MPI Layer Techniques to Improve Network Energy Efficiency

A DISSERTATION PRESENTED
BY
BRANIMIR DICKOV
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
DECEMBER 2015

Author: Branimir Dickov
Thesis director: Professor Eduard Ayguadé
Thesis co-directors: Paul Carpenter and Miquel Pericàs

MPI Layer Techniques to Improve Network Energy Efficiency

Abstract

Interconnection networks represent the backbone of large-scale parallel systems. In order to build ultra-scale supercomputers larger interconnection networks are being designed and deployed. As compute nodes become more energy-efficient, the interconnect is accounting for an increasing proportion of the total system energy consumption. The interconnect's energy consumption is, however, only starting to receive serious attention. Most of this power consumption is due to the interconnection links. The problem, in terms of power, of an interconnect link is that its power consumption is almost constant, whether or not it is actively exchanging data, since both ends stay active to mantain synchronization.

This thesis complements ongoing efforts related to power reduction and energy proportionality of the interconnection network. The thesis contemplates two directions for power savings in the interconnection network; one is the possibility to use lower bandwidth links during the communication phases and thus save energy, while the second one addresses shifting links to low-power mode during computation phases when they are unused. To address the first one we investigate the potential benefits from MPI data compression. When compression of MPI data is possible, the reduction in link bandwidth is enabled without incurring any performance penalty. Consecutively, lower bandwidth leads to lower link energy consumption. In the past, several compression techniques have been proposed as a way to improve the performance and scalability of parallel applications. Those works have shown significant speed-ups when applying compressors to the MPI transfers of certain algorithmic kernels. However, these techniques have not seen widespread adoptation in current supercomputers.

In this thesis we will show that although data compression naturally leads to improved performance, the benefit is small, for modern high-performance networks, and it varies greatly between applications. In contrast, combining data compression with switching to low-power mode preserves performance while delivering effective and consistent energy savings, in proportion with the reduction in data rate. In general, application developers view time spent in a communication as an overhead, and therefore strive to keep it at minimum. This leads to high peak bandwidth demand and latency sensitivity, but low average utilization, which provides significant opportunities for energy savings. It

Author: Branimir Dickov
Thesis director: Professor Eduard Ayguadé
Thesis co-directors: Paul Carpenter and Miquel Pericàs

is therefore possible to save energy using low-power modes, but link wake-up latencies must not lead to a loss in performance. Thus, we propose a mechanism that can accurately predict when links are idle, allowing them to be switched to more power efficient mode. Our runtime system called the Pattern Prediction System (PPS) can accurately predict not only when a link will become unused but also when it will become active again, allowing links to be switched off during the idle periods and switched back on again in time to avoid incurring a significant performance degradation. Many HPC application benefit from prediction, since they have repetitive computation and communication phases. By implementing the energy-saving mechanisms inside the MPI library, existing MPI programs do not need to be modified. We also develop more advanced version of the prediction system, Self-Tuned Pattern Prediction System (SPPS) which is capable of automatically tuning to the current application communication characteristic and shaping the switching on/off of the links in the most appropriate way.

The proposed compression and prediction techniques are evaluated using an event-driven simulator, which is able to replay the traces from real execution of MPI applications. Experimental results show significant energy savings in the IB links while the performance overhead due to wake-up latencies and additional computation time have negligible effects on the final application performance.

Contents

Listing of figures

To my parents and sister/ Mami, tati i sestri

Acknowledgments

I would like to express my gratitude to my supervisors, without whom this dissertation would not have been possible. First of all, I would like to thank Professor Eduard Ayguadé for his unconditional support, motivation and encouragement during this large journey. His guidance helped me throughout the research and writing of this thesis. Also, I would like to thank him for encouraging my research and for allowing me to grow as a research scientist.

I would like to thank Miquel Pericàs for his valuable and constructive suggestions during the planning and development of this research work, for the stimulating discussions, and most of all for his willingness to participate in the thesis development until the end.

A very special thanks goes out to Paul Carpenter without whom completion of this thesis would have been much more difficult. His assistance in writing papers and making presentations was crucial for the completion of the thesis.

Also, I would like to thanks Nacho Navarro for his suggestions and advice during the research process. His always positive attitude and support helped me a lot to persist in the completion of the thesis.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the Catalan Government with Pre-doctoral Scholarship FI (reference no. 2009FI-B00077) for the first three years and by BSC the following years.

At the same end, I would like to thanks all my friends in Barcelona, at BSC/UPC and other places, that I have met throughout all these years. They have helped me to stay sane and fresh during and most importantly after completing the thesis.

Lastly, I would like to thank my family for their love and encouragement throughout my life.

Chapter 1

Introduction

High-Performance Computing (HPC) is a crucial tool for modern science and engineering. There is a constant demand for more powerful supercomputers, leading to increasing levels of energy consumption. The industry is working toward a target of 1 ExaFlop in 20 MW, which implies a sevenfold improvement in energy efficiency, compared with today's most energy-efficient machine.* Such a large reduction in system energy consumption is only possible if there are significant improvements in the energy efficiency across all subsystems. Power-saving techniques for processors and memory are progressing quickly, but there is less progress in reducing the power consumption of the interconnect. With energy-efficient processing elements and larger networks, the interconnection network is expected to account for up to 30% of the system's total power[52]. This fraction can even reach 50%[15] for data center servers since the CPUs operate at lower levels of utilization.

Most of this power consumption is due to the interconnection links. For example, the links in an IBM eight-port InfiniBand 12× switch consume 64% of the switch power[9]. The problem, in terms of power, of an interconnect link is that its power consumption is almost constant, whether or not it

*The Shoubu machine at RIKEN leads the June 2015 Green500 list with 7.0 GF/W.

is actively exchanging data, since both ends stay active to maintain synchronization.

One possibility to save network power could be to use lower bandwidth links. In this thesis, to address this possibility, we investigate the potential benefits of MPI data compression. If compression of MPI data is possible, this will allow a reduction in link bandwidth which consequently leads to lower link energy consumption, while the original performance is preserved. Even though they are always on, high-speed channels still offer dynamic range, in the terms of their ability to vary data rate and power consumption. Considering the InfiniBand architectural specification, each link is constructed from several serialized lanes each operating at the same data rate. To reduce the energy consumption without affecting performance, the link bandwidths are dynamically adjusted by changing the number of active InfiniBand lanes. This is done in inverse proportion to the compression rate in order to maintain the original bandwidth of the uncompressed data.

Although compression can reduce considerably the necessary bandwidth and therefore, the energy consumption of the interconnect, still, a lot of power is wasted during long idle intervals. Our second goal is to provide a network that supports energy proportional communication, which means that the amount of energy consumed is proportional to the traffic intensity (volume) in the network. In general, application developers view the time spent in communication as overhead and therefore try to minimize it. This leads to high peak bandwidth demands and latency sensitivity, but low average utilisation, having the network largely idle. This provides significant opportunities for energy savings but unfortunately, as mentioned above, current interconnects are not energy proportional, so the potential energy savings are lost[59].

One approach to reduce network energy consumption during long idle periods is to turn off the network links (or put them in some low-power mode). The problem is that link state changes, from off to active, can take up to 10 μs[35]. Since state changes add to the latency of MPI messages, and many HPC applications are highly sensitive to latency, this leads to an unacceptable loss in performance. An alternative is to lower the voltage and bandwidth of links when utilization is low, which has faster

link reactivation, at about 100 ns, but the potential power saving is much lower[15]. Both mechanisms switch between power modes using low-level hardware schemes[59],[56],[17]. Common drawbacks are the inability to capture significant energy savings, as well as an unknown and uncontrollable performance penalty. In this thesis, we propose runtime power management support that identifies the structure of the MPI layer communication behavior of an application over the interconnect. Performing end-to-end, recognition of communication behavior on a higher MPI layer level, provides us with a more accurate high-level view of order and timing of the link usage. This software-managed technique will allow us to predict when the link will stay idle, instructing hardware support not only to shift links to low-power modes when idle but also to get switched back on again in time to avoid incurring a significant performance degradation.

Another option to save network power could be to force applications to use the network more efficiently. Overlapping communication and computation will lead to a steady use of the network allowing a reduction in required network bandwidth and most important will reduce time to solution. Unfortunately, overlapping communication and computation is generally hard for HPC workloads. Most HPC applications follow the bulk synchronous programming paradigm, in which application processes are synchronised, either all performing computation at the same time or all involved in communication. Instead of asking to change programs we want to propose techniques that can work with the most frequent software stacks and applications.

The contributions of the thesis are the following:

- We introduce the main concepts of compression technology for energy savings and evaluate a set of compressors. Compression is applied to the data sent and received by MPI[1] library calls. Our techniques are designed for double-precision floating-point (FP) data, which is the most common data type passed in the messages of scientific applications. We consider both lossless compression algorithms as well as lossy compression algorithms, where the compressor reduces

precision based on a target accuracy specification. When applicable, compression allows to design systems with slower components, therefore enabling a reduction in energy and installation costs.

- While compression is targeted to cover link power optimization during communication phases, additional link energy can be saved during large idle phases. The majority of the execution time in most HPC applications is spent in a large number of iterative execution phases. Since the communication pattern inside each phase is essentially the same, it is possible to observe the communication behaviour in one iteration and use the knowledge gained to predict the behaviour of the subsequent iterations. Specifically, this means detecting the patterns of MPI calls that are repeating within each MPI process. To achieve this, we propose to use a software-managed runtime prediction system that will provide us with insight on when to turn off/on the links. We developed Pattern Prediction System (PPS) which allows an on-the-fly detection of consecutive repeatable MPI communication patterns. This provides a high-level view of the order and timing of link usage, which permits the transition between different link power modes to be made with a minimum impact on performance.

- The basic Pattern Prediction System (PPS) is prone to instability if the chosen critical idle interval length does not match the critical idle interval length of the chosen application. Critical idle interval length is the minimal idle interval length during which link power can be saved. We refer to this value as the Grouping Threshold (GT), and it has an important influence on the level of prediction and therefore on energy savings, depending on the application in a complex way. A larger value of the Grouping Threshold usually leads to higher prediction accuracy, but lower values can provide better energy savings. We, therefore, propose a histogram-based self-tuned prediction mechanism that automatically determines the critical idle interval, which distinguishes short and long idle intervals. The resulting mechanism obtains energy savings in

the network edge links of up to 21% with negligible performance overhead.

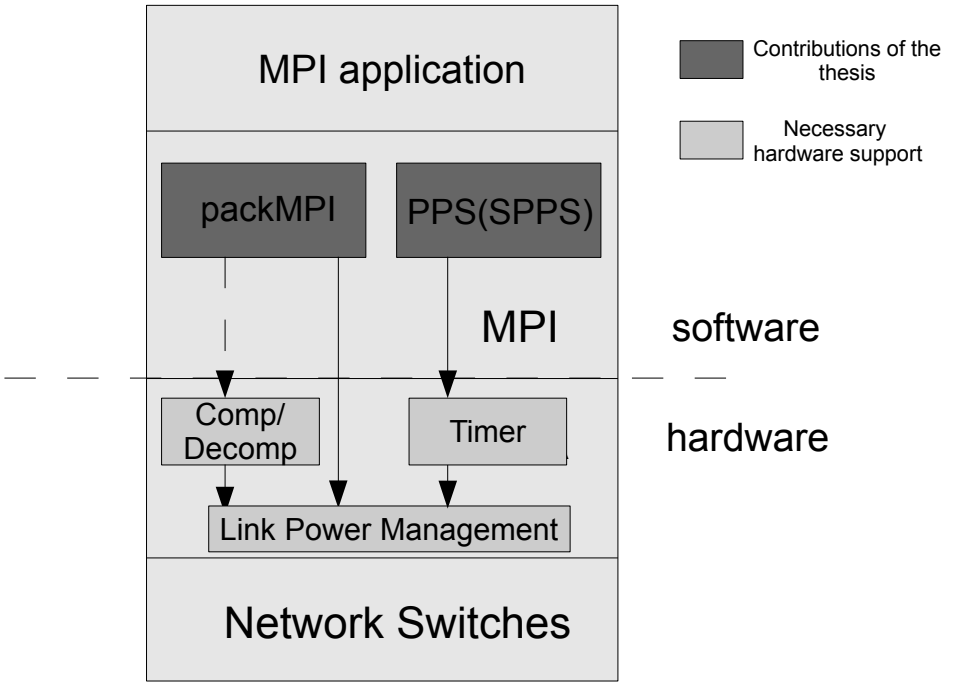


Figure 1.1: System stack architecture with our proposals in MPI layer.

Figure 1.1 summarizes our techniques developed in MPI layer. The dashed lines show the hardware approach for the *packMPI* library, in which the MPI layer is bypassed and directly communicates with the Host Channel Adapter (HCA) where the compressor (decompressor) rate is set. Therefore, when communication is started, the original data is transferred to the HCA buffers from where appropriate bit-lines are activated in correspondence with the chosen compression rate. Compression can also be done in software with the higher compression/decompression overheads introduced in the total MPI data latency. In correspondence with the compression rate, the number of lanes of InfiniBand link is also deactivated using Link Power Management (LPM) firmware on the HCA. Pattern Prediction System (PPS) exploits the MPI profiling interface PMPI to implement its functionality. PPS is wrapped around actual MPI library invocations. This makes the PPS technique portable that can be applied on

any MPI application without changing their source code. When a prediction is confirmed, switching the link to low-power mode is done using the LPM firmware.

The thesis is organized as follows. Chapter 2 provides the necessary background on network performance issues, possibilities of data compression to enhance network efficiency, network power consumption breakdown and Infiniband network technology. Chapter 3 describes the methodology and the toolchain used for the experiments. Chapter 4 introduces the main concepts of compression technology for link power savings and evaluates a set of compressors. Effects of compression on application performance are also discussed. The following Chapter 5 introduces the design of our prediction techniques for link power savings. The related work is summarized in Chapter 6. Finally, in Chapter 7 we give the main conclusions of this thesis.

Chapter 2

Background

2.1 System Area Network (SAN) - Interconnection Network for HPC

In the current High Performance Computing (HPC) landscape, clusters have become the ubiquitous architecture for accelerating many scientific and engineering applications. These systems consist of multiple computer nodes that communicate over a network. Figure 2.1 illustrates the communication path in a large-scale interconnected system from a sender (compute node 0) to a receiver node (compute node M), which consists of the following steps:

- Data are fetched from a processor local memory and sent to the memory on the Network Interface Card (NIC), also known as Host Channel Adapter (HCA) in the InfiniBand terminology, which is used for attaching the processing node to a network
- NIC transforms data messages into network packets
- Packets go through a number of switches before reaching the destination

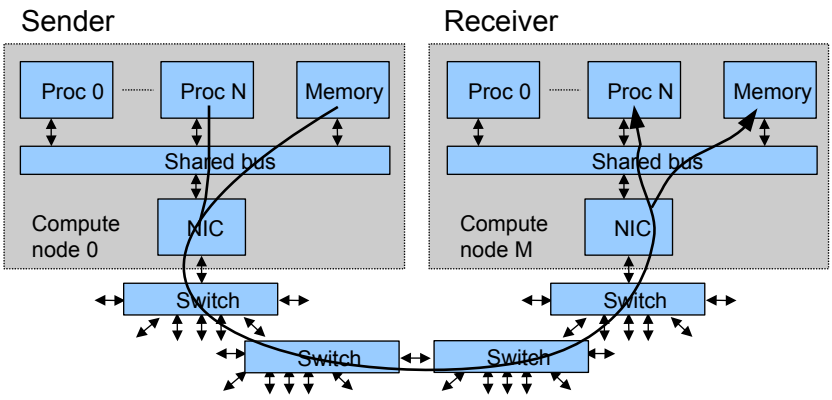


Figure 2.1: Communication path in a large-scale interconnected system.

- At the destination packets are reformatted by the receiving NIC and sent to the receiving processor's local memory

Communication is not only costly in time, energy costs also need to be taken into account. Therefore, data transfers are an important component of parallel systems that need to be extensively optimized.

2.2 Communication in Parallel Programs

Scientific applications solve complex problems by splitting the problem into several smaller parts, each assigned to a single application's process. The interconnect's role is to satisfy remote data dependencies between application's processes that are executing on different nodes. Depending on the algorithm applied to solve the problem, the processes may communicate messages following different patterns, such as, a single pair of processes at a time, i.e., point-to-point communication or multiple processes at time, i.e., usually referred as collective communications.

In general, application's communication time is the time that an application spends in communication routines. Depending on the problem size and scale, i.e., the number of processes involved in

the computation, the message sizes may vary as well, making application's communication bandwidth or latency-sensitive. Single message transfer would require the following time:

$$T_{message} = T_{latency} + \frac{Message\_Size}{B}$$

Latency is the time it takes to send zero-byte message from source to destination, whereas bandwidth is the actual speed of transmission, or bits per unit time. As communication time doesn't advance the actual computation, it is rather seen as an overhead and must be minimized to get the best performance improvement.

In order to successfully accomplish message delivery, two common message passing protocols are generally employed:

- Eager - An asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
- Rendezvous - A synchronous protocol which requires an acknowledgement from a matching receive in order for the send operation to complete.

While eager protocol reduces synchronization delays (send process does not need acknowledgement from receive process that it's OK to send message) the most important disadvantage is that it is not scalable. Significant buffering may be required to provide space for messages from an arbitrary number of senders. On the other side rendezvous protocol is scalable compared to eager but his disadvantage is that it introduces synchronization delays due to necessary handshaking between sender and receiver. Thus, eager protocol is typically used for "short" messages, while rendezvous protocol is used for "long" messages.

The performance of a parallel program can be presented by its execution time consisting of computation and communication components. Figure 2.2 shows the structure of the GROMACS application, where with light blue is shown computation phase while the communication phase is represented with all the rest of the colors.

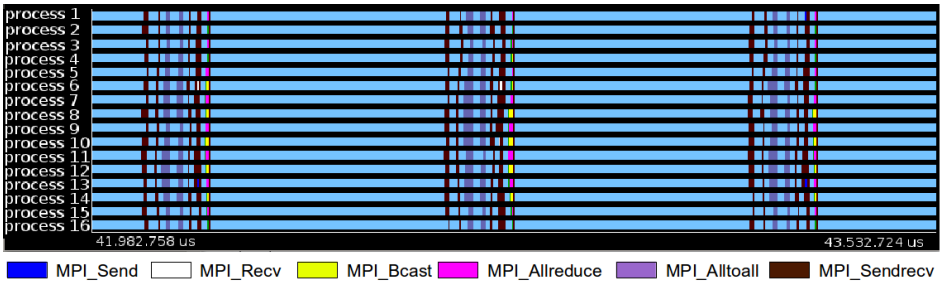


Figure 2.2: GROMACS execution trace with corresponding computation and communication phases

In this thesis, to reduce network power consumption we are mainly focused on the bandwidth component of the actual transfer time. A reduction in the bandwidth of the link will automatically reduce its power consumption, but these decisions when to switch to low-power mode are critical, especially in HPC, where the primary design objective is performance. Therefore, any proposed energy-saving technique will only be adopted if there is no significant performance deterioration.

2.3 Data Compression

Data compression, both lossless and lossy, is widely used, across many application domains, to reduce the demands on storage capacity and communication bandwidth. In HPC, data compression has been applied to messages on the network in order to minimise application execution time[30,38,23,39,41]. However, in our study, of the impact of compression on performance in Section 4.2, we concluded that the gains obtained are lower than expected by Amdahl's law for the used compression rates, where compression rate is defined as follows:

$$Compression\_rate = \frac{(uncompressed\ size)}{(compressed\ size)}$$

Thus, in this thesis we aim to apply data compression to the problem of energy efficiency.

There is an important distinction between lossless and lossy compression. Although lossless compression can be applied to scientific applications indiscriminately, always leading to correct results, our

results validate previous studies, which have shown that the resulting compression rate is low[25], [41], [29]. This is because scientific applications pass high entropy floating-point data, which is hard to compress. In contrast, applying lossy compression requires the involvement of the scientific application or library developer. Previous studies have, however, found that these experts have a good understanding of the numerical stability of their algorithms, and can determine the required data precision.

2.3.1 Lossless Compression

Lossless compression is the method of choice where preserving the numerical precision of the data is necessary for correctness, since introducing error in the system may cause a significant deviation in the final result. General purpose algorithms like GZIP[8] or BZIP2[5] can be used for floating-point data, but generally compressors designed specifically for floating point data will yield better results[22]. Usually they are based on a predictive scheme which tries to predict the next value based on the sequence of previous values. Depending on how they calculate the predicted value, lossless compression algorithms can be classified into two major groups. The first group uses arithmetic predictors based on a polynomial function to predict the next value[38] whereas the second group relies on context based predictors that store the previous values in a hash table[23]. These values are then used to predict the following ones. The difference between the predicted and true value is then computed. If the prediction is close to the true value the difference can then be encoded with fewer bits, resulting in a compressed form. This can be very effective in the case of scientific applications where communicated data represents adjacent physical quantities, such as in Finite-Difference Time-Domain (FDTD) or Lattice Boltzmann method[38], whose values tend to be highly correlated. Such data patterns may then be effectively captured by predictors.

In this thesis we used following lossless compressors:

- FPC (Floating-Point Compressor) from Burtscher et al.[23] is a lossless compression algorithm

for a linear stream of 64-bit floating-point data which relies on context based predictors. FPC reportedly achieves good compression rates and it has a compression speed one or two orders of magnitude faster than competing algorithms.

- MAF (MAkichan for Floating) is another lossless compressor developed by Tomari et al.[60]. In MAF, only the exponential part of the double-precision floating point data is compressed. This approach can be very effective if the entropy of the data is very high but the magnitudes (i.e. exponents) do not change much. It is also notable for its high speed which enables low-latency communication even in software based approaches.

- GZIP is based on the DEFLATE algorithm[27], which is a combination of LZ77[65] and Huffman coding[37].

To benefit from transferring compressed data the additional time for compression should be less than the time saved during the communication primitive. Considering that sending data involves a message setup overhead and that messages might be very small, some amount of data is established as a minimum threshold in order to apply compression. The critical size of message is defined as a product of latency and bandwidth, which means that all messages smaller than this product will have latency as a dominant factor in a transfer time and, therefore, the impact of compression would be negligible.

2.3.2 Lossy Compression

In the case of lossy compression, the most straightforward way to compress is to discard the least significant mantissa bits. However, care needs to be taken not to jeopardize the correct execution of the application. It is necessary to provide a criteria of minimum accuracy so that compression does not invalidate the final result. For the case of lossy compression it is necessary that the application experts determine the acceptable error and develop models to predict the required precision of the computation.

In this thesis, in order to apply lossy algorithms, the following criteria of accuracy has been established:

$$L^2_{relative\_error\_norm} = \frac{\sqrt{\sum_{i=1}^{N}(F_i^{52} - G_i^k)^2}}{\sqrt{\sum_{i=1}^{N}(F_i^{52})^2}}$$

where F_i is exact solution (with 52-bits of double-precision floating point mantissa) and G_i is newly calculated solution after lossy compression is applied (with k- bits of double-precision floating point mantissa).

Upon consulting with application developers we determined 1 % error from the final result to be acceptable, thus, if the newly obtained result with lossy compression is within a margin of 1% we take it as a valid result.

2.4 Interconnection Network Power Consumption

The network fabric power consumption is due to the switch fabric, HCAs, and interconnect links. The power consumption of the HCAs and switches varies with the data injection rate, being dominated by the active power of the memory elements. In contrast, the power consumption of an interconnect link is almost constant, whether or not is actively exchanging data[59], since both ends stay active to maintain synchronization. It has been shown that the majority of the total network fabric power consumption is due to the interconnect links; e.g. for an IBM InfiniBand 8-port 12× switch, the links are estimated to take 64% of the total switch power[9]. For this reason, in this thesis our focus will be on link power.

2.5 Power-saving Support in Interconnection Network Links

Most energy optimization research is focused on reducing link energy consumption in interconnection networks using different kinds of power-aware techniques. These techniques can be classified in two

classes: dynamic voltage-frequency scaling and on/off links.

The on/off technique allows entering to a deeper low-power mode, thus saving more power. The problem is that link state changes, from off to active, can take up to 10 µs[35]. Since state changes add to the latency of MPI messages, and many HPC applications are highly sensitive to latency, this leads to an unacceptable loss in performance. Also as links are turned off, a fault-tolerant routing algorithm has to be used, increasing hardware complexity and possibly introducing additional penalty in network performance. An alternative is to lower the voltage-frequency pair, thus reducing the bandwidth of links when utilization is low, which has faster link reactivation, at about 100ns, but the potential power saving is much lower[15]. Here, the advantage is that the connectivity throughout the network is preserved and the same routing algorithm can be used regardless of the power consumption level, simplifying router design.

2.6 InfiniBand technology

The adapter and switch architecture parameters used throughout the thesis are based on the current Infiniband adapter and switch architecture employed in many computing systems today. The typical link bandwidth supported in InfiniBand is 10 —40 Gbit/s and the switch latency is 100 ns.

InfiniBand links support two physical mechanisms that can be used for power saving. First one allows links to operate at different data rates, for example between Quad Data Rate (QDR), at 10 Gbit/s per serial lane, and Single Data Rate (SDR), at 2.5 Gbit/s per lane. This gives a tradeoff between bandwidth and energy consumption. The switching time between rates is small, on the order of hundreds of nanoseconds[15], but the power savings are also small.

Second power saving mechanism is derived from the actual InfiniBand link architecture, where the link is formed by aggregating one or more serial lanes. It is common to use more than one lane, in order to increase the link's bandwidth, though power consumption is also multiplied by the same

factor. Mellanox has recently announced Width Reduction Power Saving (WRPS), a technique that allows the link's number of active lanes to be dynamically reconfigured, assuming firmware support is enabled in the HCAs and switches[12]. For example, using WRPS a 40 Gbit/s 4× QDR port can run as 10 Gbit/s 1× QDR by shutting down three of its four QDR lanes. This reduction in link width reduces the power consumption of Mellanox Switch SX6036 to only 43% of its nominal power (when all four lanes are active)[12]. We use this published value of 43% in the evaluation section as the power consumption of an IB switch in low-power mode. The energy to transfer a single message remains the same, because energy is power multiplied by time. The energy savings come from relatively short idle periods between messages.

Chapter 3

Experimental Methodology

We used two classes of tests for the evaluation of application and system performance. When possible, we evaluated the performance on a production HPC machine. When our proposals depend on features that are not supported on current systems, then we used simulation tools. Since this study is exploring the HPC traffic in conjuction with the specific network technology, we used an MPI simulator that allows a replay of the application's MPI activity while respecting the communication dependencies between MPI processes which is coupled with an event-driven network simulator which simulates the network architecture in detail. In this chapter we introduce the TestBed platforms where real execution tests were done, as well as the platform used to generate input traces for the simulator. Also we describe the set of simulation tools deployed in this thesis, along with the workloads used in simulation.

3.1 TestBed platforms

The main features of the clusters and MPI implementations used for our evaluation are:

1. MareNostrum II is an compute cluster based on IBM JS21 blades, each with two dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz and with 8 GB of RAM. Communication itself occurs over a 2Gbps Myrinet network with a fat tree topology. MPICH-MX is used as message passing library, which implements the MPI-1 standard.

2. MinoTauro is a machine based on Bull B505 nodes, with six-core Intel Xeon E5649 processors at 2.53 GHz with 24 GB RAM. All nodes are connected through InfiniBand network running at 40Gbps and organized as 2-level fat tree network. OpenMPI-1.6.4 is the MPI implementation used for trace collection on MinoTauro.

3.2 HPC workloads

To evaluate the potential for link power reduction using our MPI layer techniques we have chosen a wide breadth of HPC production application and benchmarks. For the data compression evaluation just the main kernels of real applications were observed while for link power reduction using prediction system entire applications were considered.

- Real applications:

    1. ALYA[3] is a computational mechanics system that is capable of solving different physics problems. The problems that we selected in our investigation use the conjugate gradient(CG) method as the main kernel.

    2. GROMACS[7] is a molecular dynamics simulator. For the calculation of forces it uses the Particle Mesh Ewald (PME) method which is the main kernel.

    3. MILC[34] performs large scale numerical simulations to study quantum chromodynamics (QCD). QCD is the theory of the strong interactions of subatomic physics.

4. PEPC[32] is a tree code for solving the N-body problem.
5. CPMD[24] is an *ab initio* electronic structure and molecular dynamics (MD) program using a plane wave/pseudopotential implementation of density functional theory (DFT).
6. QUANTUM ESPRESSO[31] is an integrated suite of computer codes for electronic-structure calculations and materials modeling, based on density-functional theory, plane waves, and pseudopotentials (norm-conserving, ultrasoft, and projector-augmented wave).
7. WRF (Weather Research and Forecasting model)[47] is a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs.

- Kernels:

1. CG (Conjugate Gradient)[57] is an iterative method for solving systems of linear equations that arise from the finite element method (FEM).
2. PME (Particle Mesh Ewald)[53] is a method for computing long-range interactions in periodic systems. In PME the sum of long-range forces is processed in Fourier space where this sum converges much faster as compared to real space.

- Benchmarks:

1. NAS MG (Multi Grid)[13] approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method.
2. NAS BT (Block Tridiagonal)[13] is an algorithm used for solving a synthetic system of nonlinear partial differential equations.
3. NAS SP (Scalar Pentadiagonal)[13] is another algorithm used for solving a synthetic system of nonlinear partial differential equations.

3.3 Extrae - Tracing Tool

In order to collect the application communication characteristics and its performance it is necessary to insert instrumentation during its execution. Instrumentation captures information during the program execution creating an application trace. This process of receiving informative messages about the execution of an application at runtime is called tracing. To obtain traces of parallel applications at run time we used Extrae [20],the tracing tool developed at Barcelona Supercomputing Center (BSC).

To create traces of MPI calls the MPI profiling interface (PMPI) defined by the MPI standard is used. This interface allows a tool such as Extrae to interpose a library between the application and the MPI substrate and intercept one or more MPI calls. The MPI standard requires that each routine is available with both the MPI and PMPI prefix. The application calls with MPI prefix are intercepted and recorded, while PMPI calls are executed. Therefore, Extrae intercepts the MPI calls that are coded with MPI prefix. Usually the collective MPI calls are implemented using PMPI point-to-point communication calls, thus, they are not being recorded by Extrae. For our study, the internal structure of collectives is very important, thus, to instrument low-level operations of collectives we used adapted versions of MPICH-MX and OpenMPI libraries that allow the translation of low-level operations to MPI_-like names.

To intercept the MPI calls Extrae uses the LD_PRELOAD mechanism where at runtime the original symbols (MPI) are substituted by those provided by the intstrumentation package (PMPI). Aslo for the tracing of internals of collective calls the modified MPI library is loaded. In the Figure 3.1 is shown the an example script used for trace generation.

During the instrumentation, each 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 the actual computation performed are not recorded. Communication operations are recorded as send, receive, or collective operations records, including the sender, receiver, message size, and type of operation.

bsc18511@nvb127:/gpfs/scratch/bsc18/bsc18511/mnsubmit run.sh

```
#! /bin/bash
# @ job_name         = nucleo_256       #job's name
# @ initialdir       = .                #initial directory
# @ output           = mpi_%j.out       #output file
# @ error            = mpi_%j.err       #error file
# @ total_tasks      = 24               #number of processes
# @ tasks_per_node   = 6                #number of processes per node
# @ cpus_per_task    = 1                #number of processes per CPU
# @ wall_clock_limit = 00:30:00         #wall clock time 30 min

export MDRUN=/gpfs/scratch/bsc18/bsc18511/GROMACS

time srun trace.sh
```

```
export LD_LIBRARY_PATH=${MPI_INTERNALS_LIBRARY}/lib

export EXTRAE_HOME=/gpfs/apps/NVIDIA/CEPBATOOLS/extrae/openmpi/64
export EXTRAE_CONFIG_FILE=./mpitrace.xml

export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so


#Run the program
$MDRUN/bin/mdrun_d -s run_twover_200step.tpr -v -dlb yes -npme 0
```

Figure 3.1: Tracing scripts for GROMACS application.

3.4 Paraver - Visualization Tool

Paraver[14,31] is the visualization tool developed at BSC and it is used in multiprocessor systems to visualize multithreaded program traces (including MPI and OpenMP) that are obtained at runtime using the Extrae tool. It allows the user to view runtime information of function calls and hardware counters. In Paraver metrics are not hardwired on the tool but programmed. Using a filter and a semantic module, the analyst can create time-lines, profiles and histograms from trace-files to selectively display a huge number of performance metrics. The different views can be easily combined to find correlations among the causes of performance drawbacks. To capture the expert's knowledge, any set of views can be saved as a Paraver configuration file, to be reused in subsequent analyses. With Paraver it is easy to visualize communication patterns, including the number of bytes exchanged between each pair of tasks.

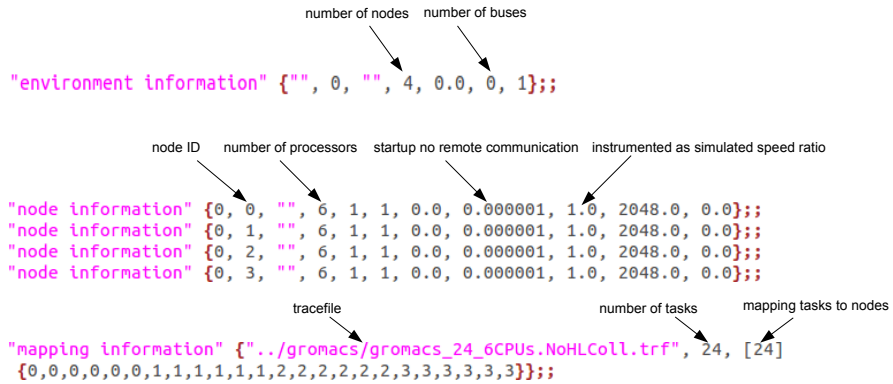


Figure 3.2: Dimemas parameters used to simulate Gromacs application with 24 MPI processes where six processes are run on each node

3.5 Dimemas - MPI simulator

Dimemas[19,42] is an event-driven simulator, which replays a trace of the application's computation bursts and MPI activity, preserving its causal relationships and timings. It is driven by traces generated by the *prv2dim* tool which converts the original traces suited for Paraver to traces expected by Dimemas. Each trace contains a sequence of operations for each thread of each task. It contains CPU intervals and MPI/communication event information (message size, identifiers, type, source-destinations) from the original execution. Dimemas models an architectural machine model with SMP nodes interconnected with a simple point-to-point network with duplex links. The model is highly parameterizable, allowing the specification of parametars such as number of nodes, number of processors per node, relative CPU speed, number of communication buses, mapping task to nodes, etc. Computation bursts are not actually performed, but represented by the time the actual computation would last. Communication operations are send and receive point-to-point communications.

In Figure 3.2 is shown an example set of parameters used in our study. Dimemas generates trace files that are suitable for Paraver enabling the user to conveniently examine any performance problems indicated by a simulator run.

3.6 Venus - Network Simulator

Venus[49,48] is a generic interconnection simulator capable of simulating many different kinds of networks. It is based on OMNET++[61] and provides a socket-based co-simulation interface to the MPI task simulator, which replays traces obtained using an instrumentation package. It is able to provide a detailed simulation of the network topology and the processing inside the switches. Detailed models of switch and adapter hardware corresponding to different networking technologies, including Ethernet, InfiniBand, Myrinet are supported.

Network topology, routing and mapping of application processes to the nodes are specified in separate configuration files. Although it can supports different network topologies our switch models are arranged in a fat tree topology - specifically an Extanded Generalized Fat Tree (XGFT). 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 example, to create a topology for a two-level fat-tree with switch radix 2 is neccesary to run the xfgt tool with following parameters:

```
xgft -m 2:2,4:1,2 > 2level_ft.map
```

Than *map2ned* tool converts a map file to an OMNEST Network Description file (2level_ft.ned) corresponding to the specified topology and a matching initialization file (2level_ft.ini) containing network address and host/switch labels.

```
map2ned -v4 2level_ft.map
```

The routing file is generated using xgft tool followed by -r option and a number that represents specific routing scheme (e.g. 3 for random routing).

Mapping is done through configuration file (.scb) that contains one hostname (as known to Venus) per line; task *n* is mapped to the host corresponding to the hostname specified on line *n*. Relation in between task mapping in Dimemas and Venus is shown in Figure 3.3.

The link bandwith is defined by two parameters where first one is the *unit_size* which is equal to the

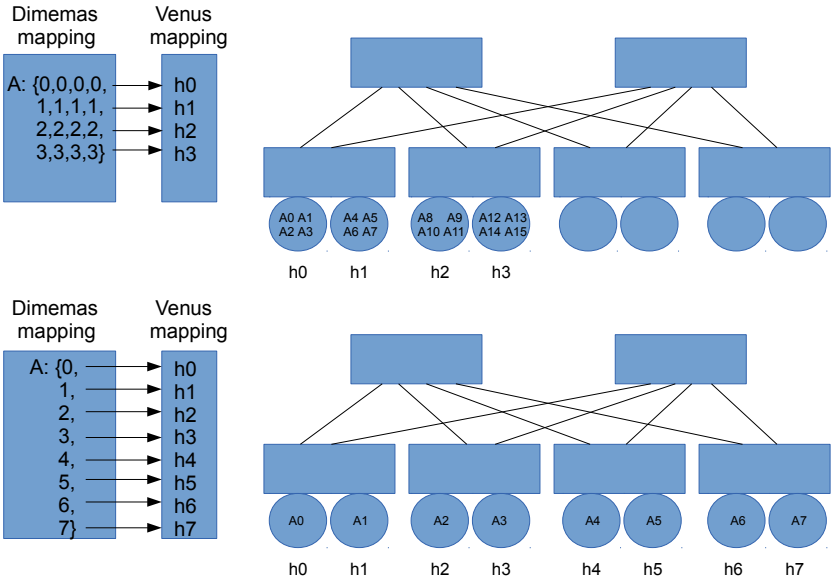


Figure 3.3: Task-to-nodes mapping in Dimemas and Venus. The upper figure shows how various Dimemas tasks are mapped to the same Venus node while the bottom figure is showing how each Dimemas task is mapped to different Venus node.

flit size of the real network while the second parameter is *unit_time* which is time needed to transfer volume of data defined by *unit_size* over the newtork link.

Dimemas has bus-based interconnect model that does not capture important network-related aspects, such as topology, routing policies, flow control, traffic contention and congestion, deadlock

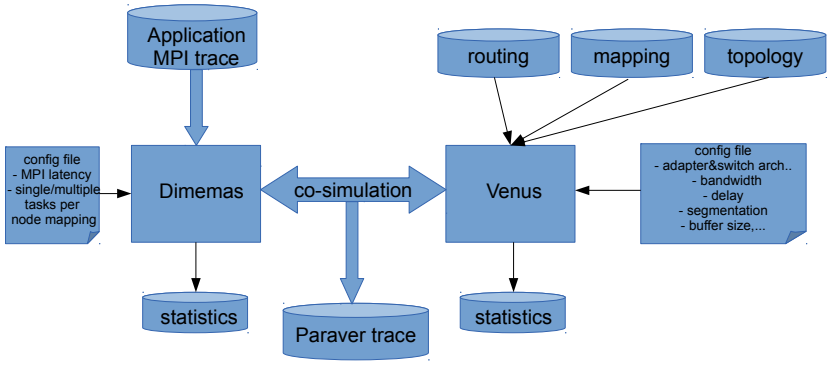


Figure 3.4: Dimemas & Venus co-simulation toolchain.

prevention, and anything relating to switch and adapter hardware implementations. Therefore, in order to increase simulation accurancy we use Dimemas integrated with Venus. The complete toolchain of Dimemas-Venus co-simulation is given in the Figure 3.4.

Chapter 4

Enhancing Network Efficiency Using MPI Data Compression

4.1 Motivation for MPI message compression

In order to motivate this work we first determined an upper bound on the improvement in performance due to data compression. Figure 4.1 shows the speedup of the two kernels, Alya's Conjugate Gradient (CG), with Input A, which consists of 500,000 doubles, and GROMACS's Particle Mesh Ewald (PME), with more than 1,000,000 doubles. This plot also shows the upper bounds, which are the ideal case where the time spent in certain MPI communication routines is reduced to zero. In this case, the absolute maximum increase in speedup, due to absence of communication, is 32% for CG (38.5× instead of 29×, for 256 processors). The absolute maximum increase in speedup for PME is 22% (42.5× instead of 34×, for 256 processors).

The `transmission time` of MPI messages occupies an important part in the total latency of the MPI routines. As the size of a message gets bigger, so does the time spent in transmission. In

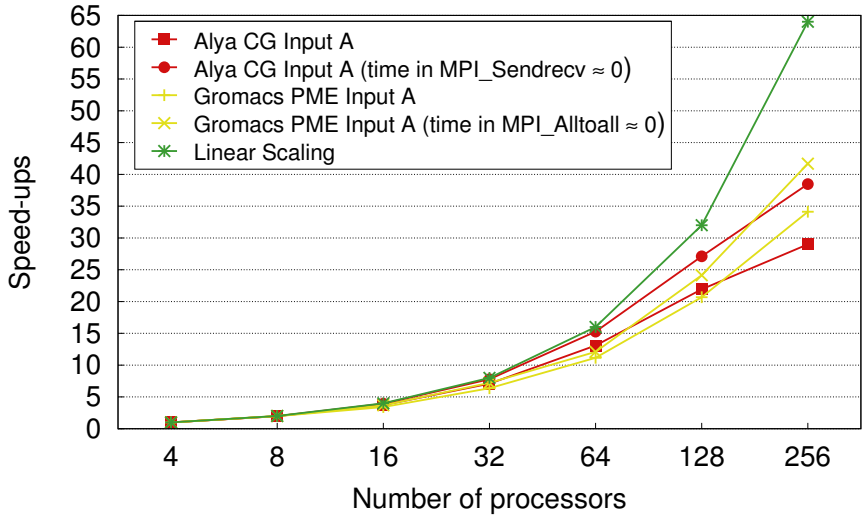


Figure 4.1: Parallel scaling of CG and PME kernel on IBM compute cluster machine with 1 process per node

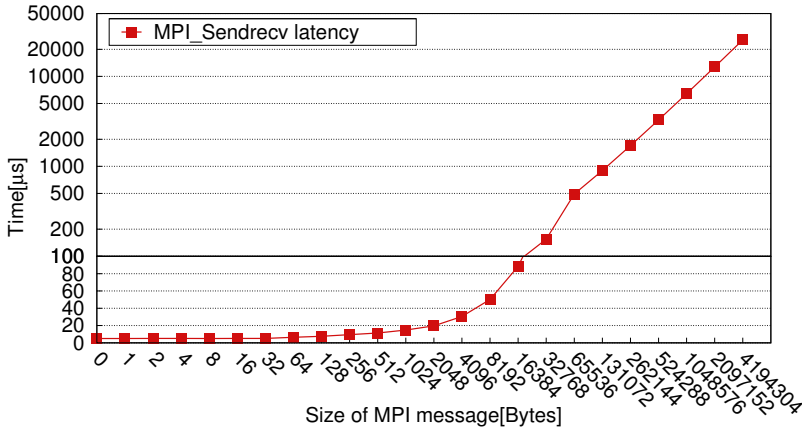


Figure 4.2: `MPI_Sendrecv` latency on IBM J21 Myrinet cluster.

order to measure the latency of `MPI_Sendrecv` and `MPI_Alltoall` calls for different sizes of messages we used the IMB-MPI1 part of Intel MPI Benchmarks[11]. We measured the performance for `MPI_Sendrecv` and `MPI_Alltoall` functions in order to understand the impact of message size on the total latency of the MPI call. Figures 4.2 and 4.3 show the latencies for messages of various sizes.

The results show that for messages smaller than 1 KB almost no savings in time can be obtained by reducing the message size. For message sizes larger than 1 KB, reducing the message size can result in

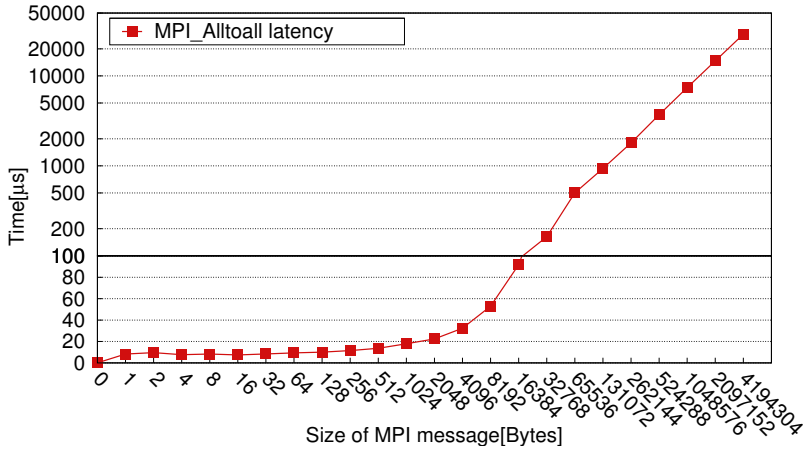


Figure 4.3: `MPI_Alltoall` latency on IBM J21 Myrinet cluster.

significantly reduced message latency. To reduce the size we apply compression to the MPI messages and analyze if the reduced `transmission time` can also result in improved application performance.

4.2 Evaluation of Performance Benefits

Compression schemes have been proposed to compress data on the network in order to reduce the overall execution time of applications. In scientific applications this data is frequently formatted as IEEE-754 double-precision floating point numbers.

In this thesis, we consider both lossless compression algorithms as well as lossy compression algorithms, where for the latter the compressor reduces precision based on a target accuracy specification. Besides our goal of using compression to increase the effective network bandwidth, it can also be used to improve memory bandwidth [38] and disk bandwidth in I/O intensive applications [63]. When applicable, compression allows to design systems with slower components, therefore enabling a reduction in energy and system costs. In the case of HPC applications it is important that compression can be done in a single pass in order to minimize the compression latency. When runtime compression of MPI messages is considered, the overheads introduced by compression and decompression opera-

tions play an important role. If the communication to computation ratio in an application is not big enough then these overheads can prevent the compression technique to achieve the expected speedups. In MPI compression it is necessary to perform one compression and one decompression routine for every single data transfer. The sum of all these transfers may lead to performance degradation. One option to remove compression/decompression overheads is to overlap the process of compressing data with the interchange of previously compressed data, exploiting parallelism to achieve negligible performance degradation as much as possible. An alternative approach is to consider streaming compression/decompression hardware. In such a scenario the task of compressing the data can be performed on the fly when data is copied from main memory to the memory on the network interface card. The procedure is inverted for decompression. Such a compressor/decompressor can appear in the form of an ASIC, FPGA or it can be a dedicated microcontroller.

4.3 Real Machine Tests

4.3.1 Methodology

To measure the impact of compression and decompression on the execution time of parallel programs we use the Paraver tool[21] to visualize the traces obtained at runtime using the `Extrae` tool (both are described in more details in Section 3.3) together with the execution times that programs themselves output. Computation time, communication time and the execution overheads due to compression/decompression are measured using `Paraver`. In order to predict execution times without compression overheads, which would be the case when assuming streaming compression hardware or overlapped compression/communication, we subtract the original average communication time from the unmodified trace and add the average communication time obtained from the traces with MPI compression. These numbers are averaged over a series of executions. The resulting number is

an approximation. The effects of compression are distributed across program threads and may also impact scheduling decisions. It is thus not possible to give an accurate prediction. However, as will be seen the trends that are observed are consistent with expectations, which raises the confidence towards the obtained results.

The programs are executed on the MareNostrum compute cluster described in more details in Section 3.1. Although the machine contains over 10k cores, we did not run simulations larger than 256 processors. Since we use rather small inputs for our workloads, we already reach a communication-dominated strong scaling scenario with a medium number of processors ($\sim$256). Each node of MareNostrum machine has two dual-core processors. Therefore, we chose to run the tests with one process per node and four processes per node. Communication between processes on the same node is done through shared memory. Our tests for the configuration with 4 processes per node starts with 8 MPI processes in order to also have inter-node communication and not just intra-node communication. When 1 process per node is used, we avoid any other effects that could be provoked by multiple process sharing the same memory and NIC. In the case of configuration with 4 processes per node, we may have more traffic through the NIC and thus, compression should help more, reducing the contention on the NIC.

In order to analyze compression we adapted several publicly available compressors to the PowerPC970 platform. Among lossless compressors, MAF and GZIP were directly available and did not require further modification. In the case of FPC, we had to adapt the code to the big-endian format of the PowerPC platform. This was not trivial as FPC uses many tricks that rely on the little-endian format. Our final port of FPC is not as efficient as the original little-endian code, but the difference is negligible (just a few additional bytes per message).

An important consideration when using lossless compression is that the resulting message size after compression cannot be deterministically computed. This is important in the case of MPI compression because it means that the receiving process does not know how many bytes it needs to read. In our

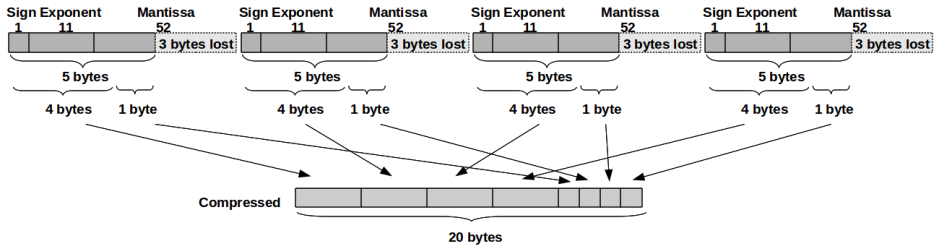


Figure 4.4: Packing algorithm [2] for `doubles` when reduction to 5 bytes is chosen

implementation we solved this problem by splitting the send procedure into two steps. In the first step just the size of the message is communicated and in the second step the message is sent. This solution is not optimal as it introduces overheads. A better solution would be to allocate the data buffers inside the MPI stack dynamically.

To apply lossy compression on CPUs we used a freely available code [2] that discards low-order bytes and packs double-precision floating point data into an array of 32-bit unsigned integers. Thus, a double-precision floating point number originally of 8 bytes can be cut to 5 or 6 bytes, which corresponds to 29 or 37 bits of mantissa. We modified the code so that double-precision floating numbers can be further cut to 3 and 4 bytes, corresponding to 13 and 21 bits precision.

Upon receiving compressed data the unpacking is done. The decompressed data is returned by extending with zeros to the original double-precision format but now having a reduced accuracy due to the quantization error resulting from cutting the mantissas. The necessary code for packing and unpacking along with the original MPI functions is inserted in an MPI wrapper function. A new library called packMPI was created with all MPI wrapper functions. They have the same arguments as the original MPI functions plus one to specify how many bytes of the original `double` value to keep. Thus, the original MPI functions can be replaced with the new wrapper MPI functions `packMPI_Alltoall` and `packMPI_Sendrecv` at any point in the application where one wants to compress the volume of transferred data. Figure 4.4 shows how the packing algorithm [2] works in a software implementation. If streaming hardware is considered, the implementation of the packing algorithm is straightforward.

It stores the words that are sent to the NIC in an input buffer and activates appropriate bit-lines corresponding to the chosen compression rate. Therefore, only a few cycles of latency will be added, which can be considered negligible.

4.3.2 Case Study: Alya

The problems that we selected in our investigation use the conjugate gradient (CG) method[57] of Alya application as the main kernel. The execution of the CG kernel consists of (i) one MPI point-to-point communication and (ii) two MPI group communications. Significant amounts of data are transferred in the `MPI_Sendrecv` point-to-point communication where local parts of the search vector are exchanged in order to obtain the final matrix-vector multiplication result. Group communication is employed for the final dot-product summation of vectors in the algorithm but only one `double` per MPI call is exchanged at this stage.

Table 4.1: Average size of MPI messages (kB)

N proc	Alya CG `Input A`	Alya CG `Input B`	Gromacs PME `Input A`	Gromacs PME `Input B`
4	14.4	47.8	559.8	4144.2
8	15.4	47.7	139.9	1036.0
16	7.9	25.8	147.8	259.0
32	3.9	17.4	55.4	350.4
64	2.2	9.2	18.4	135.5
128	1.4	5.0	8.0	53.2
256	0.8	2.9	2.6	16

For the Alya CG evaluations we have used two input data sets which we call `Input A` and `Input B`. The `Input A` has size of 5×10^5 while `Input B` is larger and has 4×10^6 nodes. The exchanged mes-

Table 4.2: Average compression rates achieved on MPI messages in the SMVM part of CG kernel

Alya CG	GZIP	FPC	MAF
Input A	1.03	1.007	1.05
Input B	1.12	1.086	1.02

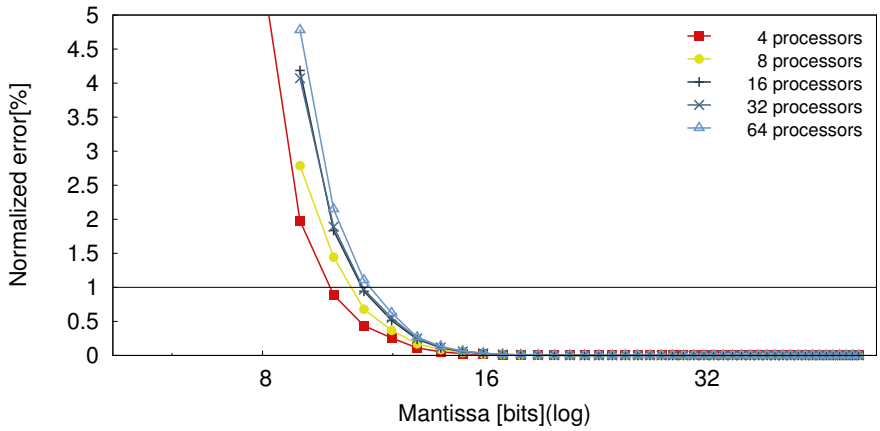


Figure 4.5: Normalized error of solution X for *Input A* as a function of size of mantissa in SMVM communication

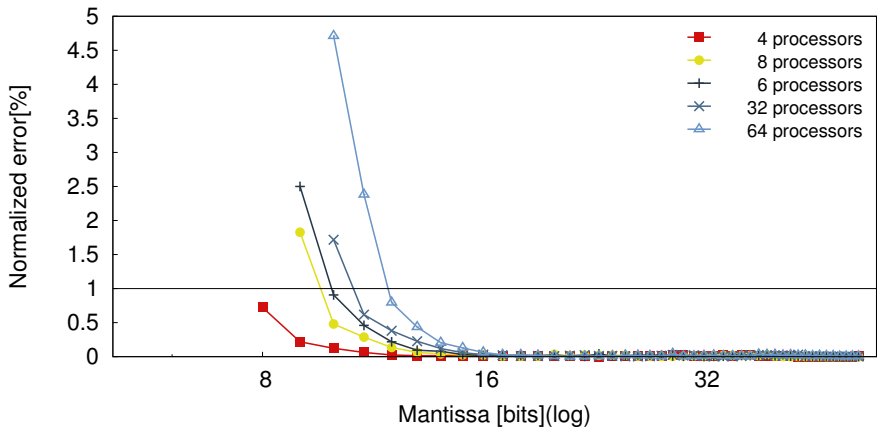


Figure 4.6: Normalized error of solution X for *Input B* as a function of size of mantissa in SMVM communication

sages are compressed before sending. After reception, decompression is performed. Table 4.2 shows the compression rates achieved by using two compression algorithms specifically designed for floating point data (FPC and MAF) and one general purpose, dictionary-based compression algorithm (GZIP). As can be seen, none of these algorithms is able to truly understand the data exchanged across nodes. Thus, the compression rates that we obtained are very poor (larger rate than 1.0 means the compressed size is smaller than uncompressed size). In addition, as execution proceeds over multiple iterations, compressibility of the data does not change significantly. Based on the obtained results,

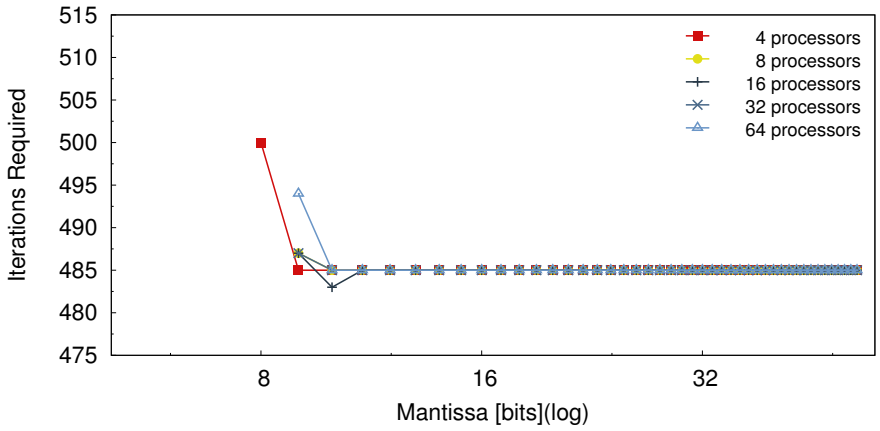


Figure 4.7: Iterations Required as size of mantissa in SMVM communication

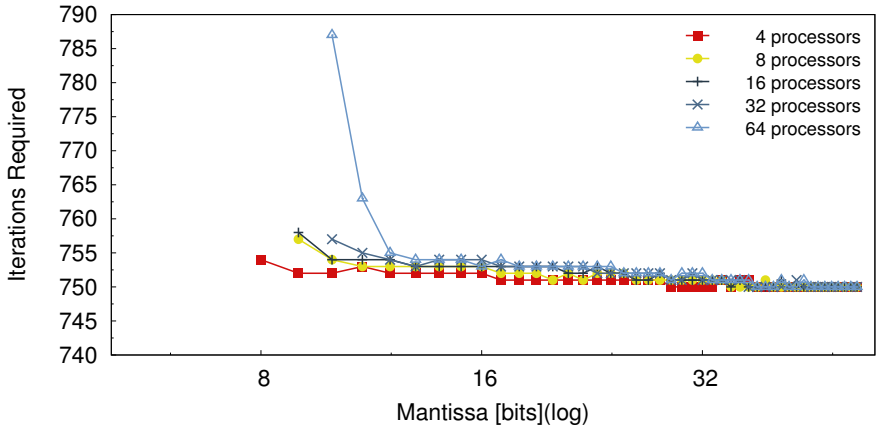


Figure 4.8: Iterations Required as size of mantissa in SMVM communication

only lossy compression is meaningful to achieve higher compression rates. In order to apply lossy compression in scientific simulation a criteria of accuracy has to be established. For this test case, the application developers determined a 1% deviation in the final result to be acceptable. The tests of accuracy have been done in order to determine how many low-order bits of every double can be discarded. Our tests show that keeping only 3 bytes of the original double-precision floating point value does still produce an acceptable result. Figure 4.5 and Figure 4.6 show how the normalized error of the resulting array changes with the precision of the mantissa. The results are obtained by using the

GNU Multiple Precision Floating-Point Reliably (MPFR) library[6] to emulate different precisions in floating point data. MPFR applies rounding when reducing the precision, meaning that these results are not based on a simple mantissa cut. When reducing the precision we need to be careful with the effects on the latency of the CG kernel. It is known that reducing precision can result in an increase of the number of iterations necessary for algorithm convergence. This increase in the number of iterations can easily offset the gains obtained thanks to compression. As we can see in Figure 4.7 and Figure 4.8 for our test cases the number of iterations until convergence increases by only 1% even when precision of mantissa is low. Another interesting effect that can be observed is that as more processors are added, more bits of mantissa are required so that the final solution of the system converges and satisfies the criteria of accuracy (1% normalized error - maximum error compared to double precision). This happens because the number of frontiers exchanged between processes increases and, therefore, discarding bits of the mantissa has more influence on the final result.

Performance result using lossy compression with 1 process per node configuration

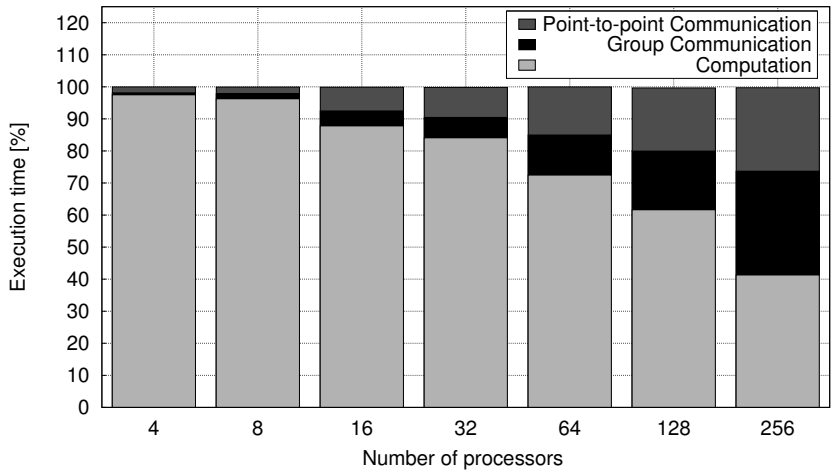


Figure 4.9: Profile of Alya CG *Input A* kernel with 1 process per node

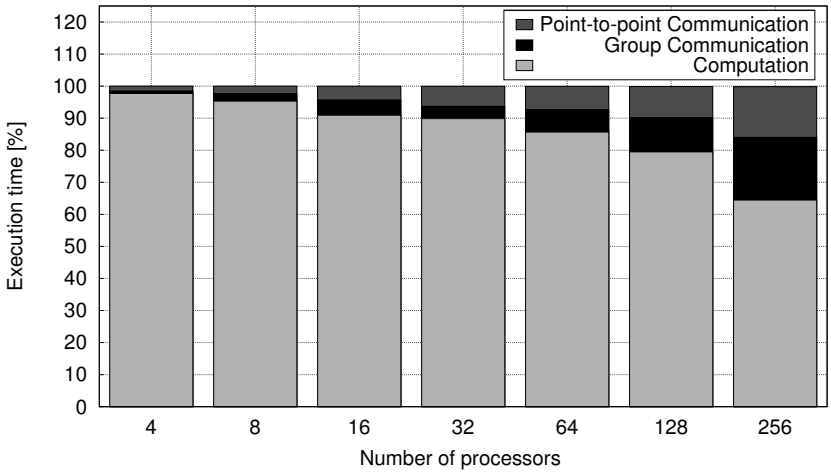


Figure 4.10: Profile of Alya CG *Input B* kernel with 1 process per node

Execution profiles of the Alya CG kernel for both input data sets are shown in Figure 4.9 and Figure 4.10. We obtained the biggest improvement of 3.6% for `Input A` when 64 processors are used. This is because the exchanged MPI messages are still sufficiently large (average size is 2.2 kB). Therefore, sufficient time can be saved when lossy compression is applied and hence, this is reflected in the speedup factor. The average sizes of the MPI messages for `Input A` and `Input B` on different number of processors are shown in Table 4.1. Since `Input B` is a larger data set than `Input A`, more computation has to be done per processor. This leads to percentually smaller communication time and thus smaller speedup factors for `Input B`. But as can be seen from Figure 4.28, speedup factors for `Input B` continue to improve, while for `Input A` (Figure 4.27) they do not follow an ascending curve. This occurs because for more than 64 processors MPI messages are too small and reducing the size of the message will only have limited impact on the overall communication times. Although the percentage of point-to-point communication time rises with the number of processors, it is not just because more communication is performed, but also because of synchronization overheads. This means that during the communication phase processes start to spend more time waiting for each other. Therefore, we observe that under a larger number of processors compression will have a higher impact and

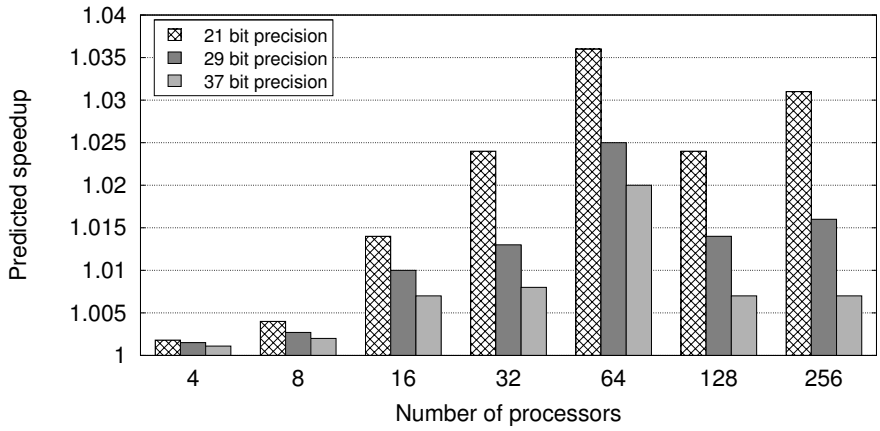


Figure 4.11: Speedup factors for *Input A* achieved applying lossy compression on MPI messages in SMVM part of CG with 1 process per node

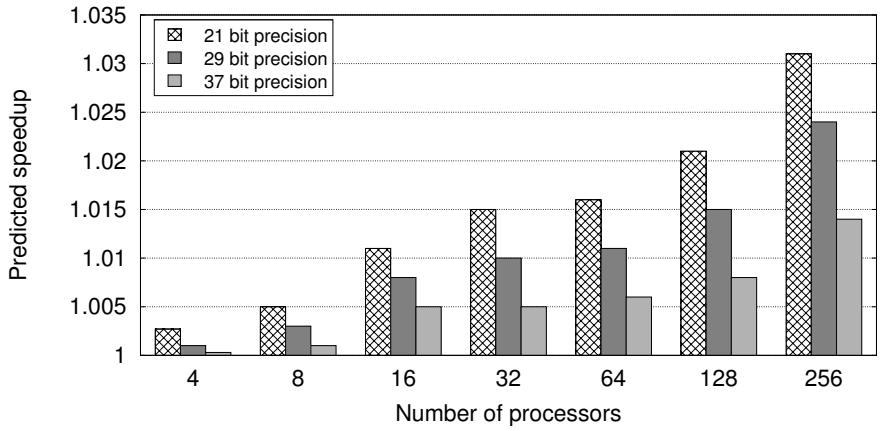


Figure 4.12: Speedup factors for *Input B* achieved applying lossy compression on MPI messages in SMVM part of CG with 1 process per node

overall execution times will decrease linearly. This will happen unless the MPI messages get so small (<1 KB) that the total latency of the MPI message becomes dominated by the `sending` and `receiving overhead` and the `time of flight` (i.e., the time for the first bit of the packet to arrive at the receiver) rather than the time corresponding to message `transmission`.

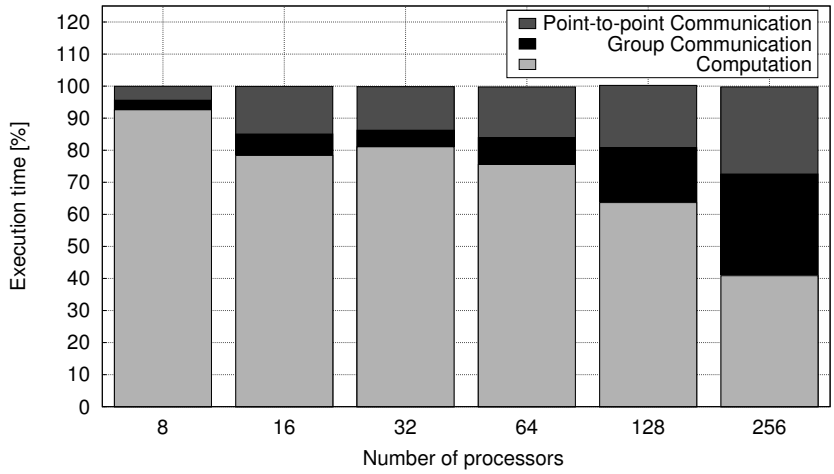


Figure 4.13: Profile of Alya CG *Input A* kernel with 4 processes per node

Performance result using lossy compression with 4 processes per node configuration

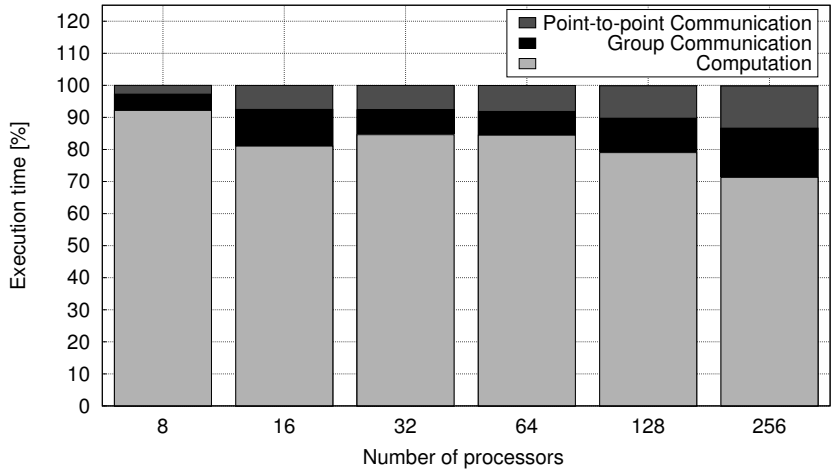


Figure 4.14: Profile of Alya CG *Input B* kernel with 4 processes per node

Depending on the number of processes assigned per node, the performance of the CG kernel will differ because the nodes on the MareNostrum cluster are SMPs. However, the communication pattern of the CG kernel depends on the domain decomposition which is done by METIS[10]. METIS is a graph

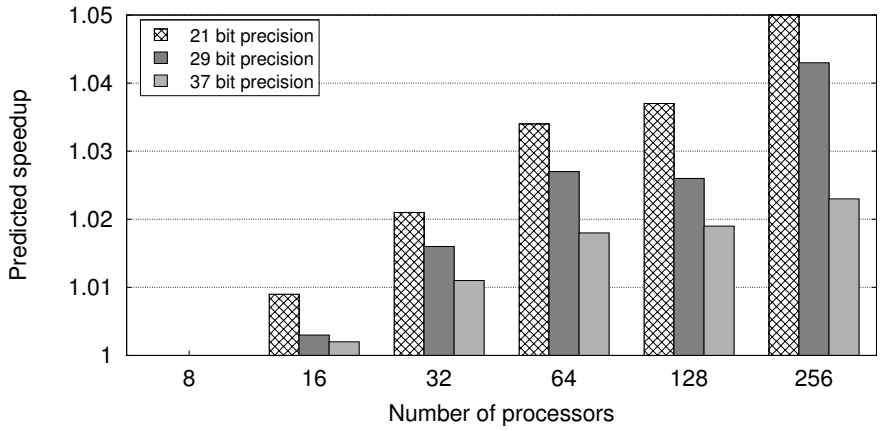


Figure 4.15: Speedup factors for *Input A* achieved applying lossy compression on MPI messages in SMVM part of CG with 4 processes per node

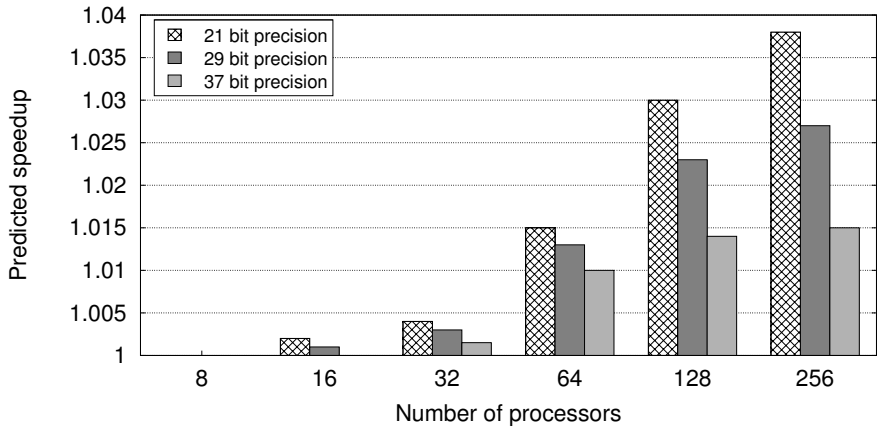


Figure 4.16: Speedup factors for *Input B* achieved applying lossy compression on MPI messages in SMVM part of CG with 4 processes per node

partitioning algorithm that partitions the domain with the goal of minimizing the communication between partitions. Precisely, because of these communication patterns processes on the node could have almost all communication inside the node or almost all outside of the node. This patterns will also show up in the profile of the CG kernel, but even more on the speedup factors achieved by the compression of MPI messages. Figure 4.13 and Figure 4.14 show the profile of the CG kernel with 4

processes per node for both input data sets. Improvement factors are smaller than in the case when the 1 process per node configuration is used because some communication will stay inside the node and will be done using shared memory. This is observed when an 8-processor configuration was used. The impact of compression was hardly measurable because just a few MPI messages passed through the NIC.

4.3.3 Case Study: Gromacs

The second application studied is Gromacs[7]. Every time-step it computes the forces for all atoms of the system. For the calculation of the forces it uses the Particle Mesh Ewald (PME) method[53]. PME is the main kernel in Gromacs so we analyze possibilities to compress MPI messages that occur there. For our tests we obtained two input sets each perform a 200-timestep simulation of a protein consisting of 145,732 and 1,094,681 atoms respectively.

In PME the sum of long-range forces is processed in Fourier space where this sum converges much faster as compared to real space. The transformation to Fourier space is performed via a 3D Fast Fourier Transform. PME contains other sections with data communication but we restrict ourselves to the FFT since this kernel is notable for requiring dense communication due to multiple `MPI_Alltoall` operations. In Gromacs, depending on the number of processors used to run the simulation, 1D or 2D domain decomposition is performed. This has influence on the 3D-FFT execution as either one or two collective communication will need to be performed during the transformation. For 4 and 8 processors a 1D domain decomposition is performed and for all other test cases a 2D domain decomposition is performed.

In the case of 1D decomposition (also called slab decomposition) the FFT is applied to the first two dimensions on each node, but for the last one it is necessary to rearrange the elements using a system wide transpose so that every processor has all elements for the third dimension to locally per-

form the FFT transformation. In the case of a 2D decomposition (also called pencil decomposition), two transpose operations are necessary, one between each of the three FFT operations. First, the FFT is applied in one dimension. Then the first transpose is performed on a subgroup of processors and then the FFT transformation in the second dimension is performed. To perform the transformation in the third dimension, a last transpose is completed between subgroups. For the transpose operation, standard `MPI_Alltoall` calls are performed.

Table 4.3 shows the compression rates achieved using the same lossless compressors as in the case of the Alya system, applied to the data of the transpose communications of the 3D FFTs. The results are similar. The unpredictability of the data results in very poor compression rates.

Therefore, we proceed to study the applicability of lossy compression algorithms. By looking at the outputs created by Gromacs we determined that only if we truncate more than 4 bytes of LSBs of mantissa in the transpose parts of 3D FFT we will observe significant deviation in the final result. Thus, we analyzed reductions to 4, 5 and 6 bytes of packed floating point data. Again, by using MPFR, we concluded that keeping as few as 16 bits of mantissa would work, but this width is not suited for general purpose processors that operate on octets. However, other platforms relying on hardware compressors could benefit from this.

Performance results using lossy compression with 1 process per node configuration

Execution profiles of the PME kernel for both input sets are shown in Figure 4.17 and Figure 4.18. When going from executions with a smaller number of processors and to a larger number of proces-

Table 4.3: Average compression rates achieved on transpose MPI messages in PME-3D FFT part of Gromacs

PME-3DFFT	GZIP	FPC	MAF
Input A	1.034	0.989	1.076
Input B	1.05	1.012	1.043

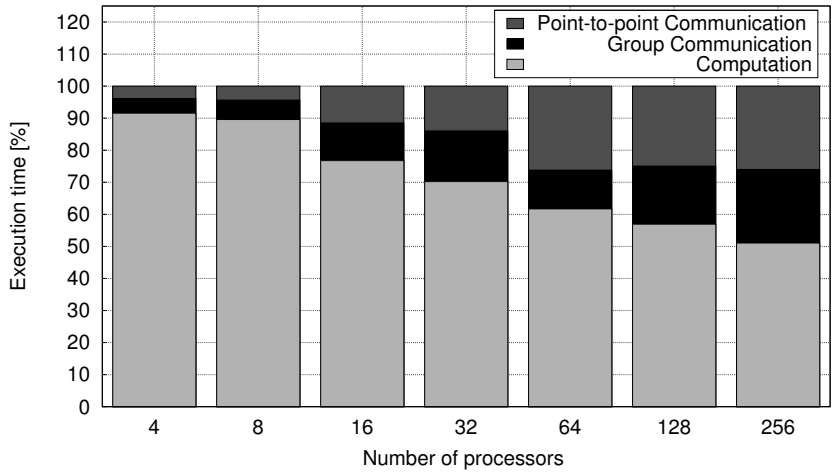


Figure 4.17: Profile of Gromacs PME *Input A* kernel with 1 process per node

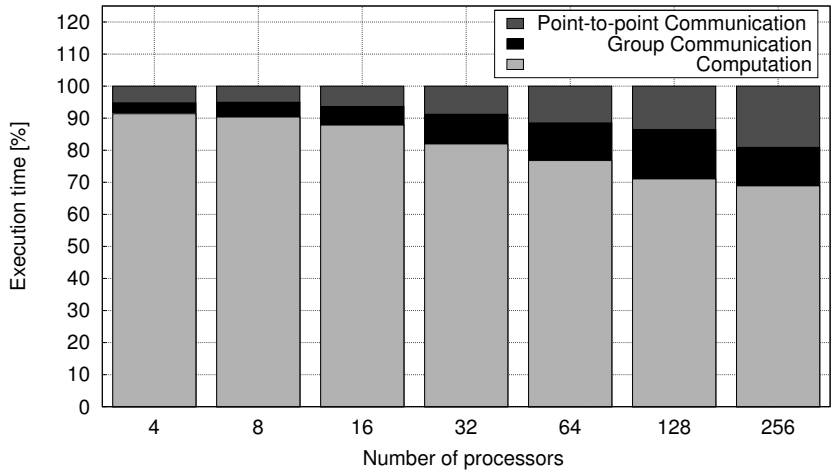


Figure 4.18: Profile of Gromacs PME *Input B* kernel with 1 process per node

sors, the contribution of group communication to the total execution time rises. Interestingly, for `Input A` with 64 processors we obtain a smaller contribution than with 32 processors. A similar phenomenon occurs with `Input B` where the contribution with 256 processors is smaller than with 128 processors. Table 4.1 provides an explanation to this behavior. As we can see, at 64 processors for `Input A` and 256 processors for `Input B` the average size of messages gets below 32KB. 32KB is a

threshold value used on the compute cluster machine to activate an optimized message passing protocol known as *Eager*. In the Eager protocol the acknowledgements used by the general *Rendezvous* protocol are removed. In this scenario data is automatically sent to the receiver assuming that he will always be able to store the message and thus no acknowledgement is waited for.

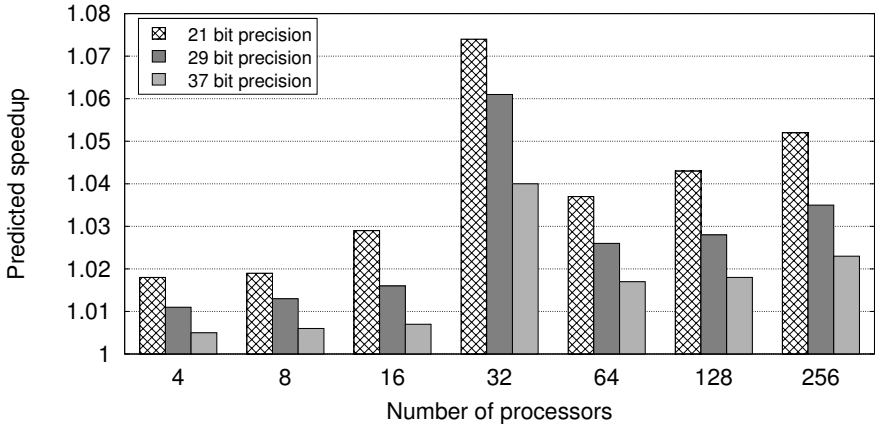


Figure 4.19: Speedup factors for *Input A* achieved applying lossy compression on MPI messages in 3DFFT part of PME kernel with 1 process per node

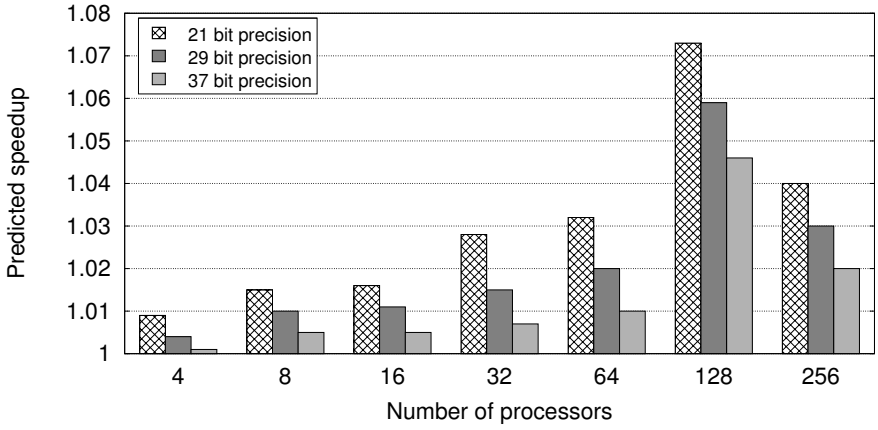


Figure 4.20: Speedup factors for *Input B* achieved applying lossy compression on MPI messages in 3DFFT part of PME kernel with 1 process per node

Due to the nature of `MPI_Alltoall`, tasks tend to be well synchronized. Thus, the applied reduc-

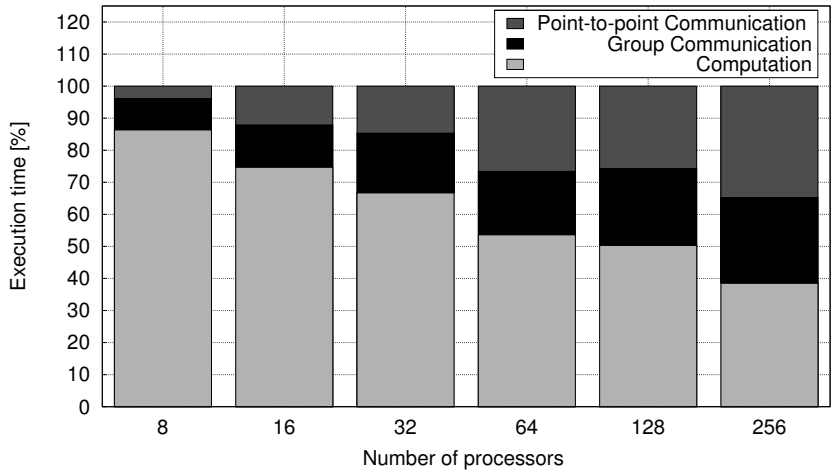


Figure 4.21: Profile of Gromacs PME *Input A* kernel with 4 processes per node

tion on the volume of data transferred has a direct influence in the decrease of time spent in communication. The results are presented in Figure 4.19 and Figure 4.20. Again we observe that speed-ups due to compression improve when more processors are used. The PME execution improves up to 7.4% for `Input A` and 7.3% for `Input B` when 50% lossy compression is applied on 32 and 128 processor execution respectively. We observe larger speed-ups on executions with 32 processors than with 64, 128 and 256 processors for `Input A`, and larger speedup factors for `Input B` with 128 processors than with 256 processors. This occurs because at that point the original MPI messages are transferred using *Rendezvous* protocol, but compressed message sizes go under the threshold value and the *Eager* message passing protocol is activated. In this case, in addition to the compression effect also the effect of optimized message protocol is added. As the number of processors increases, the 3DFFT part of PME kernel becomes less dominant in overall execution times. Thus, the percentage tends to increase very slowly when the full PME kernel is observed.

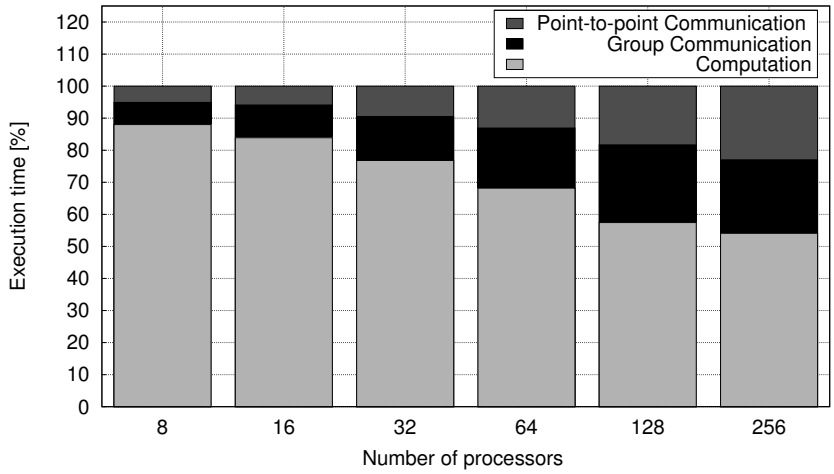


Figure 4.22: Profile of Gromacs PME kernel *Input B* with 4 processes per node

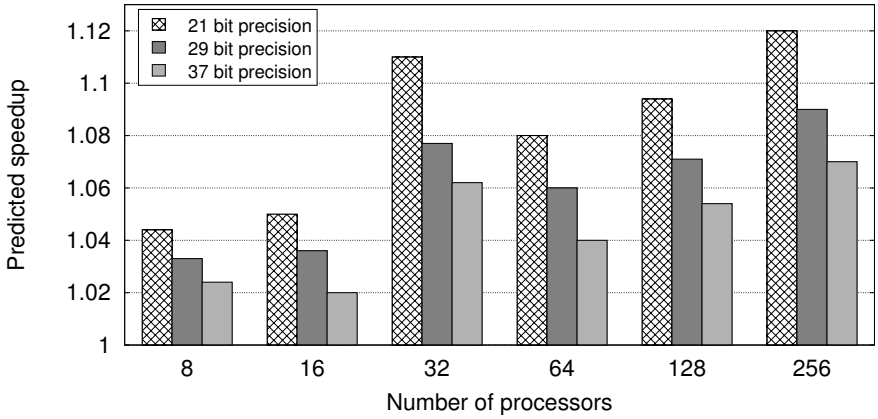


Figure 4.23: Speedup factors for *Input A* achieved applying lossy compression on MPI messages in the 3DFFT part of the PME kernel with 4 processes per node

Performance result using lossy compression with 4 processes per node configuration

The performance on a cluster of SMPs varies depending on the number of processes used per node. Beside shared memory contention, all 4 MPI processes will share the NIC on the same node. Since the `MPI_Alltoall` tends to synchronize the tasks, the processes may compete for the NIC at the same time. The intra-node communication on the tested cluster is done through shared memory, although

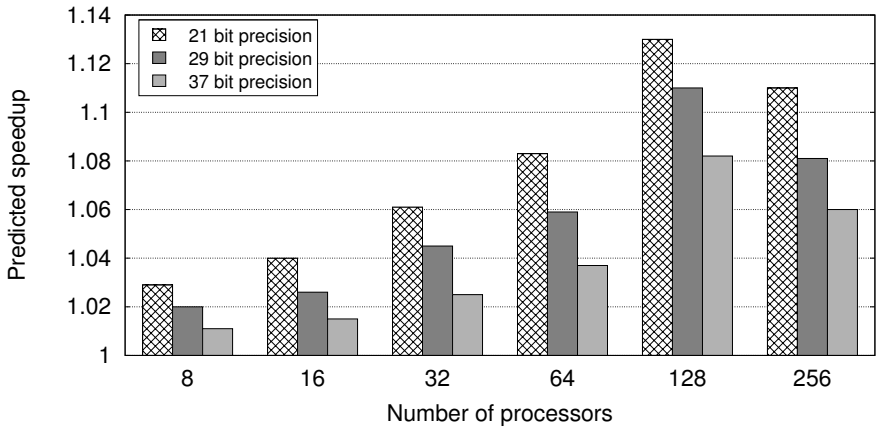


Figure 4.24: Speedup factors for *Input B* achieved applying lossy compression on MPI messages in the 3DFFT part of the PME kernel with 4 processes per node

we could use MPICH-MX library with `-noshared memory` option. We don't consider that option because it leads to slower communication on the node. The profiles of PME for both input sets are shown in Figure 4.21 and Figure 4.22. Comparing to configuration with 1 process per node, we see that the percentage of time spent on communication is increased. Thus, we are expecting to achieve greater impact on application performance by compressing MPI messages. In Figure 4.23 and Figure 4.24, we see that the trend is similar to 1 process per node configuration, but with increased speedup factors. Up to 12% is achieved for `Input A` and up to 13% for `Input B`.

4.4 Simulation Tests

In the previous section we showed the production machine tests using compression during certain inter-node communication phases in MPI application runs. Although the compression/decompression overheads are discarded from the final results they are present during the execution and can have impact on the communication patterns execution. Basically, overhead occuring on one process affects events on the other processes that are causally related. Beside that the impact of the other jobs running

on the machine which use the same communication system is not negligible. Therefore, we use simulator environment to further investigate and isolate the effects of compression on final communication times in an MPI application.

4.4.1 Methodology

To measure the impact of data compression on application execution time, we use the Venus–Dimemas simulator [49,42]. Both simulators are described in detail in Section 3.5 and Section 3.6.

Traces of the Alya and Gromacs applications were recorded on the MinoTauro machine (Section 3.1). In this architecture, allocating multiple MPI processes to the same processor implies sharing of various resources including the HCA. Data compression could alleviate contention on the HCA, especially if MPI processes communicate at similar times, which is often the case for scientific applications. In order to measure this effect, we generated traces in two configurations, first with one MPI process per processor, and second with six MPI processes per processor (one per core). Also, we generated strong scaling traces, so that as the number of processes was varied, the workload remained the same.

Table 4.4: Parameters used in Simulations

Parameter	Value
Simulator	Dimemas–Venus
Connectivity	XGFT(2;18,14;1,18)
Topology	2-level Extended Generalized Fat Tree
Switch technology	InfiniBand
Network Bandwidth	40 Gbit/s
Memory Banwidth	2 GB/s
Segment Size	2 kB
MPI latency	1 µs
CPU Speedup	1
Routing scheme	Random routing
4X-IB link port power	2W peak

Table 5.2 gives the parameters of the simulated system. We first ran the simulation without any changes in the trace, replicating the original execution times. For the performance analysis , we reduced the sizes of the MPI messages in accordance with the compression rate. Using lossy compression the message size was therefore reduced by 25%, 37.5% or 50%, equivalent to compressing the double-precision FP data by truncating the 16, 24 or 32 least-significant bits. Since this is done in hardware, only a few additional cycles would be needed, which can be considered negligible, validating the assumption of zero overhead for hardware lossy compression. When multiple MPI processes are mapped to one node, communication inside the same node is done without compression. Finally, we simulate the new traces on Venus–Dimemas, and quantify the performance.

4.4.2 Case Study: Alya-CG kernel

We first investigate the performance benefits of lossy compression for the communications in Alya[3]. Specifically, we applied compression to the Sparse Matrix–Vector Multiplication (SMVM) kernel in the Conjugate Gradient (CG) algorithm. We use two input sets, a small one denoted Input A and a large one denoted Input B.

Table 4.5: Average size of MPI messages(kB) with 1 MPI process per node

Number processes	Alya CG Input A	Alya CG Input B	Gromacs PME Input A	Gromacs PME Input B
8	15.4	47.7	139.9	1036.0
16	7.9	25.8	147.8	259.0
32	3.9	17.4	55.4	350.4
64	2.2	9.2	18.4	135.5
128	1.4	5.0	7.9	53.2

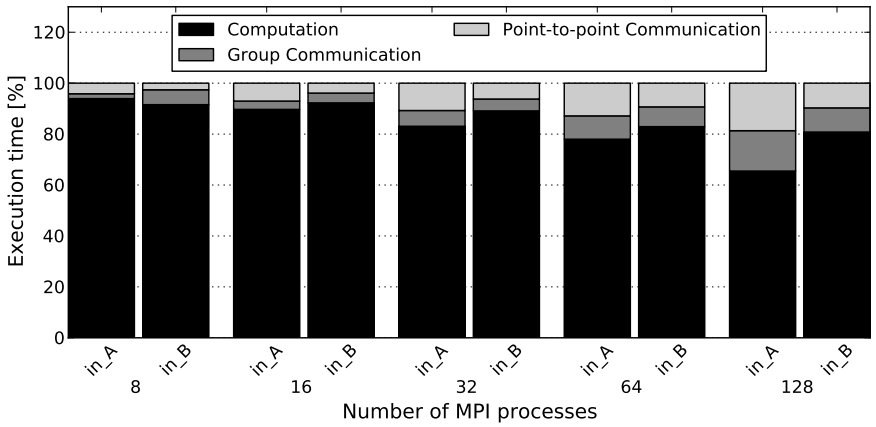


Figure 4.25: Profile of the Alya CG *Input A* kernel - 1 MPI process per node

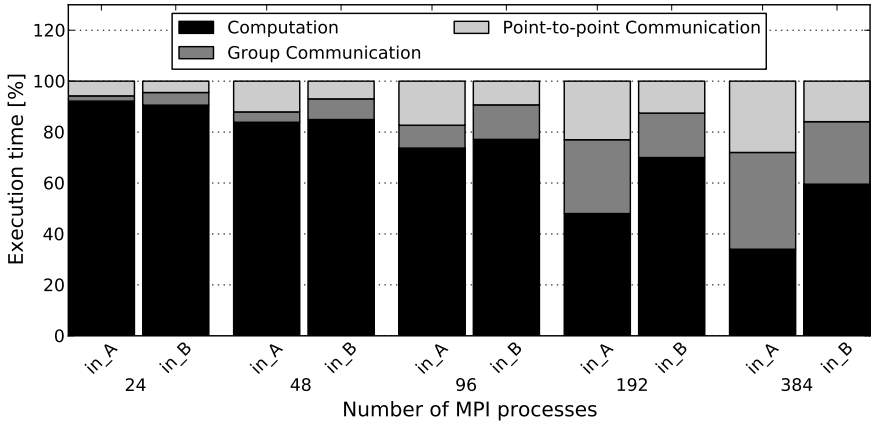


Figure 4.26: Profile of the Alya CG *Input B* kernel - 6 MPI procecesses per node

Performance using lossy compression with one MPI process per node

Figure 4.25 gives execution profiles of the Alya CG kernel for the two input data sets. For this configuration, all communication is inter-node, so all MPI messages must pass through HCAs. Compression leads to smaller MPI messages, meaning that MPI messages arrive sooner. Table 4.5 gives the average MPI message sizes, in kilobytes, for the two input sets varying the number of processes.

Figure 4.27 and Figure 4.28 shows the speedup for the Alya CG kernel, for both input sets from

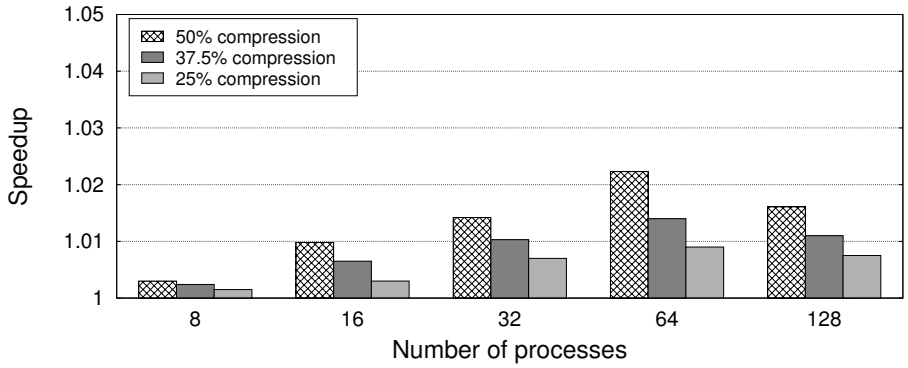


Figure 4.27: Speedup factors for *Input A* from applying lossy compression on MPI messages in SMVM kernel of CG with 1 MPI processes per node

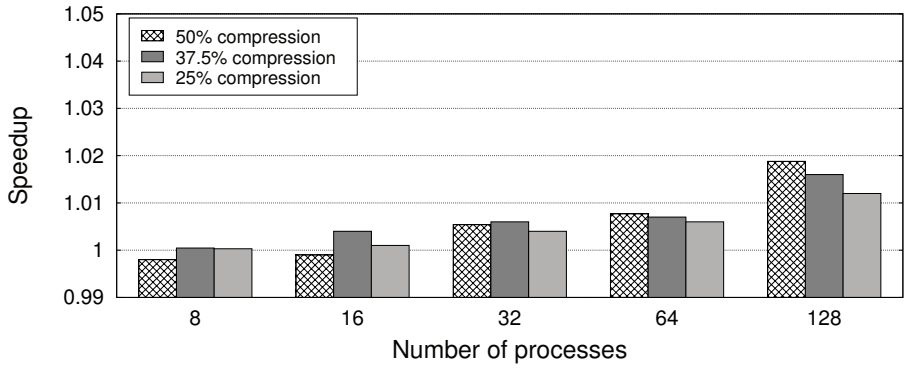


Figure 4.28: Speedup factors for *Input B* from applying lossy compression on MPI messages in SMVM kernel of CG with 1 MPI processes per node

eight to 128 processes. Speedup is measured in comparison with the corresponding baseline run without data compression. The general trend is that the speedup increases as the number of processes is increased, since the application becomes more communication-heavy, due to strong scaling. For Input A, the speedup factor slightly decreases for 128 MPI processes. This is because the size of MPI messages decreases making the application sensitive to latency rather than bandwidth. For Input A reduced precision in the computation does not result in an increase in the number of CG iterations until convergence, even at 50% compression when 32 LSB of mantissa are truncated. For Input B, reduced precision results in a slightly larger number of iterations, giving low performance improvement for

Table 4.6: Average size of MPI messages(kB) with 6 MPI processes per node

N proc	Alya CG Input A	Alya CG Input B	Gromacs PME Input A	Gromacs PME Input B
24	4.7	20.8	73.5	540.3
48	2.8	12.8	26.0	192.5
96	1.8	6.6	10.9	78.6
192	1.1	3.8	4.4	26.2
384	0.7	2.1	1.7	9.7

runs with a small number of processes. For runs with 8 and 16 MPI processes, the performance benefits of 50% compression are insuffient to compensate for the greater number of iterations, giving a degradation in performance. For the lower compression rate of 37.5% there is still a small speedup.

Performance using lossy compression with six MPI processes per node

For runs with a small number of processes, most communication in the CG kernel is between processes on the same node. As the total number of processes increases, a greater proportion of messages is between processes on different nodes. Figure 4.26 shows the profiles of the CG kernel with six MPI processes per node. Table 4.6 gives the average sizes of the MPI messages.

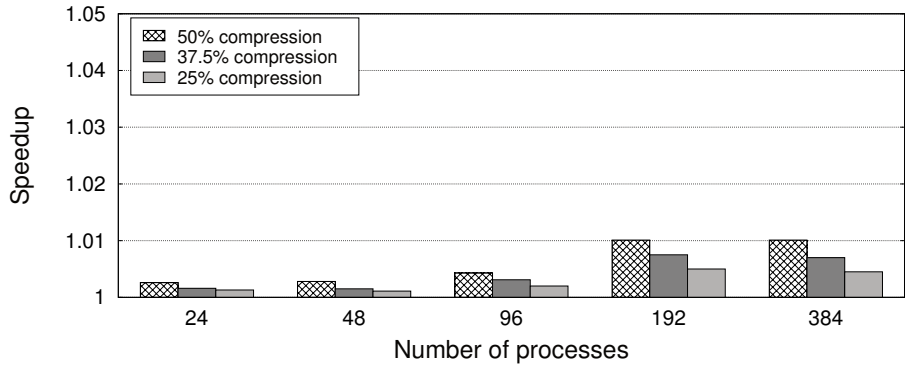


Figure 4.29: Speedup factors for *Input A* from applying lossy compression on MPI messages in SMVM kernel of CG with 6 processes per node

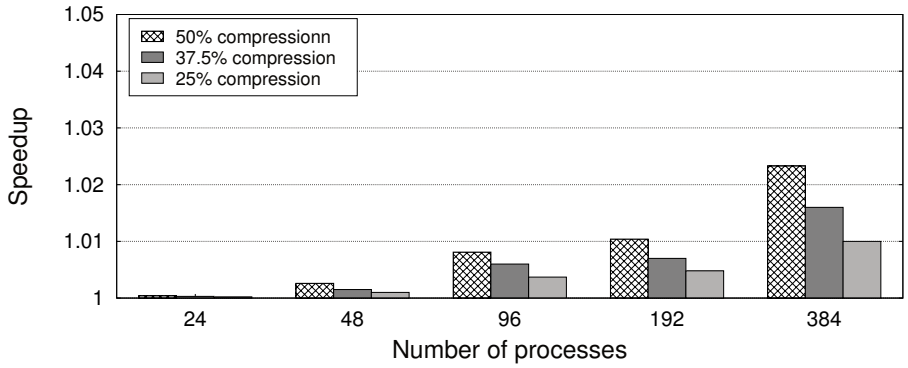


Figure 4.30: Speedup factors for *Input B* from applying lossy compression on MPI messages in SMVM kernel of CG with 6 processes per node

Since compression is only applied to data communicated between different nodes, multiple processes per node leads to smaller performance benefits from compression. Figure 4.29 and Figure 4.30 show the observed speedups. For Input A, as the inter-node to intra-node communication ratio increases, we observe larger speedup factors, especially for 192 and 384 MPI processes. Although the MPI messages become small, all inter-node communication is done via the HCAs, and the aggregate size of the MPI messages is significant. For Input B, we see that the speedup factors increase, but slowly. With 384 MPI processes, the largest performance improvement factor of just 2.3%.

4.4.3 Case Study: Gromacs-PME kernel

The second case study is Gromacs[7]. Data compression is applied to the all-to-all exchange patterns in the Particle Mesh Ewald (PME) algorithm.

Performance using lossy compression with one MPI process per node

Figure 4.31 gives execution profiles for the PME kernel for the two input data sets. Figure 4.33 and Figure 4.34 show that speedup factors tend to increase with the number of MPI processes, but for Input A with 64 MPI processes there is a small decrease in the speedup. The reason is that for this particular

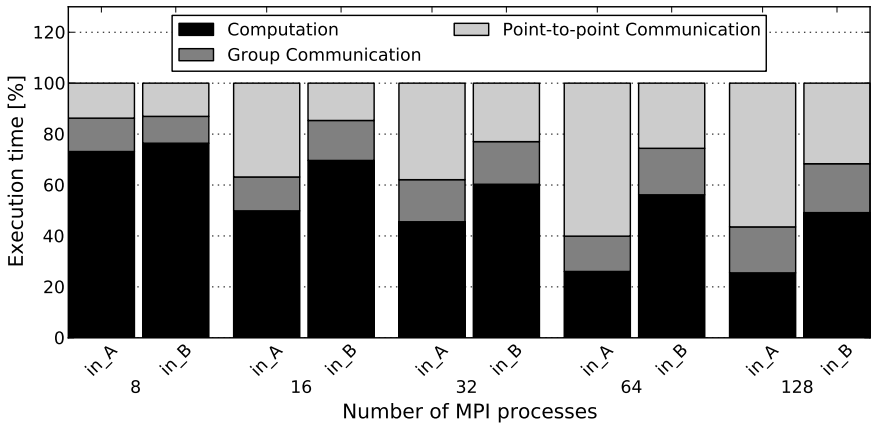


Figure 4.31: Profile of Gromacs PME *Input A* kernel

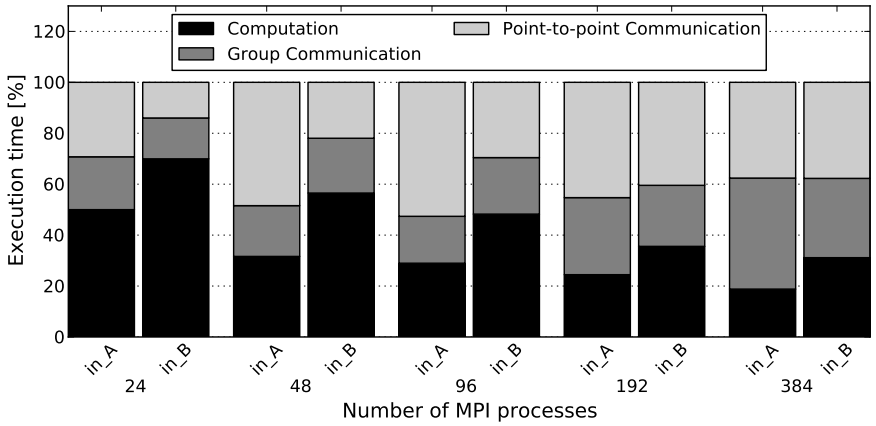


Figure 4.32: Profile of Gromacs PME *Input B* kernel

workload, Gromacs uses different domain decompositions for 32 and 64 MPI processes. This results in an inconsistent execution profile for 64 MPI processes. For the run with 128 MPI processes, we observe the maximum improvement of a little over 14%. From Table 4.5, we see that Gromacs has much larger messages than Alya, suggesting a greater performance benefit from data compression. In addition, MPI_Alltoall tends to synchronize the tasks, further increasing the performance benefits from compression.

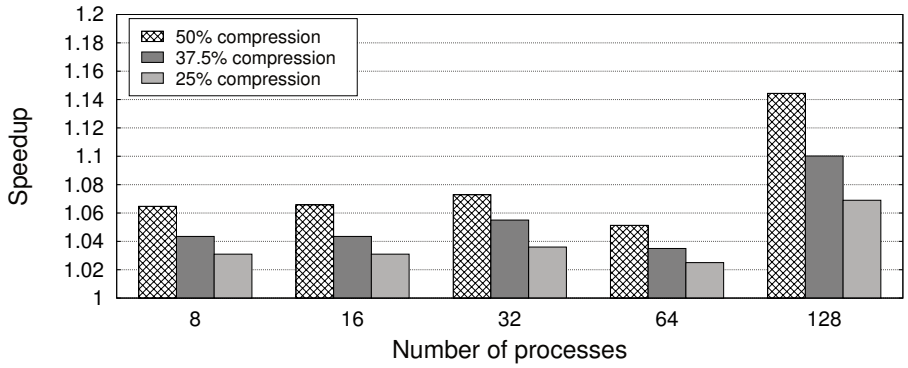


Figure 4.33: Speedup factors for *Input A* from applying lossy compression on MPI messages in the 3D FFT part of PME with one MPI process per node

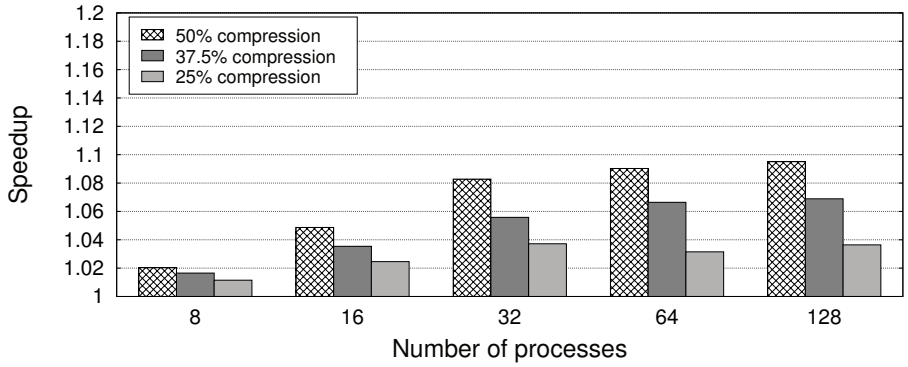


Figure 4.34: Speedup factors for *Input B* from applying lossy compression on MPI messages in the 3D FFT part of PME with one MPI process per node

Performance using lossy compression with six MPI processes per node

With six MPI processes per node, all cores are occupied. The all-to-all exchange pattern implemented in MPI_Alltoall uses non-blocking sends and receives, so each process sends data to all recipients before waiting to receive data. These non-blocking sends and receives can create contention on the HCAs. Figure 4.32 shows the profile of PME with six MPI processes per node. It is clear that a greater percentage of time is spent in communication, compared with one MPI process per node. We expect data compression to have a greater impact on application performance.

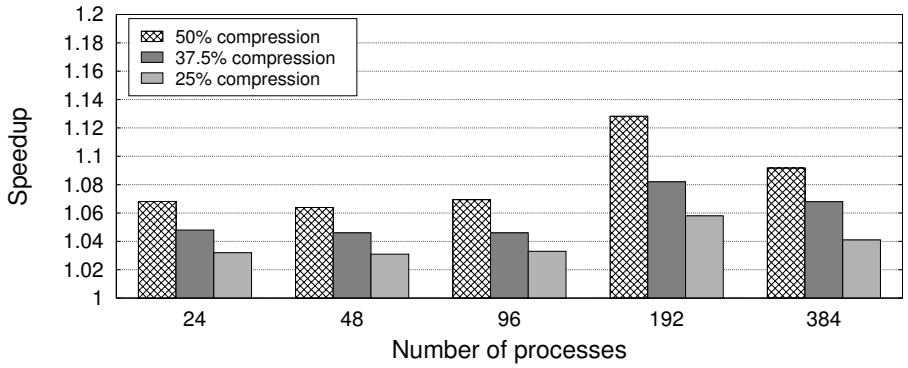


Figure 4.35: Speedup factors for *Input A* from applying lossy compression on MPI messages in the 3D FFT part of the PME kernel with six MPI processes per node

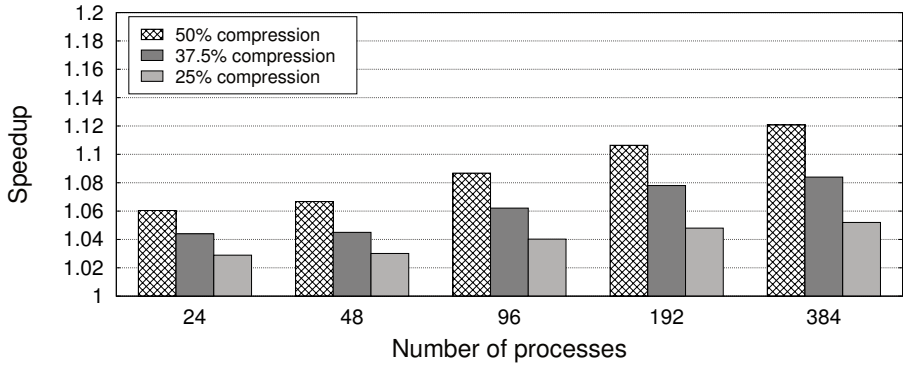


Figure 4.36: Speedup factors for *Input B* from applying lossy compression on MPI messages in the 3D FFT part of the PME kernel with six MPI processes per node

Figure 4.35 and Figure 4.36 show the speedup results, compared with the baseline with no compression. We see that for Input A, the peak speedup is for 192 processes; for 384 processes we get a lower speedup. For the largest run, the MPI messages become small ($\approx$ 1 KB) and the application becomes latency sensitive, giving a lower performance benefit. Since Input B is larger, the improvements remain consistent, even with 384 processes, giving a performance improvement of 12%.

4.5 Data compression for Network Energy Savings

4.5.1 Implementing Data Compression

Data compression and decompression often have high computational costs. Since our lossy compression scheme is simply truncation of least-significant bits (LSB) of the floating-point mantissa, it is, however, relatively inexpensive. If implemented naively in software its cost is still similar to two additional copies per message transfer (one on send and one on receive). In some cases these software copies could be merged with existing data copies. Since each extra message copy is still a significant overhead, we propose to implement truncation in streaming hardware[55,26], while the data is copied from main memory to the memory on the Host Channel Adapter (HCA). When message words arrive at the HCA, they are stored in an input buffer; the compression rate is chosen by powering down the discarded bit lines. On the receiving side, decompression is done by padding with zeros to restore to the original double-precision format. Since compression is done in the HCA, there is no data compression on messages between MPI processes mapped to the same node.

4.5.2 Power switching

In order to benefit from the potential energy savings, we used the following policy. When there is no communication, all lanes except one are switched off. If all lanes of a link were shut down, the forwarding tables would have to be updated. By having one lane remain on, management and control traffic will be always available. Also by having one lane always on, we avoid the need for complex adaptive routing schemes. When communication is about to happen, the appropriate number of lanes are powered up, depending on the compression rate. For example, if compression is not being used, then all lanes (four, in our experiments) would be powered up. Alternatively, if 50% compression is applied, half of the lanes (two lanes) are powered up, preserving the original performance. On 4×

links, the supported compression rates are therefore 50% and 25%. On 8× links, the granularity is finer, giving a wider range of possible compression rates. It is expected that the number of lanes will continue to rise. Thus, the idea is that we can have more tuned and optimized support for power savings supported through compression techniques.

To be sure that links are active when needed, we use two new MPI primitives, one to activate the appropriate links and one to deactivate unused links[44]. Each MPI primitive can use WRPS method to tune the links to required width. For turning on/off lanes we take a typical delay of up to 10 microseconds ($\tau = 10\,\mu s$)[35]. By recognizing the communication patterns or group of patterns (regions) in the parallel application, the unnecessary overheads that can appear by power switching can be avoided. This allows changes in link bandwidth (also link power consumption) to be done at coarser granularity than individual communication calls. Therefore, we apply the following policy regarding the MPI primitives for activation and deactivation:

- Activate the link to the required number of lanes before each MPI call and deactivate after the MPI call has finished.
- If the MPI call is part of larger loop where more communication exchanges are done (nearest-neighbour pattern), activate the link before the loop and deactivate when all MPI calls in the loop finish.

4.5.3 Methodology

For quantifying link energy savings same simulation environment and the same input traces are used as in the previous simulation test runs for performance analysis explained in Section 4.4. Here, we assumed that the link bandwidth was always reduced in accordance with the compression rate.

Rather than implementing new code in the simulator to dynamically modify the link bandwidths in proportion to the new message sizes, the same effect was achieved simply by keeping the original MPI

message sizes and interconnect bandwidths. It was only necessary to model the switching overheads by introducing appropriate delays in the traces.

In order to avoid unnecessary overheads that can appear by power switching, it is important to recognize communication patterns or group of patterns (regions) in the parallel application so that changes of links bandwidth (also link power consumption) are coupled with them and not with individual communication calls.

4.5.4 Analysis of Link Energy Savings

In general, previous works on network energy proportionality has focussed on powering down communication links when idle. Here, we also propose a power saving machanism during communication phases which relies on data compression. The idea is to dynamically adjust the link bandwidth, by varying the number of active lanes during communication periods. At the start of each communication, the number of lanes is increased to two, three, or four, depending on the compression rate. In any case, there is a reactivation penalty of 10 µs. During idle periods, one lane remains active.

Figure 4.37 and Figure 4.38 illustrates the behaviour of the Alya CG and Gromacs PME kernels. The

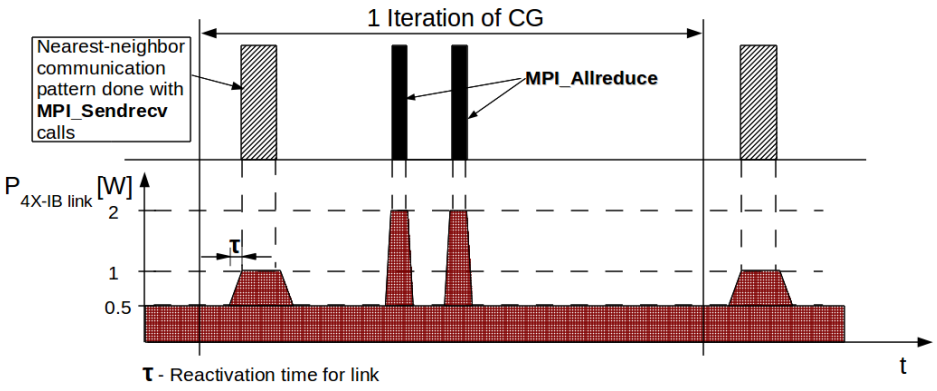


Figure 4.37: Traces showing whether Alya CG kernel is in the application or MPI library (grey or black). The lower traces show 4X-IB link power using the proposed technique

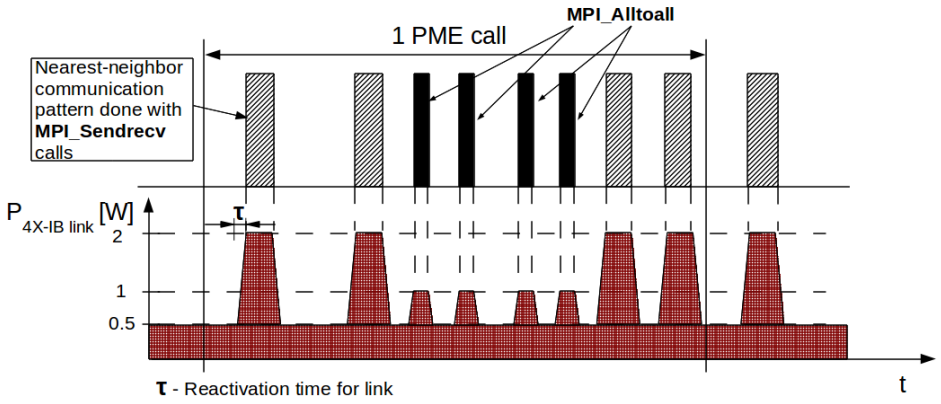


Figure 4.38: Traces showing whether Gromacs PME kernel is in the application or MPI library (grey or black). The lower traces show 4X-IB link power using the proposed technique

upper traces show whether execution is in the application (no bar) or MPI library (grey or black bar). The lower part of each subfigure shows the link power consumption, using the proposed link power reduction technique, applying our policy for link activation and deactivation. We see that link power reduction is possible during computation phases and, if compression is used, also during communication phases. Whenever compression cannot be used; e.g. for the application-driven communication in Gromacs, the links are fully operative, consuming full power.

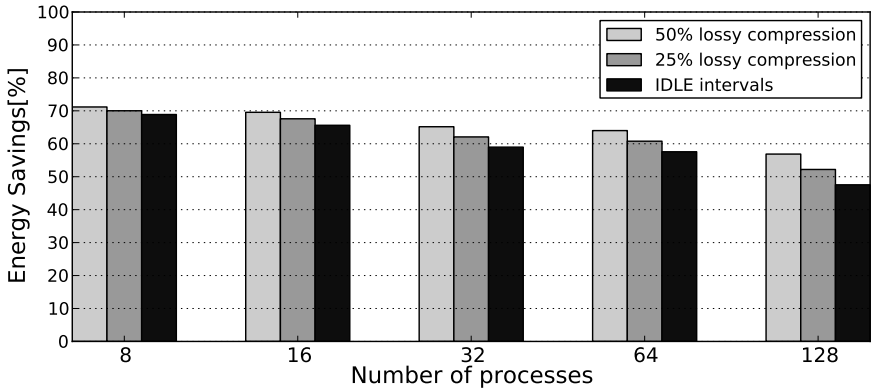


Figure 4.39: IB edge switch link energy savings for Alya CG kernel for one MPI process per node with *Input A*

Figure 4.39 and Figure 4.40 show the energy savings for the Alya CG kernel. The energy reduction

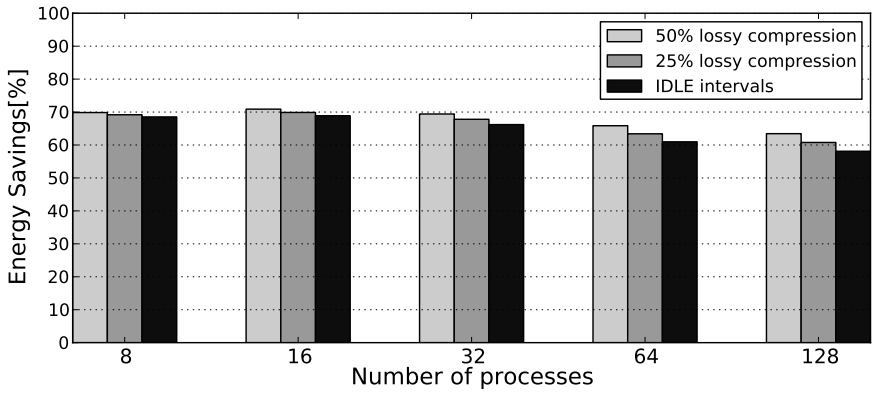


Figure 4.40: IB edge switch link energy savings for Alya CG kernel for one MPI process per node with *Input B*

is about 70% for eight processes, reducing to about 50% for 128 processes. The total energy savings decrease with the number of processes, since, assuming strong scaling, the computation phases become shorter, reducing the lengths of the idle periods. There are, however, greater relative energy savings from data compression. With 50% compression (Input A), the difference in energy savings between the smallest and largest runs is just 10%. Considering only the benefits from idle link periods, this difference increases to about 20%.

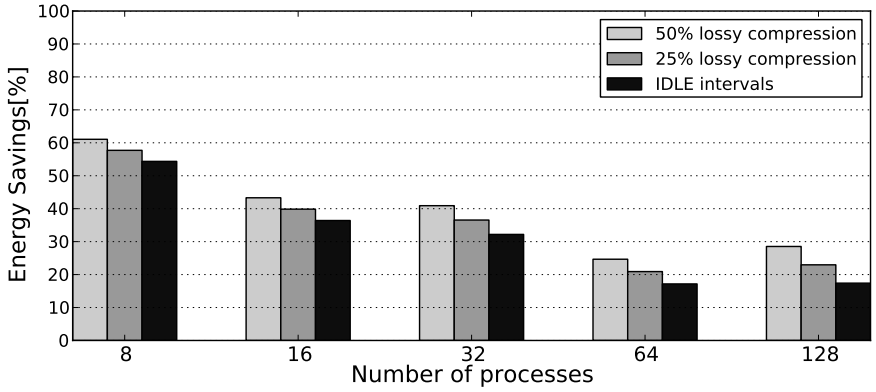


Figure 4.41: IB edge switch link energy savings for Gromacs PME kernel for one MPI process per node with *Input A*

The larger Input B has a lower drop in energy savings from increasing the number of processes. Data

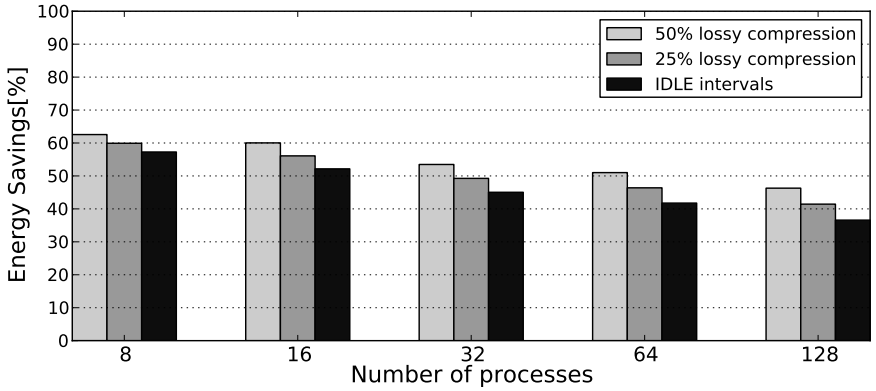


Figure 4.42: IB edge switch link energy savings for Gromacs PME kernel for one MPI process per node with *Input B*

compression will, however, still be important if the number of processes is increased much above 128. The same observation is relevant for Gromacs, but to a lesser extent, as shown in Figure 4.41 and Figure 4.42. The communication phases for which compression can be used make up a smaller part of the overall execution time.

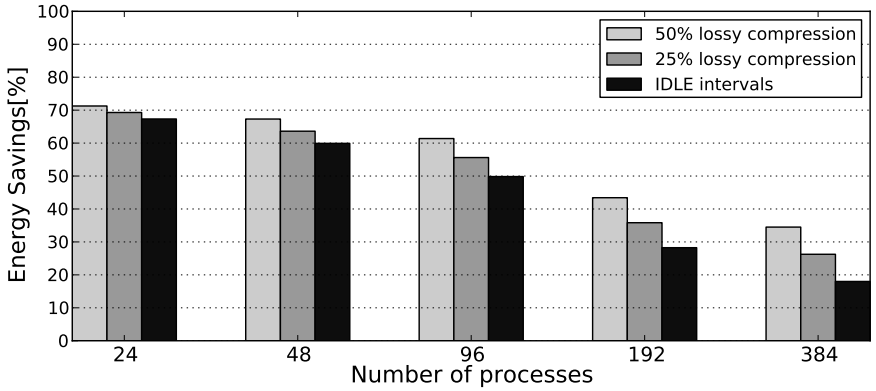


Figure 4.43: IB edge switch link energy savings for Alya CG kernel for six MPI processes per node with *Input A*

If more than one MPI process is allocated to a node, the link will be powered up if any of these processes needs to communicate; i.e. it is powered up when the first process on the node starts to communicate and powered down when the last process finishes communication. This should lead to

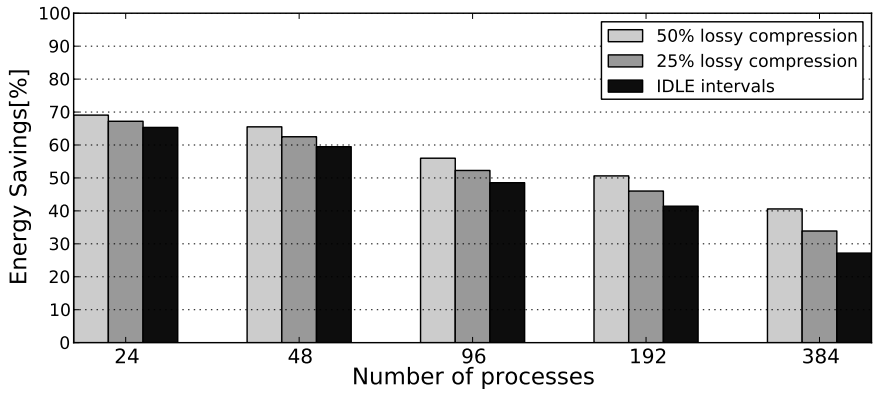


Figure 4.44: IB edge switch link energy savings for Alya CG kernel for six MPI processes per node with *Input B*

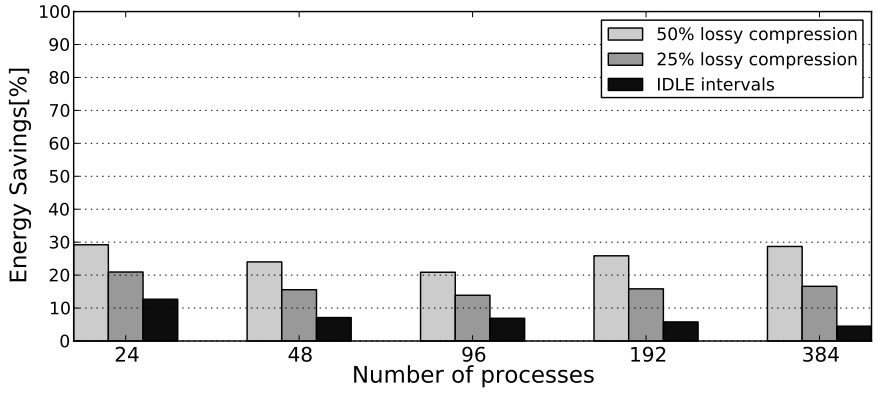


Figure 4.45: IB edge switch link energy savings for Gromacs PME kernel for six MPI processes per node with *Input A*

higher link utilization, decreasing the potential for link energy savings, compared with a single process per node. Figure 4.43 and Figure 4.44 show the results for the Alya CG kernel with six processes per node, which are little different from the results with one process per node. In contrast, Figure 4.45 and Figure 4.46 show the results for the Gromacs PME kernel, with six processes per node. Here, a higher link utilization leads to lower link energy savings. This is clearest for Input A, which is considerably smaller, soon reaching the limits of strong scaling. In this case, the link energy savings from compression, in percentage terms, increase with the number of processes. This is simply because communication phases account for an increasing percentage of the total execution time.

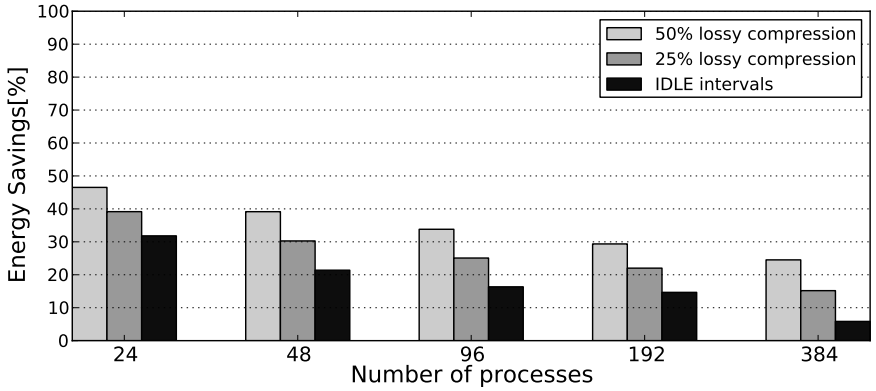


Figure 4.46: IB edge switch link energy savings for Gromacs PME kernel for six MPI processes per node with *Input B*

Finally, we investigate the performance overheads caused by the link deactivation and activation penalties. With one process per node, as shown in Figure 4.47, the maximum increase in execution time was 6%, but it was usually considerably less. With six processes per node, in Figure 4.48, the execution time was increased by less than 0.5%. This is because the reactivation penalties were each time paid by the first process to communicate, which is generally the fastest process, and therefore not on the critical path. The communication on the critical path incurs no penalty, preserving the original performance. The performance loss of Alya CG for Input A is because lossy compression provokes a

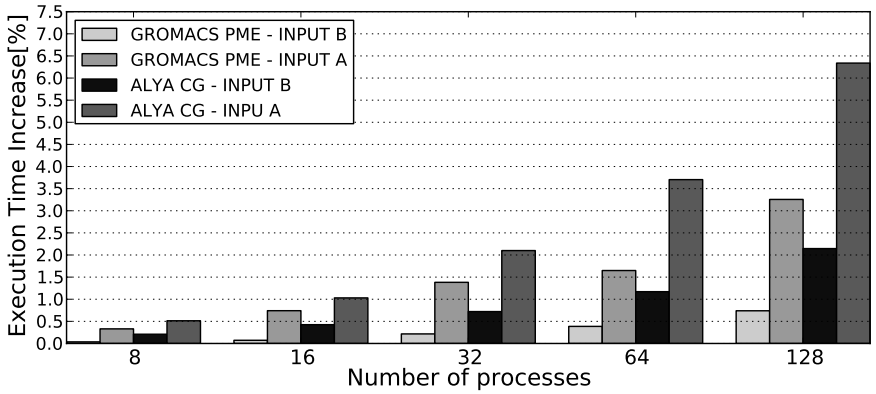


Figure 4.47: Applications kernels execution time increase due to reactivation time penalty and lossy compression for one process per node configuration

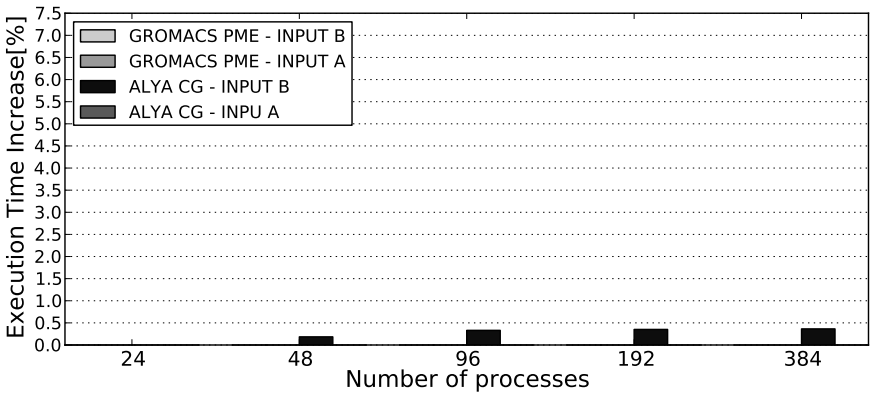


Figure 4.48: Applications kernels execution time increase due to reactivation time penalty and lossy compression for six processes per node configuration

small increase in the number of CG iterations until convergence.

4.6 Conclusions

In this chapter, we evaluated the benefits and trade-offs of using MPI compression techniques in two environments, where one is a real production HPC machine and another is an HPC cluster simulator. For the application kernels evaluated, we showed that the exchanged data does not follow any predictable pattern which leads to bad lossless compression ratios. To overcome this problem, we used lossy compression (reducing up to 50%), verifying that the remaining accuracy still leads to correct results.

Using simulation we obtained similar results as with runs on the HPC production machine. The gains obtained are lower than expected by Amdahl's law for the used compression rates. The blocking nature of point-to-point MPI calls in the nearest-neighbour pattern, where only a single message is outstanding in communication between each pair of processes, does not overload network resources at the HCA. More time is spent on scheduling and synchronization inside the communication pattern than on the actual data transfer. Also, when the size of the messages and the number of neighbour-

hood processes for each process are variable, the total time of communication is also affected. On the other hand, patterns like all-to-all tend to synchronize the tasks, leading to a larger speedup. This communication pattern loads the HCA channel with multiple MPI messages, so a reduction in size improves performance.

On the real machine, for configurations with 4 processes per node executing `MPI_Alltoall` calls from PME kernels, the speedup factor get closer to the expected values according to Amdahl's law. For instance, we get speedups of 12% and 13% for `Input A` and `Input B` respectively while `MPI_Alltoall` calls were occupying 26% and 24% of the total execution time (when 50% lossy compression was applied). The speedup factor also get closer to the expected values for simulation environment where configurations with 6 processes per node are considered. This suggests that contention created on the NIC, which can occur serving all 4 (6 in case of simulation tests) processes at the same time, was dominant and reducing the size of the message was clearly reflected in obtained speedups. As a side effect, we observed that using compression may result in switching from a *Rendezvous* protocol to the more optimal *Eager* message passing protocol, further reducing the execution times. Although speedup factors were not encouraging, correct result obtained suggests that lower bandwidth networks may be implemented, leading to energy and installation cost savings.

To the best of our knowledge, this is the first time that data compression is investigated for link energy savings. Using compression allows the number of active lanes to be reduced in proportion to the compression rate. Thanks to compression, even with reduced network bandwidth, the application performance is not affected. Reactivation delays typically increased execution time by just a few percent. Using 50% compression, we obtained in the lowest (edge-level) network links energy savings of up to 71% for the Alya CG kernel and 63% for Gromacs PME. We also show that strong scaling runs, in particular, have a large benefit from data compression.

Chapter 5

Runtime Software-Managed Power Savings in IB Links

This chapter describes and evaluates our approach to reduce network energy consumption, by switching links off during long idle periods, and using prediction to avoid the wake-up latency. The Static Pattern Prediction System, presented in Section 5.2, introduces the basic prediction techniques and methods for link power management. The Self-Tuned Pattern Prediction System, presented in Section 5.3, automatically configures itself, in order to build a fully self-contained algorithm.

5.1 Motivation

As discussed above, HPC applications typically follow the bulk synchronous programming model, in which network traffic is concentrated into distinct communication phases. It is reasonable to expect that, since the network links are idle during computation phases, there is an automatic opportunity to enter power-saving mode. It is, however, important to take account of the overhead in changing

Table 5.1: Distribution of link idle intervals with strong scaling

	N° proc	$T_{idle} < 20\,\mu s$			$20\,\mu s < T_{idle} < 200\,\mu s$			$T_{idle} > 200\,\mu s$		
		N° intervals	Intervals [%]	Time [%]	N° intervals	Intervals [%]	Time [%]	N° intervals	Intervals [%]	Time [%]
GROMACS	8	3277	58.56	0.001	6	0.11	0.009	2313	41.33	99.99
	16	3595	54.98	0.002	606	9.27	0.078	2338	35.75	99.92
	32	5052	53.72	0.006	1523	16.2	0.304	2829	30.08	99.62
	64	5046	53.68	0.011	2228	23.7	0.779	2126	22.62	99.21
	128	9067	68.5	0.11	2276	17.2	1.01	1893	14.3	98.88
ALYA	8	771	22.57	0.024	82	2.4	0.006	2563	75.03	99.97
	16	1744	34.08	0.013	811	15.85	0.077	2563	50.08	99.91
	32	3642	58.52	0.07	818	13.14	0.99	1763	28.33	98.94
	64	6754	71.97	0.27	827	9.29	0.9	1758	18.73	98.83
	128	8497	76.72	0.4	1644	14.84	5.05	934	8.43	94.55
WRF	8	209357	94.31	0.05	2201	0.99	0.14	10419	4.69	99.81
	16	209423	94.34	0.11	2051	0.92	0.26	10503	4.73	99.63
	32	209414	94.34	0.3	4014	1.81	0.73	8549	3.85	98.97
	64	209284	94.28	1.07	5050	2.28	1.48	7643	3.44	97.45
	128	209442	94.36	1	6697	3.02	0.51	5833	2.63	98.49
NAS BT	9	9664	78.63	0.009	9	0.07	0.001	2618	21.3	99.99
	16	13286	77.63	0.022	5	0.03	0.008	3824	22.34	99.97
	36	20522	76.68	0.031	5	0.02	0.009	6236	23.3	99.96
	64	27750	76.21	0.094	13	0.04	0.006	8648	23.75	99.9
	100	34996	75.98	0.13	161	0.35	0.22	10902	23.67	99.65
NAS MG	8	5468	54.66	0.095	3794	37.92	3.055	742	7.42	96.85
	16	5119	54.85	0.18	3729	39.96	5.87	484	5.19	93.95
	32	5503	58.7	0.46	3600	38.4	11.38	271	2.89	88.16
	64	5775	60.79	0.97	3458	36.4	8.37	267	2.81	90.66
	128	7082	84.65	7.04	1123	13.42	6.71	161	1.92	86.25

power mode, which is approximately 10 µs[35]. There can be no energy savings from idle periods that are shorter than the total time to turn the link off and then back on again. A significant energy saving is only possible if the idle period is much longer than this overhead. For simplicity in exposition, we assume that the time to turn the link off is the same as the time to turn it back on again, and therefore denote both by T_{react}. In summary, energy savings are only possible for idle periods with $T_{idle} > 2 \times T_{react}$.

We evaluated the potential for link power reduction by analysing traces of typical HPC applications (Gromacs[7], Alya[3], WRF[47] and two NAS Parallel Benchmarks[13]) running on the MinoTauro production machine (described in Section 3.1). We configured the applications to use one MPI process per processor. We used strong scaling, in which the same workload was used irrespective of the number

of processors.

The results are shown in Table 5.1. We see that, for almost all applications, 99% of the link idle time is inside idle intervals that are longer than 20 μs, which is twice the typical value of T_{react}. Even more importantly, in the majority of cases, more than 90% of the total link idle time is in longer idle intervals of duration $T_{\text{idle}} > 200\,\mu\text{s}$, where significant power can be saved. Only the NAS MG benchmark when running with a large number of processes, has a figure lower than 90%. Since the goal is a reduction in operational costs over the lifetime of the supercomputer, the important consideration is average potential energy savings over all applications. All the results in this thesis are for the typical and more challenging case of strong scaling. Better results are expected for weak scaling. Nevertheless, although, for strong scaling, the number of short intervals ($T_{\text{idle}} < 20\,\mu\text{s}$) rises with the number of MPI processes, short intervals still contribute a small proportion of the total idle time. Since long idle intervals account for most of the idle time, reducing link power only during the long idle intervals is sufficient to obtain most of the potential energy savings, resulting in close to energy proportionality.

While deactivating IB lanes can be overlapped with computation, reactivation may incur a latency in subsequent communication. In an ideal case, the IB link lanes would be turned on in time to avoid a latency penalty on the next message. We solve this problem by providing the necessary knowledge using a prediction algorithm.

5.2 Pattern Prediction System

5.2.1 Design

This section describes our energy-saving mechanism, which reduces link power consumption during idle periods, with negligible impact on execution time. Figure 5.1 is a high-level view of our proposal, which consists of two parts. The first part, the Pattern Prediction Component (PPC), is invoked be-

fore every MPI event. It contains the Pattern Prediction Algorithm (PPA), described in detail below, which builds a table, known as the *pattern list*. This table contains repeatable MPI communication patterns. It also generates an output flag, *patternPrediction*, which is true whenever PPA has determined that the program is following a known repeatable pattern. If *patternPrediction* is false, then no prediction is active, meaning that the link remains in full-power mode.

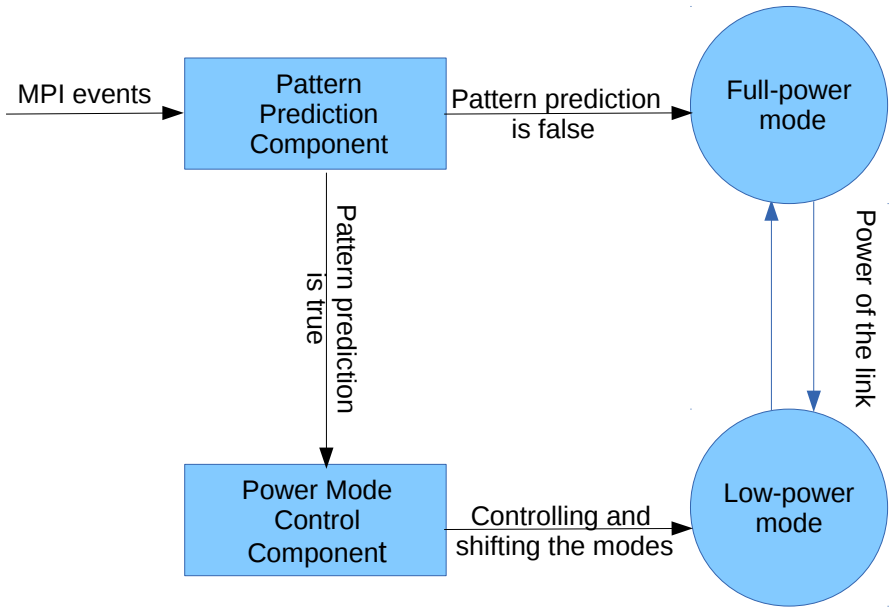


Figure 5.1: Simplified diagram of MPI process pattern prediction system that reduces power consumption in interconnection links

When *patternPrediction* is true, however, control of the link's power modes is transferred to the second part, the Power Mode Control Component (PMCC). Whenever this component is active, it is invoked *after* every MPI event. It compares the actual MPI events with those expected from the pattern. So long as they continue to match, the length of the next idle interval can be read from the pattern. At the start of expected long idle intervals, the link is put into low-power mode for the appropriate amount of time. As long as the program continues to follow the pattern, there is no need to invoke PPA, since the pattern is already known. It is only necessary to continue updating the idle

intervals with recent values, allowing some adaptation to varying application characteristics. If the current MPI event does not match the pattern, however, PMCC sets *patternPrediction* to false. In that case, PPA is reactivated and the link is kept in full-power mode until the next repeatable pattern.

5.2.2 Pattern Prediction Component

The algorithm uses the concept of *n*-grams, which is extensively used in the area of natural language processing. The *n*-gram extraction approach has been used to efficiently detect DNA patterns[64] and patterns in musical notes[50]. An *n*-gram is defined to be a subsequence of *n* items in a sequence. In our case, the sequence of items, known as grams, is derived from the MPI events in the program's execution. Each *gram* is one or more consecutive MPI events that are separated only by short idle intervals, whereas the idle intervals between different grams are long. An *n*-gram is a sequence of *n* consecutive grams. Note that PPA works on the MPI events in a single process. Although it is outside the scope of this thesis, if there are multiple MPI processes per node, prediction should be done inside each MPI process separately, with their outputs combined using a single PMCC per node.

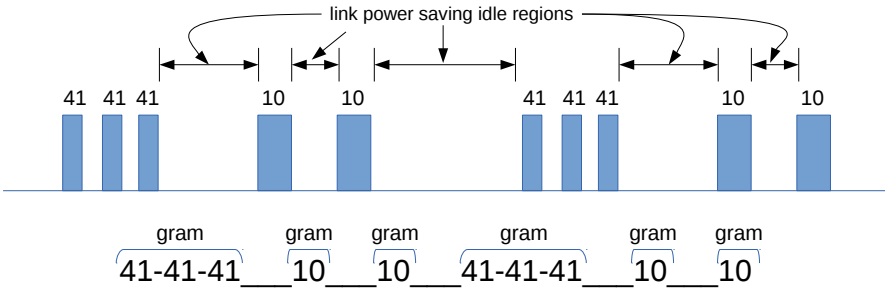


Figure 5.2: Forming the array of grams from the MPI event stream (Alya). Event IDs are 41 for *MPI_Sendrecv* and 10 for *MPI_Allreduce*

Before the PPA algorithm is invoked, the grams need to be formed. Algorithm 1 performs the grouping of MPI events into grams, based on the idle time interval between adjacent MPI events. Two consecutive MPI events are considered to be part of the same gram whenever the idle time sep-

arating them is less than a threshold known as the *grouping threshold* (GT). The intention is that the link enters low-power mode between grams but not inside them, so this grouping threshold should be larger than the critical value of $2 \times T_{\text{react}}$ discussed in Section 4.1.

Algorithm 1 Forming Array of Grams

Input: Current MPI event, *eventType*, and length of preceding idle interval, *previousIdleTime*.
Output: Predicted pattern of MPI events, *predictedPattern*, and the current partial gram, *currentGram*, required by Algorithm 4.

```
Global: pos : Integer(0) ; patSize : Integer(2) ; posNext = patSize
1: if previousIdleTime > groupingThreshold then
                                        ▷ Insert completed gram into array
2:     array[pos] = (currentGram, previousIdleTime)
3:     pos = pos + 1
4:     clear currentGram
5: append eventType to currentGram
6: if patternPrediction is false then
7:     if pos ≥ (posNext + patSize) then
8:         call PPA()
```

The input to the algorithm is the current MPI event type, *eventType*, and the length of the idle time preceding it, *previousIdleTime*. The output of the algorithm is the predicted pattern of MPI events, *predictedPattern*, and the current partial gram, *currentGram*, required by PMCC.

Here, an array, *array*, of tuples is created. Each tuple in this array holds the list of MPI events in the current gram, as well as the length of the idle interval that *follows* the gram. Note that the current gram can only be inserted into *array* when this latter length is known; i.e. after the idle interval following it, which is on the first MPI event of the next gram.

Figure 5.2 illustrates the effect of Algorithm 1. Each set of three consecutive *MPI_Sendrecv* calls is grouped together to form a single gram, while each *MPI_Allreduce* call is isolated as a separate gram. These grams will be used as building blocks to construct the repeatable communication patterns. The building of patterns is done by the PPA algorithm, which is invoked on line 9, only when there is no currently repeating pattern and a sufficient number of grams has been seen.

5.2.3 Pattern Prediction Algorithm

A *repeating pattern* is a sequence of grams that has been observed to occur at least twice consecutively. We established the following policy to discover these repeating patterns and accurately predict their continuation:

- After observing three consecutive occurrences of the same pattern, it is predicted to continue to repeat for a long time, meaning that the Power Mode Control Component is activated.
- On misprediction, the Power Mode Control Component is deactivated. However, observing the pattern once more causes it to be detected, meaning that the Power Mode Control Component is reactivated.

This policy is implemented by Algorithm 2, the Pattern Prediction Algorithm (PPA). It is based on an algorithm proposed by Alawneh for the detection of process patterns[16]. We modified the algorithm to adapt it to detect continuous repetitions of patterns in program execution and the prediction of pattern appearance based on previous appearances.

The input to the PPA algorithm is the array of tuples, *array*, from Algorithm 1. Each tuple in the array corresponds to a completed gram, holding the list of MPI events inside it, as well as the length of the idle interval that follows it. The PPA algorithm builds a *uthash*[33] hash table, known as the *pattern list*, with key the pattern sequence (list of grams) and value a tuple giving the pattern's length, its positions in the array, its frequency, the list of idle intervals between grams and the total number of MPI calls in the sequence. In addition, there are two indices into *array*, *posCur*, initially zero, which points to the current pattern, and *posNext*, initially equal to *patSize*, which starts with the value two.

The PPA algorithm is best understood using an example. Figure 5.3 illustrates the execution of the algorithm for the Alya workload. At the top, in Figure 5.3(a), is the list of MPI events grouped into grams; it is an extension of the example in Figure 5.2. Next, in subfigure (b), is shown the progress of

Algorithm 2 Pattern Prediction Algorithm (PPA): Algorithm runs for each MPI process separately identifying consecutive repeating patterns based on which the prediction is done

```
1: if newPattern is false and checkConsec is false
2:     and patSize < maxPatternSize then
                                                    ▷ Grow the pattern by one gram
3:     checkPrevious = true
4:     match = false
5:     nextPattern = currPattern + array[posCur + patSize]
6:     patSize = patSize + 1
7:     newPattern = updatePL(nextPattern, curPos)
8:     if newPattern is false then
9:         if PL[nextPattern].wasUsed is true then
                                                    ▷ Reactivate previously used pattern
10:            predictedPattern = nextPattern
11:            patternPrediction = true
12:            return
13:    Check all previous positions listed in PL[currPattern]
14:    if All can be extended to match nextPattern then
15:        match = true
16:    checkConsec = true
17:    currPattern = nextPattern
18: if checkConsec is true then
                                    ▷ Check whether the pattern is repeated at posNext
19:    if array[posNext : posNext + patSize] Equals currPattern then
20:        consecutiveRepeats + = 1
21:        UpdatePL(currPattern, posNext)
22:        match = true
23:        if consecutiveRepeats Equals 2 then
24:            maxPatternSize = patSize
25:            PL[currPattern].wasUsed = true
26:            predictedPattern = currPattern
27:            patternPrediction = true
28:        posCur = posCur + patSize
29:        posNext = posNext + patSize
30:    else
31:        checkConsec = false
32:        if newPattern is true and match is false then
33:            posCur = posCur + (patSize − 1)
34:            posNext = posNext + (patSize − 1)
35:    if match is false and checkPrevious is true then
36:        remove nextPattern from PL
37:        patSize = 2
38:        checkPrevious = false
39: else                                            ▷ Insert current gram into pattern list
40:    currPattern = array[posCur : posCur + patSize]
41:    newPattern = updatePL(currPattern, posCur)
42:    checkConsec = true
```

the algorithm, with each row corresponding to an MPI event. For simplicity, the lengths of the idle intervals have been omitted from *array*. At the bottom, in Figure 5.3(c) is shown the insertions into the pattern list.

Algorithm 3 function *updatePL*

Input: current pattern, *currPattern* and current position, *posCur*
Output: Return true if *curPattern* is a new pattern, else if already exists in PL returns false

```
1: if currPattern in PL then
2:     PL[currPattern].frequency += 1
3:     append posCur to PL[currPattern].position
4:     return false
5: else
6:     add to PL[currPattern, posCur]
7:     return true
```

We now follow the progress of the algorithm in Figure 5.3(b). The PPA algorithm will not be executed until there are sufficient completed grams in *array* (line 9 of Algorithm 1). Since the initial values of *patSize* and *posNext* are both two, the number of formed grams becomes large enough only on the ninth MPI call (line 9 in the PPA execution in Figure 5.3(b)). At this point, since *newPattern* is true and *checkConsec* is false, the only action is to insert the current gram into the pattern list (lines 48 to 50). The first bi-gram, `41-41-41_10`, is therefore read from *array*, and added to the pattern list (lines 48 and 49). This insertion is shown in Figure 5.3(c). The return value from *updatePL* (described in more detail in Algorithm 3) indicates whether this is the first insertion of that particular pattern sequence. It is, so *newPattern* is true.

On the next MPI event, *newPattern* and *checkConsec* are both true, so the first action is to check whether there are two consecutive identical patterns in the array (line 23). The comparison is between the bi-grams `41-41-41_10` and `10_41-41-41` at the beginning of the array. These do not match, so control passes to lines 36 to 40, where *checkConsec* becomes false, and both *posCur* and *posNext* are shifted one position. On the 11th MPI call, the second bi-gram `10_10` is added to the pattern list, in a similar manner to the first. On the 13th MPI call, the third bi-gram is added.

On the 15th MPI event, the `41-41-41_10` bi-gram is encountered for a second time. Since it was already present in the pattern list, *newPattern* is set to false (line 49). Inside *updatePL*, the frequency count, shown in the third column in the insertions list in Figure 5.3, is increased to two and the list of

positions is extended to be [0, 3]. Next, on the 16th MPI event, *newPattern* and *checkConsec* are both true, but, as before, there is no consecutive repeat of the bi-gram 41-41-41_10. Therefore, *checkConsec* is set to false, but, as now *newPattern* is set to false, it is necessary first to check whether the enlarged pattern can detects its repetitions, before we shift both indices by the $patSize - 1$.

On the 17th MPI event, *newPattern* is still false, and *checkConsec* is now false, since the sequence of grams, 41-41-41_10 has been seen twice, but they are not consecutive. For this reason, the algorithm increases the size of this pattern by one gram (lines 3 to 14); in this case, to the tri-gram 41-41-41_10_10. If this pattern had previously been used for prediction, then it would be immediately reactivated (lines 9 and 11), according to the second statement in the policy at the beginning of this section. This is not the case, so instead, line 15 checks whether all previous occurrences of the bi-gram 41-41-41_10 can be extended to the new tri-gram. If the newly constructed tri-gram cannot be detected at any previous position of its prefix bi-gram and there's no consecutive repeats, than it will be removed from the pattern list (line 43) and the size of a *n*-gram will be set to the minimal value, 2 (bi-gram). Here, it is not the case, so *match* is set to true.

Eventually, on the 17th call, the first consecutive repetition of the tri-gram 41-41-41_10_10 is found. At this point, *consecutiveRepeats* is incremented to 1, and both *posCur* and *posNext* are advanced by the pattern size. When PPA is next invoked on the 21st MPI event, the second consecutive repeat is seen. The pattern is assigned to *predictedPattern* and *patternPrediction* is set to true (lines 28 to 31), since the PPA algorithm has successfully found the repeating pattern.

In order to recognize the natural (real) iteration in the application and predict each iteration based on the behaviour of the previous one, we must avoid merging multiple application iterations into a single pattern. This is done by setting the maximum pattern size to be the length of the current pattern (line 28). If this were not done, and increasing numbers of application iterations were combined into a single pattern, prediction accuracy would suffer, since idle intervals would be predicted based on older values from many iterations previously. The pattern size can therefore vary from the smallest bi-gram

(a) Array of grams of MPI events formed during an MPI process:

0	1	2	3	4	5	6	7	8
41-41-41	10	10	41-41-41	10	10	41-41-41	10	10
41-41-41	10	10	41-41-41	10	10	41-41-41	10	10
9	10	11	12	13	14	15	16	17

(b) PPA execution:

#	MPI ID	Array of grams	Current Gram	Action on MPI event	Pattern prediction
1	41		41	Not enough grams	false
2	41		41-41	Not enough grams	false
3	41		41-41-41	Not enough grams	false
4	10	[41-41-41]	10	Not enough grams	false
5	10	[41-41-41,10]	10	Not enough grams	false
6	41	[41-41-41,10,10]	41	Not enough grams	false
7	41	[41-41-41,10,10]	41-41	Not enough grams	false
8	41	[41-41-41,10,10]	41-41-41	Not enough grams	false
9	10	[41-41-41,10,10,41-41-41]	10	Add pattern to PL	false
10	10	[41-41-41,10,10,41-41-41,10]	10	Check consecutive-no	false
11	41	[41-41-41,10,10,41-41-41,10,10]	41	Add next pattern to PL	false
12	41	[41-41-41,10,10,41-41-41,10,10]	41-41	Check consecutive-no	false
13	41	[41-41-41,10,10,41-41-41,10,10,]	41-41	Add next pattern to PL	false
14	10	[41-41-41,10,10,41-41-41,10,10,41-41 -41]	10	Check consecutive-no	false
15	10	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10]	10	Add next pattern to PL- match detected	false
16	41	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10]	41	Check consecutive-no	false
17	41	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10,]	41-41	Add gram Consecutive-yes	false
18	41	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10]	41-41-41	Not enough grams	false
19	10	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10, 41-41-41]	10	Not enough grams	false
20	10	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10, 41-41-41,10]	10	Not enough grams	false
21	41	[41-41-41,10,10,41-41-41,10,10,41-41 -41,10,10, 41-41-41,10,10]	41	Check consecutve-yes	true

(c) Insertions into Pattern List:

#	pattern	frequency	idle intervals	position	pattern size	Nº MPI calls
9	41-41-41_10	1	t_1,t_2	0	2	4
11	10_10	1	t_1,t_2	1	2	2
13	10_41-41-41	1	t_1,t_2	2	2	4
15	41-41-41_10	2	t_1,t_2	0, 3	2	4
17	41-41-41_10_10	1	t_1,t_2,t_3	3	3	5
17	41-41-41_10_10	2	t_1,t_2,t_3	3, 6	3	5
21	41-41-41_10_10	3	t_1,t_2,t_3	3, 6, 9	3	5

(d) Prediction possible:

predicted pattern	idle intervals	from position
41-41-41_10_10	[t_1,t_2,t_3]	12

Figure 5.3: Example execution of the PPA algorithm for Alya workload

to the size defined by *maxPatternSize* value.

Algorithm 4 Power Mode Control Component

Input: Predicted pattern, *predictedPattern* and the current (partial) gram, *currentGram*

```
 1: index : Integer(0)
 2: if patternPrediction is true then
 3:     predictedPatternGram = predictedPattern[index]
 4:     idleTimeArray = PL[predictedPattern].idleTime
 5:     if len(currentGram) Equals len(predictedPatternGram) then
 6:         if currentGram Equals predictedPatternGram then
 7:             idleTime = idleTimeArray[index]
 8:             safetyLimit = idleTime × displacementF + T_react
 9:             predictIdleTime = idleTime − safetyLimit
10:             WRPS_method(predictedIdleTime)
11:             index = (index + 1) mod len(predictedPattern)
12:         else
13:             patternPrediction = false
14:             index = 0
```

5.2.4 Power Mode Control Component

The Power Mode Control Component is responsible for switching between link power modes, according to the current repeatable pattern. The algorithm is presented as Algorithm 4, which is executed only when *patternPrediction* flag is true. The first input to the algorithm is the predicted pattern from Algorithm 2. This pattern is described by two arrays. The first array is the sequence of grams, *predictedPattern*, and the second array is the sequence of idle time intervals following those grams, *idleTimeArray*. The other input to the algorithm is the current gram being built by Algorithm 1.

Algorithm 4 works with the partial gram, and considers it to be complete when it has the correct length (line 5). If, in addition, the MPI events in the actual gram match the prediction (line 6), then the predicted length of the upcoming idle period, *idleTime*, is read from the array. It is modified by the *displacement factor*, as described below, and the T_{react}, obtaining the final prediction, *predictIdleTime*. The resulting value can be passed as the argument to *WRPS_method*, giving the time to remain in low-power mode. If, on the other hand, the actual gram does not match the prediction, then the current pattern has finished, and PPA is reactivated by setting the *patternPrediction* flag to *false*.

The displacement factor, mentioned above, is a safety factor, used to take account of variability in

the link idle intervals. To reduce the likelihood that the link is not turned on too late, the predicted idle time is reduced using the displacement factor (line 8 of Algorithm 4). It is a value between 0 and 1, where 0 means that the predicted idle time is not reduced, and 1 means that it is reduced all the way to zero. For simplicity in presentation, the displacement factor is expressed as a percentage (so a displacement factor of 5% is equivalent to a value of 0.05 in the algorithm).

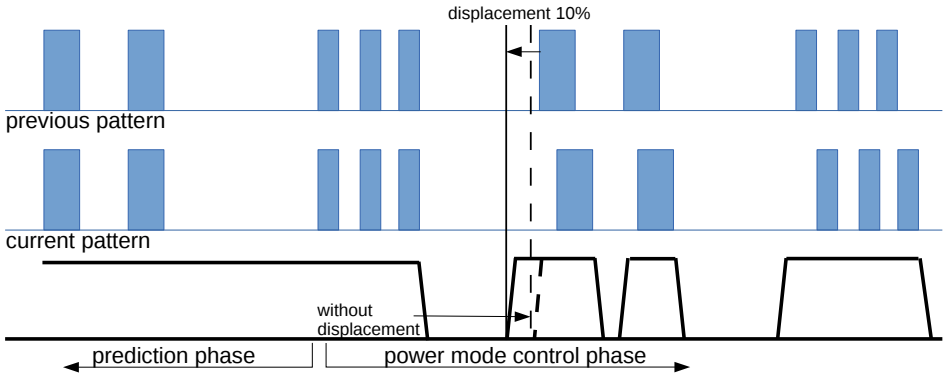


Figure 5.4: Controlling IB link power mode during execution of Alya, with *displacement factor* of 10%. Real idle interval turned out to be larger than expected.

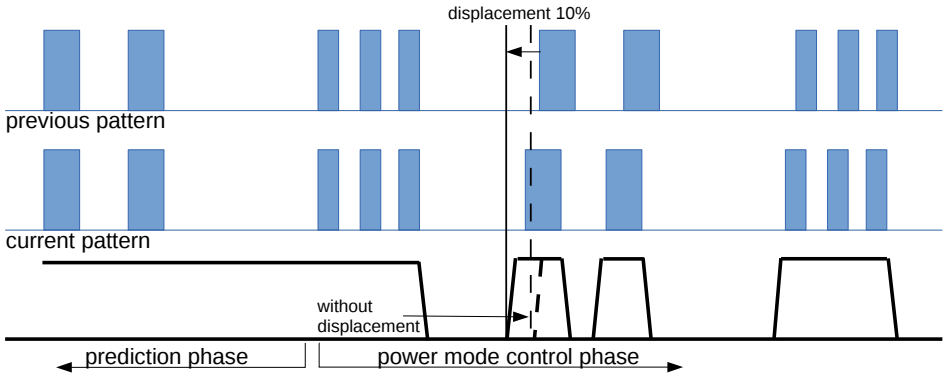


Figure 5.5: Controlling IB link power mode during execution of Alya, with *displacement factor* of 10%. Real idle interval turned out to be shorter than expected.

The function of the displacement factor is illustrated in Figure 5.5 and Figure 5.4. Figure 5.4 is the case when the current pattern has an idle interval slightly larger than predicted. In this case, a

displacement factor of 10% reduces the energy savings by slightly more than 10%, compared with optimal. Figure 5.5 is the case when the current pattern has an idle interval shorter than predicted. In this case, the displacement factor of 10% has avoided the latency penalty that would have been incurred by switching on the link too late. In general, in the context of HPC, it is better to reduce the energy savings than risk a noticeable degradation in performance. Varying the displacement factor exposes a trade-off between the two.

5.2.5 Hardware Support

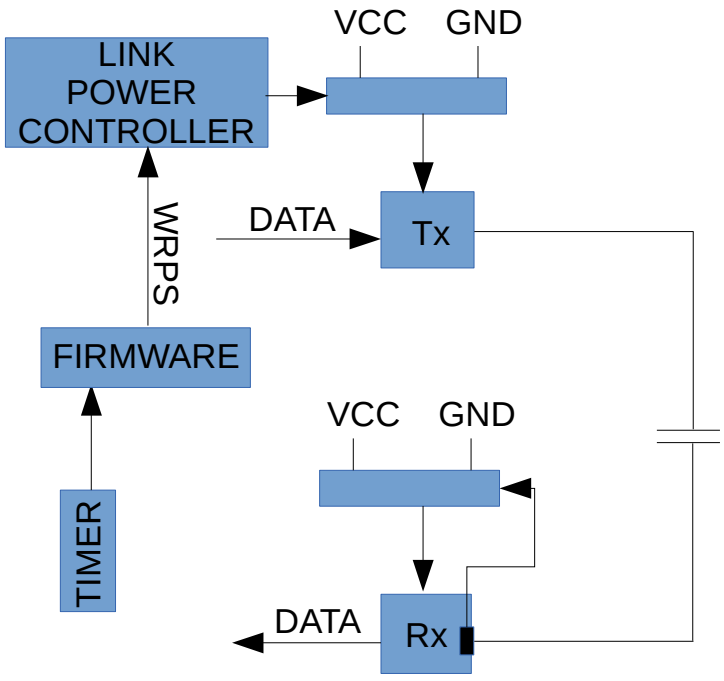


Figure 5.6: Link Block diagram

Figure 5.6 shows the hardware support that is required for IB link power management. A special command is required, which enables user code to request that the link enters low-power mode once any ongoing communication has completed. In order to avoid interrupting the CPU when it is time to

wake up, we propose adding one hardware timer associated with the link. This timer is programmed using the predicted idle time. After the programmed delay elapses the timer will generate an interrupt to the firmware, which will reactivate the lanes. Communication between PMCC and the hardware is unidirectional, meaning that there is no feedback to the system on the correctness of prediction.

5.2.6 Experimental Evaluation

Methodology

The link-level power management described in Section 2.6 is still under development by the InfiniBand switch and NIC vendor, so it is not yet supported in the devices' firmware. We therefore evaluate our prediction-based mechanism and its impact on performance and energy savings using a simulation environment. We decided to use the Venus–Dimemas[49,42] simulator (both are explained in detail in Section 3)

Dimemas was fed with five representative HPC applications traces obtained on a MinoTauro machine (described in Section 3.1). The applications were configured with one MPI process per node and strong scaling (i.e. a fixed workload). The parameters of the simulated system are given in Table 5.2.

We first ran the simulations without modifying the traces, in order to check that the original execution times were reproduced. Next, we apply PPA to the traces, inserting new events that mark when prediction is possible and events that mark when links are in low-power mode. When mispredictions happen delays due to reactivation of a lanes are inserted in the traces. All other overheads associated with the power saving mechanism are inserted, including the time to execute the PPA algorithm, as well as the overheads of data collection. Finally, we simulate the new traces on Venus–Dimemas, in order to quantify the resulting performance and energy savings.

Using the Paraver tool[14], we measure the total amount of time for which the IB links are fully active, as well as the time that the links are in low-power mode. Figure 5.7 shows a trace from Paraver. The

Table 5.2: Parameters used in Simulations

Parameter	Value
Simulator	Dimemas-Venus
Connectivity	XGFT(2;18,14;1,18)
Topology	Extended Generalized Fat Trees (two levels of switches)
Switch technology	Infiniband
Network Bandwidth	40 Gbit/s
Segment Size	2 kB
MPI latency	1 µs
CPU Speedup	1
Routing scheme	Random routing
Switch power consumption	43% when in low-power mode[3]

dark blue regions represent durations during which the IB links are in low-power state, and bright blue regions show when IB links are in full-power state. Energy savings are somewhat different for the various MPI processes. The times used for evaluation are averaged over all MPI processes.

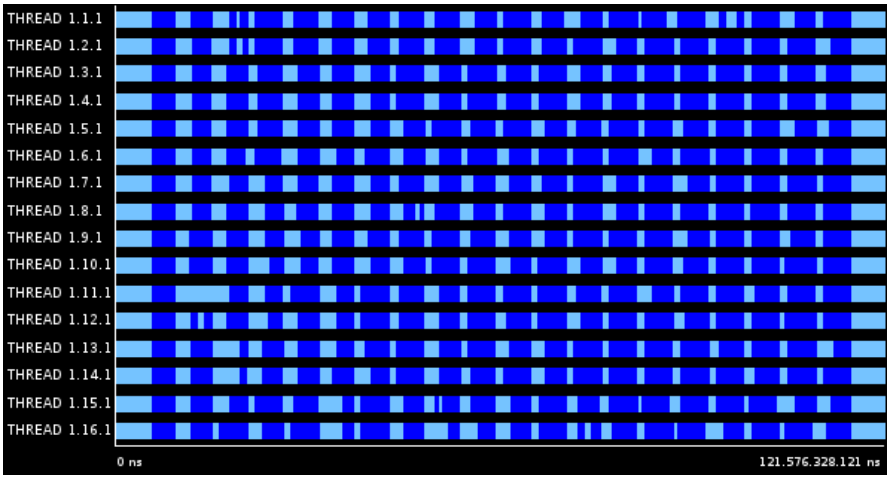


Figure 5.7: Execution trace of the Gromacs application with 16 MPI processes, showing when IB links enter low-power mode

Results

This section presents and analyzes the experimental results, in terms of execution time and energy savings. For all benchmarks except NAS BT, we show results for runs with 8, 16, 32, 64 and 128 MPI processes. Since NAS BT requires the number of processors to be square, we instead run it with 9, 16, 36, 64, and 100 MPI processes.

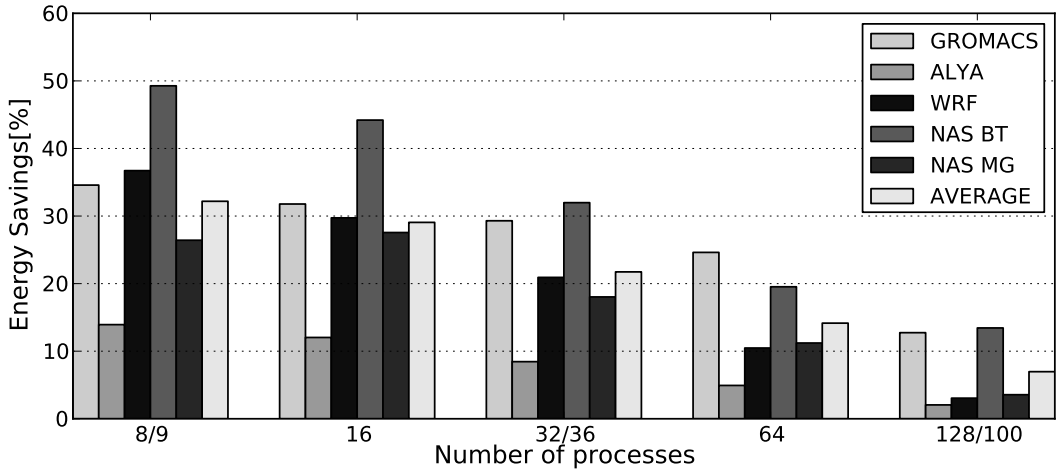


Figure 5.8: Energy savings in IB edge switch links - strong scaling results with medium *displacement factor*

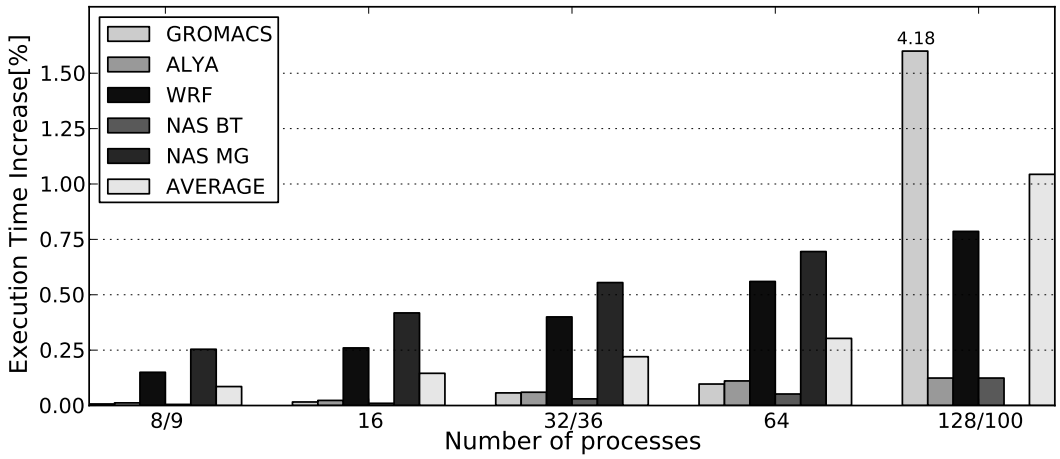


Figure 5.9: Applications execution time increase - strong scaling results with medium *displacement factor*

Figure 5.8 and Figure 5.9 show the energy savings and performance impact respectively for a medium value of the *displacement factor* equal to 5%. Since we used strong scaling workloads, the amount of communication relative to computation increases with the number of nodes, inevitably reducing the opportunities for energy savings. We expect this problem to not occur with weak scaling. For the same reason, larger scale runs suffer from a larger increase in execution time, but still the maximum average increase, across applications, is around 1%. Due to larger inter-process communication the delays introduced in the system coming from our power saving mechanism can accumulate between processes. Depending on the communication pattern during execution, this could bring the agglomeration of delays and create a total delay in the entire application that is much larger than a single local delay on one MPI process. This can be seen for the Gromacs application, where in a run with 128 processes, we see more than 4% increase in execution time.

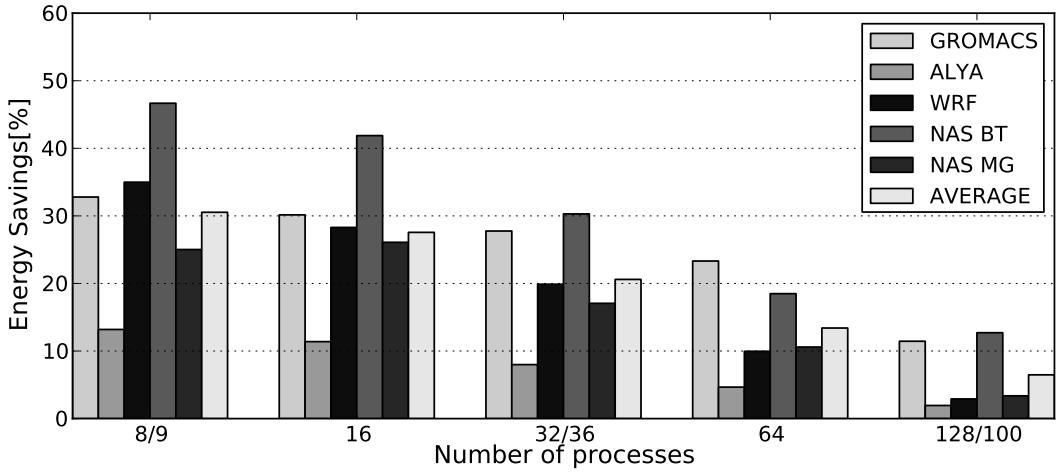


Figure 5.10: Energy savings in IB edge switch links - results when large *displacement factor* of 10% is employed

Figures 5.10 and 5.11 and Figures 5.12 and 5.13 explore the trade-off in varying the *displacement factor*. Choosing a larger *displacement factor* reduces the overheads incurred by waking the link up too late, at the cost of reduced time in the low-power mode. The results for a large displacement of 10%, in Figure 5.10 and Figure 5.11 show that the average energy reduction is lower at 30.6%, with an almost

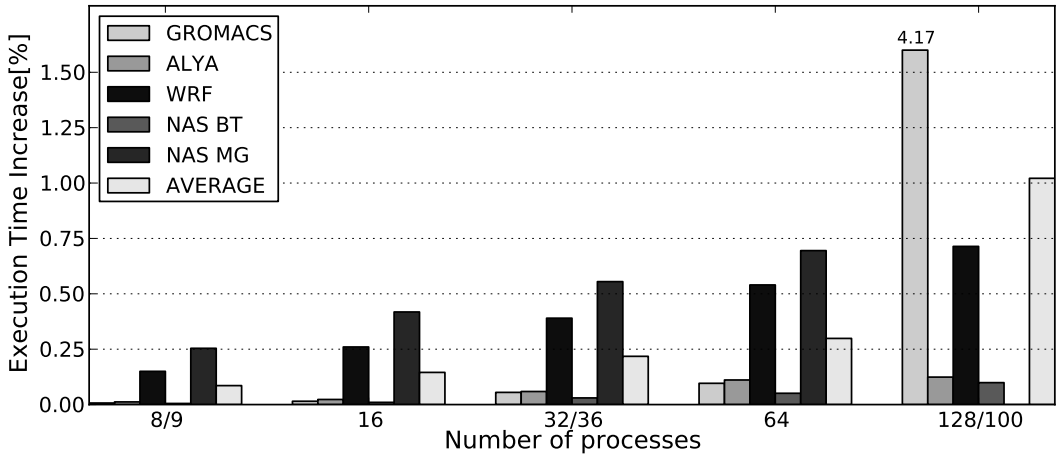


Figure 5.11: Applications execution time increase - results when large *displacement factor* of 10% is employed

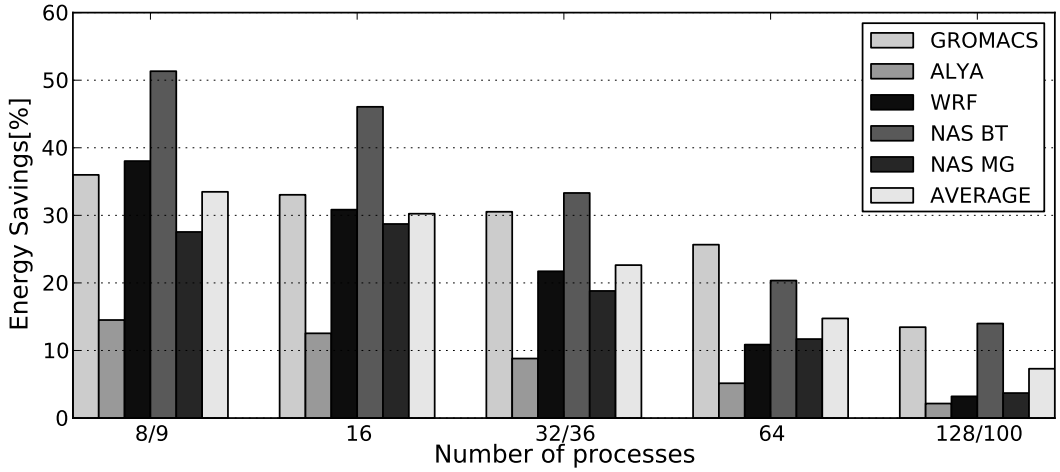


Figure 5.12: Energy savings in IB edge switch links - results when small *displacement factor* of 1% is employed

negligible increase in execution time, compared with the original. Using the smaller displacement factor of 1%, in constrast, shown in Figure 5.12 and Figure 5.13 gives the largest average energy savings of 33.5%, at the cost of potentially larger impact on execution time.

The energy consumption of the interconnection network can be reduced further if other components in the switches can be turned off; e.g. the input buffers and crossbars. The reactivation times of these elements are much longer, at up to a millisecond, which could cause an unacceptably large

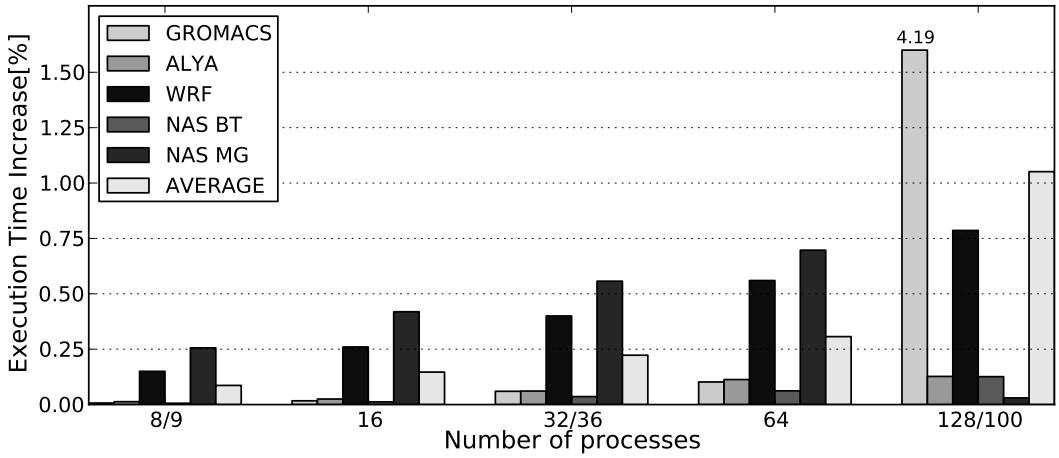


Figure 5.13: Applications execution time increase - results when small *displacement factor* of 1% is employed

increase in execution time. We expect that our power saving mechanism can better amortize larger reactivation times and allow switches to go to deeper low-power modes without any major negative effect on the execution times.

Grouping Threshold (GT) Value

An important parameter in the PPA algorithm is the grouping threshold (GT) value, which determines whether two consecutive MPI calls should be considered as part of the same gram. Since there are no opportunities for power savings during idle periods shorter than $2 \times T_{\text{react}}$, the value of GT should be greater than this value.

Table 5.3 shows the values of the grouping threshold that were used for evaluation, as well as the resulting prediction accuracy. Prediction accuracy is averaged over all MPI calls, including those outside the iterative parts of the application, which correspond to less predictable initialization and finalization phases. This is an important consideration for WRF and partially for Gromacs, while for Alya, NAS BT and NAS MG, the majority of calls are inside the iterative phase and the prediction accuracy is rather large. It is interesting that although the WRF application has the lowest prediction accuracy,

it has the second-largest power savings; see Figure 5.12. This is because the majority of large idle intervals are inside the iterative phase, while idle intervals in the other parts of the application are quite small. The opposite is true for the Alya application, where the prediction accuracy is large but the power savings are smaller. Here, the mayority of the large idle intervals are not in the iterative part of the application.

Table 5.3: Chosen GT across HPC applications

	Num proc	Grouping Threshold, GT (μs)	MPI call hit rate (%)
GROMACS	8	20	42
	16	222	44
	32	20	48
	64	22	44
	128	136	59
ALYA	8	20	93
	16	72	93
	32	36	93
	64	36	93
	128	20	93
WRF	8	56	25
	16	30	33
	32	30	32
	64	36	31
	128	22	31
NAS BT	9	20	97
	16	22	98
	36	46	98
	64	20	98
	100	50	98
NAS MG	8	300	74
	16	382	79
	32	300	70
	64	290	74
	128	150	74

Table 5.4: Average PPA overheads: 16 MPI processes

HPC workload	MPI calls when PPA is invoked	Overhead when PPA invoked (μs)	Average overhead per MPI call (μs)
Gromacs	4.7%	25.1	2.1
Alya	1.2%	16.1	1.2
WRF	0.4%	7.8	1.1
NAS BT	3.7%	6.9	1.1
NAS MG	0.5%	26.4	1.05
Average	2.1%	16.5	1.3

System Overheads

To measure overheads we relied on the system clock using the *gettimeofday* system call. The costs of overheads associated with interception of the MPI call and reading the system time are approximately 1 μs. These overheads occur every MPI call while overheads that come from power saving system are different and do not occur on every MPI call. When the algorithm predicts the repeating pattern allowing power saving mechanism to shut down inactive lanes, the PPA is disabled, waiting for pattern misprediction to be relaunched again. Also, if the number of necessary grams is not enough the PPA will not be invoked. For activation/deactivation of the IB lanes, we chose a typical latency of 10 μs. While the deactivation will be overlapped with computation, the reactivation penalty in case of misprediction has to be paid. The penalty could be equal or smaller than reactivation time if reactivation has already been started. The PPA overheads are also varying and depend on pattern size and number of all possible patterns detected during the execution. We used *uthash*[33] hash table to store the pattern objects where pattern is used as a key. Table 5.4 shows the average overheads of PPA through the HPC applications. Although the overhead per MPI call on first sight can seem very large, it only occurs on small number of MPI calls (average 2.1%). The overheads associated with PPA can be further reduced by using faster hash tables.

5.3 Self-Tuned Pattern Prediction System

5.3.1 Effect of Grouping Threshold (GT) value

The Grouping Threshold (GT) defines the threshold between short and long idle intervals. An MPI event that is preceded by a long idle interval; i.e. one longer than the GT, becomes the first MPI event in a new gram. All consecutive MPI events that are preceded by short intervals will be included in this same gram. This grouping of MPI events into grams was shown in Figure 5.2.

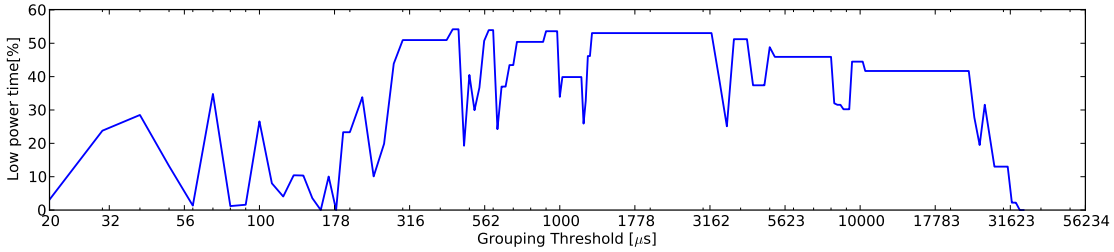


Figure 5.14: Effect of Grouping Threshold on time in low power mode, for NASMG benchmark with 32 MPI processes, showing data of process 0

Figure 5.14 shows the effect of varying the Grouping Threshold for one process of the MG NAS benchmark. The *y*-axis is the fraction of time that the link stays in low-power mode, and higher values are clearly better. The *x*-axis is the value of the GT, varying from the minimum value of 20 µs, which is the total time required by the hardware to enter and leave the low-power mode, to an upper value of around 56 ms. After 32 ms, the curve reaches zero as even the largest (the one that can be predicted) idle interval is classified as short i.e. below the GT value. We observe that the amount of time that the link stays in low-power mode remains steady around GT values of 400 µs, 2 ms and 15 ms.

We wish to find the best value of GT automatically, but the erratic behavior of the function in Figure 5.14 implies that it will be difficult to optimize, without additional context. There is, however, a connection between the sharp dips in this function and the frequency histogram of the idle interval lengths, as described in Section 5.3.2, which can be used for optimization.

5.3.2 Design

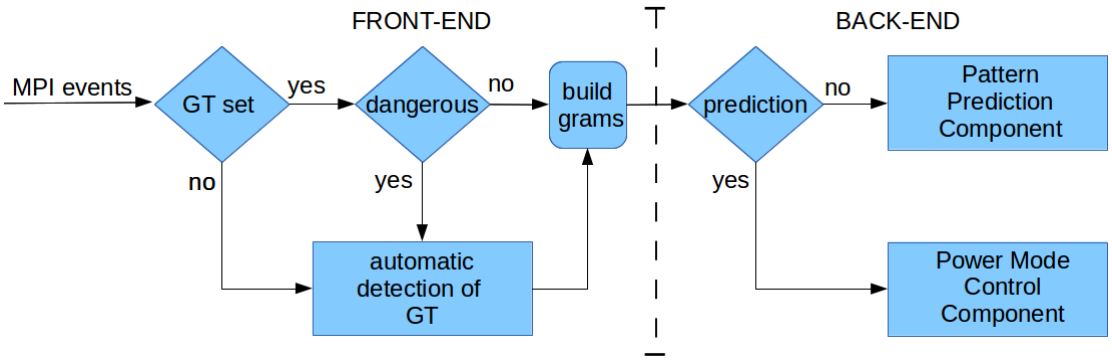


Figure 5.15: Simplified block diagram of Self-Tuned Pattern Prediction System (SPPS)

This section describes the Self-Tuned Pattern Prediction System (SPPS), a software-managed interconnect energy saving mechanism executed in the PMPI profiling layer of the MPI library. The overall structure of SPPS is shown in Figure 5.15. The front-end groups the MPI events into grams, each of which contains one or more MPI events separated only by short idle intervals. The first task is to determine the Grouping Threshold (GT) which distinguishes between long and short intervals. For reasons given in Subsection 5.3.2, idle intervals of length close to the GT may cause problems with prediction. The front-end therefore checks whether any idle intervals appear close to the GT, in a region known as the *dangerous zone*, and, if so, the GT must be recalculated. Otherwise, the MPI events are processed according to PPS [28], by grouping them into grams and passing them to the back-end. In the back-end, the Pattern Prediction Component detects repeatable communication patterns and it uses these patterns to predict the link's idle intervals. If prediction is possible, then the Power Mode Control Component (PMCC) shifts between link power modes.

GT detection algorithm approach

This section gives an informal description of the intuition behind the SPPS algorithm. As already described, the Grouping Threshold (GT) is used to classify each idle interval as short or long. In order

for the prediction algorithm to function well, this classification must be consistent, meaning that each time the pattern repeats, corresponding MPI events should always be classified in the same way.

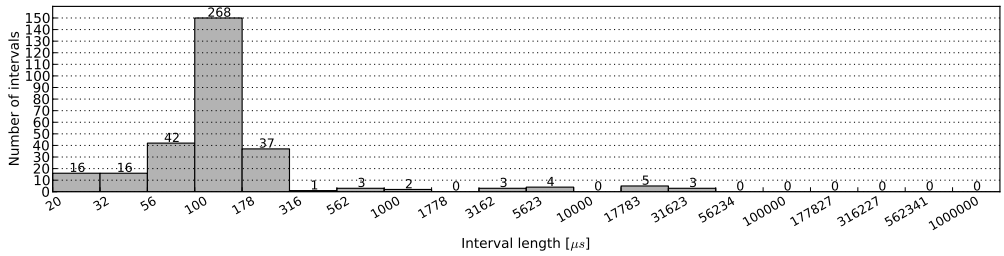


Figure 5.16: *Histogram*$_f$: Histogram of number of IDLE intervals for NASMG benchmark with 32 MPI processes, showing data of process 0

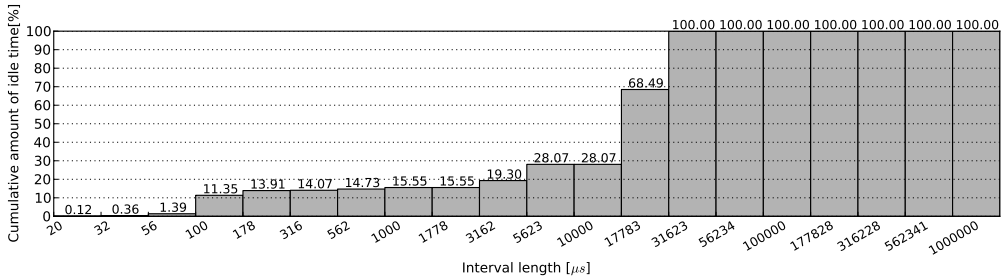


Figure 5.17: *Histogram*$_{cumt}$: Cumulative histogram of IDLE interval time for NASMG benchmark with 32 MPI processes, showing data of process 0

Figure 5.16 is a histogram of the idle intervals, known as *Histogram*$_f$, for the same example used in Figure 5.14. Each bin counts the number of idle intervals in the relevant range. For example, there are 268 idle intervals of length between 100 μs and 178 μs. There are a large number of idle intervals of length less than about 300 μs, with no clear internal structure. There are also three smaller peaks, with clear gaps separating them from the other peaks. The region below 300 μs is the same region, visible in Figure 5.14, for which the GT gives poor and unreliable results. We see similar behaviour across many of the benchmarks.

This suggests that the algorithm should avoid values of the GT close to many idle intervals. If the GT is close to a large number of idle intervals, it is likely that when a pattern is repeated, some

of the idle intervals will be classified differently each time, i.e. they will be on different sides of the GT. Such behaviour will mean that the pattern is not recognised, leading to a low prediction rate. To avoid this we should aim to set the GT in an area without idle intervals, which will cover the region of at least GT±10% (in the rest of the text we refer to this region as the *dangerous zone*). The reason for the size of ±10% is that the lengths of idle link intervals are always subject to noise, which makes corresponding idle intervals have slightly different lengths. The tolerance of ±10% is proposed based on offline clustering experiments, which have shown that idle intervals tend to be in groups of at most this size.

Figure 5.17 is a cumulative histogram of the idle time, which is known as $Histogram_{cumt}$. Each bin is the percentage of the total idle time that is found in idle intervals of length less than or equal to the upper bound of the bin. That is, each bin gives the percentage of the total idle time mapped to either the given bin or a "smaller" bin. The behaviour in the example is typical, as most of the idle time is contained in long intervals. This supports the decision to put the interconnect into low-power mode only during comparatively long intervals. The GT should however be small enough to capture most of the energy savings.

The approach is therefore to use the histogram of the number of idle events, given in Figure 5.16, to choose only values of GT for which there are zero or few idle intervals of a similar length, and then to use the cumulative histogram of the total idle time, shown in Figure 5.17, to choose a GT value with acceptable energy savings.

Description of the GT detection algorithm

The algorithm that finds the best Grouping Threshold (GT) value is known as Automatized Detection of Grouping Threshold (ADGT) algorithm, and given in pseudocode in Algorithm 5. This algorithm is divided into three phases, which are described in the following subsections.

Configuration and Build-Histogram Phase: The first phase collects a sample of the initial idle in-

terval lengths, and it builds the two histograms defined in Section 5.3.2. Since many applications start with an initialization phase significantly different from the rest of the application, line 1 skips the first few idle intervals. The number of skipped idle intervals is given by *firstIdleIntervals*, and is set to 100 in the evaluation (in a more sophisticated system, the initialization phase could be determined dynamically). Line 2 collects the sample of idle intervals, and lines 5 and 6 build the two histograms. The number of bins for the histograms is given by a common rule of thumb, which uses the square-root of the number of idle intervals in the sample. Both histograms should use a log-scale representation of the idle intervals, due to an empirically observed skewness towards shorter intervals. In both histograms, the last bin, corresponding to the longest idle intervals, should include all idle intervals longer than its lower bound; i.e. they should also include the idle events longer than the histogram's nominal upper bound of 1.7 s. Lastly, we set the demanded time of a link in low-power mode, *demCumIdleTime*, to the initial value of 90%. This value was chosen based on our previous study[28], which found that 90% of idle time is spent in relatively large idle intervals. This value is, however, decreased if no solution is found, as elaborated in the next phase.

Main Phase: The main phase finds a histogram bin containing at most a few idle intervals that still leads to acceptable energy savings. Line 13 excludes all values of GT that would lead to unacceptable energy savings, i.e. those where the energy savings would be less than *demCumIdleTime*. This is done by pointing *indexBin* to the "smallest" bin that does not have acceptable energy savings. There is at least one such bin, since the construction of $Histogram_{cumt}$ ensures that its "largest" bin always has the value 100%. Hence, the set of values given to the min operator is non-empty, and since all elements are valid, the value of *indexBin* is valid. Line 16 determines the bin with acceptable energy savings that contains the smallest number of idle intervals. The resulting value of *Bin* is always valid because the argmin operator always returns a non-empty set of valid bins. It is not empty because the case $indexBin = 0$, which would cause argmin to be evaluated on an empty set, is specifically excluded by line 14. If several bins count the same number of idle intervals, then the "largest" bin is chosen.

Algorithm 5 Automatized Detection of Grouping Threshold (ADGT) value

Input: *firstIdleIntervals*, *numberIdleIntervals*, *demCumIdleTime*, α
Output: Grouping Threshold (*GT*).

▷ Configuration and Build-Histogram Phase

skip *firstIdleIntervals*
idleIntervals = collect *numberIdleIntervals*
nBin = $\lfloor\sqrt{numberIdleIntervals}\rfloor$

▷ Use Coarse Grain Log-Scale (20μs, 1s)

dangNum = $\lfloor\alpha \times (28.6 \div nBin)\rfloor$
$Histogram_f$ = freqhistogram(*idleIntervals*, *nBin*)
$Histogram_{cumt}$ = cum-timehistogram(*idleIntervals*, *nBin*)

▷ Main Phase

aFind = *false*
do
    do
        binGT = *false*
        if *demCumIdleTime* < 0.1 then
            return None

        indexBin = min b
            $b \in \{0, \cdots, nBin - 1\}$
            $Histogram_{cumt}[b] \geq (100\% - demCumIdleTime)$
        if *indexBin* = 0 then
            return None
        Bin = max argmin $Histogram_f[b]$
            $b \in \{0, ..., indexBin - 1\}$
        if $Histogram_f[Bin] < dangNum$ then
            binGT = *true*
        else
            demCumIdleTime = *demCumIdleTime* × 0.95
    while (*binGT is false*)

▷ Finalization Phase

    minval = $Histogram_f.minval[Bin]$
    maxval = $Histogram_f.maxval[Bin]$
    idleIntervalsInBin = $\{x \in idleIntervals,$
      $minval < x < maxval\}$
    sIdleIntervals = sort(*idleIntervalsInBin*)
    sIdleIntervals = $[minval] +\!\!+ sIdleIntervals +\!\!+ [maxval]$
    for $i \leftarrow 0$ to *len*(*sIdleIntervals*) − 2 do
        dist = (*sIdleIntervals*$[i+1] \div$ *sIdleIntervals*$[i]$)
        if *dist* ≥ 1.494 then
            aFind = *true*
            idx = *i*
            break
    if *aFind* = *false* then
        demCumIdleTime = *demCumIdleTime* × 0.95
while (*aFind is false*)
GT = 1.1 × *sIdleIntervals*$[idx] \div 0.9$
return *GT*

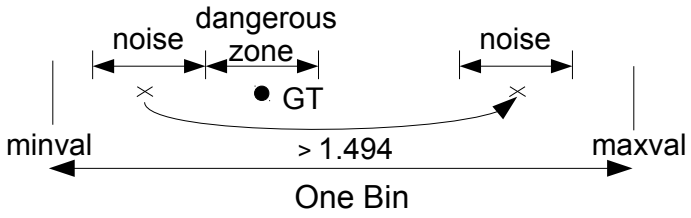


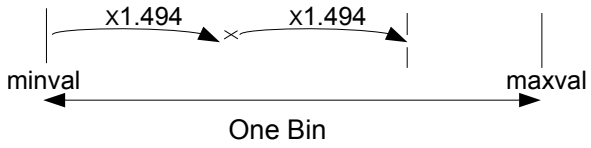
Figure 5.18: Necessary gap in the finalization stage for successful GT allocation

Line 17 checks the number of intervals in the bin, and if it is sufficiently low, then there is a high chance that the finalization phase will be able to find a valid GT value inside the bin. The bin is accepted if the number of intervals is less than a value known as *dangNum*, whose calculation is explained in Section 5.3.2. When a satisfactory bin has been found, the algorithm proceeds to the finalization phase by setting *binGT* to *True* in Line 18. Otherwise, in Line 20, the value of *demCumIdleTime* is decreased by 5%, and the loop is repeated. It is possible, though in practice unlikely, that no bin has less than *dangNum* idle intervals, in which case, line 12 causes the algorithm to fail.

Finalization Phase: When an acceptable bin has been found, the final stage is to choose the value of GT inside that bin. Lines 22 and 23 determine the shortest and longest idle interval lengths mapped to the bin. This defines the acceptable range for the GT value. Line 26 calculates a sorted list of idle interval samples, with the *minval* and *maxval* delimiting the endpoints of the bin. The loop on lines 27 to 32 searches for the first sufficiently large gap with no idle intervals. As shown in Figure 5.18, the gap between two consecutive idle interval lengths (represented with crosses) should be large enough to contain the *dangerous zone* around the GT, as well as the $\pm$10% noise margin around the adjacent idle interval lengths. It is therefore necessary that the larger idle interval length should be at least 1.494 times the smaller idle interval length.

If there is such a gap, then line 36 calculates the GT value, as illustrated in Figure 5.18, setting it to a safe distance above the shorter idle interval lengths, rather than placing it in the middle, in order to maximise the potential energy savings. If it is not possible to find such a gap, then line 34 reduces

a.) with n idle intervals (n=1):



$$\left(maxval = minval \times \left(100.000\right)^{\left(\frac{1}{nBin}\right)}\right)$$

always have large enough gap if:

$$maxval \geq 1.494^{(n+1)} \times minval$$

$$1.494^{(n+1)} \leq \left(100.000\right)^{\left(\frac{1}{nBin}\right)}$$

$$n+1 \leq \lfloor \left(\frac{1}{nBin}\right) \frac{\left(\log 100.000\right)}{\left(\log 1.494\right)} \rfloor$$

$$n < \lfloor \left(\frac{1}{nBin}\right) \frac{\left(\log 100.000\right)}{\left(\log 1.494\right)} \rfloor \approx \left(\frac{28.6}{nBin}\right)$$

b.) Simplifying and introducing tolerance factor α:

$$n < \lfloor \alpha \times \left(\tfrac{28.6}{nBin}\right) \rfloor := dangNum$$

Figure 5.19: Value of *dangNum* used in main phase of ADGT. α is tolerance factor, in our tests $\alpha = 4$

demCumIdleTime by 5%. This is an error condition that returns to the main phase. If no such gap is ever found, then line 12 will eventually cause the algorithm to fail.

Calculation of dangNum

As remarked in the previous section, a candidate bin is only accepted if the number of idle intervals that it contains is less than a value known as *dangNum*. This value is determined as described below. Figure 5.19(a) shows a single histogram bin containing n idle intervals. As explained above, a value of GT can be successfully chosen whenever the ratio of two consecutive idle intervals is greater than or equal to 1.494. There is always at least one such gap whenever $maxval \geq 1.494^{n+1} \times minval$. This is illustrated in Figure 5.19(a), which shows that if the first n gaps are smaller than 1.494, then the last

gap must be larger than this value—so they cannot all be smaller. The rest of Figure 5.19(a) simplifies this equation, using the correct value of the bin's width.* We find, however, that using this value is too restrictive, so the value is increased slightly using a tolerance factor, α, obtaining the final formula for *dangNum* shown in Figure 5.19(b). For example, if the sample contains 400 idle intervals, then $nBin = \sqrt{400} = 20$, so if $\alpha = 4$, then *dangNum* equals five.

Restart mechanism

The restart mechanism causes the Grouping Threshold (GT) to be recalculated when doing so is likely to be beneficial. A restart is needed when the application changes phase. It is also needed when random factors cause the ADGT algorithm to choose a bad value of the GT.

As mentioned in Section 5.3.2, the GT values should not be close to many idle intervals. If several idle intervals are seen within GT±10%, which is known as the *dangerous zone*, then the prediction algorithm is likely to become unstable. In order to restore the proper functioning of the prediction algorithm, it is restarted, which causes a new automatic detection of the GT. An important question is how sensitive to make the restart mechanism since, on the one hand, no energy savings are possible during restart, but on the other hand, it is worthless to continue using a bad value of GT for too long. On balance we tolerate a single idle interval in the dangerous zone, and restart if a second one appears within an interval of three times the current pattern size.

*Referring to Figure 5.16 and Figure 5.17, there are *nBins* bins in the histogram, which covers the range from 17 µs at the bottom of the first bin (although the figure specifies 20 µs since intervals less than 20 µs are ignored) to 1.7 s at the top of the last bin (this is unmarked, since the last bin contains all idle intervals longer than 1 s). Hence the ratio between the upper and lower limits of each bin is given by $(1.7\,\text{s}/17\,\mu\text{s})^{1/nBin} = (100,000)^{1/nBin}$.

5.3.3 Experimental Evaluation

Methodology

We use the methodology previously described in Section 5.2.6. The traces of ten representative HPC workloads were collected on the MinoTauro machine (described in Section 3.1). The applications were configured with one MPI process per node.

The benchmarks cover a broad range of scientific workloads, with the number of MPI processes varying between 16 and 128. MILC and QUANTUM were executed with 16 MPI processes, GROMACS, ALYA, WRF and NASMG with 32 MPI processes, and NASSP and NASBT had 36 MPI processes. PEPC and CPMD had larger runs with 64 and 128 MPI processes respectively.

We measured the execution time overheads for the SPPS algorithm on a real system, by reading the system clock using the **`gettimeofday()`** system call. We use the execution time overheads as described in Section 5.2.6 (execution time overheads that arise from the Pattern Prediction Component (PPC), were on average about 1 µs to 2 µs per MPI call). We measured the additional overheads that arise from collecting the idle intervals to be far less than 1 µs. We also measured the time to calculate the GT value, following the algorithm in Section 5.3.2, which was between 5 µs and 10 µs. We included all these overheads in the traces by increasing the lengths of the relevant computation bursts.

Results

This section discusses the experimental results, including a comparison with the existing PPS algorithm [28]. The results for PPS use a single best value of GT taken from a manual sweep. Since the SPPS algorithm introduced in the current paper extends the previous algorithm to automatically determine the GT value, the optimal PPS algorithm results should be seen as ideal results to try to match, rather than the current state-of-the-art to improve upon.

The main experimental results are shown in Figure 5.20, which gives the increase in execution time,

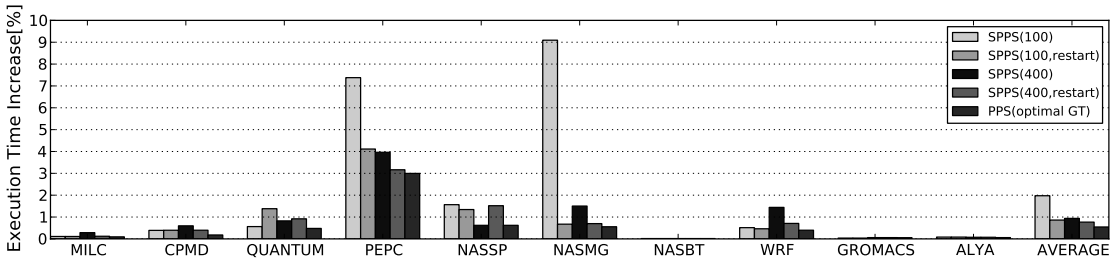


Figure 5.20: Applications execution time increase

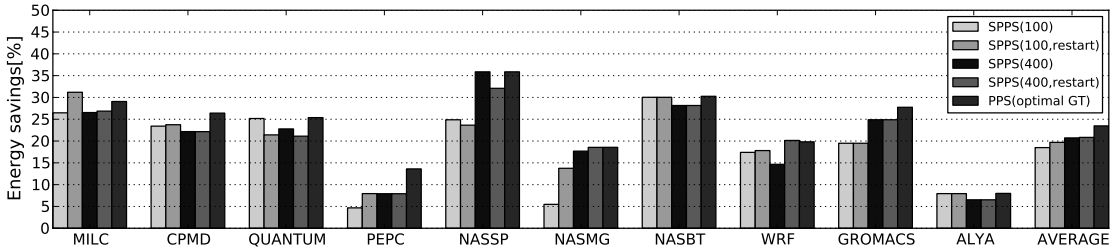


Figure 5.21: Energy savings in InfiniBand edge switch links

and Figure 5.21, which gives the energy savings. For each benchmark we show five results. The first four results are for various configurations of the SPPS algorithm, with a sample size of 100 or 400 idle intervals, and with or without the restart mechanism. The final result, "PPS (optimal GT)", shows the best possible result from the existing PPS algorithm.

The results for a sample size of 400 have a worst case overhead of about 4% and an average overhead of less than 1% (Figure 5.20), both only marginally above that of PPS−Optimal. Regarding the energy savings, the average interconnect edge link energy savings is 21% (Figure 5.21), which is within 3% of the average from PPS−Optimal algorithm. The difference between a sample size of 100 and 400 idle intervals is, for most benchmarks, small, but there are two benchmarks (PEPC and NASMG) for which the smaller sample size initially chooses a bad value of the GT. These benchmarks, however, benefit from restart, which allows the algorithm to recover by calculating a new GT value, in which case it quickly chooses a good value. These benchmarks also benefit from restart when the sample size is 400, but to a lesser extent.

When comparing the dynamic SPPS algorithm against the results for the static PPS−Optimal algorithm, there are two effects to bear in mind. Firstly, the static PPS−Optimal algorithm has the advantage of using the best global value of GT. Secondly, the dynamic SPPS algorithm has the ability to dynamically adjust the GT value taking into account application phases, so long as restart is enabled. Among our benchmarks, only MILC has multiple identifiable phases, but the available benefit is small, amounting to about 5% in energy for the 100-interval case. This is the only result for which the SPPS algorithm has better energy savings than PPS−Optimal. We expect this dynamic adaptability to lead to a larger benefit when running large workloads with multiple phases.

The results show that restart should be enabled, mainly in order to avoid commiting to a poorly-chosen GT value. As already remarked, PEPC and NASMG show a large reduction in overheads, when restart is enabled, especially with a 100-interval sample. These overheads are caused by the Pattern Prediction Component (PPC), which has larger overheads than the Power Mode Control Component (PMCC), together with the large number of mispredictions. The remaining benchmarks show little difference in performance or energy, although QUANTUM and NASSP show a drop in energy savings of about 5%. The average results show that restart provides a large improvement in performance, although this effect is mostly due to the large benefit for PEPC and NASMG.

5.4 Conclusions

In this chapter we have presented a software-directed mechanism for interconnect link energy proportionality. We propose the PPA algorithm, to be executed within the PMPI layer of MPI. Putting the intelligence in the MPI library allows differentiation by the system integrator and customisation by the operations department, while avoiding any need for modifications to the user's source code. This allows energy savings to be achieved for unmodified existing MPI applications.

The PPA algorithm detects the repetitive communication patterns that are typical of modern scien-

tific applications, and it uses this knowledge to predict the durations of the link idle periods. The links are put into low-power mode during idle periods until a short time before they are expected to become active again, leading to a significant reduction in the average link energy consumption at negligible loss in performance.

We evaluated the possible power savings with strong scaling runs, which gives pessimistic results, since network utilization, which should be proportional to energy consumption, increases with the number of nodes. In addition, strong scaling leads to shorter computation periods, meaning that constant overheads in changing power mode are amortized over short idle periods. Weak scaling runs would therefore lead to larger observed energy savings. Nevertheless, the results show the possibility for significant energy savings in IB edge switch links of up to 33%, with a negligible increase in execution time of around 1%.

We also show possibilities for further switch energy savings, by powering down other elements in the switch. Such elements take much longer to change power state, requiring up to a millisecond to wake, increasing the need for accurate prediction mechanisms such as the PPA algorithm.

In the second part of this chapter we present Self-Tuned Pattern Prediction System (SPPS), which is an automatic software-directed mechanism that is aimed for interconnect link energy proportionality. SPPS automatically recognises and exploits repetitive patterns in the application's communication behaviour. Like PPS, it is implemented in the MPI library, which allows all MPI programs to benefit, without needing changes to their source code.

Our automatic mechanism tries to identify and predict those link idle intervals where the majority of idle link power is spent. Discovering the critical idle interval length, defined by the Grouping Threshold (GT) value, allows to filter out irrelevant idle intervals in the communication patterns. We propose to detect GT for an application process based on its characteristics. We analyze and find the GT value at runtime without knowing any details about the application. We also show that the SPPS system is capable of adjusting to different application regions selecting the new customized GT for

each region, while keeping the overheads at the minimum.

We evaluate SPPS using an event-driven simulator with traces from a production supercomputer. When compared to baseline unmanaged execution, our simulations show average energy savings in the network edge links of up to 21% with an average execution time penalty less than 1%.

The obtained results show that our runtime self-tuned algorithm can work cooperatively with the on-the-fly dynamic management. Experimental results confirm the effectiveness of our pattern prediction approach on large number of applications and benchmarks. Therefore, we believe that our solution, with its significant energy-saving potential and low performance overheads, can accelerate the adoption of runtime power management techniques in HPC systems.

Chapter 6

Related Work

Enhancing Network Efficiency Using MPI Data Compression

In the Chapter 4 where compression is proposed for enhancing network efficiency the applications and its kernels worked on IEEE 754 double precision data which is the general data type for numerical programs. A large body of related work focuses on optimizations of this type of data. Some works have applied compression to improve the bandwidth to the I/O subsystem during large scale molecular dynamic simulations [63], while other works focus on compressing MPI messages that are exchanged in distributed systems.

Burtscher et al. [23] proposed the lossless FPC compression algorithm for double precision data. The FPC algorithm works well for pattern based data sets. The algorithm tries to capture these patterns and store them in hash tables. In their work they applied FPC to datasets that are the result of numerical simulation, on datasets that contain MPI messages (from NAS Parallel Benchmark (NPB)[18] and ASCI Purple applications [4]) and on observational datasets that comprise measurements from scientific instruments. In some cases they obtain very good compression rates, but in others the compression rates are very poor. This shows that FPC is not a general FP compression algorithm, but works well

only for data that follows certain patterns. Previously, the authors proposed the DFCM compressor[39] for double precision data which is a subpart of the later developed FPC compressor. They incorporated DFCM in an MPI library to speed up MPI programs running on a cluster of workstations. This cluster used a low performance network (Fast Ethernet) which created a high communication to computation ratio when the application was executed on a large number of processors. They therefore achieved good speedups, of up to 98%.

Tomari et al.[60] developed the lossless MAF compression algorithm which eliminates redundancy in the exponents of arrays of numbers formatted as IEEE 754 double precision numbers. They reported that they achieved a compression ratio of 1.2× against data stream in FFT benchmark.

Another lossless algorithm based on arithmetic prediction was proposed by Katahira et al.[38]. The algorithm is implemented in hardware and is used to enhance the memory bandwidth of LBM (Lattice Boltzman Method) stream-computing accelerators. They achieved compression rates up to 3.5×. Like FPC this algorithm is based on prediction patterns and we expect it to behave similarly on other data sets.

Filguera el al.[30] used different types of compression algorithms which were applied to MPI transfers at runtime depending on the type of data. They achieved good results, but mostly for integer and single precision FP data.

There is also a large body of work on general-purpose lossless data compression, including GZIP[8], BZIP2[5], etc. However, as described in Section 4.3, these algorithms, specifically GZIP, achieved poor compression rates for double-precision scientific data (of at most 1.12). In addition, these compression algorithms are quite complex, making their use a poor trade-off, even in combination with lossy compression.

The work most similar to ours was done by Kumar et al.[41]. In their work they also contemplate the improvements by using both lossless and lossy compression schemes for `doubles`. Lossless compression schemes are similar to proposed by Burtscher et al[23] whereas lossy compression is done by

reducing the number of mantissa bits, as in our work. They applied both compression schemes in the communication intensive part of the Community Atmosphere Model application. Using a lossless compression scheme run on 32 processes they achieve a speedup of 53.58%. Such large speedups were obtained primarily because they ran the tests on a cluster that uses a low performance network (Gigabit Ethernet). The cluster was therefore much more sensitive to the message size. On the other hand, we ran our tests on a TOP500 supercomputer which uses the high performance Myrinet interconnect.

In order to use lossy compression algorithms one has to ensure that the final result will not deviate significantly from the correct result. Lopes et al. [46] used model predictive control to achieve a satisfactory degree of correctness in the final result from a conjugate gradient algorithm. This work should be extended, to cover a greater number of applications.

Runtime Software-Managed Power Savings in IB Links

Optimization of the interconnection network energy consumption optimization is an important target for HPC system designers. Hoefler [35] gives an overview of the power problem and related aspects of interconnect power, with a focus on supercomputers. Power models for the interconnection network, which characterize the power profile of network routers and links, have been proposed, enabling further research into power-efficient techniques [62]. Several network power-saving techniques have been proposed, most of which are hardware approaches based on the runtime behaviour at a single switch or link. Shang et al. [56] proposed a history-based dynamic voltage scaling (DVS) policy, where past network utilization is used to predict future traffic. Alonso et al. [17] propose a power-saving mechanism for regular interconnection networks built with high-degree switches and port trunking (also known as link aggregation). Each trunk link is composed of multiple links that can be individually turned on and off depending on the load. There is at least one link on, at all times, in order to maintain connectivity. Kim et al. [40] use both dynamic voltage scaling and the powering down of under-utilized links. This technique requires adaptive routing, in order to avoid deadlocks. Saravanan et al. [54] proposes an

algorithm to turn links on and off using an idle period predictor, which detects repetitive behaviour at the packet level. Our technique, in contrast, is done at the application level, and before injecting data into the network. By predicting end-to-end traffic we are able to predict bursty data transmission more accurately than if prediction is done at the packet level.

Although previous work shows that hardware-based schemes can be effective, they share a common drawback that they may be to slow to adjust to sudden changes in the network traffic. Soteriou et al.[58] show that hardware-based approaches can incur large performance overheads and propose a compile-time technique, as part of a parallelizing compiler flow, that generates instructions to dynamically control network power reduction. Li et al.[43] propose another compiler-based technique, which inserts instructions to turn off communication links at the point in each loop nest when they are last used. The link is reactivated on-demand the next time it is used.

Jian Li et al.[44] are focused on non-prediction power-saving techniques. Links are powered up just before they are needed, by relying on hints from the built-in system events or from macros in MPI source code. Here, a separate control network is needed which is always on, to enable link activation messages to flow through. In our approach, we rely on InfiniBand architecture with links that offer a dynamic range in terms of performance and power.

In the work of Abts et al.[15], the authors propose energy-proportional datacenter networks. Link data rates are selected on the basis of traffic intensity in the network. They use the congestion sensing heuristic to sense traffic intensity, dynamically activating links as they are needed. While this work is focused on datacenter applications, which can tolerate small changes in latency, HPC applications cannot afford such performance loss.

Huang et al[36] propose a table-based traffic predictor (ATPT) which can predict the amount of data injected in the network allowing the suitable working frequency and the corresponding voltages of the links to be set in advance. The main problem lies in the unknown time interval in tracking and predicting traffic which if not chosen correctly can affect the prediction acccuracy. Also the amounts

of data transmitted are quantized according to the number of discrete voltage-frequency (VF) levels implying that more adjustable voltage levels will incur more costs.

Finally, the work of Lim et al.[45] is complementary to ours, in the sense that both are power saving techniques in the MPI run-time system. Whereas our technique turns off communication links during computation periods, Lim et al. reduces CPU power consumption during the communication phases. The run-time system identifies communication regions and adjusts the processor frequency to minimize the energy–delay product, without needing profiling or training.

Chapter 7

Conclusion

One of the biggest challenges in high-performance computing is to reduce the power and energy consumption. Significant improvements in energy efficiency are required across all subsystems, and, as processors and memory have become more energy efficient, attention is turning to the interconnect.

In the first part of the thesis, we evaluated the benefits and trade-offs of using MPI compression techniques in HPC production environments. We first tried using general purpose lossless data compressors, such as GZIP. In all cases, we found that the data exchanged between processes had few, if any, exploitable patterns, which led to poor compression ratios. To overcome this problem, we used lossy compression (with a compression ratio of up to 2), and we verified that the remaining accuracy was still sufficient to obtain correct results. We first evaluated the effect of data compression on application performance. Our results showed that the effect on performance was generally limited, with speed-up factors rather lower than expected by Amdahl's law for the used compression rates. The blocking nature of point-to-point MPI calls in the nearest-neighbour pattern, where only a single message is outstanding in communication between each pair of processes, does not overload network resources at the HCA. More time is spent on scheduling and synchronization inside the communication pattern

than on the actual data transfer. Also, when the size of the messages and the number of neighborhood processes for each process are variable, the total time of communication is also affected. On the other hand, patterns like all-to-all collectives tend to synchronize the tasks, leading to a larger speedup. This communication pattern loads the HCA channel with multiple MPI messages, so a reduction in their size improves performance.

To the best of our knowledge, we are the first to apply data compression to link energy savings. Using compression allows the number of active lanes to be reduced in proportion to the compression rate. Thanks to compression, even with reduced network bandwidth, the application performance is not affected. Reactivation delays typically increased execution time by just a few percent. Using 50% compression, we obtained in the lowest (edge-level) network links energy savings of up to 71% for the Alya CG kernel and 63% for Gromacs PME kernel. We also show that strong scaling runs, in particular, have a large benefit from data compression.

In the second part of the thesis we propose techniques to make high performance interconnects energy proportional, that is, the amount of energy consumed is proportional to the traffic intensity in the network. Now that energy consumption is beginning to account for a significant fraction of an HPC system's total cost of ownership, there is pressure for all system components to become more energy efficient. An important characteristic of energy-efficiency is energy proportionality. Although processors and memories are now close to achieve energy proportionality, high-performance interconnects are not.

This thesis presents a software-directed mechanism for interconnect link energy proportionality. We propose the Pattern Prediction System (PPS) which is executed within the PMPI layer of MPI. Putting the intelligence in the MPI library allows differentiation by the system integrator and customisation by the operations department, while avoiding any need for modifications to the user's source code. This allows energy savings to be achieved for unmodified existing MPI applications.

Pattern Prediction System (PPS) with its main PPA algorithm detects the repetitive communica-

tion patterns that are typical of modern scientific applications, and it uses this knowledge to predict the durations of the link idle periods. The links are put into low-power mode during idle periods until a short time before they are expected to become active again, leading to a significant reduction in the average link energy consumption at negligible loss in performance.

We evaluated the possible power savings with strong scaling runs, which give pessimistic results, since increasing the number of nodes increases the communication-to-computation ratio, which reduces the proportion of energy that can be saved through energy proportionality. In addition, strong scaling leads to shorter computation periods, meaning that constant overheads in changing power mode are amortized over short idle periods. In contrast, weak scaling runs would lead to larger observed energy savings. Nevertheless, the results show the possibility for significant energy savings in IB edge switch links of up to 33%, with a negligible increase in execution time of around 1%.

To completely automate the prediction process we further develop the Self-Tuned Pattern Prediction System (SPPS), an automatic software-directed mechanism for interconnect link energy proportionality. SPPS automatically recognises and exploits repetitive patterns in the application's communication behaviour.

Our automatic mechanism tries to identify and predict those link idle intervals where the majority of idle link power is spent. Discovering the critical idle interval length, defined by the Grouping Threshold (GT) value, allows to filter out irrelevant idle intervals in the communication patterns. We propose to detect GT for an application process based on its characteristics. We analyze and find the GT value at runtime without knowing any details about the application. We also show that the SPPS system is capable of adjusting to different application regions selecting the new customized GT for each region, while keeping the overheads at the minimum.

We evaluate SPPS using an event-driven simulator with traces from a production supercomputer. When compared to baseline unmanaged execution, our simulations show average energy savings in the lowest (edge-level) network links of up to 21% with an average execution time penalty less than 1%.

The obtained results show that our runtime self-tuned algorithm can work cooperatively with the on-the-fly dynamic management. Experimental results confirm the effectiveness of our pattern prediction approach on large number of applications and benchmarks. Therefore, we believe that our solution, with its significant energy-saving potential and low performance overheads, can accelerate the adoption of runtime power management techniques in HPC systems.

We also show possibilities for further switch energy savings, by powering down other elements in the switch. Such elements take much longer to change power state, requiring up to a millisecond to wake, increasing the need for accurate prediction mechanisms such as the PPA algorithm.

Finally, it is important to note that the principles of our system are not restricted to Infiniband. Many modern interconnect technologies, like Infiniband, have multiple lanes at the physical layer. For example, 40GbE Ethernet has four lanes at 10 Gb/s each, although there is currently no standard for turning lanes on and off individually. Proposals like ours may have an impact on future standardisation efforts.

Chapter 8

Future Work

This thesis proposed two orthogonal directions for reducing the energy consumption of the network links. During the communication bursts, we use data compression, as described in Chapter 4, to reduce the energy consumption without affecting performance. During the computation bursts, on the other hand, we use prediction, as described in Chapter 5, to reduce the number of active lanes while the network is idle. These directions are compatible, but in this thesis they were evaluated separately. In future work, they should be combined into a unified technique, and evaluated as a whole.

The energy-saving techniques are intended to work with the lowest (edge-level) network links. We would like to extend these techniques to also support power saving in the higher-level links. Since a single high-level link could be used by multiple MPI processes and potentially multiple applications, deciding how to combine the individual decisions due to different compression rates and/or idle time predictions is an open research question.

We restricted attention to the network links, which, as stated in the introduction, correspond to about 64% of a (baseline) switch's power consumption. After obtaining significant energy savings in the links, it is necessary to reduce the energy consumption in the remaining parts of the switch and

NIC. Other components, such as the memory elements, which are dominant in the HCA, excluding the links, have larger reactivation times that can cause significant performance penalties. We would like to show, however, that the intelligent power saving mechanisms presented in this thesis could also be beneficial in this context.

Chapter 9

Publications

- B. Dickov, P. M. Carpenter, M. Pericàs, E. Ayguadé. Self-Tuned Software-Managed Energy Reduction in InfiniBand Links. The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2015.

- B. Dickov, M. Pericàs, P. M. Carpenter, N. Navarro, E. Ayguadé. Software-Managed Power Reduction in InfiniBand Links. The 43rd Annual Conference International Conference on Parallel Processing (ICPP), 2014.

- B. Dickov, M. Pericàs, P. M. Carpenter, N. Navarro, E. Ayguadé. Analyzing Performance Improvements and Energy Savings in Infiniband Architecture using Network Compression. 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2014.

- B. Dickov, M. Pericàs, G. Houzeaux, N. Navarro, E. Ayguadé. Assesing The Impact of Network Compression on Molecular Dynamics and Finite Elements Methods. The 14th IEEE International Conference on High Performance Computing and Communications (HPCC),

2012.

- B. Dickov, M. Pericàs, N. Navarro, E. Ayguadé. Row-Interleaved streaming data flow implementation of Sparse Matrix Vector Multiplication. 4th HIPEAC Workshop on Reconfigurable Computing, 2010.

- B. Dickov, M. Pericàs, N. Navarro, E. Ayguadé. Mapping Sparse Matrix-Vector Multiplication (SMVM) on FPGA – Reconfigurable Supercomputing, HIPEAC ACACES summer school, 2009.

References

[1] (1994). MPI forum: MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 165–414.

[2] (1999). pack_doubles.c. http://chips.ncsu.edu/~luw/version2.1.5/HTML_SOURCE/pack_doubles.c.html.

[3] (2011). Alya system - large scale computational mechanics. http://www.bsc.es/plantillaA.php?cat_id=552.

[4] (2011). asci. http://www.llnl.gov/asci/purple.

[5] (2011). bzip2. http://www.bzip.org/.

[6] (2011). Collaborative project: Multi-precision floating point library. http://www.mpfr.org.

[7] (2011). Gromacs(GROningen MAchine for chemical simulations). http://www.gromacs.org.

[8] (2011). gzip. http://www.gzip.org/.

[9] (2011). Ibm. ibm infiniband 8-port 12x switch. http://www-3.ibm.com/chips/products/infiniband.

[10] (2012). Family of graph and hypergraph partitioning software. http://glaros.dtc.umn.edu/gkhome/views/metis.

[11] (2012). Intel® mpi benchmarks. http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[12] (2013). Power savings features in mellonox products. http://www.mellanox.com/related-docs/whitepapers/.

[13] (2014). Nas parallel benchmarks 3.3. http://www.nas.nasa.gov/Resources/Software/npb.html.

[14] (2014). Performance tools. http://www.bsc.es/computer-sciences/performance-tools/.

[15] Abts, D., Marty, M. R., Wells, P. M., Klausler, P., & Liu, H. (2010). Energy proportional datacenter networks. *SIGARCH Comput. Archit. News*, 38(3), 338–347.

[16] Alawneh, L. & Hamou-Lhadj, A. (2011). Pattern recognition techniques applied to the abstraction of traces of inter-process communication. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on* (pp. 211–220).

[17] Alonso, M., Coll, S., Martínez, J.-M., Santonja, V., López, P., & Duato, J. (2010). Power saving in regular interconnection networks. *Parallel Comput.*, 36(12), 696–712.

[18] Baileya, D., Harris, T., Saphir, W., v.d. Wijngaart, R., A.Woo, & Yarrow, M. (1995). *The NAS Parallel Benchmarks 2.0*. Technical Report NAS-95-020, NASA Ames Research Center.

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

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

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

[22] Burtscher, M. & Ratanaworabhan, P. (2007). High throughput compression of double-precision floating-point data. In *DCC* (pp. 293–302).: IEEE Computer Society.

[23] Burtscher, M. & Ratanaworabhan, P. (2009). FPC:a high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computer*, 58(1), 18–31.

[24] Car, R. & Parrinello, M. (1985). Unified approach for molecular dynamics and density-functional theory. *Phys. Rev. Lett.*, 55, 2471–2474.

[25] Cortes, T., Becerra, A., & Cervera, R. (2000). Swap compression: resurrecting old ideas. *Software, Practice and Experience*.

[26] Das, R., K.Mishra, A., Nicopoulos, C., Park, D., Narayanan, V., Iyer, R., Yousif, M. S., & Das, C. R. (2008). Performance and power optimization through data compression in network-on-chip architectures. In *HPCA*.

[27] Deutsch, P. (1996). Deflate compressed data format specification version 1.3.

[28] Dickov, B., Pericàs, M., Carpenter, P., Navarro, N., & Ayguadé, E. (2014). Software-managed power reduction in infiniband links. In *Parallel Processing (ICPP), 2014 43rd International Conference on* (pp. 311–320).: IEEE.

[29] Dickov, B., Pericas, M., Houzeaux, G., Navarro, N., & Ayguade, E. (2012). Assessing the impact of network compression on molecular dynamics and finite element methods. In *HPCC* (pp. 588 –597).

[30] Filgueira, R., Singh, D. E., Calderón, A., & Carretero, J. (2009). CoMPI:enhancing mpi based applications performance and scalability using run-time compression. In *EUROPVM/MPI* Espoo, Finland.

[31] Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G. L., Cococcioni, M., Dabo, I., Dal Corso, A., de Gironcoli, S., Fabris, S., Fratesi, G., Gebauer, R., Gerstmann, U., Gougoussis, C., Kokalj, A., Lazzeri, M., Martin-Samos, L., Marzari, N., Mauri, F., Mazzarello, R., Paolini, S., Pasquarello, A., Paulatto, L., Sbraccia, C., Scandolo, S., Sclauzero, G., Seitsonen, A. P., Smogunov, A., Umari, P., & Wentzcovitch, R. M. (2009). Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39), 395502 (19pp).

[32] Gibbon, P. (2003). *PEPC : Pretty Efficient Parallel Coulomb-solver*. Sonstiger Interner Bericht ZAM-IB-2003-05, ZAM, Jülich, Forschungszentrum.

[33] Hanson, T. D. (2013). *uthash*: A hash table for c structures. http://troydhanson.github.io/uthash/.

[34] 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.

[35] Hoefler, T. (2010). Software and hardware techniques for power-efficient hpc networking. *Computing in Science Engineering*, 12(6), 30–37.

[36] Huang, Y.-C., Chou, K.-K., & King, C.-T. (2013). Application-driven end-to-end traffic predictions for low power noc design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(2), 229–238.

[37] Huffman, D. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9), 1098–1101.

[38] Katahira, K., Sano, K., & Yamamoto, S. (2010). FPGA-based lossless compressors of floating-point data streams to enhance memory bandwidth. In *ASAP*.

[39] Ke, J., Burtscher, M., & Speight, E. (2004). Runtime compression of mpi messages to improve the performance and scalability of parallel applications. SC '04 (pp. 59–). Washington, DC, USA.

[40] Kim, E. J., Yum, K. H., Link, G. M., Vijaykrishnan, N., Kandemir, M., Irwin, M. J., Yousif, M., & Das, C. R. (2003). Energy optimization techniques in cluster interconnects. In *ISLPED*.

[41] Kumar, V. S., Nanjundiah, R., Thazhuthaveetil, M., & Govindarajan, R. (2008). Impact of message compression on the scalability of an atmospheric modeling application on clusters. *Parallel Computing*, 34(1), 1–16.

[42] Labarta, J., Girona, S., Pillet, V., Cortes, T., & Gregoris, L. (1996). Dip: A parallel program development environment. In *Euro-Par'96 Parallel Processing* (pp. 665–674). Springer Berlin Heidelberg.

[43] Li, F., Chen, G., Kandemir, M., & Karakoy, M. (2005). Exploiting last idle periods of links for network power management. In *EMSOFT* (pp. 134–137). New York, NY, USA: ACM.

[44] Li, J., Huang, W., Lefurgy, C., Zhang, L., Denzel, W., Treumann, R., & Wang, K. (2011). Power shifting in thrifty interconnection network. In *HPCA*.

[45] Lim, M. Y., Freeh, V. W., & Lowenthal, D. K. (2006). Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *SC* New York, NY.

[46] Lopes, A. R., Shahzad, A., Constantinides, G. A., & Kerrigan, E. C. (2009). More flops or more precision? Accurancy parametrizable linear equation solvers for model predictive control. In *Proceedings of the 17th IEEE International Symposium on Field-Programmable Custom Computing Machines* NAPA, CA, USA.

[47] Michalakes, J. (2004). The weather research and forecast model: Software architecture and performance. In *In proceedings of the 11th ECMWF Workshop on the Use of HPC In Meteorology,*.

[48] 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.

[49] 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* ICST, Brussels, Belgium.

[50] Patel, N. & Mundur, P. (2005). An n-gram based approach for finding the repeating patterns in musical data. In *EuroIMSA* Grindelwald, Switzerland.

[51] 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.

[52] P.M.Kogge (2008). Architectural challenges at the exascale frontier(invited talk). *Simulating the Future: Using One Million Cores and Beyond*.

[53] Sagui, C. & Darden, T. A. (1999). Molecular dynamics simulations of biomolecules: long-range electrostatic effects. In *Annu Rev Biophys Biomol Struct*, volume 28 (pp. 155–179).

[54] Saravanan, K. P., Carpenter, P. M., & Ramirez, A. (2014). A performance perspective on energy efficient hpc links. In *Proceedings of the 28th ACM international conference on Supercomputing* (pp. 313–322).: ACM.

[55] Sathish, V., Schulte, M. J., & Kim, N. S. (2012). Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12 (pp. 325–334).: ACM.

[56] Shang, L., Peh, L.-S., & Jha, N. K. (2003). Dynamic voltage scaling with links for power optimization of interconnection networks. In *HPCA*.

[57] Shewchuk, J. R. (1994). *An Introduction to the conjugate Gradient Method Without Agonizing Pain*. Technical report, School of Computer Science, Carnegie Mellon University.

[58] Soteriou, V., Eisley, N., & Peh, L.-S. (2007). Software-directed power-aware interconnection networks. *ACM Trans. Archit. Code Optim.*, 4(1).

[59] Soteriou, V. & Peh, L.-S. (2004). Design-space exploration of power-aware on/off interconnection networks. In *ICCD* (pp. 510 – 517).

[60] Tomari, H., Inaba, M., & Hiraki, K. (2010). Compressing floating-point number stream for numerical applications. In *Proceedings of the 1st IEEE International Conffference on Networking and Computing* (pp. 112–119). Hiroshima, Japan.

[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] Wang, H.-S., Peh, L.-S., & Malik, S. (2002). A power model for routers: modeling alpha 21364 and infiniband routers. In *High Performance Interconnects, 2002. Proceedings. 10th Symposium on* (pp. 21–27).

[63] Yongpeng, L., Hong, Z., Yongyan, L., Feng, W., & Baohua, F. (2011). Parallel compression checkpointing for socket-level heterogeneous systems. In *HPCC*.

[64] Zheng, J. & Lonardi, S. (2005). Discovery of repetitive patterns in DNA with accurate boundaries. In *Fifth IEEE Symposium on Bioinformatics and Bioengineering (BIBE)* (pp. 105–112).

[65] Ziv, J. & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3), 337–343.