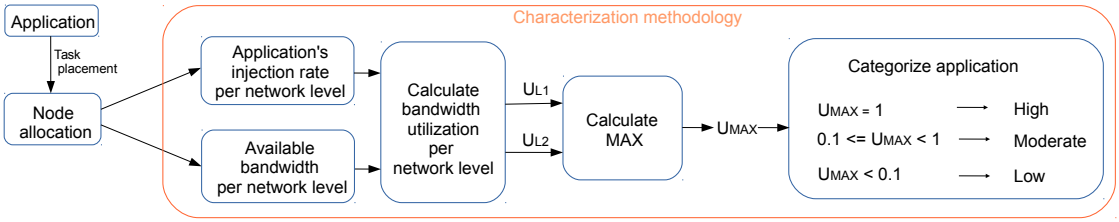

Figure 3.2: Methodology for characterization of application's network utilization; bandwidth utilization of application per levels L1 and L2 is U_{L1} and U_{L2}, respectively

produce if there was no any resource limitation in the network except end-point contention. L0 plots show the injection rates of the applications' load at level 0 of the fat-tree, i.e., total traffic injection rate. In terms of total traffic, the order of applications' communication intensity is the same as presented in Figure 2.3. L1 plots show the injection rate of the portion of the traffic that was sent outside of the first level-switches, while L2 plots show the injection rate of the portion of the traffic that was sent outside of the second-level switches. Therefore, observing the relation between L0 and L1 can give us insight whether an application is communicating locally or globally. The case of F16 node allocation gives us the information on application inherited locality. In that regard, FT, MILC and BT most of its traffic communicate outside of the first-level switch, while CG, WRF and CGPOP communicate locally. Note that the y-axis is at logarithmic scale. As we spread applications from F16 to F2, the locality of the applications changes and they start to communicate most of its traffic at L1 and L2. Thus, we can say that node allocation deviates the nature of application communications behavior.

Figure 3.4 and Figure 3.5 show the available bandwidth when a job of the size 256 and 512 nodes, respectively, are allocated on different fragmentations. As the job is more spread in the system it can "see" more bandwidth both at L1 and at L2. However, the benefit in terms of more bandwidth when spreading decreases as the network becomes slimmer. When allocated on F16, a job of size 256 never uses L2 bandwidth.

The metric that we use to classify applications based on their network demands is bandwidth utilization. Bandwidth utilization we obtain by dividing injection rate (Figure 3.3) with available bandwidth for a given allocation (Figure 3.4 and Figure 3.5, the latter in the case of MILC). Figure 3.6 shows

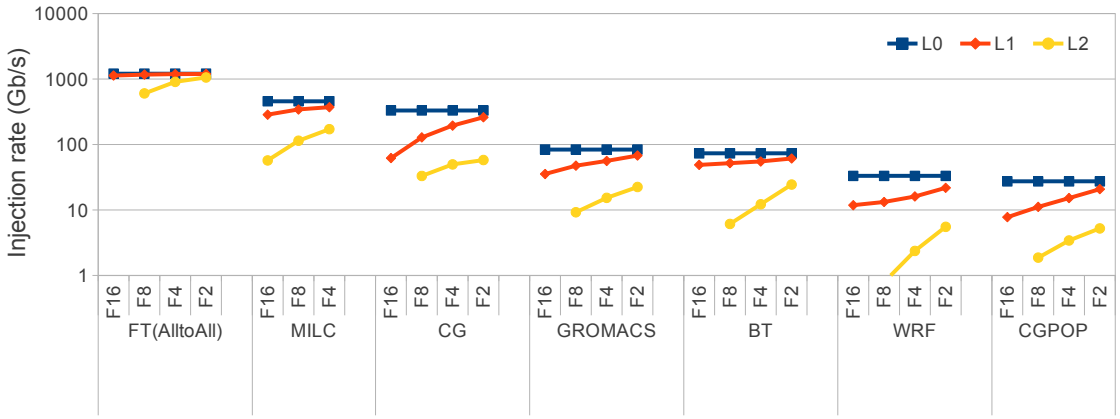

Figure 3.3: Applications' injection rate per level of xgft(3;16,16,8;1,w,w) under different task placements.

the utilization per level for our seven applications of interest for a case of S2 network and F2 allocation. We can do a coarse grain classification of applications (Figure 3.7) in extremely demanding applications as FT(Alltoall). followed by MILC and CG as moderately demanding and finally, GROMACS, BT, WRF and CGPOP having very low network demands.

3.3 Exploring the sensitivity to task placement and bisection bandwidth

We will separate the presentation of the sensitivity of applications in two sets of graphs. First, we will analyse each application's sensitivity to the topology and fragmentation when running alone, without considering interference from other applications. Then we will introduce the cases of mixing with other applications.

3.3.1 Impact on intra-application contention

Figure 3.8 shows the sensitivity of CGPOP to fragmentation and slimming. From left to right, each group shows the impact of decreasing connectivity. Five different topologies, i.e., slimming factors

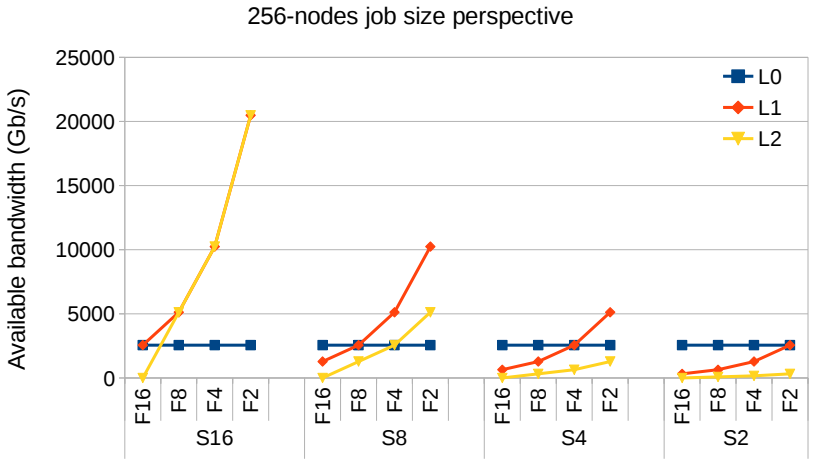

Figure 3.4: Available bandwidth from the perspective of a job of size 256-nodes per each level of a fat-tree network for different slimming factor.

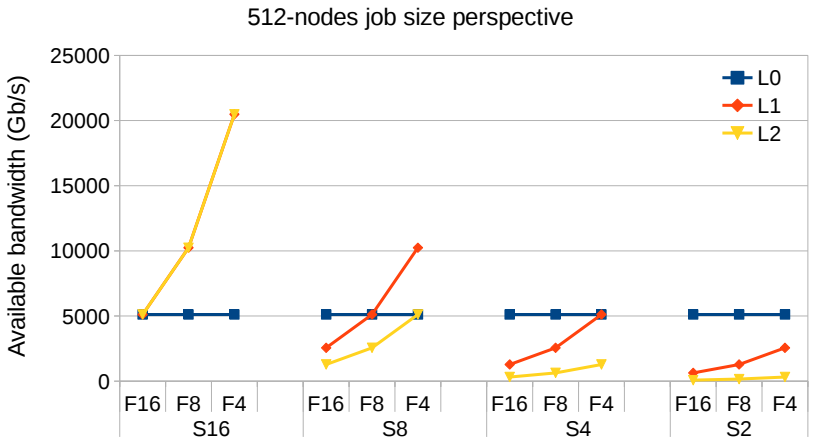

Figure 3.5: Available bandwidth from the perspective of a job of size 512-nodes per each level of a fat-tree network for different slimming factor.

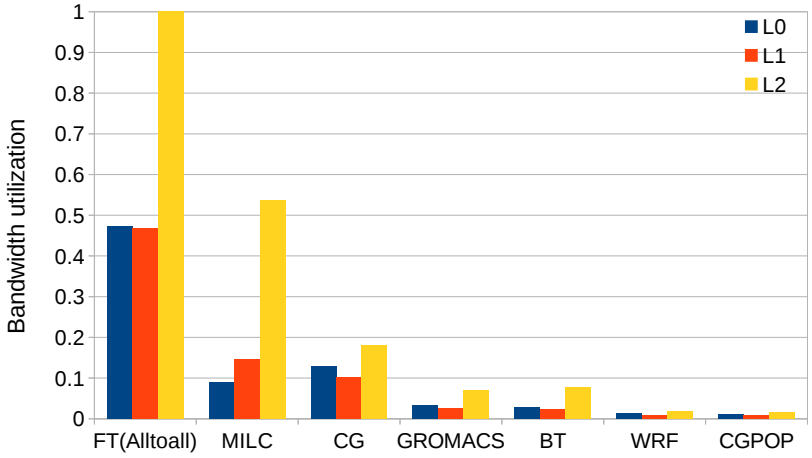

Figure 3.6: Bandwidth utilization per level for applications on xgft(3;16,16,8;1,2,2) fat-tree network and node allocation on F2 fragmentation.

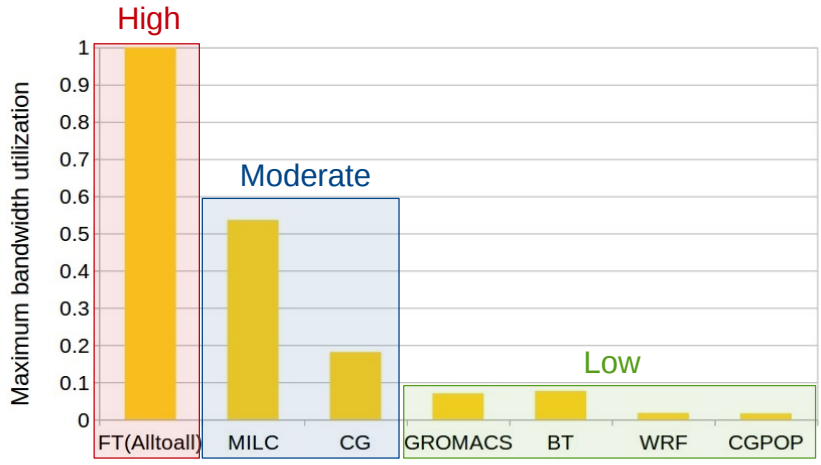

Figure 3.7: Classification of applications based on their maximum utilization.

are presented, starting with the fat tree network with full bisection bandwidth and ending with the network with the minimum connectivity. CGPOP experiences degradation of at most 3% due to slimming with respect to the case of full bisection fat tree. For each of the topologies the application was simulated using several fragmented allocations – random and regular. RF_AVG is the average value from ten runs under different random allocations. RF_STDEV is standard deviation for these ten runs. Finally, STDEV shows the standard deviation among all the presented task allocations. The trend of increasing variability when slimming the network is observed. Still, CGPOP's sensitivity to fragmentation is very low – the maximum standard deviation of the execution times for one of the topologies is 0.14%. Overall, CGPOP may be categorized as an application insensitive to both available bandwidth and fragmentation. WRF exhibits a similar behavior being less sensitive than CGPOP (Figure 3.9). WRF's degradation due to slimming is up to 2% and the standard deviation for different fragmentations is up to 0.04%.

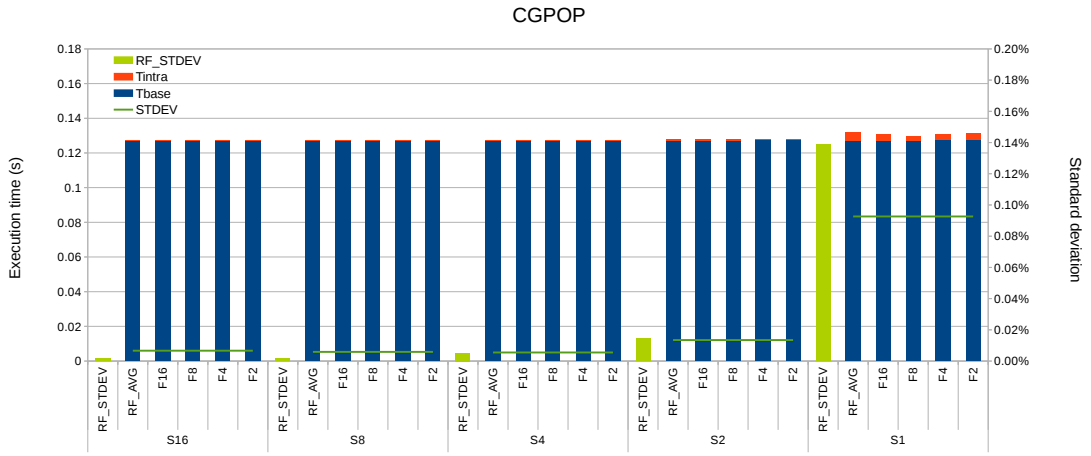

Figure 3.8: Impact of fragmentation and slimming to CGPOP performance variability when running alone in the system.

Figure 3.10 shows the results for the same set of experiments done with the application GROMACS. Regarding the effect of slimming, GROMACS can loose up to 47% of performance on the S1 topology, but in most of the presented cases (from S16 to S2) the loss is less than or around 10% compared to the case of network with full bisection bandwidth. GROMACS has an increased sensitivity to the

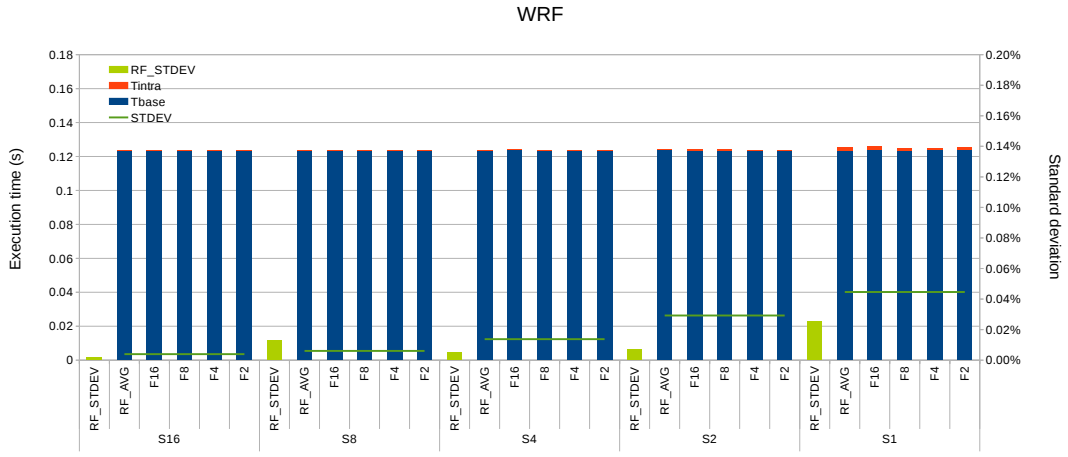

Figure 3.9: Impact of fragmentation and slimming to WRF performance variability when running alone in the system.

choice of fragmentation compared to CGPOP. The standard deviation is still less than 5%. BT fits in this category of applications with an impact of slimming up to 14% and a standard deviation due to fragmentation of 0.90% (Figure 3.11).

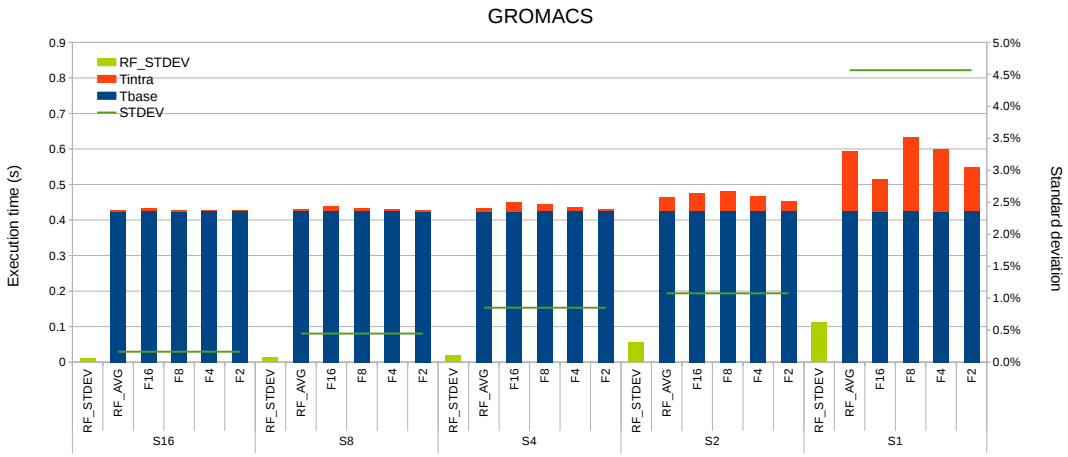

Figure 3.10: Impact of fragmentation and slimming to GROMACS performance variability when running alone in the system.

Figure 3.12 presents the case of FT(Alltoall). FT shows extreme sensitivity to slimming. Only by

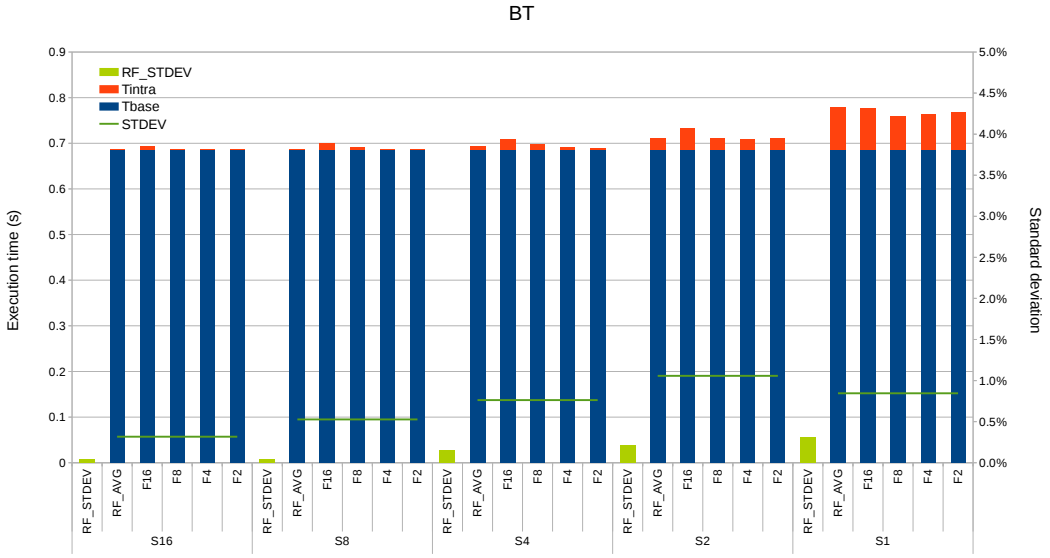

Figure 3.11: Impact of fragmentation and slimming to BT performance variability when running alone in the system.

halving the links i.e. going from S16 to S8 it experiences a degradation of 15%-66% and from S16 to S1 of 2554.92%. FT's variability due to different task allocations is extreme as well, ranging from 2.14% for S16 to 214% for S1. The FT belongs to category of extremely sensitive applications to both slimming and fragmentation. The applications CG (Figure 3.13) and MILC (Figure 3.14) also present high sensitivity to fragmentation and slimming. The degradations when going from S16 to S1 are 173% and 295% for CG and MILC, respectively. The standard deviation when using different allocations for MILC is up to 26%, while CG's performance when running alone does not vary much - standard deviation is up to 5%.

In summary, the applications with extremely low bandwidth utilization at all levels of fat-tree (WRF and CGPOP) show extreme insensitiveness to both task placement and bisection bandwidth. GROMACS and BT start to show slight sensitiveness under very slimmed network which is in line with their per-level utilization profile for the very slimmed network, i.e., they have slightly higher utilization than WRF and CGPOP. FT(Alltoall) is extremely sensitive to both fragmentation and slimming which is inline with its high-utilization at L2. The same conclusions hold for CG and MILC.

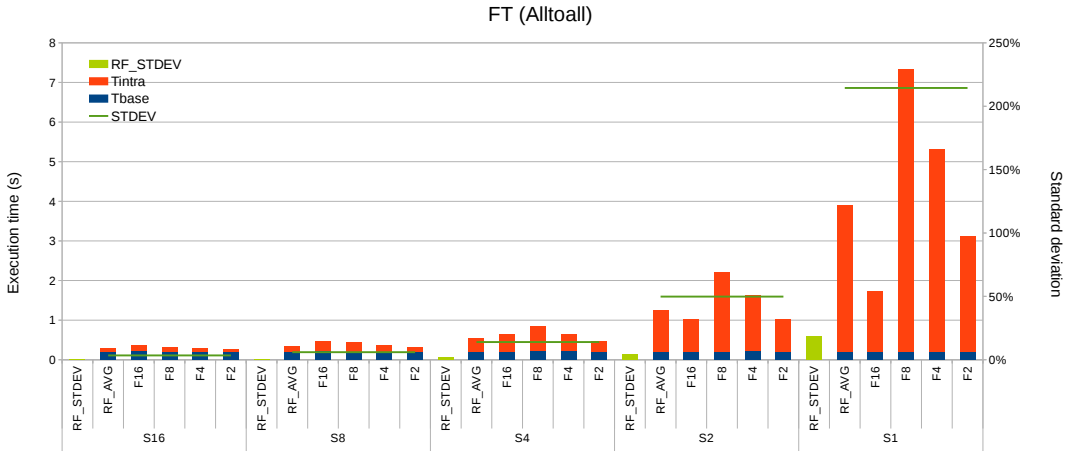

Figure 3.12: Impact of fragmentation and slimming to FT performance variability when running alone in the system.

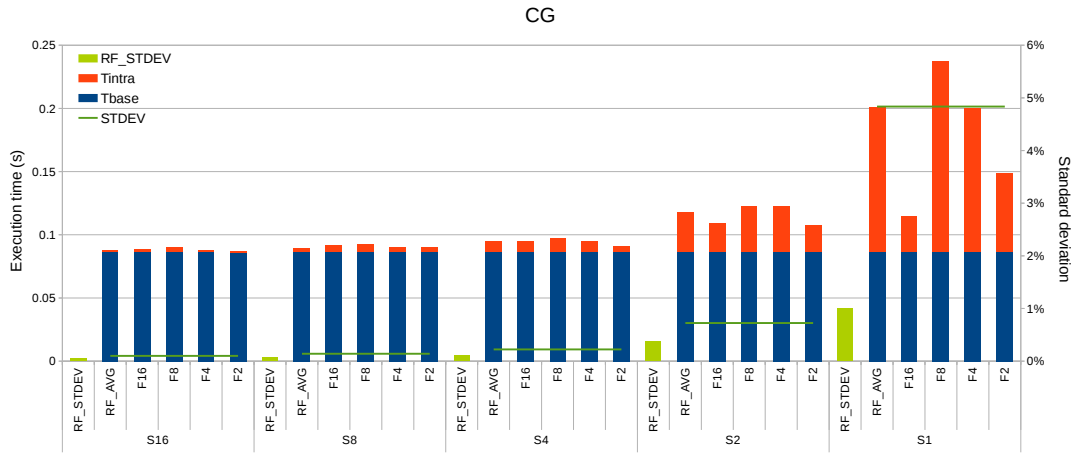

Figure 3.13: Impact of fragmentation and slimming to CG performance variability when running alone in the system.

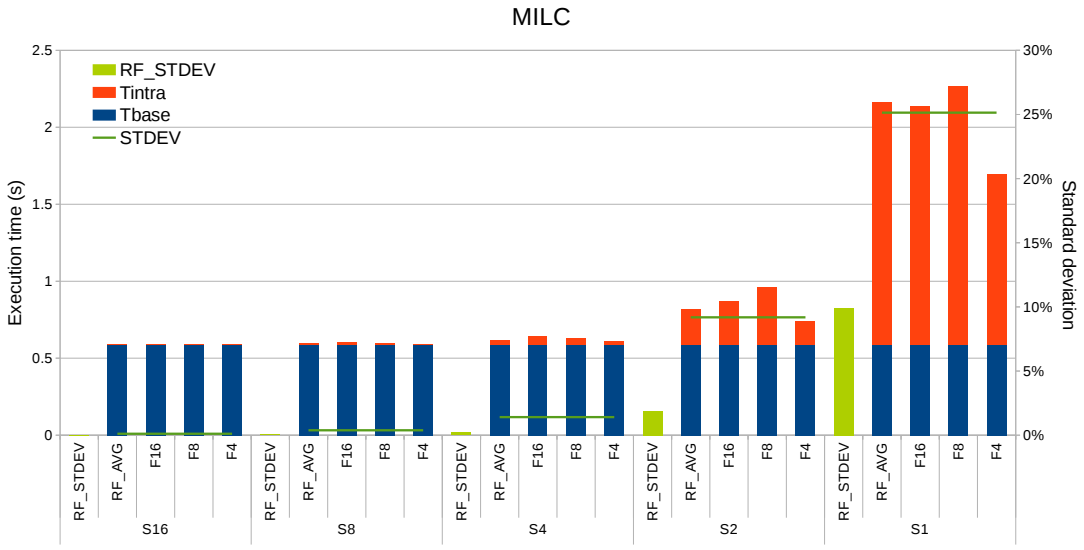

Figure 3.14: Impact of fragmentation and slimming to MILC performance variability when running alone in the system.

Therefore, our characterization methodology is able to very well predict the sensitiveness of the applications to these two important factors.

3.3.2 Impact on inter-application contention

Now that we have identified and decoupled the main factors, namely, fragmentation and slimming, that influence inter-application contention, we can present how these applications behave when they share network resources. First we categorize workloads depending on their demands and sensitiveness to bandwidth and task allocation and thus their sensitiveness to inter-application contention. In order to do this, we ran different workloads consisting of the same or different applications on ten different random allocations and on three different topologies (S8, S4 and S2).

Figure 3.15 shows the broken-down application normalized execution times of a workload created by mixing together all studied applications. The execution time of each application when running with other applications normalized with respect to T_{base} (i.e., execution time in a one-hop fully connected network). The execution time is the average value of the runs on ten different random alloca-

tions. RF_STDEV_SHARED is the standard deviation of the ten runs, while RF_STDEV_ALONE is the standard deviation of the ten runs of the application running alone (the same value as shown per application in the previous Section 3.3.1). Comparing these two values we see a clear trend of increase in performance variability, when going from scenario "alone" to a scenario of mixing applications together sharing the network. In other words, inter-application contention leads to more performance variability. Regarding slimming, from left to right we see the expected trend of performance degradation for all applications when reducing the number of links in the fat tree topology. However, not all applications "react" in the same way. We can recognize two groups: highly impacted ones (FT, CG, MILC and GROMACS) and less impacted ones (BT, CGPOP and WRF) making this seven-applications workload moderately sensitive. Note that applications that were almost completely insensitive when running alone, like CGPOP, now, sharing resources with other applications experiences 80% degradation for topology S2. The reason is the presence of applications with high communication demands (Figure 2.1) in the network. Still, in the case of higher connectivity, S8, the degradation is limited to 5% for insensitive applications and to 11% (the case of FT) for sensitive applications. In summary, slimming makes the effect of inter-application contention the biggest problem in order to increase system throughput.

Figure 3.16 shows the broken-down normalized execution times of a workload created with eight instances of the CGPOP application. The average values from ten runs on different random allocations are presented. When only CGPOPs run together the performance variability increases in the shared environment, but it is low and limited to 0.10% (standard deviation). When decreasing connectivity there is a slight increase in execution time due to inter-application contention. The maximum performance degradation is limited to 5% (the case of S2). The workload created of eight insensitive applications is quite insensitive even with very low connectivity.

Figure 3.17 shows the broken-down normalized execution times of a workload consisting of eight CG instances. The performance variability between ten runs on different random allocations is not high, standard deviation is up to 0.9%. Slimming has a huge impact on performance of CGs. Even with S8 the performance loss is around 30% for each of CG instances. The results suggest that mixing

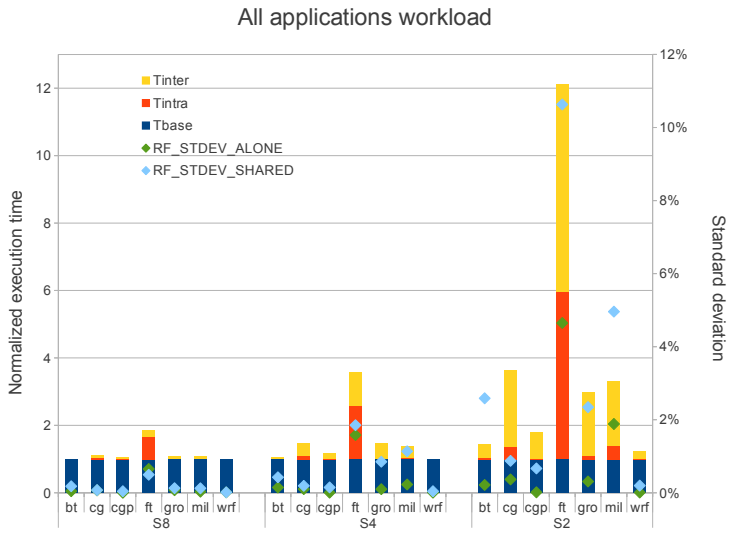

Figure 3.15: Mixing all applications together on different random allocations for three different slimmed topologies.

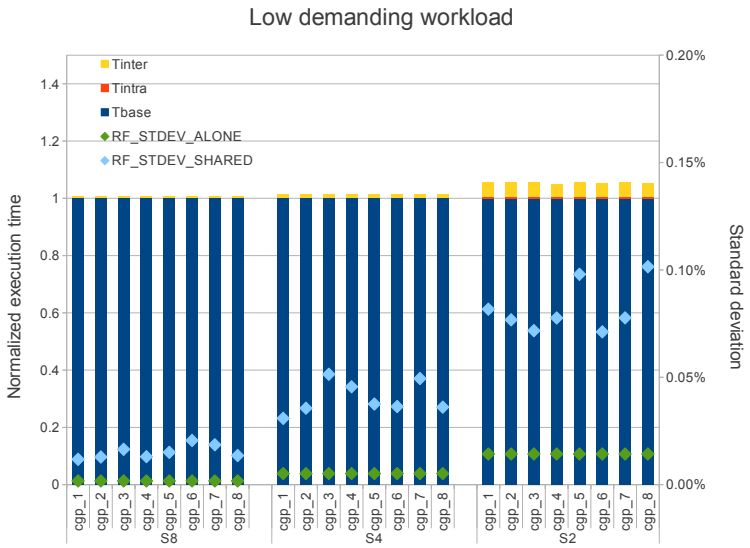

Figure 3.16: Mixing eight CGPOPs together on different random allocations for three different slimmed topologies.

sensitive applications together on random fragmentations can lead to poor performance.

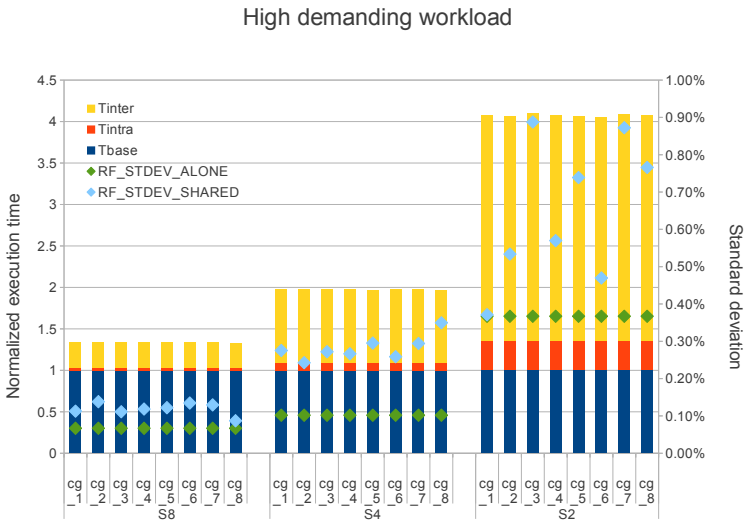

Figure 3.17: Mixing eight CGs together on different random allocations for three different slimmed topologies.

3.3.3 Trade-offs of multiple applications in a shared environment

Whenever the job scheduler needs to decide on a node allocation for a new job, assuming that several jobs are already running, several node allocations may be available; each of them may have a different impact on system performance. This is due to two main factors, namely,

- the performance of the application running on this particular node allocation, which depends on the application's communication demands and the available network bandwidth, and,
- the performance sensitivity of the mix of applications consisting of those present and the one being allocated.

To characterize these factors, let T_{inter} be the amount of time by which the application completion time increases because of network resource sharing (inter-application network contention), and let T_{intra} be the increase in completion time due to the contention produced in the system solely by the

messages that belong to the same application (intra-application contention). Finally, we will denote by T_{base} the time it would take an application to execute on a system with a crossbar network topology thus providing full bisection bandwidth and eliminating all types of contention except end-point contention. Therefore, the runtime of an application A given a particular job fragmentation F_A can be expressed as

$$T^A(F_A) = T^A_{base} + T^A_{intra}(F_A) + T^A_{inter}(F_A). \tag{3.2}$$

The completion time for all n applications plus the target application A running concurrently in the system at the moment of scheduling application A is described as

$$T(F_A) = T^A(F_A) + \sum_{j=1}^{n}(T^j_{base} + T^j_{intra}(F_j) + T^j_{inter}(F_j) + T^j_{inter}(F_A)). \tag{3.3}$$

F_j is a fragmentation allocated for application j. Note that $T^j_{inter}(F_j)$ is the inter-application produced in application j due to interference with all other applications in the system except A, while $T^j_{inter}(F_A)$ is the additional inter-application produced in application j owing to the node allocation of application A, F_A.

System throughput is inversely proportional to completion time. Therefore, an optimal node allocation F_{opt} for a given application A minimizes T. Formally, we can express this optimization as the difference in completion times from all applications between two fragmentations, $T(F_A) - T(F_{opt}) > 0$ which results in

$$T^A_{intra}(F_A) - T^A_{intra}(F_{opt}) + \sum_{j=A,1}^{n} T^j_{inter}(F_A) - \sum_{j=A,1}^{n} T^j_{inter}(F_{opt}) > 0, \tag{3.4}$$

which can be rewritten as

$$\sum_{j=A,1}^{n} T^j_{inter}(F_A) - \sum_{j=A,1}^{n} T^j_{inter}(F_{opt}) > T^A_{intra}(F_{opt}) - T^A_{intra}(F_A). \tag{3.5}$$

Therefore, obtaining a good node allocation of the target application (a node allocation that increases system throughput) assumes trade-off between intra- and inter-application contention. The increase of intra-application contention should be lower than the aggregate reduction of the inter-application contention for all the applications.

Node allocations that distribute tasks over more leaf switches have the advantage of providing more bisection bandwidth from the point of view of a given application, which may reduce intra-application contention. However, spreading tasks of the same job across the system induces the following two negative effects: i) The probability to interfere with other applications increases, which will cause more inter-application contention, ii) The mean distance between tasks increases, implying that the load on the upper levels of the tree (which have the lowest bandwidth) will increase, thus also leading to more inter-application contention. Therefore, spreading an application's tasks throughout the network is not likely to be a good node allocation policy for applications with high communication demands (unless it is alone in the system). In this case, a more contiguous node allocation would be more desirable.

3.4 Exploring the ways to reduce inter-application contention using task placement

When allocating nodes for a new job, the scheduler has two basic options:

1. Use the nodes that are currently available, or
2. hold the job for a certain amount of time until the preferred node allocation becomes available.

The first option should be chosen when the node allocation for the target application is good enough according to Equation 3.5. In the second case, the target application should be put on hold in the scheduler queue until the preferred node allocation is obtained i.e. the jobs currently running release needed nodes. Here we explore the strategies a scheduler may consider when choosing among several

possible allocations in order to reduce interference between applications. The practical feasibility of the strategies will be discussed later.

Figure 3.18 shows the same case of eight CGs workload on different regular allocations including the already analyzed random allocation on a topology with contention ratio 2:1 (the case S8). From left to right we show the trend when the spreadness of the applications' tasks is reduced. Reducing the spreadness correlates with reducing number of applications sharing first level switch (F2 case - eight applications per switch, F4 - four applications, F8 - two applications and F16 -only one application). However, the total number of applications in the system is always the same - eight. The results suggest that in the case of high demanding workload assigning random available nodes to application's tasks brings the highest degradation to their performance (up to 30%). Further, the optimal case is isolating each application on its own sub-tree (F16 case). Practically this approach is hardly feasible, since finding the non-fragmented allocations is not always available or it would require introducing certain waiting time in the queue until non-fragmented allocation becomes available. However, according to the results, another more feasible strategy would be to reduce number of applications sharing the first level switch as much as possible, since in that case the trend of reducing degradation is observed. When we move from the RF case to other less spread allocations we can get improvements of 13%, 18% and 20% in the performance of each CG instance for the cases of F4, F8 and F16, respectively.

Figure 3.19 shows the cases of different decisions on applications that share a switch. The experiments are done on a topology S8 and all applications are allocated on F8 fragmentation, thus two applications are sharing a switch and there are four applications in total in the system. Grouping two FTs on a switch even with high connectivity leads to a degradation of 47% with respect to the case of running alone on the same allocation and topology. On the other hand, two CGPOPs, two WRFs or two BTs when sharing a switch experience negligible impact on each others performance. A decision of mixing FT and the other low sensitive applications (CGPOP, WRF or BT) using the same resources leads to very significant improvement. FTs degradation in that case is only 2-3%, while for low sensitivity applications it is negligible.

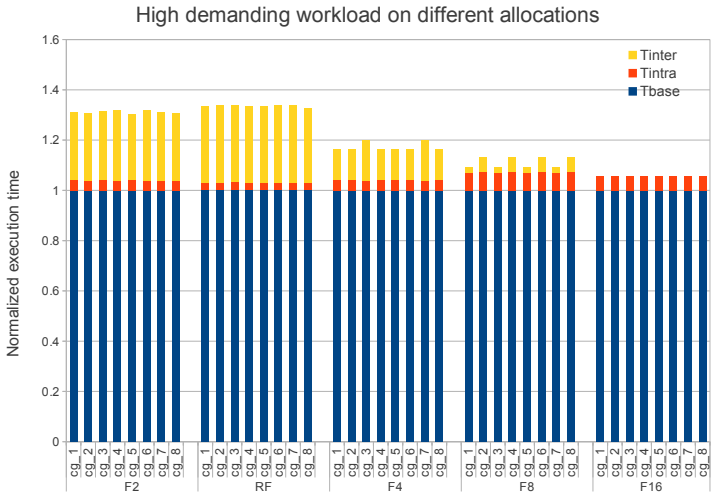

Figure 3.18: The mix of eight CGs on different allocations - random and regular.

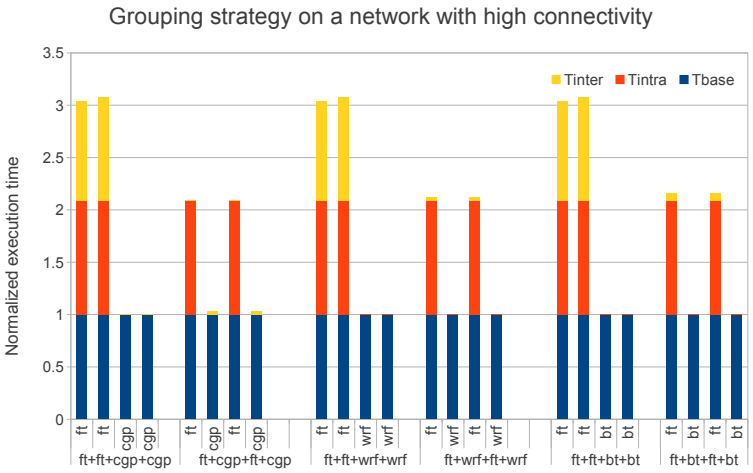

Figure 3.19: Improvement due to mixing high sensitive and low sensitive applications. The mixes of four applications: two FTs and two CGPOPs, two FTs and two WRFs and two FTs and two BTs on S8 topology.

Figure 3.20 shows the case of two FTs and two CGPOPs workload. The case of grouping FT and CGPOP on the same switch is compared to the case of isolating each of the four applications on its own sub-tree. From left to right we show results obtained on different topologies, going from less to more slimmed networks (S8 to S2). Note that the first group of the results (S8, F8 - case) is already analyzed in the previous figure. On S8 topology the high demanding applications (FT) may still benefit from using F8 fragmentations comparing to the case of non-fragmentation (F16). However, when moving towards networks with lower connectivity (S4 and S2) isolation brings better performance to both FT (reducing intra- and inter-contention) and CGPOP (reducing its inter-application contention component). The improvements are for FT 22% and 55%, and for CGPOP 13% and 40%, for topologies S4 and S2, respectively. Therefore, for the case of lower connectivity (S4 and S2) if a non-fragmented allocation is available, the best strategy is to isolate the most demanding applications at least, since all the applications will benefit from that.

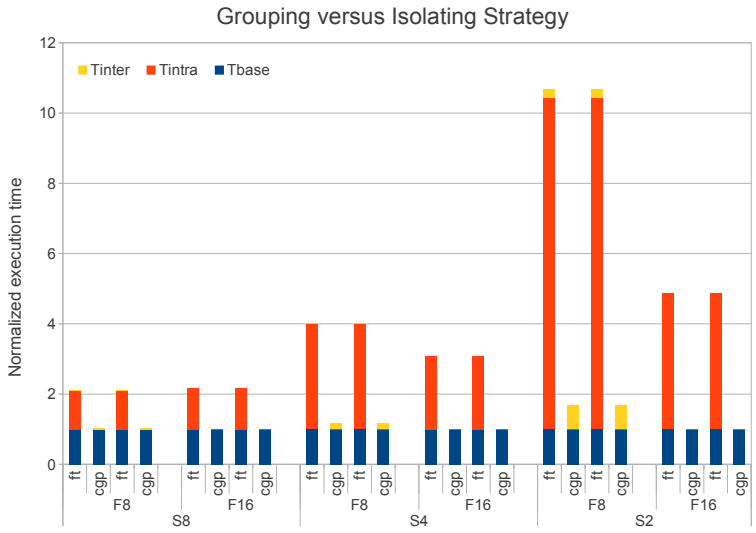

Figure 3.20: Comparing strategies of grouping and isolating for different slimming levels, S8, S4, S2. The mix of two FTs and two CGPOPs on F8 and NF allocations.

3.5 Conclusions

It was demonstrated that scientific applications may be substantially degraded by the impact of inter-application network contention that occurs when multiple applications are running concurrently in the system and thus sharing the same network resources. We have shown that the performance of some scientific applications experiences huge variability under different task allocations and fragmentation scenarios, and that this performance variability is tightly coupled to the degree of connectivity in blocking networks. We identified different classes of applications based on sensitiveness to these factors. The performance degradation may go from up to 5% for insensitive applications to up to 20 times increase in execution time for sensitive applications. Also, we found a correlation between communication intensity of applications at different network levels and their sensitiveness to fragmentation and slimming. Mixing an application with other applications in the system makes these two factors even more important since the decision on task allocation of one application impacts other applications due to inter-application contention and vice versa. The common trend in all applications is that inter-application network contention is becoming the major factor of performance degradation when the network connectivity is reduced which might be the case in future exascale computers. The workloads executed in HPC systems may be various in terms of communication intensity of their applications. Having a communication intensive application in the workload may lead to inter-application contention becoming a dominant factor in the degradation of all applications in the workload. Our results suggest two strategies for the workloads with applications of different characteristics: 1) when the network is not very slimmed the best strategy is to choose applications that share resources (e.g first level switch) carefully, mixing communication intensive with less comunication intensive applications and 2) when the network is very slimmed, then isolating application on a non-fragmented allocation is more important than careful mixing. For the workloads in which all applications are sensitive, random task allocation leads to the largest degradation - 30% in case of eight CGs workload even on a topology with high connectivity. The strategy for this kind of workloads is to keep the number of applications sharing a switch as few as possible, making the isolation of each application the optimal

case. Finally, the workloads of insensitive applications perform well even with low connectivity, and low-cost networks are a good choice for the systems that execute this kind of workloads.

However, in practice, the strategy of mixing applications based on their communication intensity is hard to achieve due to the following reasons. First, at the point of scheduling the application its communication behavior is typically unknown and therefore it cannot be decided which applications that are already running in the network is the best to mix it with. Second, due to the dynamic nature of the system, it is difficult to track for how long the applications that are already running will share the resources with the application to be scheduled. Third, even by being able to get the mentioned pieces of information, it is difficult to precisely predict which of the currently available fragmentations or the ones available in near future is the best option without running the application before. Especially, in the case of considering to wait for an allocation in the future, we cannot make good estimate on whether it pays off at all in terms of saving the performance. In summary, the mixing strategy is extremely complex approach. Thus, the strategy that remains more practically feasible is isolation. As we have seen isolation is a good strategy in the case of demanding workloads and also, isolation is becoming the prefered strategy in the blocking networks with lower connectivity. However, providing an isolated (exclusive) portion of the network to the application does not come for free. Namely, it leads to increased queueing time. In the next chapter we describe the proposed technique for solving the trade-off between providing the isolated resources and achieving reasonable queueing time.

4

System-level resource management

Supercomputers are typically shared by many parallel jobs with different resource requirements. Before execution, each job is granted the desired number of computing nodes by the system resource manager. The allocated nodes communicate through the system interconnection network. Thus, the computing nodes' physical locations will define the part of the network used by a job.

Ideally, jobs would be allocated on a set of nodes interconnected by a minimum number of switches and separated by the minimum number of hops, keeping communication between tasks of the job contained within the minimum required part of the network that satisfies the job's communications

demands. For example, if a job requires a number of nodes less or equal than a switch size, an optimal allocation would be a set of nodes within a single switch; if a job size fits in two switches it should be allocated only in two switches, preferably switches that are themselves also as close as possible (i.e., in a tree-like network, within the lowest encompassing sub-tree), etc. This kind of allocation is assumed to be ideal because it has a two-fold effect on job performance. By reducing the number of hops communication latency is reduced[60], and by allocating job's nodes as close as possible interference from nearby jobs is also reduced, increasing performance predictability[38].

However, as jobs with different resource requirements (mainly: number of nodes and execution time) arrive and leave the system in an non-deterministic fashion, allocating them to maximize system utilization will lead to a fragmentation of the resources assigned to the jobs: e.g., a job requiring a large node-count will be placed in nodes left free by previously running jobs requiring less nodes that have already finished. This problem becomes even more important in multi-stage networks, such as fat-trees (a common network topology in both the commercial and HPC domains), where fragmentation becomes more relevant the more spread out a job is on the system, as every stage increases the communication latency (up to a factor of 1.5 in state-of-the-art three-level multi-stage supercomputers[60]), and it also increases the probability of harmful interference leading to significant application performance degradation[30].

Our own analysis of the system resource manager logs from the MareNostrum supercomputer* reveals that the actual job allocations are far from the optimal ones described in the previous paragraph and that jobs are rather allocated on free fragments of switches (free portions of non-fully occupied switches) spread out onto a much higher number of switches than would be optimally required. The more and the further the fragments allocated to a job are, the higher the fragmentation in the system.

Looking only at the case of the jobs smaller than the switch size (each switch can accommodate 18 computing nodes in Marenostrum), reveals that a high percentage of the jobs that can fit in a single switch and thus not use the rest of network resources are actually spread on multiple switches (Figure 4.1).

*The MareNostrum supercomputer is a large supercomputing resource used for research in many different areas. It is run by the Barcelona Supercomputing Center, Barcelona, Spain.

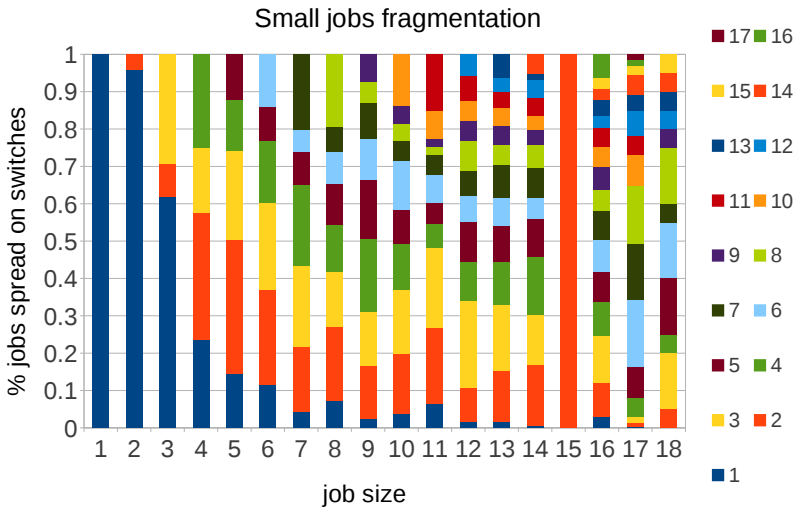

Figure 4.1: The small jobs spreadness in MareNostrum supercomputer. Percent of jobs of size x (x-axis) spread on s switches (given in the legend).

Similar conclusions are drawn for job sizes greater than a switch. The jobs requiring between 2 and 4 switches (between 36 and 72 nodes) can be found spread out to up to 55 switches (Figure 4.2), whereas the jobs whose size range from 5 to 16 switches can populate up to 96 switches (Figure 4.3).

This analysis and previous ones clearly highlight the high level of fragmentation that is being induced by the job management software in current supercomputers. Even more, as several authors have noted in great detail [58,31,18], the most negative side-effect of fragmentation, i.e., interference from other applications, cannot be accurately measured in a running system. Impact of interference cannot be accurately measured because it is unrealistic to reserve an isolated portion of the network to run the same application with the same task placement, input parameters, the same routing function, etc., to measure its execution time in the absence of interference and then compare its performance to that obtained in the presence of interference.

However, although it is practically impossible to accurately measure the impact of interference directly in a production system, several authors have managed to measure it indirectly, by looking at the performance variability across several executions of the same job, arriving to the conclusion that

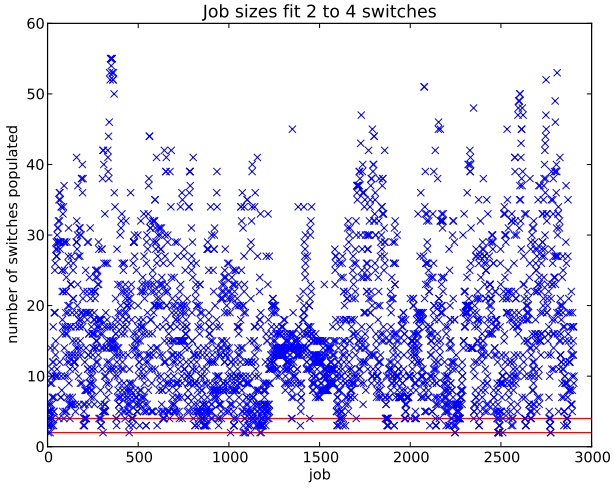

Figure 4.2: The actual number of populated switches for the job sizes range from 19 to 72 computing nodes in MareNostrum; the jobs that would ideally fit in 2 to 4 switches. The red lines are at the switches 2 and 4. 8% of jobs fit within the 2-4 switches boundary.

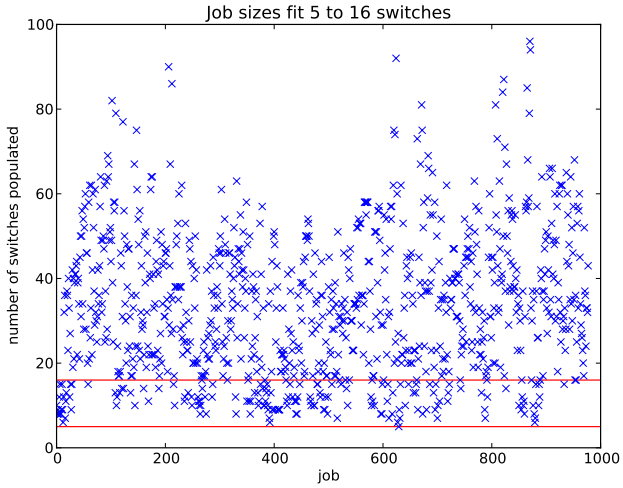

Figure 4.3: The actual number of populated switches for the job sizes range from 73 to 288 computing nodes in MareNostrum; the jobs that would ideally fit in 5 to 16 switches. The red lines are at the switches 5 and 16. 19% jobs fit within the 5-16 switches boundary.

interference is a main contributor to performance loss[31,8,30].

In part because of the difficulty to measure the impact of interference, and in part because there

hasn't been any proposal that can maintain a high system utilization while providing isolation job schedulers are to a great extent still oblivious to the spatial fragmentation induced by their policies.

In this work we attempt to reconcile system utilization and application isolation. The main insights and contributions of this work are the following:

- A model that defines the relationship between the job's allocation and the number of job's it could share network with per fat-tree level.
- We identify the causes of the high fragmentation in HPC systems based on the analysis of actual system resource manager logs from an HPC system.
- We propose and evaluate scheduling polices that can control fragmentation and reduce the number of jobs sharing the network.
- As the side-effect of the proposed policy, a portion of the network switches can be either turned off or eliminated reducing power and cost.

4.1 Network sharing as a function of job allocation

Having jobs spread out on a high number of switches consequently increases the probability of sharing the network with more jobs. Figure 4.4 shows the maximum number of jobs that a job spread on multiple switches has been sharing the network with. In particular, a job of size 8 spread on 8 switches is the worst case of fragmentation scenario for a job of this size. This means that a single node was allocated at every switch used by the job. The number of sharing jobs at second-level was 50 in this case. Similarly, the data from the system reveals that the jobs of size 64 can be spread on up to 56 switches and can share up to 120 jobs at the second level.

At the third-level, as shown in Figure 4.5, the jobs are quite fragmented, as well. A small job of size 8 can populate up to 8 different subtrees and thus use all the tree levels for every single communication. Also, at the third level the number of sharing jobs is significantly increased comparing to the previously

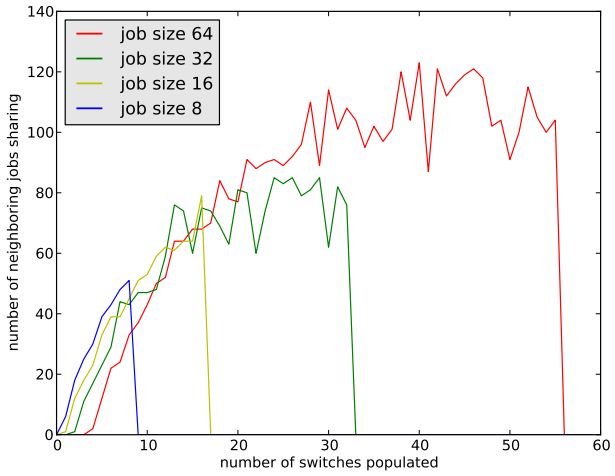

Figure 4.4: The relationship between the number of switches populated by a job and the number of jobs it shared the second-level of the MareNostrum network with. The four lines correspond to each of four typical job sizes, 8, 16, 32 and 64. For example, out of all jobs of size 16 that were spread on 10 switches, some job shared the second-level with 50 other jobs, being that the maximum number of jobs a job of size 16 spread in 10 switches shared the second-level of the network with.

seen second-level case. In particular, it ranges from 158 jobs sharing in the worst case of job size 32 to 176 sharing jobs in the worst case of job size 64.

In light of the results seen from the actual machine in Figures 4.4 and 4.5, we derive the maximum bound of the number of different jobs that could be sharing the fat tree at a certain level. Let us assume that a job of size N is spread out in the network assuming that for this job the fragments of n_{k-1} nodes are allocated at the k^{th} level. In case each of the remaining free nodes at the k^{th} level is allocated to a different job that communicates through k^{th} level our job will share the network with the maximum number of jobs. Then, the maximum number of sharing jobs at k^{th} level can be calculated as:

$$\#\text{sharingJobs}^k = \frac{N}{n_{k-1}} \cdot (m_1 \cdot \ldots \cdot m_{k-1} - n_{k-1}) \tag{4.1}$$

Simplifying formula by introducing number of populated switches and assuming equal switch radices

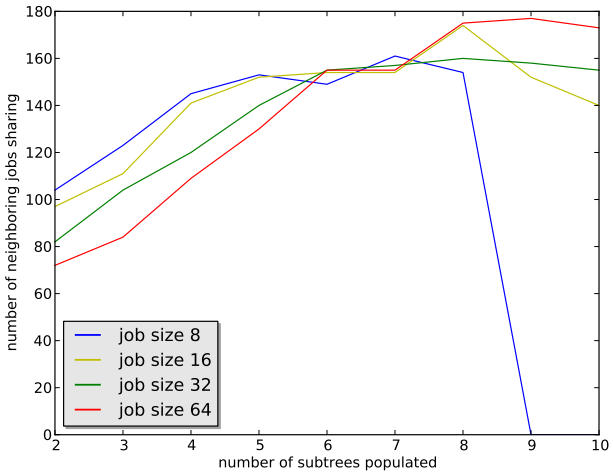

Figure 4.5: The relationship between the number of subtrees populated by a job and the number of jobs it shared third-level of MareNostrum network with. The four lines correspond to each of four typical job sizes, 8, 16, 32 and 64.

at each level, we get:

$$\#\text{sharingJobs}^k = \#\text{populatedSwitches} \cdot m^{k-1} - N \tag{4.2}$$

Thus, the worst case in number of sharing jobs for the 2nd level ($k = 2$) is for $n_1 = 1$:

$$\#\text{sharingJobs}_{MAX} = N \cdot (m - 1) \tag{4.3}$$

And similarly for the 3rd level ($k = 3$) links when $n_2 = 1$:

$$\#\text{sharingJobs}_{MAX} = N \cdot (m^2 - 1) \tag{4.4}$$

A simple example of the worst case fragmentations at the 2nd and the 3rd level in a xgft$(3; 2, 2, 2; 1, 1, 1)$ are given in Figure 4.6. In the case of the example on the left maximum number of jobs that the job A can share the 2nd level links with is $4 \cdot (2 - 1) = 4$, while the 3rd level links are shared with $\frac{4}{2} \cdot (2^2 - 2) = 4$. In the case of the example on the right, 2nd level links are shared with $2 \cdot (2 - 1) = 2$

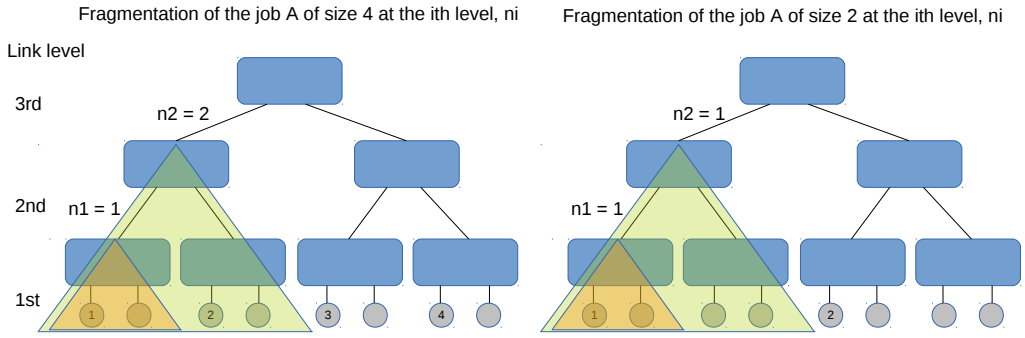

Figure 4.6: Fragmentation size at different levels of fat-tree.

jobs, whereas the 3rd level links are shared with $2 \cdot (2^2 - 1) = 6$. Note that if we divide the number with the number of switches we get the maximum number of interfering jobs at a single link. Thus, left case gives us at a 2nd level link $2 - 1 = 1$ interfering jobs and at a 3rd level link $(2^2 - 2) = 2$, whereas for right case we have $(2 - 1) = 1$ and $(2^2 - 1) = 3$ interfering jobs at a 2nd and a 3rd level link, respectively. Therefore, the number of interfering jobs increases with level and with the reducing the fragmentation size.

This model just presents the maximum amount of jobs that could be sharing a certain level of the fat tree; in reality, fortunately, the actual values are lower, despite being still very high. However, it gives us an idea of the scale of the problem fragmenation may create if not addressed properly.

4.2 Quiet neighborhoods via Virtual network blocks

Based on the analysis of the MareNostrum scheduler logs we were able to identify several causes of node fragmentation in the system:

- A significant portion of the system workload are jobs that can fit in less than a switch size, i.e, small jobs.

- The typical sizes of scientific jobs are powers of two, whereas the switch size of 18 nodes is not

power of two.

- Only 2% of the jobs bigger than switch size, i.e., big jobs, are multiple of switch size.

Therefore, for almost 100% of the workload, even an optimal contiguous job allocation will leave a number of free (not populated) nodes at the switches, i.e., fragments. We define the mechanisms to address each of the factors that contribute to fragmentation.

4.2.1 Removing the impact of fragmentation created by small jobs

To remove the fragmentation created by small jobs, two virtual network blocks are created on top of the physical topology, one block for small jobs and another for big jobs. The size of the blocks is adjusted in time based on the demand for resources from each of the two job-size groups. The limit between two virtual blocks is implemented as shown in Figure 4.7. The big jobs populate system from the first switch on, whereas the small jobs populate system from the last switch backwards. The big job's block begins at the first switch and ends up at a switch before the lowest occupied switch of small job's block. Similarly, the small job's block begins at the last switch and ends up at a switch before the highest occupied big job's block. The limit between the big job's block and the small job's block is updated dynamically. Each time a job arrives or leaves the system, the status of the highest occupied node for big job's block and the lowest occupied node for the small job's block is checked and the limits are updated if there was a change. Figure 4.8 shows the limits for small and big blocks at every time a new job comes to the system. As it can be observed the limit for small jobs was at around 160 at the begining and moved down to 139 switch index in order to accomodate more small jobs in the system.

In addition, a constraint for small jobs allocation is built, i.e., always allocate a whole small job in a single switch. Therefore, even though there will be a lot of fragmentation in the small job's block, it is not a harmful fragmentation, since there the jobs will communicate within the switch and will not use the upper levels of the network. Thus, small jobs will never experience nor create interference with nearby jobs.

One of the important benefits of this allocation algorithm is the possibility to switch-off the upper network part of the small job's block, since it will never be used. The portion of the network will depend on the ratio of small jobs in the system workload. Figure 4.8 gives an insight on the size of the small jobs block for the workload in the MareNostrum scheduler log.

4.2.2 Addressing the mismatch between the switch size and power of two big job sizes

In the big jobs block we introduce virtual switch partitions. On top of the switches we create two virtual switches containing each one 16 and 2 nodes, respectively. Based on these virtual switches we created two virtual partitions within big job's virtual block, one that contains all 16-node virtual switches and another that contains the 2-node virtual switches.

A set of constraints are built on who and how can use these partitions in order to control the fragmentation of the jobs allocated to them. First, the 16-nodes partition is not fragmentable, i.e., it can only be fully populated by a single job. Second, the 2-node partition is fragmentable.

The scheduler allocates jobs to these partitions based on the following rules:

1. Big jobs that are power of two, i.e., of sizes 32, 64, 128, etc.; they will be allocated on $n = \frac{\text{jobsize}}{16}$ 16-node virtual partitions on n switches. Note that these jobs do not have any fragmentation at the second-level switches in a fat-tree.

2. Big jobs that are not a power of two; they will be allocated on both 16-node and 2-node virtual partitions taking the whole physical switch size, thus they will need $n = \frac{\text{jobsize}}{18}$ full switches and an additional switch for the remaining nodes. Note that the remaining nodes will create fragmentation; this issue is addresed in 4.2.3.

3. Small jobs of the size 1 and 2 nodes will be allocated only on the 2-node virtual partition if an allocation for them is not available at the small jobs virtual network block. The small jobs must

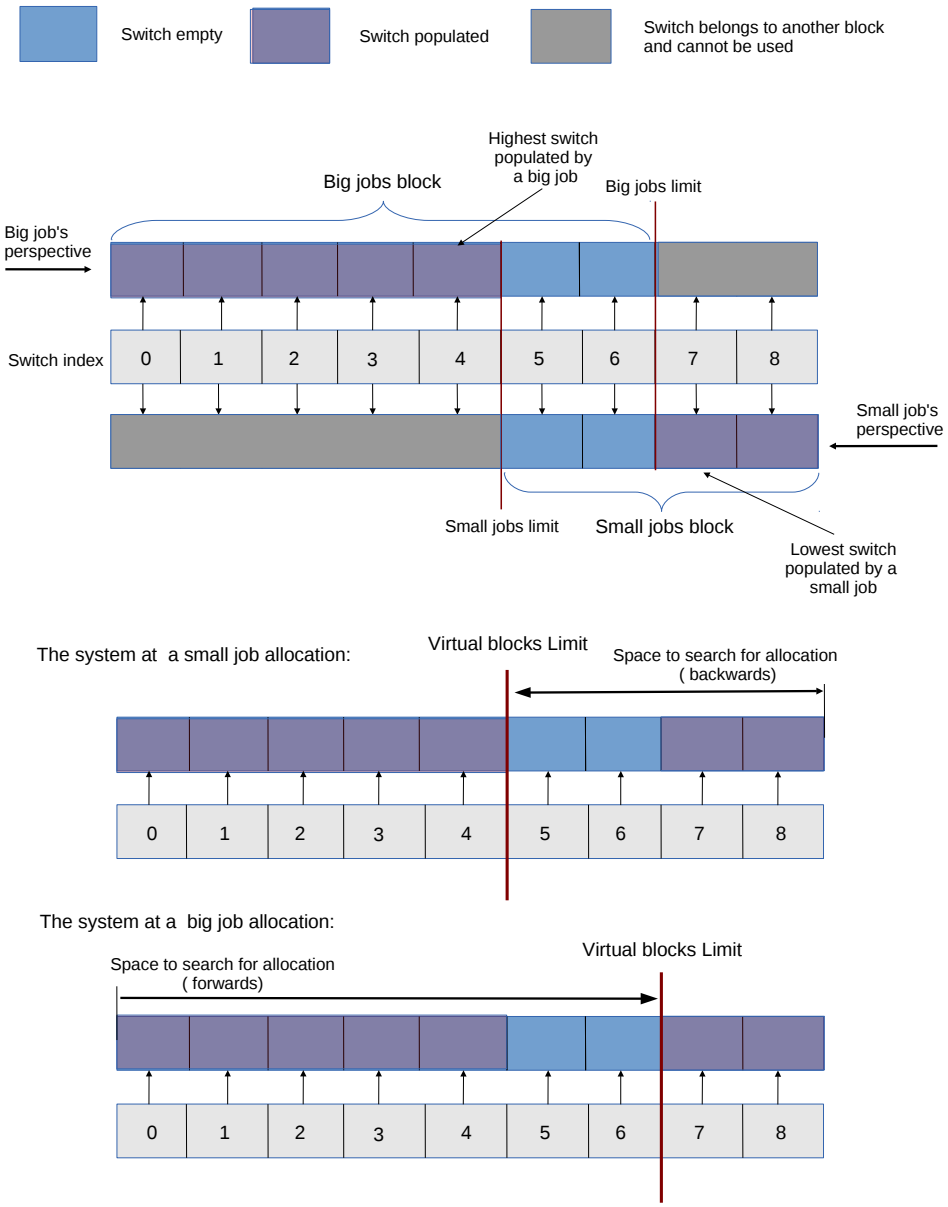

Figure 4.7: Dynamically adjusted limit between the big jobs block and small jobs block. The case of a small system of 9 switches. Big jobs are populating system from the first switch on, whereas the small jobs are populating system from the last switch backwards. In the empty system the big jobs limit would be equal to the highest switch index, i.e., 8 for the system in the figure, and the small jobs limit would be equal to the lowest switch index, i.e., 0

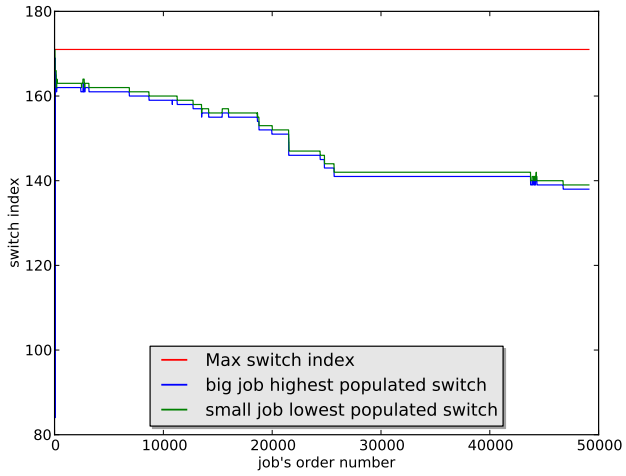

Figure 4.8: Change of dynamic limit in time. A value for each of the two limits was taken upon the allocation of new arriving job. Maximum switch index is 171. The switches above the limit do not have to be interconnected at higher levels; the percent of the switches for which higher level interconnect can be switched-off increases up to 19% over time

always retain the constraint to fit in a single switch, thus there will be no sharing of network resources even when a 16-node partition is occupied by another job.

4.2.3 Addressing the remainings nodes of other big jobs

To acommodate the remainings of the big jobs that are neither multiple of 16 nor of 18 several (rule 2) fragmentable switches are allowed per subtree, i.e., rem switches. The remainings follow the same rule as small jobs to always fit in a single switch. Since, the nodes allocated for the remainings of big jobs will communicate outside of the switch and in addition, there can be several remainings from different jobs at the same rem switch, there will be sharing of network resources in this case. Therefore, the only network sharing at second-level will be at the switches that accomodate remainings of big jobs (rem switches). Figure 4.9 illustrate how the physical switches are divided into virtual partitions within the big jobs block and the switch types in the small job's and big job's block.

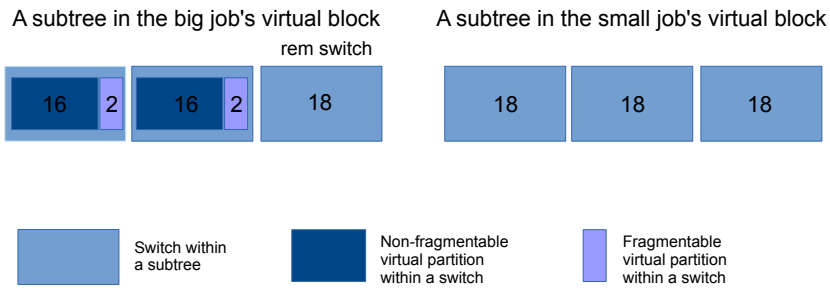

Figure 4.9: Illustration of the virtual partitions in the actual switches for the small jobs and the big jobs blocks. In the small job's block there are only fragmentable switches of size 18 nodes without virtual switch partitions, whereas in the big job's block there are both virtually partitioned switches and non-partitioned, rem switches. The example is given for the system with subtrees of three switches size.

4.2.4 Addressing the limit flexibility issue

Since the flexibility of the virtual blocks limit is bound to the duration of "the highest" job in a big block or "the lowest" job in a small block, an exception is made for small jobs that cannot find a suitable allocation during the period of the locked limit. Thus, for these jobs it is allowed to look for allocation within the already fragmented rem switches of the big job's block. Still, the small jobs in this case will retain all the constraints for small jobs and will not create any fragmentation. Moreover, allocating small jobs that do not communicate outside of the switch at rem switches will reduce probability of second-level sharing at these switches, that was identified as an issue in 4.2.3.

4.2.5 Addressing fragmentation at third-level

To reduce the fragmentation at the third-level in a fat-tree a contiguous allocation strategy is applied that gives priority to allocate switches that are contiguous in the system. This helps, for example, to prevent 32-nodes jobs being allocated in two switches, each one in a different subtree, even when two switches within the same subtree are available.

Note that the only source of sharing are due to the remainings nodes from big jobs allocations, and also fragmentation at third-level from any job as well.

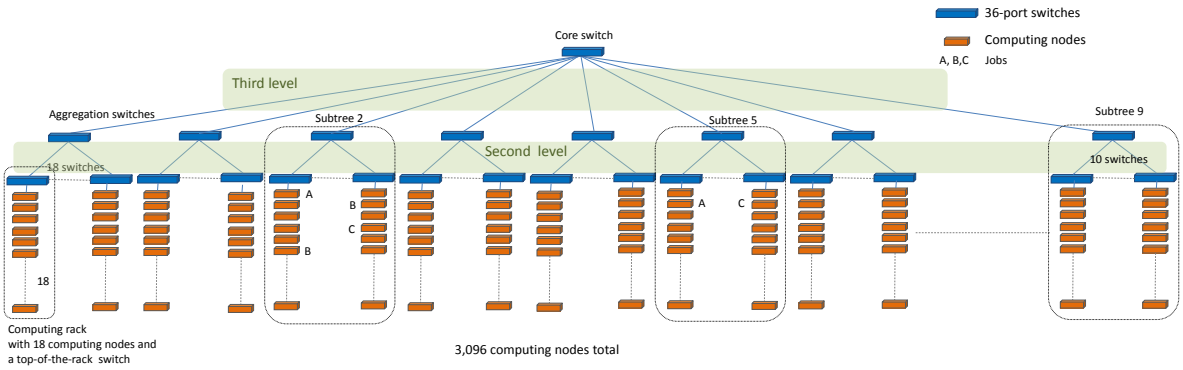

Figure 4.10: Overview of the MareNostrum system.

Note also, that it might happen that some jobs cannot be allocated to the system even when there is available computing nodes because of the constraints that the proposed scheduling is imposing. In this case, jobs have to wait in the scheduler waiting queue until enough nodes become available on their corresponding virtual partitions. This issue can be addresed by setting up carefully the amount of rem switches. This is achieved dynamically based on the demand for these types of jobs during the normal system operation.

4.3 Experiments

The simulator explained in Chapter 2 models the resource allocation in the Marenostrum system. There is a total of 3,096 computing nodes in Marenostrum organized in several racks. Figure 4.10 depicts the Marenostrum system. The system network topology is a 3-level fat-tree. Each computing rack contains 18 computing nodes that are connected to the top-of-the-rack switch. Racks are grouped into ten subtrees using the agregator switches. Each subtree contains 18 computing racks except for the last one that contains ten racks. A core switch provides connectivity to the aggregator switches. The network topology is a fully non-blocking.

The metrics that report network sharing by jobs are calculated per each level in order to see how

different policies are loading the different network levels. An example of the calculation of this metric is ilustrated in Figure 4.10 where three jobs (A,B,and C) each using two computing nodes are in different racks of the system. Jobs A and C are being placed in two different subtrees, 2 and 5, whereas job B is being placed solely in subtree 2. Job A shares the up link at level 2 in the top-of-the-rack switch with job B and the up link at level 3 in the aggregation switch with job C. Job B shares the up link at level 2 in the top-of-the-rack switch with A and C. And finally, job C shares with B the up link at level 2 in the top-of-the-rack switch in subtree 2, but shares with only job A in the up link at level 3 in the aggregation switch in the subtree 5. The resulting network sharing per job is two because each job shares the network with other two different jobs. The total number of sharing jobs would be three, each job shares with another job. The network sharing per level 2 would be two job pairs (A with B and B with C), and for level 3 would be one job pair, only A and C share the network.

For comparison purposes several scheduling policies have been implemented in order to compare with our proposed policy. They are described below.

- First available. Jobs are allocated to the first available computing nodes starting from the first switch to the last switch in the system. This policy does not look if the job completely fits in some switch of the system so it is prone to generate fragmentation.

- First contiguous. This policy chooses switches that fit completely the job first and tries to allocate computing nodes together in contiguous switches as much as possible. In case there are no contiguous switches then chooses computing nodes as in the First available policy.

- Exclusive. It allocates jobs in computing nodes that do not share the network with other jobs. Therefore, this policy achieves no network sharing at all, so it is the best policy in terms of minimizing network sharing. In order to achieve that, small jobs must be allocated to a single switch and big jobs must be allocated to one or more subtrees where there are no other big jobs allocated there already. Note that small jobs and big jobs can share the same subtree because small jobs will not use network resources. In case there is no available computing nodes that fulfill the aforementioned rules then the jobs must wait on the queue. As it can be expected

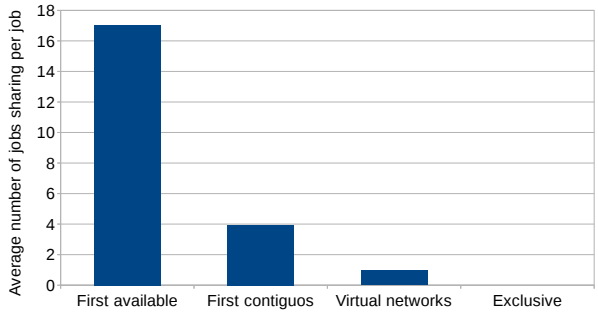

Figure 4.11: Average number of jobs that shared the network with a single job during the execution of the 49107 jobs workload from MareNostrum log for each of the four evaluated scheduling policies.

this policy is prone to increase job's waiting queue time.

4.4 Evaluation

This section shows the evaluation of the proposed policy. The evaluation is performed using a trace of the workload obtained recently from the MareNostrum system during 32 consecutive days. This trace contains a list of 49,107 different jobs from multiple users that were running during that period in the system. The proposed policy is compared with the other traditional policies, First available and First contiguous. It is also compared with the Exclusive policy that achieves no job sharing at all.

Figure 4.11 shows the average number of jobs that are shared per job for the different policies evaluated. As it can be seen, the First available policy is the one that shows the highest value, 17 shared jobs on average. On the other hand, the Exclusive policy shows no sharing per job because it guarantees that no other job will use the job's part of the network before allocating the computing nodes to it. As we will see later, this advantage comes with a substantial penalty on the completion time.

The First contiguous policy reduces by a factor of 4.3× the amount of network sharing with respect to First available policy. And even better is the reduction achieved by the Virtual network policy that achieves an additional factor of 4× with respect to First contigous policy achieving on average only

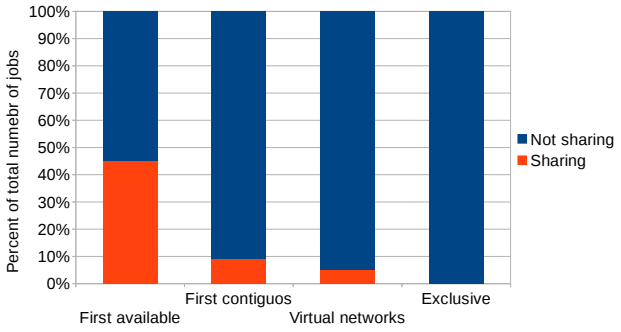

Figure 4.12: Percent of jobs that shared network with other jobs during the execution of the 49107 jobs workload from MareNostrum log for each of the four evaluated scheduling policies.

0.9 jobs sharing per job.

Figure 4.12 shows the percent of jobs that shared network with other jobs from the whole trace totaling 49,107 jobs. As expected, the Exclusive policy achieves no sharing at all, whereas First available policy achieves the highest number of jobs shared. However, our proposed policy achieves a significant reduction on the total number of jobs sharing achieving only 5% which corresponds to a reduction factor of 9× with respect to First available and a 45% reduction with respect to First contiguous policy.

The above results clearly show that our proposed policy is quite efficient in achieving a significant reduction on the amount of jobs sharing the network simultaneously. The reason for this is the segregation of jobs per job-size in the isolated virtual networks blocks as described in Section 4.2. However, jobs might be waiting more time in the waiting queue in order to get space available on a virtual block. This effect is going to be evaluated as follows.

The results in Figure 4.13 show the number of job pairs that were sharing the network at each of the levels of the tree. First available is worse than the other policies at both the second and the third level. There is a huge improvement of First contiguous over First available, in both the second and the third level. On the contrary from First available, First contiguous is having more sharing on the third level than on the second level, since it is not solving the fragmentation at the third level. Our policy improves over First contiguous policy by a factor 8.4× at the second level and by the factor of 3.9× at

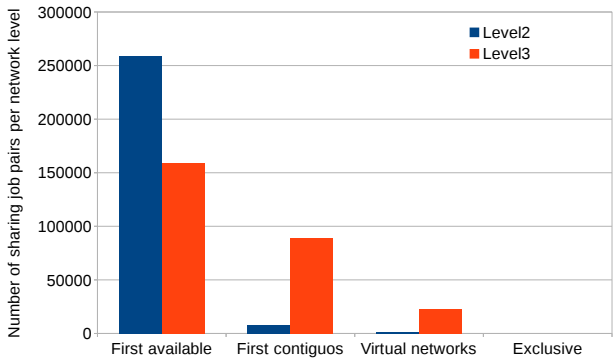

Figure 4.13: Number of job pairs that shared network at the 2nd and at the 3rd network level during the execution of the 49107 jobs workload from MareNostrum log for each of the four evaluated scheduling policies.

the third level. The Exclusive policy, as previously seen, does not introduce any network sharing.

Figure 4.14 shows the completion time for the various scheduling policies. As can be seen, both policies, First available and First contiguous, show the lowest completion time, 32 days. On the other hand, the Exclusive policy shows the largest completion time, 39 days. A 23% increase of time due to the restriction to allocate exclusive subtrees for parallel jobs larger than 18. This policy delays substantially completion time. However, our policy obtains a completion time of only 33 days which corresponds an increase of only a 4% with respect to the first two policies. The increase in completion time of our policy is almost negligible and significantly lower than for the Exclusive policy, while maintaing a high level of isolation between applications, effectively eliminating interference between applications that in practice could reduce the execution time of each job.

This increase of the completion time is due to fact that our policy holds jobs on the queue longer time waiting for the proper computing nodes to become available. This is illustrated in Figure 4.15. As can be seen, Virtual networks policy increases the queue time of jobs on average around sixteen minutes. The Exclusive policy introduces an increase in the average waiting queue time per job of up to 28 minutes.

Figure 4.16 shows the portion of the system computing nodes being allocated to jobs in average for all four policies. As can be seen, although our policy increases the job queue time, the system utilization

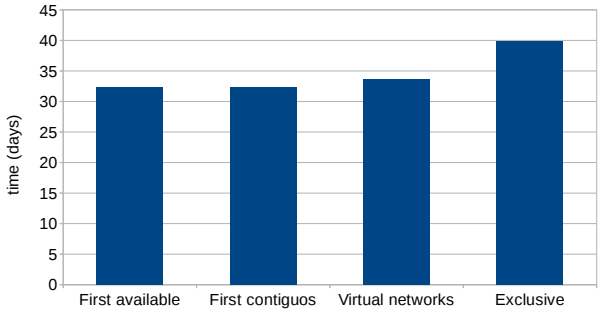

Figure 4.14: Completion time of the 49107 jobs workload from MareNostrum log using each of the four evaluated scheduling policies.

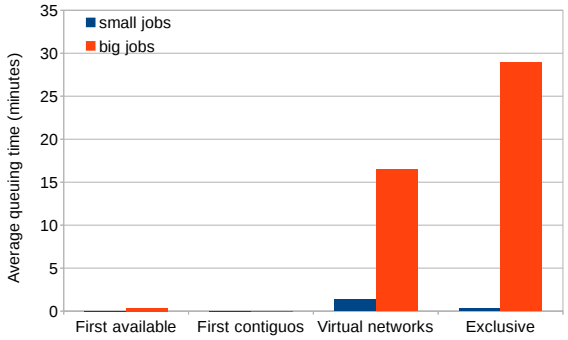

Figure 4.15: Average time a job was waiting in the queue for the allocation. The data shown for small and for big jobs. The data was obtained from the execution of the 49107 jobs workload from MareNostrum log for each of the four evaluated scheduling policies.

is reduced by only 1%. Virtual networks achieve a similar utilization to the other two policies that does not show a significant waiting queue time (First available and First contiguous). The system utilization for these policies is 84%. On the other hand, a much lower node utilization is observed for the Exclusive policy, only 76% which represents a 10% less utilization than for the rest of the policies. This respresents a significant loss of computing power in the system which is not desirable.

Figures 4.17, 4.18, 4.19, and 4.20 display the job allocation for each policy, First available, First contigous, Virtual network block, and Exclusive, respectively, when the number of jobs processed in the

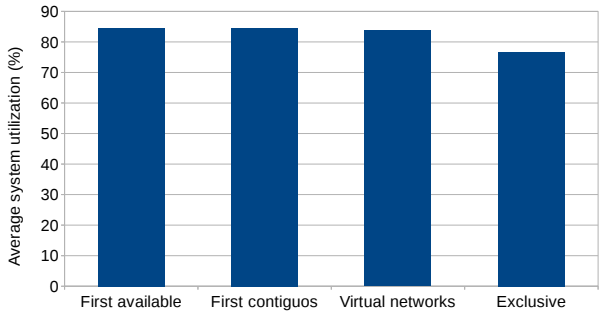

Figure 4.16: Average system computing node utilization during the execution of the 49107 jobs workload from MareNostrum log for each of the four evaluated scheduling policies.

trace is around one half, i.e. at the 25000th job of the trace. The figures show all 172 computing racks in Marenostrum grouped into all ten subtrees. Every row of racks corresponds to a subtree. Thus, the racks within a row are interconnected with the second level network and the racks at different rows are interconnected with the third level network. In order to easily visualize the job allocation in the system each job is colored with a different color. Black color is used for jobs with a number of computing nodes equal or less than the amount of computing nodes available per switch (18), i.e., small jobs, whereas other colors, except grey color, is used to visualize jobs with a number of computing nodes larger than 18. Grey is a special color used to display that the computing node has not yet any job allocated on it.

In Figure 4.18 that shows the First avaible policy it can be observed that some computing racks are displayed with multiple different colors meaning that multiple different parallel jobs are being allocated to these racks increasing the sharing among jobs. Moreover, black colors are spread out among multiple different computing racks which may point out one of the cause of the high level of sharing of the racks identified before.

On the other hand, on Figure 4.18 that shows the First contiguous policy it shows a different bahavior on job allocation than the First avaiable shown before. Now, the computing racks are clearly showing much less sharing of the computing racks by multiple jobs as racks are not shown as multiple

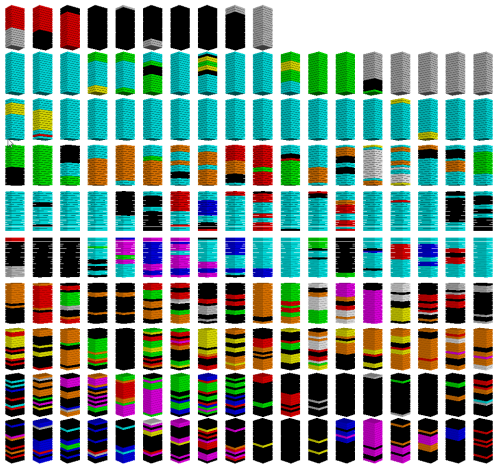

Figure 4.17: First available policy. Status of the system population in the middle of simulation. Grey depicts the node that is not used by any job. Black color depicts small jobs, i.e., less or equal than 18. Other colors depict big jobs.

Figure 4.18: First contiguous policy. Status of the system population in the middle of simulation. Grey depicts the node that is not used by any job. Black color depicts small jobs, i.e., less or equal than 18. Other colors depict big jobs.

colors as before. Nevertheless, still jobs of the size equal or less than 18 are spread out through the entire system which may cause high fragmentation for larger job sizes.

On the other hand, Figure 4.19 shows the Virtual network block policy where it can be seen that jobs of the size equal or lower than 18 are more concentrate into an particular area of the system, on the last two subtrees. This area is the small jobs virtual block, explained in 5.1 In addition, it can be

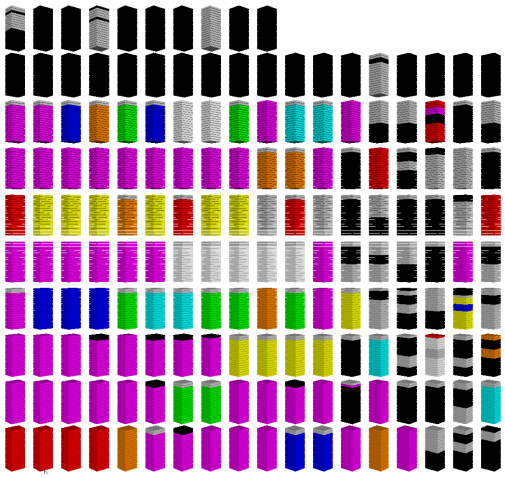

Figure 4.19: Virtual network block. Status of the system population in the middle of simulation. Grey depicts the node that is not used by any job. Black color depicts small jobs, i.e., less or equal than 18. Other colors depict big jobs.

observed that racks are shared by less number of jobs than First available policy. This policy shows the advantage of the First contigous buy fixing the spreadness of the jobs whose size is equal or less than 18.

And finally, the Exclusive policy is shown on Figure 4.20. It can be seen that racks only share large jobs with job sizes equal or less than 18, but at the expense of showing a higher number of computing nodes without any job allocated as can be seen on grey nodes.

4.5 Conclusions

In this work we identified key characteristics of the distribution of job sizes in shared HPC systems, consistent with many previous studies characterizing the dynamic and diverse workloads that their job schedulers have to cope with.

We abstracted the main characteristics having an impact on the spatial axis of decisions for the job schedulers, namely, that a non-negligible portion of jobs at any time in the system are small jobs that fit in a single switch, and that most large jobs usually have very concrete sizes.

From this abstractions we were able to propose a novel, simple technique that, by virtualizing net-

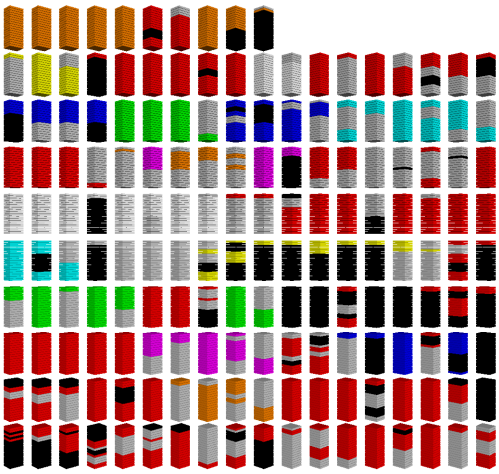

Figure 4.20: Exclusive policy. Status of the system population in the middle of simulation. Grey depicts the node that is not used by any job. Black color depicts small jobs, i.e., less or equal than 18. Other colors depict big jobs.

work resources at different levels, is able to transform the chaotic arrival distribution of job sizes and durations into a stable spatial segregation of jobs with a dynamic boundary that can adapt to changes in the arrival distribution.

Our policy, by being able to hierarchically and flexibly place jobs within contained parts of the network minimizing interference is able to achieve very high system utilization, similar to topology and interference oblivious techniques, while at the same time achieving an unprecedented level of isolation and thus performance predictability for the individual applications.

Moreover, the segregation technique led to two very beneficial side-effects yet to be explored in detail: 1) potential hardware savings, in the form of requiring less or no virtual channels to provide Quality of Service features to isolate applications, as we are able achieve it through the scheduling decisions, or by reducing the capacity of the second-level and third-level switches for the part of the network that accepts only small jobs, or alternatively, instead of reducing the capacity, 2) potentially saving power by selectively switching off the corresponding part of the network that accepts only the small jobs.

As we have shown, our policy for the sake of flexibility and system utilization allows a small percent of jobs to share the network resorces. Potential interference impact on these jobs' performance and

consequently the system performance is mitigated by additional resource management techniques at the link-level described in the next chapter. The two techniques at system-level and at the link-level are compatible.

5

Link-level resource management

Interconnection networks for high-performance computers (HPC) have traditionally been operated on a best-effort basis. Network capacity is based on peak traffic load estimates, so that networks have generally been over-provisioned. This approach is simple for systems with predictable traffic loads where the performance achieved fits the user expectations for a small set of selected applications of interest.

However, as systems are being scaled out, they can proportionally accommodate more applications running concurrently, which implies that multiple messages of different applications compete for the

same network resources.

Inter-application contention depends on many factors such as routing, task mapping, application communication patterns, and the relative start times of applications. The lack of predictability of the effects of inter-application contention leads to low throughput, high latency and high-jitter, as the competition of applications for network resources disrupts the performance of the applications more sensitive to contention degrading the overall system throughput. We evaluated the degradation in HPC systems reporting the system throughput loss of 10%-30% [29].

The problem of inter-application contention is already well-known in the Internet arena where different data flows might compete for the same network resources. The solution in the context of Internet applications was to use Quality-of-Service (QoS) mechanisms to guarantee certain levels of performance to specific data flows such as real-time video streaming. This effort was referred to as Differentiated Services [9] by the Internet Engineering Task Force (IETF). A classification of the different traffic types was performed in [47] and revised and implemented in InfiniBand networks in [3]. In particular, the InfiniBand QoS mechanism is based on the use of virtual lanes (VLs) and the corresponding arbitration tables that define the bandwidth allocated to each VL. Additionally, InfiniBand supports three priority levels, where packets on the higher priority are served first regardless of the bandwidth allocated for the lower priority. By default the highest priority is reserved for subnet management traffic, and the rest for user traffic. The methodology used in [3] was based on allocating time-sensitive traffic to high-priority VLs, and using low-priority VLs for other traffic. In the latter case, it attempts to guarantee that traffic is allocated to the VL that matches its bandwidth needs.

Recently, QoS is being used in HPC, as well in order to reduce the impact of inter-application contention for scientific codes. HPC system administrators can apply specific QoS policies on a per-job basis by means of system resources management tools such as the Unified Fabric Manager Software [40] (UFM) from Mellanox. However, the QoS policies employed are still quite basic and coarse-grained, and only benefit a small set of applications. Basically, applications are classified into two broad traffic classes, namely latency-sensitive and bandwidth-sensitive applications, where latency-sensitive applications are mapped to high-priority VLs and the rest to low-priority VLs.

In this work we propose an effective quality-of-service policy for capacity HPC systems. HPC applications are usually not real-time and are different from the internet-like traffic analyzed in previous works [47,3]. Depending on the application, its characteristics lie on a continuum between latency-sensitive and bandwith-sensitive traffic.

The technique that we propose provides a wider classification of applications, and hence can leverage QoS mechanisms for a large number of applications resulting in an increase of system performance. The proposed QoS policy provides a method to effectively map applications to VLs, and provides an effective distribution of bandwidth for each of these VLs. The proposed techniques are fully supported in InfiniBand and do not require any additional hardware capability. Specifically, the contributions of this work are as follows:

- A method to map applications into VLs focused on minimizing inter-application contention. This method dynamically maps applications during runtime to approach the optimal mapping based on behavior of the applications that are running using characterization methodology proposed in Chapter 3.

- We show that segregating applications on different VLs significantly reduces inter-application contention. Reductions in contention time from 10% - 60% are achieved.

- We also provide a method to group applications into virtual lanes in case there are more applications than virtual lanes based on the previously presented characterization.

- We provide a technique to effectively distribute available network bandwidth to virtual lanes. It is demonstrated that for HPC codes it is very important to detect communication intense applications at the critical point of the network and separate them from less intense applications by assigning an exclusive VL. In this way, the burstiness of such an application can be regulated tuning the VL's bandwidth.

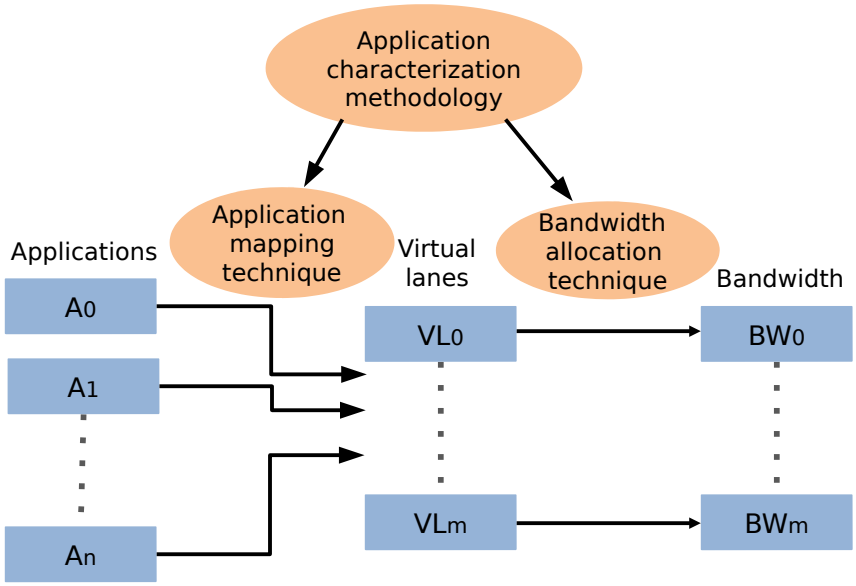

Figure 5.1: Techniques developed for the new proposed QoS policy.

5.1 Proposed Quality-of-Service Policy

The proposed QoS policy is depicted in Figure 5.1. It is based on two techniques: (i) a mapping technique that determines the most suitable VL for each application; (ii) a method to properly allocate bandwidth to each VL. The descriptions of these techniques are presented below.

5.1.1 Application Mapping to VLs

The method to properly map applications to the available VLs is depicted in Figure 5.2.

The method distinguishes between two cases. When the number of applications is less than or equal to the number of VLs, each application will be allocated to an independent VL to partially eliminate the performance degradation caused by inter-application contention. The reason for this is

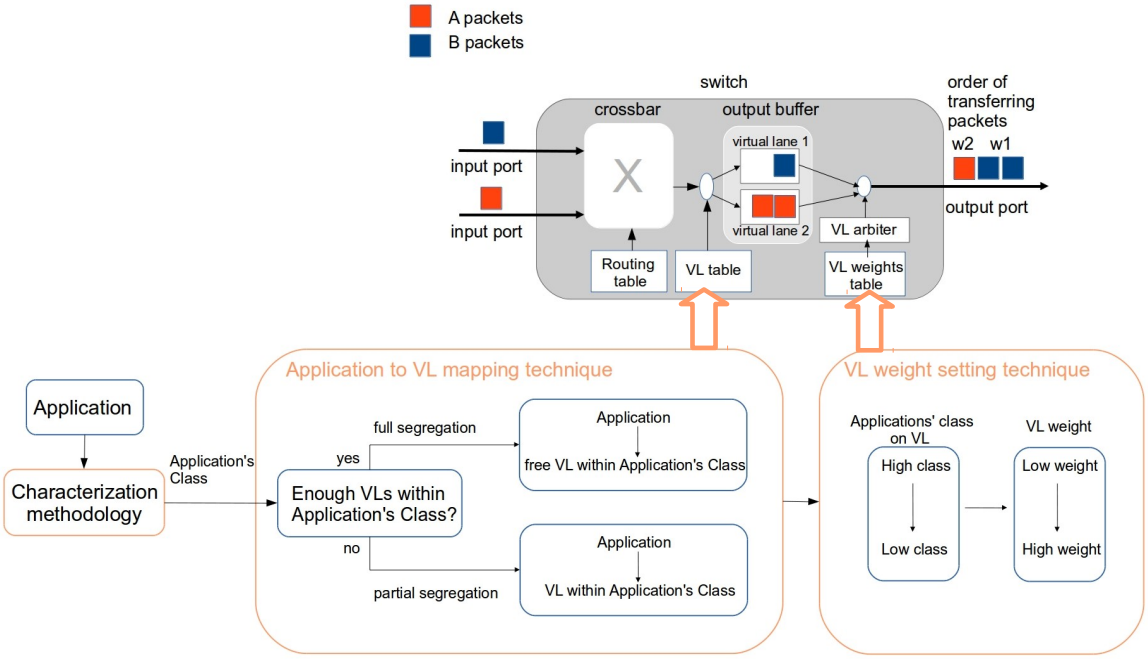

Figure 5.2: Algorithm for mapping applications into VLs.

simple: segregating applications into VLs helps to partially reduce the interactions between them. For example, if two streams of data – A and B – share the same VL and B is coming last to the network, it will suffer the inter-application contention from A according to Equation 1.3.

Therefore, when B is allocated to another VL the VL arbiter can transmit B packets just after the current A transmission finishes if it was not blocked. Therefore, segregating applications into VLs eliminates the $T^{A}_{congestion}$ and $T^{A}_{blocking}$ terms from the inter-application contention, but not the current $T^{A}_{transfer}$.

In the case of having more applications sharing network resources than available VLs, the method of mapping applications into VLs is based on grouping applications that are compatible into groups, where number of groups are equal to number of available VLs. For a measure of compatibility, we use the characterization described in Chapter 3.

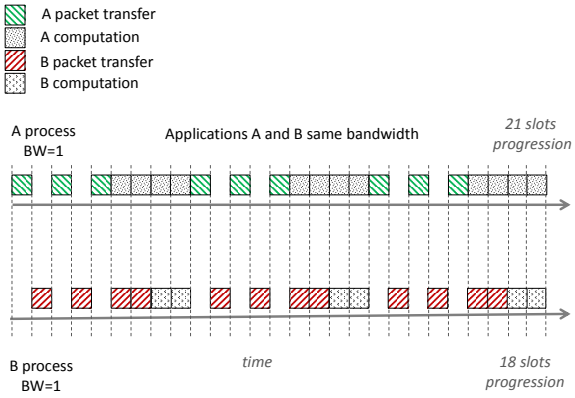

Figure 5.3: Timeline showing the progression of two applications, A and B, where to each application is assigned the same bandwidth.

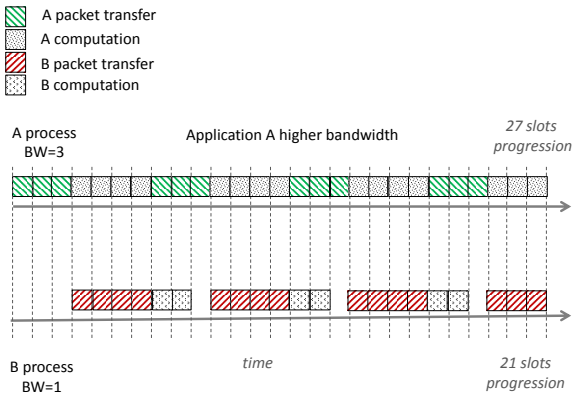

Figure 5.4: Timeline showing the progression of two applications where to A is assigned higher bandwidth than B.

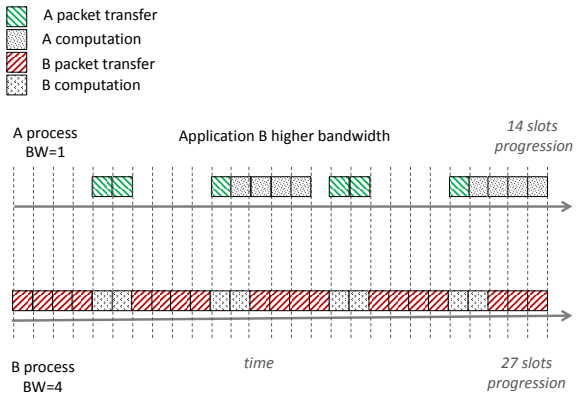

Figure 5.5: Timeline showing the progression of two applications where to B is assigned higher bandwidth than to A.

5.1.2 Understanding the impact of VL weights on reducing of interference

In this section, we will show the necessity for bandwidth allocation technique on VLs in order to further reduce inter-application contention.

The technique is based on assigning more bandwidth to VLs that have applications that have low communication demands. This heuristic may be seen counter-intuitive because one might expect that

applications that are less communication demanding should be assigned less bandwidth. However, as we will see below, it is more beneficial to assign more bandwidth to the applications with low communication demands.

The principle behind this effect is illustrated in the following Figures. 5.3, 5.4 and 5.5. The examples in figures are based on behavior that we observed in the traces of two applications' simultaneous executions using our visualization tool and are simplified here for clarity. We will give concrete examples of real applications and their communication behavior later on. Each figure shows the execution timeline of two applications with quite different communication demands, A and B, when performing several iterations over a period of 27 time slots. One iteration is composed of three packet transfers and four computation slots for A, whereas for B, which has higher communication demands, each iteration comprises four packet transfers and two computation slots. In each time slot, applications can either perform computation or communication, or stall waiting for the network to become available. To model network contention, we assume that only one application can communicate in each time slot. As shown in Figure 5.3 when both applications are assigned the same bandwidth ($BW_A = BW_B = 1$), the transfer of packets of these applications are interleaved in time because they have the same bandwidth. This interleaving of packets due to network contention is unfortunately delaying both applications. The resulting progression of A and B is 21 and 18, respectively, out of the total 27 available slots.

In the other case shown in Figure 5.4, A is given three times more bandwidth than B, so it can transfer three consecutive packets before B can transfer any packet ($BW_A = 3, BW_B = 1$). In this case, A is not suffering any network contention. Moreover, although the communication of B is delayed at the beginning, it is not suffering as much delay as in the previous case, because most of its transfers occur when A is computing. The result is that both applications are able to make more progress, and hence the system achieves higher overall throughput. In particular, the progress of A and B is 27 and 21 slots, respectively.

Finally, the example shown in Figure 5.5 shows that giving more bandwidth to B, which is more communication demanding, is not as efficient as the previous case. In particular, B can communicate

as much as it needs in each iteration so $BW_B = 4, BW_A = 1$. As a result, the performance of B is boosted at the expense of A. The resulting progress for B and A is 27 and 14 slots, respectively. Therefore, assigning more bandwidth to the more communication demanding application is not the best strategy.

The bandwidth allocation technique that we are proposing is based on assigning more bandwidth to applications that have a higher computation to communication ratio. Therefore, we can still use the metric for characterization as proposed earlier.

Nevertheless, determining the optimal bandwidth for each VL might be quite complex, because multiple applications may be assigned to a given VL. And also, giving more bandwidth than necessary to low-communication demanding applications might degrade performance of high-bandwidth applications. Therefore, we recommend in practice a trial-and-error approach to carefully assign more bandwidth to less communication-intensive applications and observe the impact on performance to more communication-intensive applications to achieve an optimal balance.

To implement this bandwidth allocation technique in InfiniBand, applications have to be mapped to the available low-priority VLs based on the grouping technique proposed in Section 5.1.1. Then, the weights in each VL should be assigned in such a way that VLs containing less communication-intensive applications have more weight than those containing more communication-intensive applications.

5.2 Simulation

In this section, we introduce the metric used to measure inter-application network contention, and we describe the simulation environment and the applications used to evaluate our proposal. Subsequently, we present the results showing a) the effect of mapping applications to VLs and b) the impact of bandwidth allocation.

5.2.1 Inter-application contention metric

Impact of inter-application network contention is measured as an increase in the absolute execution time that an application experiences due to sharing network resources with other applications. We want to measure this contention to identify the optimal QoS policy. As we know, a QoS policy Q_j is a function of three factors: 1) assigning applications or groups of applications to SL classes, 2) mapping SLs to VLs and 3) allocating bandwidth (weight) assigned to VLs. Note that in our proposed QoS policy factors 1) and 2) are managed by our application-to-VL mapping technique. We employ the notation "VLnxm" to indicate that VL n has relative weight m.

To quantify this impact for a particular set of QoS policy parameters Q_j, we used the metric explained in Chapter 2. Namely, the application was simulated in the same system twice. First, it was executed alone, i.e., without any inter-application contention. The resulting execution time is used as a reference time T_{alone} (see Table 5.3). Then, it was run simultaneously with another application (or several applications) sharing the system and thus experiencing inter-application contention. We will refer to the completion time of an application in the latter scenario as $T_{sharing}$. Therefore, the inter-application contention for an application i can be calculated as

$$T^{i}_{contention}(Q_j) = T^{i}_{sharing}(Q_j) - T^{i}_{alone}. \tag{5.1}$$

Note that all system parameters and settings (e.g., size of the network, task allocation, routing, MTU size, etc.) have to be the same in both scenarios so that the increase in the execution time of the application can be attributed solely to the inter-application contention, and not to a coupled effect of contention and other factors. Moreover, to obtain a view of the amount of inter-application contention for a given combination of applications we compute the total inter-application contention time $T_{total\text{-}contention}$ of n applications as follows:

$$T_{total\text{-}contention}(Q_j) = \sum_{i=1}^{n} T^{i}_{contention}(Q_j). \tag{5.2}$$

If all the applications use the same number of compute nodes C, each compute node having com-

pute power P GFLOPS, then the system-level compute power waste, P_w, in GFLOPs (s stands for plural, and not for "per second") is calculated as

$$P_w = C \cdot P \cdot \sum_{i=1}^{n} T^i_{\text{contention}}(Q_j). \tag{5.3}$$

The QoS policy Q_{opt} is the most efficient one within a set of N QoS policies if it satisfies

$$T_{\text{total-contention}}(Q_{opt}) = \min_{j \in N}(T_{\text{total-contention}}(Q_j)). \tag{5.4}$$

5.2.2 Simulation setup

The simulated systems parameters are given in the Table 5.1. The network topology used is a blocking ("slimmed") three-level fat tree. We simulate networks of large scale (details in Table 5.2). The slimmed networks are usually characterized by contention factor—ratio between the input ports coming from the lower level of the fat-tree and the output ports that go to the upper level of fat-tree in a switch.

The trace was originally collected on a machine with IBM's 2.5 GHz PowerPC 970 processors from 2005 with a peak performance of 27,6 Gflops per chip. Today's IBM processors such as Power7 achieve a peak of 264 Gflops. This results in roughly 10X faster computations, which is the speed up factor used in our simulations. In addition, the simulated network (10 Gb/s) is five times faster than the original network (2 Gb/s Myrinet). The workloads executed on the system consisted of a mix of two, three, or four applications. The settings for the experiments with different number of applications in the mix were different in terms of total number of nodes, task allocation, etc. (see Table 5.2).

One MPI application process is allocated per node in our simulator. We are assuming that multiple cores within a node are fully utilized speeding up its computation phase which is the typical scenario in emerging hybrid parallel programing languages, i.e. MPI+OpenMP, MPI+StarSs.

To mimic the effects of non-contiguous node allocation (fragmentation), we interleave the tasks belonging to different applications such that each set of nodes attached to the same first-level switch

are evenly distributed across the applications. For example, in the case of two applications, each application uses eight nodes per switch. We use the notations F4 and F8 to indicate that four and eight nodes are used per application at the switch, respectively.

In the experiments, the SL and VL are set at the adapter and are not changed at any point in the system during the whole execution of application. In our simulator the weights for each VL are set as the number of packets that can be served per turn for that VL.

Table 5.1: Parameters used in the simulations

Simulator	Venus-Dimemas
Topologies	Extended Generalized Fat Trees (XGFT) [45,49]
Switch Technology	InfiniBand
Switch Size	32-ports
Buffers	Input/Output
Network Bandwidth	10 Gbits/s
Segment size	4 KB
MPI Latency	1 μs
CPU Speedup	10x
Routing scheme	Random routing [24,21]

5.2.3 Applications

In our experiments we employed four NAS Parallel Benchmarks from the NPB3.3-MPI release: FT, BT, CG and MG. The details on applications' traces cuts used for the experiments are given in Table 5.3.

The bandwidth utilization per level obtained by our characterization methodology is shown for each of the considered applications in Figure 5.6 and Figure 5.7 for the case of node allocation F8 and F4, respectively. When on F8 fragmentation FT and CG fall into category of moderately intensive

Table 5.2: System settings and task allocations for various workload sizes

#Apps	XGFT topology	#Nodes (used/idle)	Contention ratio	Task allocation	#VLs/Buffer size per VL	Reference times
2	(3; 16, 16, 2; 1, 4, 4)	512/0	4:1	F8	1/64KB, 2/32KB	T_{alone}(buff. 32KB) see Table 5.3
3	(3; 16, 16, 4; 1, 4, 4)	768/256	3:1	F4	1/64KB, 4(3 used)/16KB	T_{alone}(buff. 16KB) see Table 5.3
4	(3; 16, 16, 4; 1, 4, 4)	1024/0	4:1	F4	1/64KB, 4/16KB	T_{alone}(buff. 16KB) see Table 5.3

Table 5.3: Applications traces details and reference times (T_{alone}).

App.	Problem size	Msg sizes	#Tasks	Orig. trace	Used trace cut	#Iters.	T_{alone} (buff.32KB,1VL,F8)	T_{alone} (buff.16KB,1VL,F4)
FT	Class D	512 KB	256	415 s	75 s	5	9.111861 s	8.192807 s
CG	Class D	732 KB	256	882 s	75 s	337	8.191318 s	7.95694 s
BT	Class D	761 KB, 158 KB, 26 KB	256	620 s	75 s	22	7.534967 s	7.459007 s
MG	Class D	260 KB, 128 KB, 65 KB, 32 KB	256	111 s	75 s	36	not used	7.304353 s

applications, while BT and MG are low communicating applications. In the case of F4 fragmentation, FT is still moderately intensive, whereas CG, BT and MG are low intesive applications. Note that the CG's utilization at Lo of fat-tree is much higher than the one at other levels and also it is higher than Lo utilization of the rest of applications. However, Lo traffic is not relevant for the inter-application contention since all applications have exclusive computing node-switch network links at Lo, i.e., a node is exclusively allocated to one application.

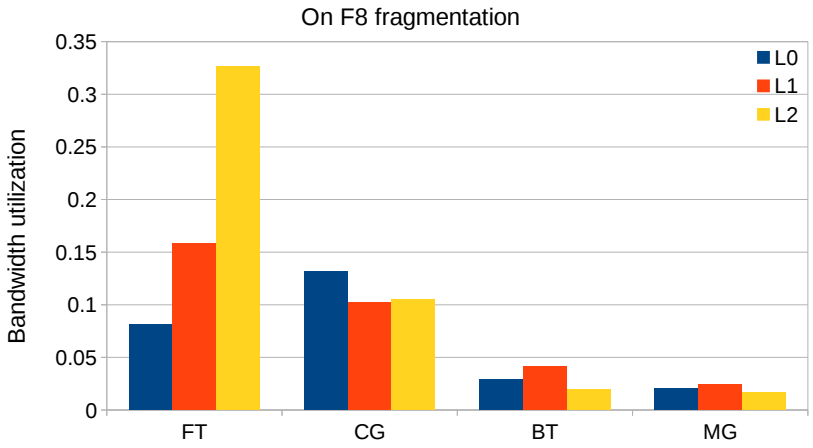

Figure 5.6: Bandwidth utilization per level for applications on xgft(3;16,16,4;1,4,4) fat-tree network and node allocation on F8 fragmentation.

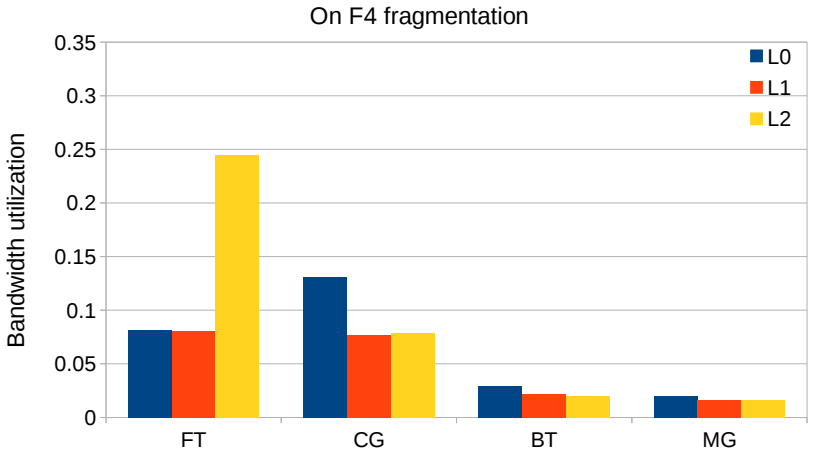

Figure 5.7: Bandwidth utilization per level for applications on xgft(3;16,16,4;1,4,4) fat-tree network and node allocation on F4 fragmentation.

5.3 Results of the proposed techniques

5.3.1 The benefit from segregation of applications into VLs

We evaluate the scenario in which the number of VLs configured in the system is equal to or greater than the number of applications that are being executed on the system.

For this scenario, we evaluate the benefits of segregating applications into VLs in such a way that each application is assigned to an independent VL exclusively for its entire execution. Also, applications are assigned to VLs in the following order, the first application in the workload indicated on the x-axis is mapped to VL0, the second to VL1, etc. Here we assume all VLs are of low priority unless specified differently.

Uniform bandwidth allocation

In this part we will evaluate the case when the weights assigned to all VLs are the same and allow each VL to send one packet in each turn (we annotate it as vl0x1, vl1x1). We refer to this as interleaving of VLs. We compare the results of segregating with those achieved by assigning all applications to the same VL (vl0). Note that the total buffer space should remain the same size in the case of using one VL and using several VLs. Table 5.2 provides the details on per VL buffer size for each set of experiments.

Figure 5.8 shows the total inter-application contention time for the case of using only one VL and the case of segregating applications on two VLs for the two-application mixes. As can be seen, by segregating applications into two VLs a substantial improvement is achieved. In particular, reduction of the contention time ranges from 7% for two FT applications executed together to 59% when FT shares the network with BT. This result indicates that in the case where there are as many VLs as applications, each application should be mapped to a different VL. The case of CG and BT where the total contention time is increased upon segregation is most probably caused by dividing up the available buffer size. Namely, when CG runs with BT on VL0 they share 64KB buffer. As BT communicates much less then CG, CG can use the whole buffer most of the time. When we segregate two

applications on VL0 and VL1 each can use only half of the buffer space - 32KB.

Figure 5.9 shows the normalized execution time for each application for the same mixes as before. We can see that segregating onto different VLs significantly improves the performance of both applications in most cases. Also, there are some cases in which the performance of one application is substantially improved at the cost of slightly degrading the other one. In particular, when sharing the network with FT, CG may be improved by 8% through segregation, and FT is only degraded by 1%. On the other hand, we can obtain improvements for both applications by segregation when the two are the same applications (e.g., FT with FT, CG with CG).

Figure 5.10 shows the total contention time for the three-application mixes. When increasing the number of applications in the workload we again obtain substantial reductions in the contention time owing to the segregation. Specifically, this reduction is 22% for the case of a mix of FT, CG, and BT. Moreover, a workload comprising four applications increases the individual contention time of each application and thus total contention time. However, also in this scenario segregating achieves a reduction in inter-application contention. In particular, a reduction of 32% is observed compared with the case of non-segregating.

Figure 5.11 shows the impact of segregation on each application's overall performance in the three-applications mix (on the left) and the four-applications mix (on the right). The improvement in performance is seen for all applications but one - FT. However, degradation of FT, around 1% is less than the improvement of any other application in both three- and four-application mixes.

Non-uniform bandwidth allocation

Figures 5.12, 5.13 and 5.14 show the total contention time for the FT+CG, FT+BT, and CG+BT mixes, respectively, when assigning different weights to VLs as explained in Sec. 5.1.2. Note that we are assuming full segregation of applications into VLs. Also, the mapping of applications to VLs is constant (vl0 - blue, vl1- red). Additionally, we show the two extreme cases of assigning full link bandwidth to VL0 and VL1 which corresponds to assigning a high priority to these VLs. The inter-application contention gradually diminishes in the application as its bandwidth allocation increases, but at the same

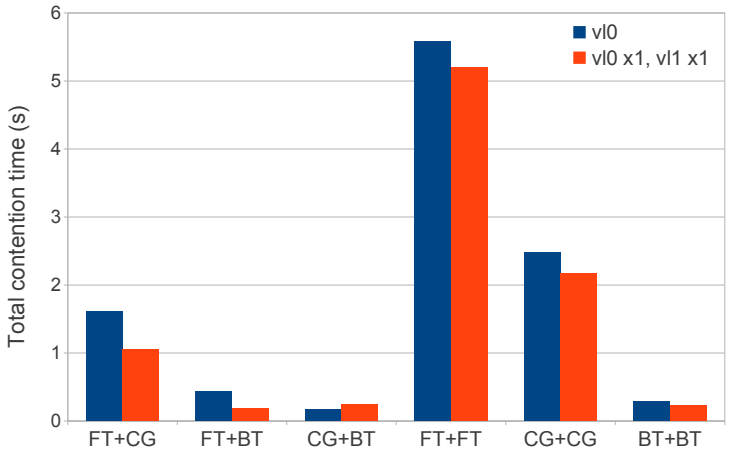

Figure 5.8: Total contention time when using one and two VLs for the two-application mixes.

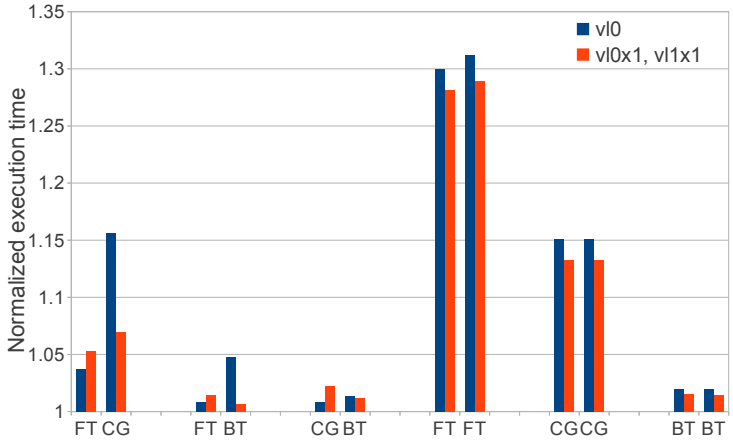

Figure 5.9: Impact on the execution time of each application when using one and two VLs for the two-application mixes.

time increases the contention of the other application. In other words, the ratio between individual application contention times is changing when favoring one VL (vl0, on the left) or another VL (vl1, on the right). However, a sweet spot is reached around the middle of the graphs on where the total

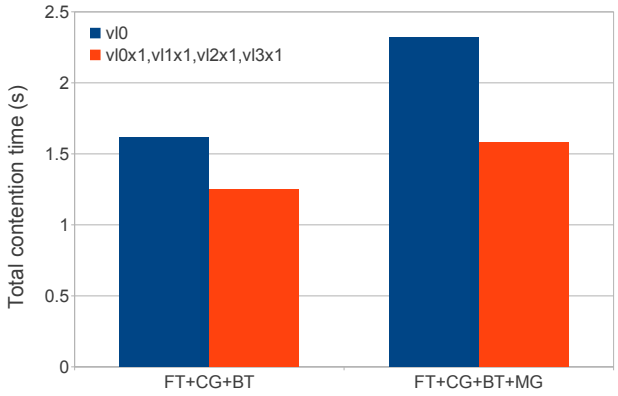

Figure 5.10: Total contention time when using one and three/four virtual lanes for the three/four-application mixes. In FT+CG+BT, FT on VL0, CG on VL1 and BT on VL2. In FT+CG+BT+MG, FT on VL0, CG on VL1, BT on VL2 and MG on VL3.

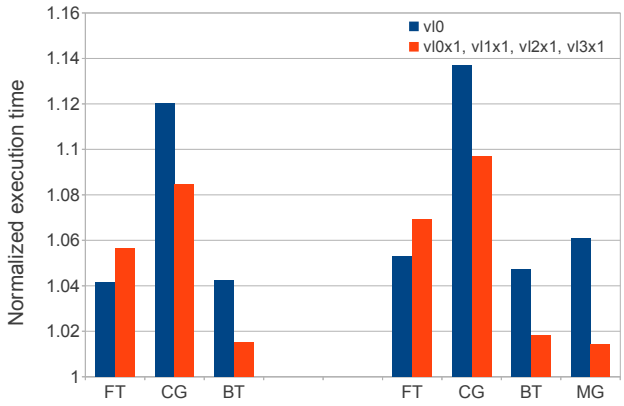

Figure 5.11: Impact of segregation on the execution time of each application in three/four application mixes.

contention time is lowest.

In particular, for the FT+CG mix the optimal strategy is achieved when FT (vlo) sends one packet and CG (vl1) sends two packets per turn (vlo x1 vl1 x2). The reduction of total contention time is around 7% compared to case of equal-weight interleaving (each VL sends one packet in a turn). In application classes terms, this is the case of moderate-intensity application (FT) sharing resources with

another moderate-intensity application (CG). By assigning less bandwidth to application of higher intensity we are, in fact, performing regulation of FT's flow. Similar behavior is also observed for the mixes FT+BT where the minimum total contention is observed at (vl0 x1 vl1 x4). Since BT is of lower intensity than CG the mutual impact is lower than in the case of FT+CG. The case of CG and BT sharing the network reaches minimum contention at (vl0 x1 vl1 x8), but overall the impact of their interference on the performance is not too high. This result suggest that the total contention time is smaller when the application that has lower communication demands is favored. Note that this bandwidth allocation strategy improves the performance of one application without excessively degrading the other one, thereby resulting in an overall decrease of the total contention time.

The case of three-application mixes is shown in Figure 5.15. For this case a similar behavior is observed. The largest reduction in contention is achieved when the bandwidth allocation favors the application with lower communication demand and restricts the bandwidth of applications of higher intensity (FT). In this case we can see that the optimum is achieved for vl0 x1 vl1 x2 vl2 x4 which gives more weight to the lowest communication demanding application (BT), then to the second lowest (CG) and finally, the least weight to most demanding application (FT). The improvement observed is 9% with respect to the case of equal-weight interleaving (vl0 x1 vl1 x1 vl2 x1). Note that any deviation from this bandwidth allocation leads to a suboptimal QoS strategy. For example, applying the opposite strategy, i.e., giving more bandwidth to more demanding applications, leads to an increase of 23% in total contention time compared to interleaving.

5.3.2 The effect of non-fully segregation of applications (grouping) on VLs

When there are fewer VLs than applications our strategy is to group some applications on one VL as we explained in Sec. 5.1.

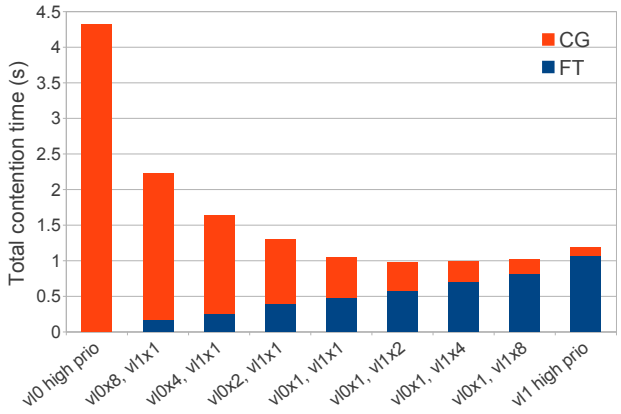

Figure 5.12: Total contention time when assigning different weights to VLs for the FT+ CG mix. FT on VL0 and CG on VL1.

Uniform bandwidth allocation

Figure 5.16 shows the effect of various grouping decisions for the three-application mix. The first bar on the left corresponds to the case of fully segregating applications and serves only for comparison. Here, vlo x1 (FT+CG), vl1 x1 (BT) indicates the policy which groups FT and CG and maps them on VL0, while BT is mapped to VL1, allowing each VL to send one packet per turn. Also, in this graph we show a breakdown of contention time per application to be aware of the fairness of the grouping decision. The optimal grouping decision is achieved when CG is grouped with BT into the same VL. This case improves the total contention time by 23% with respect to combining FT and CG. This result shows that the penalty of choosing non-optimal grouping of applications is quite significant.

Using the previously defined application classes we can explain the results as follows. Applications with low communication intensity, CG and BT, are compatible, while the application of moderate intensity FT is better to be separated in exclusive virtual lane.

Non-uniform bandwidth allocation

Finally, Figure 5.17 shows the total contention time for the FT+CG+BT mix when FT is on VL0, and CG and BT are on VL1 for different bandwidth allocations. We can see that the bandwidth alloca-

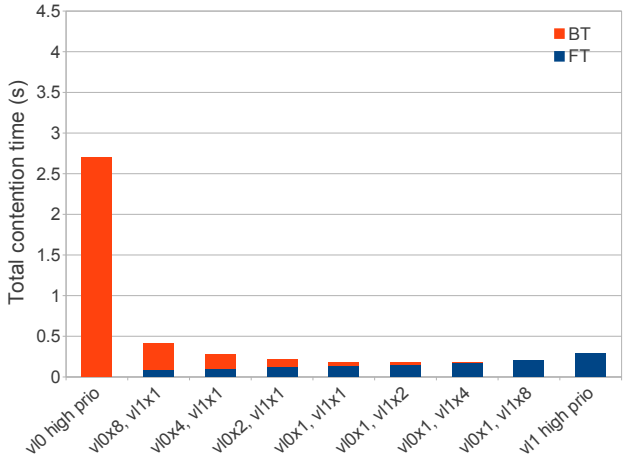

Figure 5.13: Total contention time when assigning different weights to VLs for the FT+BT mix. FT on VL0 and BT on VL1.

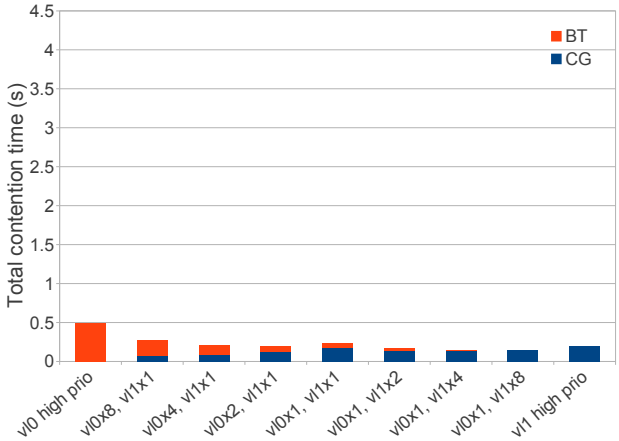

Figure 5.14: Total contention time when assigning different weights to VLs for the CG+BT mix. CG on VL0 and BT on VL1.

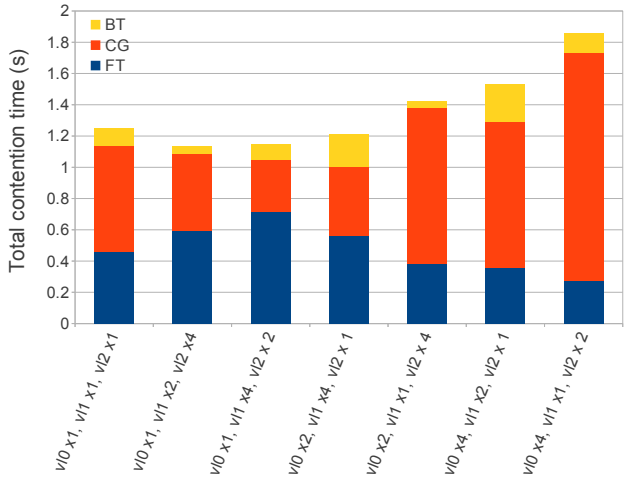

Figure 5.15: Total contention time when assigning different weights to VLs for the FT+CG+BT mix. FT on VL0, CG on VL1 and BT on VL2.

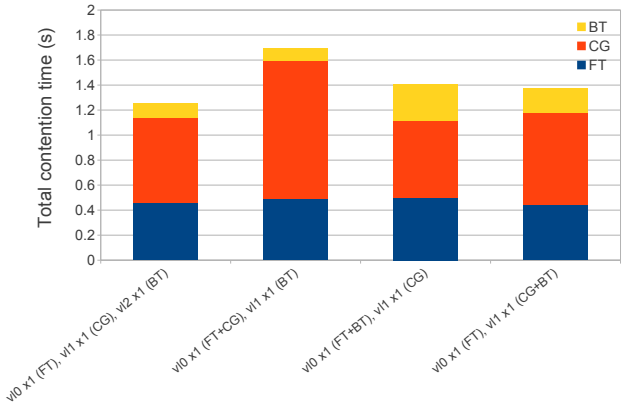

Figure 5.16: Total contention time for various application grouping decisions and also for fully segregating applications for the three-application mixes.

tion that achieves lower contention is when higher bandwidth is assigned to VL1, the VL with two low communication demanding applications. In particular, an improvement of 8% is achieved with respect to interleaving.

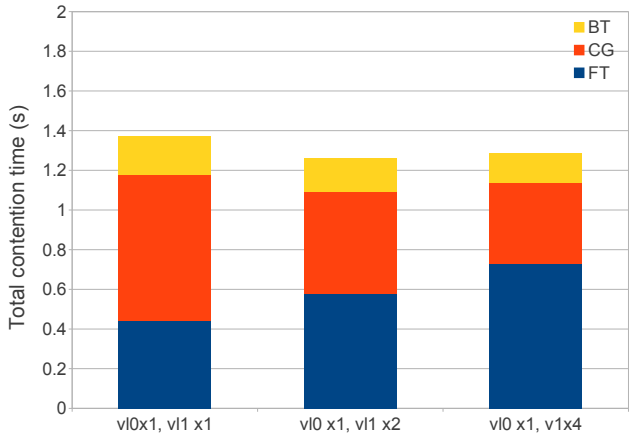

Figure 5.17: Total contention time when assigning different weights to VLs for the FT+CG+BT mix where FT is on VL0, and CG and BT are on VL1.

5.4 Conclusions

The use of QoS policies to mitigate the negative effects of inter-application network contention in capacity HPC systems has been explored. We have shown that the proposed QoS policy can significantly reduce this contention, thus achieving higher throughput in large systems.

The proposed QoS policy is based on an effective mapping of applications to VLs and allocating bandwidth to each VL. The proposed techniques for mapping and bandwidth allocation are based on the earlier proposed characterization of applications. This characterization can be easily performed at the nodes following the methodology in Section 5.1 and the InfiniBand subnet manager can change dynamically at runtime the VLs of the application without stopping its execution.

We have shown that segregating applications into VLs significantly reduces inter-application contention. Improvements of the inter-application contention time of up to 59% are achieved. In addition, significant improvements are observed when allocating bandwidth based on the proposed application characterization. We have shown that a strategy to assign less weights to VL with higher-intensity applications is more effective since in that way their traffic is being regulated. Additional improvements of up to 9% are observed for this technique.

Moreover, we have proposed a method to group applications into VLs in case there are more applications than VLs. In this case, we have shown that applications can be effectively grouped by means of the same characterization methodology. Applications with low intense communication behavior should be grouped into the same VL. Improvements of up to 23% are achieved by this strategy.

6

Related work

Kramer and Ryan[35] studied the performance variability of parallel applications (NAS Parallel Benchmark kernels) in four systems, a Cray 3TE, an IBM SP, a Compaq SC, and an Intel cluster. They run hundreds of times the same benchmarks, using fully packed nodes (but not all nodes) and found larger variation than expected. They found that a part of the varition could be attributed to the node alone (as one of the benchmarks, EP, did not use the network). Network performance variability was one of responsible factors for variation. They concluded that there is not a single parameter alone that can control or predict the variation, it is a combinations of many factors.

Evans et al.[19] concentrated more specifically on the application performance variability introduced

by sharing the network. The authors used the user-requested wall-clock times (to assume similar input sizes to the jobs running the same application) and actual execution times (to study the variability) from the run-time logs obtained from the resource management systems in several clusters. The authors identified task allocation as one of the sources of the variability and run a synthetic benchmark measuring the mean and standard deviation of the send time for different messages sizes and increasing the number of communicating pairs.

Petrini et al.[48] discovered in the ASCI Q machine that even a small amount of noise in the system could have a significant impact in the performance. The authors realized that increasing the performance of the communications alone does not lead to the expected improvement in time due to coupled effect of noise and synchronizations, implictly present in most parallel applications due to the data exchanges.

Huerta et al.[65] approached the analysis of application performance sensitivity from a more theoretical perspective, using symbolic models of the parallel application and the resources, evaluating this model for a particular set of parameters. The authors proposed the use of the term external contention to account for the contention suffered by applications by factors external to their own execution. They claim that a model of external contention is extremely hard if not impossible. However, this technique is of very limited applicability to large systems and the authors did not apply the technique to evaluate the effect of sharing the network resources with other applications.

Argawal et al.[2] contributed a theoretical analysis of the effect of noise in the scalability of a single computation phase. The authors analyzed the impact that a noise distribution and its intensity have in the weak-scalability of applications (increasing the number of nodes but maintaining the computational load).

Chandu and Singh[13] did a systematic analysis of the impact of network noise on the performance of a set of parallel applications by injecting controlled perturbations in the execution, in an attempt to model non-local interference in MPI applications. The authors measured the ability of application to absorb noise.

Hoefler et al. in[27] used a microbenchmark that introduces a controlled perturbation during the

execution of several MPI collective operations and study the impact on performance for several perturbation ratios and network sizes. By perturbation ratio the authors refer to the number of tasks involved in the perturbating MPI collective with respect to the number of tasks involved in the measured MPI collective. A later work[28] explores the impact of noise (Operating System noise) on the large scaling of collectives (up to millions of threads) and of applictions (up to tens of thousands of threads). This latter work shows that even a small amount of injection noise per task will introduce an inflection point in the scalibility of collectives upon which the noise becomes the bottleneck. A similar conclusion had been obtained before by Ferreira et al.[20] that also investigated the impact of noise in the scalability of large systems by MPI applications for sytems of up to 512 nodes.

Finally, Koop et al.[34] studied the impact on performance of mixes of NAS benchmarks sharing the same node, i.e., intra-node contention. The authors found that, when maximizing node utilization, mixing applications with complementary communication characteristics could lead to a better global performance by breaking the synchronized nature in which tasks of the same application will access the network adapter. This study restricts itself to the study of the contention of tasks of the same or different applications sharing the same multi-core node to access the network.

Our work, in contrast, focus on the impact of inter-application contention at the system level, using detailed simulation of the whole network topology with traces from actual HPC applications. We decouple the factors that can introduce variability and study their effect separately. We used seven HPC application traces with very different communication characteristics, that can clearly be classified in three groups depending on their perturbation power, and on the ability to absorb perturbations.

There have been a large number of proposals on job characterization and job scheduling to improve system utilization and user satisfaction of large shared computing centers. Subhlock[58] in 1996 characterized, for three different systems, the typical workloads that job schedulers cope with in such systems, consisting in a mix of a relatively large fraction of small (in number of nodes) jobs that use a small portion of the system competing for a smaller fraction of large jobs that occupy a larger fraction of the system for a considerable amount of time. Recent works show that such characterization, to a significant extent, still holds today[67,18].

These large facilities have to deal with a huge diversity in the characteristics and demands of scheduled jobs: different node-counts, memory requirements, execution times, prioritizations, unpredictable arrival distribution, different sensitivity to noise, etc.[58,31]. Due to all these factors, improving job scheduling and measuring the impact of an improvement is a very challenging problem involving a large number of factors (thoroughly analyzed in[31]). It is extremely difficult to obtain a clear picture of the impact that interference has on a job or collection of jobs. Measuring the impact of interference directly (by measuring the execution time in the presence and absence of interference) is impractical, and researchers have resorted to indirect methods to measure interference (measuring variability when running with different jobs or under different allocations) to attempt to improve job scheduling in such systems[31,18]. In[31] low-overhead monitoring is used to then proceed to structured pairwise comparisons and application classification. In[18], targeted at heterogeneous systems, a machine learning algorithm (singular value decomposition, SVD) is used, to, similarly to recommendation systems (Netflix Challenge[7]), extract features from the workloads and recommend the nodes or set of node that should best match the application.

Subhlok[58] claimed in 1996 that "in most modern parallel machines, the physical location of the nodes on which a job executes does not significantly affect the execution, and hence the logged execution time can be taken as the execution time for simulations.". Such a claim is not considered true any more. Even for fat tree networks, where the assumption has been that, due to its high bisection bandwidth, the allocation (fragmented or not) should have a minimal impact, it has been demonstrated[55,44,60] that the allocation strategy plays an important role in application performance: in reducing communication latency between communicating tasks[60] and in reducing interference from other applications[30]. For other topologies, such as meshes or tori, the problem of interference is even worse[63,8,46,56]. Delimitrou[18] show workloads slow-downs to up to 34% when interference-oblivious schedulers are used.

Some works have suggested[18] that packing workloads "in a small number of servers when the overall load is low, the rest of the servers can be turned off to save energy". However none of these works have provided a simple method, requiring very little information and interaction with the job sched-

uler, that can reduce interference and save energy for fat tree networks taking into consideration the consistent distribution of job sizes in most of the workloads analyzed and the negative impact that interference between applications can cause. Our work attempts to fill this gap.

There have been a lot of work on dynamic application-aware QoS provisioning in on-chip networks based on performance counters[17,14,16]. Although some concepts from this works might be applied to the area of computing clusters, application interference in live large-scale distributed systems poses new challenges to even measure application interference, let alone to mitigate it[31].

The use of QoS in InfiniBand was mainly studied in the area of Internet applications. The classification of applications is made based on their latency and bandwidth requirements[47,3].

Pelissier[47] explains the basic QoS method for four classes of application: latency-sensitive, bandwidth-sensitive, best-effort and challenged; it basically consists of assigning high priority lane to latency-sensitive applications and low priority lanes to the rest that maybe further differentiated by assigning weights to these lanes. The strategy would be to give more weights to applications that require more bandwidth. The study did not provide any evaluation of this QoS strategy.

Following the proposal in[47], Alfaro et al.[3] point out the need for wider classification of bandwidth demanding applications into low, very low, high and very high bandwidth demanding subclasses. In the same study they propose and evaluate the strategy for filling the arbitration table, assigning weights to virtual lanes based on mean bandwidth required by the application assigned to that virtual lane. Although a similar classification of applications based on the bandwidth may be used for HPC applications, giving more time slots to demanding application may slow down messages from lower demanding applications. This delay may greatly impact the performance of the parallel applications. Therefore, a throughput fairness policy, such as the max-min algorithm[15] or similar, is needed to achieve equilibrium alocation. Our work is in that direction and it shows (Figs. 5.12, 5.13, 5.14) that common Internet approach i.e. favoring high-demand applications is not feasible for HPC systems.

In[25] a routing algorithm was proposed for distributing flows belonging to different source-destination switches pairs over different lanes to avoid the Head-of-Line blocking due to end point congestion. This approach does not guarantee fair sharing of the network resources when several applications are

running simultaneously. As we showed 5.1.1, there are other sources of contention between applications besides HoL blocking. The proposal was evaluated using randomly generated traffic or one application together with this synthetic traffic. Our experiments are based on simultaneous execution of several real applications with real communication patterns and computation/communication ratios.

In [59] the improvement was gained for one application in the system by using a basic technique of evenly distributing bandwidth among the VLs in round-robin fashion. We went further showing (Figs. 5.16, 5.17) that with proper bandwidth allocation per virtual lane and mapping further improvements can be seen. They evaluate impact of contention when multiple microbenchmarks are competing for resources for different message sizes, but using the same message size for all microbenchmarks within one experiment. These microbenchmarks do not include all the characteristics of real applications. Message sizes may vary, as well as communication patterns during the application execution and thus the inter-application contention.

7

Conclusions

Capacity computing systems are usually running a large variety of jobs that are arriving and leaving the system in an unpredictable fashion having quite different compute and unpredictable network resource requirements. In such a scenario, jobs end up sharing network resources between themselves suffering the effect of network contention and congestion degrading their performance severely. The problem addressed in this thesis is focused on protecting job performance from interference with other jobs while maintaining high system utilization and low queue time. Finding a solution to this problem is not trivial due to the dynamic behavior and lack of predictability described earlier.

In this thesis, we are providing promising solutions to address this problem that could be easily

deployed in current systems as they could improve the current system resource management techniques. In order to solve this complex problem, there is no a single technique to satisfy both individual job performance and system performance. Therefore, our solutions are based on simple but efficient strategies that combine the allocation of an application on isolated resources and selectively grouping several applications on the same shared resources such that the effect of job's interference is minimized or even fully removed. We propose the solutions based on a fully understanding gained through this thesis on the main causes of the performance loss due to applications interference in the network – applications communication behavior and task placement. We have found that the job's communication requirements are fully defined by how logically distant communication peers are located and also on how much traffic is exchanged between them. The task placement of the application further determines how physically distant are its communication peers are and consequentially, which part of the network and how much is utilized by the application.

In this thesis, we have proposed a methodology to characterize jobs based on, first, how much it utilizes the network and second, which part of the network. The former will be used to identify the jobs with high utilization, since, as we show, they can have a significant impact on other jobs performance. The latter defines with how many other jobs it will share the network. Thus, it reveals the criticality of that job for overall effect of interference on the system performance. Having these pieces of information at any level of resource allocation decision would be of extreme importance for effective protection of job performance. This job characterization is fundamental to implement grouping strategies effectively to selectively choose jobs to share network resources from which system utilization would benefit. In order to isolate the critical applications or to selectively group the applications that do not impact each other, the information on communication behavior is the key.

However, at the point of node allocation for a new-arriving job to the system, its communication behavior is typically unknown. Therefore, our approach is to apply isolation strategy to as many applications as possible using the pieces of information available at the point of node allocation such as the job size. First, we identified the causes of node fragmentation, which is the main obstacle for having isolated allocation timely available. We have addressed each of them through the proposed concept of

virtual network blocks enabling in that way more control over the node fragmentation and achieving isolation with a slight penalty in system utilization and queuing time. For the sake of flexibility, this proposal allows network sharing among a small percent of applications.

The impact of sharing can be further addressed by applying the QoS mechanism using virtual channels available in InfiniBand networks employed in systems nowadays. The proposed policy is to apply a strategy of isolation to the virtual channels available in each network link by segregating jobs to different virtual channels. Additionally, once the job is allocated on the selected set of nodes, its communication behavior can be profiled according to the proposed characterization methodology. Based on the characterization the QoS mechanism of virtual channels can be enabled to effectively tune the bandwidth of each channel and further protect job performance by regulating the traffic of the too demanding applications. Finally, in the case there are more jobs sharing the link than the virtual channels, the information from the characterization process helps identify the applications that should be grouped together on a virtual channel in order to protect their performance.

Overall, task placement is the resource management technique that is capable of reducing most of the interference among applications, while quality-of-service techniques at the link-level can further help to protect the performance. The combination of both resource management techniques is promising to guarantee job performance predictability effectively.

8

Future work

As future work we would like to take several directions based on the knowledge gained in this thesis. First, we would like to extend our characterization methodology for parallel applications, explore its feasibility and implement it in run-time. In particular, we would like to achieve the following steps:

- Develop an efficient algorithm for obtaining a global characterization of the application based on local information gathered at the computing nodes.

- Explore algorithms to characterize application sensitivity in order to extend the characterization methodology to a 2D characterization.

Having such an insightful run-time characterization and classification will open new directions in improving our QoS policy. We would like to explore and implement the QoS mechanisms at the network switch. Namely, the QoS policy will decide on proper VL weights settings based on either hardware counters at each VL in the switch (local information) or based on the classes of applications assigned to VLs in the switch (centralized entity information). The second approach may increase the overhead in the switch. However, the two approaches can be used in a hybrid way. For example, it would be more beneficial to use application-aware approach in the case of low-utilization applications and the hardware counter approach in the case of applications with high utilization.

Regarding the job scheduling policy we would like to take our technique to the actual job scheduler used in MareNostrum, LSF. Also, we would like to continue exploring the interference-aware and energy-aware job scheduling policies and estimate potential savings, in cost and energy. Beside fat-trees, we would like to extend the research to other network topology families, such as, tori, dragonflies, etc. Further, we learned in this thesis that the job scheduling decision is missing an important piece of information on application network requirements. An effort in developing algorithms to transfer the insights on application communication internals before actually running it in the system would be an interesting path of research since it would be possible to mix more efficiently the jobs that are going to share the network. Or, based on the application classes and providing lower migration costs in the future, it would make sense to explore the benefits of application node re-allocation techniques after the application has been already scheduled, run for some time and profiled.

9

Publications

- A. Jokanovic, J. C. Sancho, G. Rodriguez, C. Minkenberg, and A. Lucero. Quiet Neighborhoods: Key to Protect Job Performance. 29th IEEE International Parallel and Distributed Processing Symposium, 2015. [under review]

- B. Prisacari, G. Rodriguez, A. Jokanovic and C. Minkenberg. Randomizing task placement and route choice does not randomize traffic (enough). Journal Design Automation for Embedded Systems, 2014.

- A. Jokanovic, B. Prisacari, G. Rodriguez and C. Minkenberg. Randomizing task placement does not randomize traffic (enough). ACM 7 th Workshop on Interconnection Network Ar-

chitecture: On-Chip, Multi-Chip (INA-OCMC), 2013.

- A. Jokanovic, J. C. Sancho, G. Rodriguez, C. Minkenberg, R. Beivide and J. Labarta. On the Trade-off of Mixing Scientific Applications on Capacity High-Performance Computing Systems. Journal IET Computers and Digital Techniques, 2013.

- A. Jokanovic, J. C. Sancho, G. Rodriguez, C. Minkenberg, and J. Labarta. Effective Quality of Service Policy for Capacity High-Performance Computing Systems. 14th International IEEE Conference on High-Performance Computing and Communications (HPCC), 2012. [Best Paper Award]

- A. Jokanovic, J. C. Sancho, G. Rodriguez, C. Minkenberg, R. Beivide and J. Labarta. Contention-aware node allocation policy for high-performance capacity systems. ACM 6 th Workshop onInterconnection Network Architecture: On-Chip, Multi-Chip (INA-OCMC), 2012.

- Jose Carlos Sancho, Ana Jokanovic and Jesús Labarta. Scalable Fault-tolerant Interconnection Networks for Large-scale Computing Systems. In 4th Workshop on Design for Reliability (DFR), 2012.

- Ana Jokanovic. Network Contention Management for HPC Systems. In 7th ACM International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC), Poster Session, 2012.

- Jose Carlos Sancho, Ana Jokanovic and Jesús Labarta. Reducing the Impact of Soft Errors on Fabric-based Collectives. In 4th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids, 2011.

- A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta. Impact of inter-application contention in current and future HPC systems. 18 th IEEE Symposium on High-Performance Interconnects (HOTI), 2010.

References

[1] (2007). Infiniband architecture specification, volume 1, release 1.2.1.

[2] Agarwal, S., Garg, R., & Vishnoi, N. (2005). The impact of noise on the scaling of collectives: A theoretical approach. In D. Bader, M. Parashar, V. Sridhar, & V. Prasanna (Eds.), High Performance Computing – HiPC 2005, volume 3769 of Lecture Notes in Computer Science (pp. 280–289). Springer Berlin Heidelberg.

[3] Alfaro, F. J. & Sánchez, J. (2002). A strategy to compute the infiniband arbitration tables. In in International Parallel and Distributed Processing Symposium (IPDPS). April 2002.

[4] Arimilli, B., Arimilli, R., Chung, V., Clark, S., Denzel, W., Drerup, B., Hoefler, T., Joyner, J., Lewis, J., Li, J., Ni, N., & Rajamony, R. (2010). The percs high-performance interconnect. In Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects, HOTI '10 (pp. 75–82). Washington, DC, USA: IEEE Computer Society.

[5] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., & Weeratunga, S. K. (1991). The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications.

[6] Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., & Sancho, J. C. (2008). Entering the petaflop era: the architecture and performance of roadrunner. In SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (pp. 1–11). Piscataway, NJ, USA: IEEE Press.

[7] Bell, R. M. & Koren, Y. (2007). Lessons from the netflix prize challenge. SIGKDD Explor. Newsl., 9(2), 75–79.

[8] Bhatele, A., Mohror, K., Langer, S. H., & Isaacs, K. E. (2013). There goes the neighborhood: Performance degradation due to nearby jobs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13 (pp. 41:1–41:12). New York, NY, USA: ACM.

[9] Blake, S., Back, D., Carlson, M., Davies, E., & Wang, Z. (1998). An Architecture for Differentiated Services, RFC 2475.

[10] BSC (2014a). Dimemas: internals and details (slides).

[11] BSC (2014b). Extrae: User guide manual for version 2.5.0.

[12] BSC (2014c). Paraver internals and details (slides).

[13] Chandu, V. & Singh, K. (2007). Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, chapter Sensitivity analysis of parallel applications to local and non-local interference, (pp. 469 – 474).

[14] Chang, K. K.-W., Ausavarungnirun, R., Fallin, C., & Mutlu, O. (2012). Hat: Heterogeneous adaptive throttling for on-chip networks. In SBAC-PAD (pp. 9–18).

[15] Dally, W. & Towles, B. (2003). Principles and Practices of Interconnection Networks. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[16] Das, R., Ausavarungnirun, R., Mutlu, O., Kumar, A., & Azimi, M. (2013). Application-to-core mapping policies to reduce memory system interference in multi-core systems. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), HPCA '13 (pp. 107–118).

[17] Das, R., Mutlu, O., Moscibroda, T., & Das, C. R. (2009). Application-aware prioritization mechanisms for on-chip networks. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42 (pp. 280–291). New York, NY, USA: ACM.

[18] Delimitrou, C. & Kozyrakis, C. (2013). Paragon: Qos-aware scheduling for heterogeneous datacenters. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13 (pp. 77–88). New York, NY, USA: ACM.

[19] Evans, J. J., Hood, C. S., & Gropp, W. D. (2003). Exploring the relationship between parallel application run-time variability and network performance in clusters. In Proceedings of the 28th Annual IEEE Conference on Local Computer Networks (LCN'03).

[20] Ferreira, K. B., Bridges, P., & Brightwell, R. (2008). Characterizing application sensitivity to os interference using kernel-level noise injection. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08 (pp. 19:1–19:12). Piscataway, NJ, USA: IEEE Press.

[21] Flich, J., Malumbres, M., Lopez, P., & Duato, J. (2000). Improving routing performance in myrinet networks. In Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International (pp. 27–32).

[22] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., &

Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting (pp. 97–104). Budapest, Hungary.

[23] Goedecker, S. & Hoisie, A. (2001). Performance optimization of numerically intensive codes. Software, environments, tools. Philadelphia, Pa. Society for Industrial and Applied Mathematics.

[24] Greenberg, R. I. & Leiserson, C. E. (1985). Randomized routing on fat-tress. In Proceedings of the 26th Annual Symposium on Foundations of Computer Science, SFCS '85 (pp. 241–249).

[25] Guay, W. L., Bogdanski, B., Reinemo, S.-A., Lysne, O., & Skeie, T. (2011). vftree - a fat-tree routing algorithm using virtual lanes to alleviate congestion. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11 (pp. 197–208).

[26] He, J., Kowalkowski, J., Paterno, M., Holmgren, D., Simone, J., & Sun, X.-H. (2011). Layout-aware scientific computing: A case study using milc. In Proceedings of the Second Workshop on Scalable Algorithms for Large-scale Systems, ScalA '11 (pp. 21–24). New York, NY, USA: ACM.

[27] Hoefler, T., Schneider, T., & Lumsdaine, A. (2009). The impact of network noise at large-scale communication performance. In Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on (pp. 1–8).

[28] Hoefler, T., Schneider, T., & Lumsdaine, A. (2010). Characterizing the influence of system noise on large-scale applications by simulation. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10 (pp. 1–11). Washington, DC, USA: IEEE Computer Society.

[29] Jokanovic, A., Rodriguez, G., Sancho, J. C., & Labarta, J. (2010). Impact of inter-application contention in current and future hpc systems. In Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects, HOTI '10 (pp. 15–24).

[30] Jokanovic, A., Sancho, J. C., Rodriguez, G., Minkenberg, C., Beivide, R., & Labarta, J. (2013). On the trade-off of mixing scientific applications on capacity high-performance computing systems. IET Computers & Digital Techniques, 7(2).

[31] Kambadur, M., Moseley, T., Hank, R., & Kim, M. A. (2012). Measuring interference between live datacenter applications. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12 (pp. 51:1–51:12).

[32] Kamil, S., Oliker, L., Pinar, A., & Shalf, J. (2010). Communication requirements and interconnect optimization for high-end scientific applications. IEEE Trans. Parallel Distrib. Syst., 21(2), 188–202.

[33] Kim, J., Dally, W., Scott, S., & Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. In Computer Architecture, 2008. ISCA '08. 35th International Symposium on (pp. 77–88).

[34] Koop, M., Luo, M., & Panda, D. (2009). Reducing network contention with mixed workloads on modern multicore, clusters. In Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on (pp. 1–10).

[35] Kramer, W. T. C. & Ryan, C. (2003). Performance variability of highly parallel architectures. In Proceedings of the 2003 International Conference on Computational Science: PartIII, ICCS'03 (pp. 560–569). Berlin, Heidelberg: Springer-Verlag.

[36] Labarta, J., Girona, S., Pillet, V., Cortes, T., & Gregoris, L. (1996). Dip: A parallel program development environment. In Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II, Euro-Par '96 (pp. 665–674).

[37] Leiserson, C. E. et al. (1992). The network architecture of the Connection Machine CM-5. In Proc. of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures (pp. 272–285). San Diego, CA, USA.

[38] Lysne, O., Reinemo, S.-A., Skeie, T., Solheim, A., Sodring, T., Huse, L., & Johnsen, B. (2008). Interconnection networks: Architectural challenges for utility computing data centers. Computer, 41(9), 62–69.

[39] McKeown, N., Mekkittikul, A., Anantharam, V., & Walrand, J. (1999). Achieving 100 Communications, IEEE Transactions on, 47(8), 1260–1267.

[40] Mellanox (2011). Unified fabric manager software for data center management.

[41] Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., & Wang, W. (2004). The weather research and forecast model: software architecture and performance. In Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology, volume 25 (pp.29).: World Scientific.

[42] Minkenberg, C., Denzel, W., Rodriguez, G., & Birke, R. (2012). End-to-end modeling and simulation of high-performance computing systems. In S. Bangsow (Ed.), Use cases of discrete event simulation: Springer.

[43] Minkenberg, C. & Rodriguez, G. (2009). Trace-driven co-simulation of high-performance computing systems using omnet++. In Proceedings of the 2nd International Conference on Simulation Tools and Techniques, Simutools '09 (pp. 65:1–65:8).

[44] Navaridas, J., Pascual, J. A., & Miguel-Alonso, J. (2009). Effects of job and task placement on parallel scientific applications performance. In D. E. Baz, F. Spies, & T. Gross (Eds.), PDP (pp. 55–61).: IEEE Computer Society.

[45] Öhring, S. R., Ibel, M., Das, S. K., & Kumar, M. J. (1995). On generalized fat trees. In Proceedings of the 9th International Symposium on Parallel Processing, IPPS '95 (pp.37).

[46] Pascual, J., Navaridas, J., & Miguel-Alonso, J. (2009). Effects of topology-aware allocation policies on scheduling performance. In E. Frachtenberg & U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, volume 5798 of Lecture Notes in Computer Science (pp. 138–156). Springer Berlin Heidelberg.

[47] Pelissier, J. (2000). Providing quality of service over infiniband architecture fabrics. In In Proceedings of the 8th Symposium on Hot Interconnects (pp. 127–132).

[48] Petrini, F., Kerbyson, D. J., & Pakin, S. (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC '03 (pp. 55–). New York, NY, USA: ACM.

[49] Petrini, F. & Vanneschi, M. (1997). k -ary n -trees: High performance networks for massively parallel architectures. In Proceedings of the 11th International Symposium on Parallel Processing, IPPS '97 (pp. 87–). Washington, DC, USA: IEEE Computer Society.

[50] Pillet, V., Labarta, J., Cortes, T., & Girona, S. (1995). PARAVER: A Tool To Visualise And Analyze Parallel Code. In Proceedings of WoTUG-18: Transputer and occam Developments, volume 44 (pp. 17–31). Amsterdam: IOS Press.

[51] Prisacari, B., Rodriguez, G., Jokanovic, A., & Minkenberg, C. (2014). Randomizing task placement and route selection do not randomize traffic (enough). Design Automation for Embedded Systems, (pp. 1–12).

[52] Pronk, S., Páll, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M. R., Smith, J. C., Kasson, P. M., van der Spoel, D., et al. (2013). Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics, 29(7), 845–854.

[53] Rodriguez, G. (2011). Understanding and Reducing Contention in Generalized Fat Tree Networks for High Performance Computing. PhD thesis, Computer Architecture Department, Technical University of Catalonia (UPC), Barcelona, Spain. Advisors: Jesus Labarta and Ramon Beivide.

[54] Rodriguez, G., Beivide, R., Minkenberg, C., Labarta, J., & Valero, M. (2009). Exploring pattern-aware routing in generalized fat tree networks. In ICS '09: Proceedings of the 23rd international conference on Supercomputing (pp. 276–285). New York, NY, USA: ACM.

[55] Sem-Jacobsen, F. O., Solheim, Å. G., Lysne, O., Skeie, T., & Sødring, T. (2011). Efficient and contention-free virtualisation of fat-trees. In IPDPS Workshops (pp. 754–760).: IEEE.

[56] Srinivasan, T., Seshadri, J., Chandrasekhar, A., & Jonathan, J. B. S. (2004). A minimal fragmentation algorithm for task allocation in mesh-connected multicomputers. In Proceedings of IEEE International Conference on Advances in Intelligent Systems – Theory and Applications – AISTA 2004 in conjunction with IEEE Computer Society, IEEE Press , ISBN (pp. 2–9599).: Press.

[57] Stone, A., Dennis, J., & Strout, M. M. (2011). The CGPOP Miniapp, Version 1.0. Technical Report Technical Report CS-11-103, Colorado State University.

[58] Subhlok, J., Gross, T., & Suzuoka, T. (1996). Impact of job mix on optimizations for space sharing schedulers. In Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on (pp. 54–54).

[59] Subramoni, H., Lai, P., Sur, S., & Panda, D. K. D. (2010). Improving application performance and predictability using multiple virtual lanes in modern multi-core infiniband clusters. In Proceedings of the 2010 39th International Conference on Parallel Processing, ICPP '10 (pp. 462–471).

[60] Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., & Panda, D. (2012). Design of a scalable infiniband topology service to enable network-topology-aware placement of processes. In High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for (pp. 1–12).

[61] Varga, A. (2001). The omnet++ discrete event simulation system. In Proceedings of the European Simulation Multiconference (pp. 319–324). Prague, Czech Republic: SCS – European Publishing House.

[62] Varga, A., Sekercioglu, Y. A., & Egan, G. K. (2003). A practical efficiency criterion for the null message algorithm. In Proceedings of the European Simulation Symposium (ESS 2003).

[63] Weisser, D., Nystrom, N., Vizino, C., Brown, S. T., & Urbanic, J. (2006). Optimizing job placement on the Cray XT3. In Proceedings of the Cray User Group Meeting, CUG '06.

[64] Yang, C.-Q. & Miller, B. (1988). Critical path analysis for the execution of parallel and distributed programs. In Distributed Computing Systems, 1988., 8th International Conference on (pp. 366–373).

[65] Yero, E. J. H. & Henriques, M. A. A. (2006). Contention-sensitive static performance prediction for parallel distributed applications. Perform. Eval., 63(4), 265–277.

[66] Yoo, A., Jette, M., & Grondona, M. (2003). Slurm: Simple linux utility for resource management. In D. Feitelson, L. Rudolph, & U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, volume 2862 of Lecture Notes in Computer Science (pp. 44–60). Springer Berlin Heidelberg.

[67] You, H. & Zhang, H. (2013). Comprehensive workload analysis and modeling of a petascale supercomputer. In W. Cirne, N. Desai, E. Frachtenberg, & U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, volume 7698 of Lecture Notes in Computer Science (pp. 253–271). Springer Berlin Heidelberg.

[68] Zahavi, E. (2011). Fat-trees routing and node ordering providing contention free traffic for mpi global collectives. In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on (pp. 761–770).

[69] Zahavi, E., Johnson, G., Kerbyson, D. J., & Lang, M. (2007). Optimized infiniband fat-tree routing for shift all-to-all communication patterns.

[70] Zhou, S., Zheng, X., Wang, J., & Delisle, P. (1993). Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Softw. Pract. Exper., 23(12), 1305–1336.