TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik

Resource-Elasticity Support for Distributed Memory HPC Applications

Isaías Alberto Comprés Ureña

Vollständiger Abdruck von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigten Dissertation.

Vorsitzender: Prof. Bernd Brügge, Ph.D.

Prüfende der Dissertation:
1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Michael Georg Bader

Die Dissertation wurde am 23.06.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2017 angenommen.

TECHNICAL UNIVERSITY OF MUNICH

Dissertation

Resource-Elasticity Support for Distributed Memory HPC Applications

Author:	Isaías Alberto Comprés Ureña
First examiner:	Prof. Dr. Hans Michael Gerndt
Second examiner:	Prof. Dr. Michael Georg Bader

The dissertation was submitted to the Technical University of Munich on 23.06.2017, and was approved by the Faculty of Informatics on 12.07.2017.

I hereby declare that this thesis is entirely the result of my own work except where other-wise indicated. I have only used the resources given in the list of references.

Garching, 5.5.2017 Isaías Alberto Comprés Ureña

Acknowledgments

First, I want to thank to Prof. Gerndt. It was because of a recommendation of his that I originally had the opportunity to engage in message passing research at a reputable research institution. He later gave me the opportunity to pursue this doctorate, with an expanded scope that includes resource management and scheduling. In addition, the quality of this work has largely improved thanks to his diligent supervision and advice.

I would also like to thank the people in my academic environment. To all my colleagues that provided me with new ideas to consider, I am forever grateful. To the staff of the Technical University of Munich, for providing a great environment for work and research. To the Leibniz Supercomputing Center, for granting me access to the supercomputing resources needed for this type of research. Finally, to the Invasive Computing Transregional Collaborative Research Center for providing the theoretical background and necessary funding for this work.

I would also like to take this opportunity to thank all my friends and relatives, in no particular order, who have directly or indirectly positively influenced my life. I would like to express my gratitude to Manuel and Gloria Cocco, who helped me during moments of adversity. I am thankful to my mother Yvette Ureña, whose lifelong interest in my well being has no parallels. I also want to thank my uncle Miguel Ramón Ureña for his constant advice and support. Finally, I want to express gratitude to my aunt Miguelina Ureña, who has helped me in many ways over the years.

Abstract

Computer simulations are alternatives to the scientific method in domains where physical experiments are unfeasible or impossible. When the amount of memory and processing speed required is large, simulations are executed in distributed memory High Performance Computing (HPC) systems. These systems are usually shared among its users.

A resource manager with a batch scheduler is used to fairly and efficiently share the resources of these systems among its users. Current large HPC systems have thousands of compute nodes connected over a high-performance network. Users submit batch job descriptions where the number of resources required by their simulations are specified. Batch job descriptions are queued and scheduled based on priorities and submission times.

The parallel efficiency of a simulation depends on the number of resources allocated to it. It is challenging for users to specify allocation sizes that produce adequate parallel efficiencies. A resource allocation can be too small and the parallel efficiency of the application may be adequate, but its performance may not be scaled to its maximum potential. A resource allocation can be too large and therefore the parallel efficiency of the application may be degraded due to synchronization overheads. Unfortunately, in current systems these resource allocations cannot be adapted once the applications of a job start.

A resource manager and MPI library combination that adds resource-elasticity support for HPC applications is proposed in this work. The resource manager is extended with operations to adapt the resources of running applications in jobs; in addition, new scheduling techniques are added to it. The MPI library has been extended with operations that enable resource adaptations as changes in the number of processes in world communicators. The goal is to optimize system-wide efficiency metrics through adjustments to the resource allocations of running applications. Resource allocations are adjusted continuously based on performance feedback from running applications.

Contents

List of Figures

1 Introduction

Large High Performance Computing (HPC) systems are essential tools for multiple research areas today. These systems can require large amounts of funds for their initial purchase as well as long term maintenance. Due to the costs, these systems are usually shared among a large number of scientists and professionals from multiple institutions.

A resource manager with a scheduler is typically used to share the resources of an HPC system across the compute jobs submitted by its users. Schedulers can operate in time-sharing or space-sharing mode. In time-sharing mode, the tasks of multiple jobs can get simultaneous access to resources. Current schedulers typically operate in space-sharing mode; in this mode, jobs get exclusive access to their allocated resources for the entirety of their execution time. Space-sharing mode provides an environment with predictable performance for applications that run as part of a job, and has been an adequate solution up to the present time.

Near future HPC systems are expected to reach exaFLOPS of performance. Due to current trends in computer hardware, this requires the aggregation of ever greater numbers of nodes that have internally ever greater amounts of hardware parallelism. Along with the increase of parallelism of HPC hardware, distributed system software as well as user software will need to be updated to make efficient use of the increased number of individual hardware resources.

Future applications are expected to become more dynamic. For example, their processing requirements will vary at runtime with the use of Adaptive Mesh Refinement (AMR) methods. These applications have varying amounts of available parallelism at runtime, due to the number of elements in their meshes being altered. Since refinements occur in a distributed manner, it can also lead to load imbalances.

Current applications that have limited available parallelism will also pose challenges, even if their available parallelism is constant during runs. Strong scaling applications where their input determines their computational domain (size and geometry) can receive resource allocations that result in low parallel efficiency. It is difficult to predict good resource allocations for applications before collecting enough empirical data at a wide range of processing resource counts. In many cases, these applications will run only once per input set. Applications may also be composed of multiple phases, where each phase has its own available parallelism.

Parallel applications operate at different efficiencies depending on their current available parallelism and their current resource allocation. Inadequate resource allocations may lower the efficiency of applications. This work addresses efficiency losses due to inadequate resource allocations by introducing a resource-elastic execution environment for distributed memory HPC systems. The parallel efficiency of applications is estimated with a performance model and their resources are periodically increased or reduced based on a new heuristic.

The resources allocated to a job can be adjusted at runtime in a resource-elastic execution environment. Resource counts may be adjusted to prevent low parallel efficiency in individual applications, if it can be detected. In the case of applications that use AMR methods, resources can be adjusted based on updated mesh element counts and communication patterns. Strong scaling applications can have their resource allocations adjusted based on estimations to their parallel efficiency. The adjustment of resources can also produce a surplus of resources that can be allocated to waiting jobs.

The proposed solution is divided into two main parts: a communication library and a resource manager. The goal is to improve efficiency mainly at the system level, while providing improvements to individual applications as much as possible. The resource manager and the communication library interact to adjust the resources allocated to running applications in order to improve their efficiency. The adjustment is a continuous activity during the runtime of jobs in a distributed memory system.

The communication library supports the Message Passing Interface (MPI) standard. The standard has preexisting support for expansions of resources in applications. This can be achieved through the use of the dynamic processes operations introduced in version 2 of the standard. Due to the performance costs and limitations of these operations, these have not been widely used by application developers. A notable limitation is that resources cannot be reduced.

An extension to the MPI standard is proposed in this work. It consists of four new operations. These operations allow for the dynamic modification of the number of processes of an application, matching any changes in its resources at runtime. These new operations differ in the way they allow resource adaptations, when compared to the ones provided by the standard dynamic processes support in MPI. The core new concept is the creation of adaptation windows, where resources are incorporated or removed from distributed applications. Resources are abstracted as processes in the `MPI_COMM_WORLD` communicator. Additionally, the adaptations are initiated by the resource manager and not the MPI application. Adaptation windows are defined by a begin and a commit operation (similarly to transactions) and cannot be nested.

A short description of the four proposed operations is provided here:

- `MPI_INIT_ADAPT`: Initializes the library in adaptive mode and indicates the status of the local process: new or joining. A process is new when it is created as part of the initial application launch, or joining when created as part of an expansion by a resource manager.

- `MPI_PROBE_ADAPT`: Indicates whether the application is required to adapt. If an adaptation is required, it also provides the status of the preexisting process. This status can be staying or leaving.

- `MPI_COMM_ADAPT_BEGIN`: Begins the adaptation window. This operation provides helper communicators that enable communication between preexisting and newly added process groups. It also provides additional information to aid repartitioning schemes.

- `MPI_COMM_ADAPT_COMMIT`: Completes the adaptation window. All staying preexisting processes and all joining processes become members of the `MPI_COMM_WORLD` communicator after this operation, while any leaving preexisting processes are removed.

Malleable applications are those that can have their computing resources adjusted at runtime. These operations can be used to create new malleable MPI applications or to convert existing ones.

The design of the operations in the MPI extension allows MPI library implementations to hide all latencies related to resource management and the creation of new processes from preexisting processes. Preexisting processes are only notified about adaptations once new processes are ready and blocking in the begin operation.

The resource manager applies new resource allocations to running applications through a reallocation message. Each application is sent a reallocation message and its status is changed from running to adapting, in its job metadata. A six step adaptation sequence is started with this message. In the final step, the application notifies back the resource manager when its adaptation has been completed. The resource manager then updates the status of the job back to running, from adapting. This is done to prevent the nesting of adaptations, since nesting is currently not supported by the design.

Sets of applications can have their resources adapted with the use of multiple adaptation messages simultaneously. There is a time window where processes from two applications may reside in individual nodes. This occurs when resources are being moved from one application to another. Once both applications complete their adaptation, exclusive access to resources is again ensured. In summary, exclusive access to resources is guaranteed after resource adaptations are completed, but not during adaptations.

Only things that can be measured or estimated can be managed. Some means to estimate the efficiency of applications at their current resource allocations was needed. Without any estimation on how efficiently applications are performing with specific resource allocations, scheduling decisions that alter resource counts would be of little value or detrimental to performance. An estimation on the expected change in efficiency of an application given a new resource allocation is necessary to support resource-elastic scheduling.

The resource manager and the communication library interact periodically during elastic application runs. During these interactions, performance data is collected and used to generate performance models. These models are used to estimate the efficiency at which each application is operating, given its current resource allocation. If the efficiency is estimated to be low, the resources in its allocation are reduced. If its efficiency is estimated to be high, the resources in its allocation may be preserved or increased, based on a heuristic.

Currently only one performance model has been developed: the SPMD-Phase model. As its name suggests, it only supports Single-Program Multiple-Data (SPMD) distributed patterns. These SPMD phases need to be first identified in distributed applications. These distributed patterns are detected by first building a partial Control Flow Graph (CFG) of the application at each individual MPI process. The algorithm updates a partial CFG as the application runs. The CFG is only complete when the application terminates; therefore, the detection is always operating on incomplete CFGs during the runtime of applications.

The generation of the CFGs relies on unique MPI call site markings that are introduced during compilation with the use of the MPI compiler wrappers. These markers eliminate the need of backtracing at runtime. Backtracing is a source of significant CFG detection overheads. Previous CFG detection solutions rely on backtracing.

The first step to generate the model is to identify loops in the partial CFG of each process of the distributed application. These are then matched, reduced and annotated with the collected performance data. In the SPMD-Phase model, the performance data is used to compute the proportion of MPI to compute time of the distributed loop. This proportion is then used to produce a range of possible resource allocation sizes for the application, where its estimated parallel efficiency is predicted to either improve or remain stable. This is done for each application in a set of candidates for resource adaptations, and a Resource Range Vector (RRV) is produced. The SPMD-Phase model is applied per application and produces the entries of the vector individually. New models are expected to be added in the future for different parallel execution patterns.

The proposed scheduling heuristic takes as input the RRV. Its output is a Resource Scaling Vector (RSV) of concrete resource counts to be applied to the set of running elastic applications considered. It is assumed that the ranges produced by the performance model do not degrade the efficiency of the applications significantly. Additionally, the performance of the applications are assumed to scale linearly within the supplied ranges. In other words, the assumption is that the parallel efficiency of the application improves or remains similar within the provided ranges.

The scheduler makes resource adaptation decisions on sets of applications in order to improve system-wide performance metrics. The performance of individual applications may also be improved in the process. The interactions between the resource manager and the MPI library are only done if applications are resource-elastic. Applications that are not resource-elastic are simply ignored.

The new scheduling heuristic is an extension to the current batch scheduling and backfilling approach found in most distributed schedulers. Its implementation is split into two software components: the Elastic Batch Scheduler (EBS) and the Elastic Runtime Scheduler (ERS). The EBS was not implemented in time to be presented in this document; however, its role in the current design has been determined and is described.

The EBS will implement moldable batch scheduling. This type of scheduling is well understood and already used in related resource-elastic scheduling research. The resources at which jobs start are flexible with this type of scheduling. Moldable batch scheduling is not concerned with what happens to resource allocations after applications start.

The ERS implements what is referred to as elastic backfilling in this document. Elastic backfilling consists of resource adaptation operations that can be applied to minimize idle node counts, while improving the overall estimated efficiency of the system. The trade-off between estimated parallel efficiency and the number of idle node counts generated by the technique can be configured by setting thresholds. The generation of idle nodes by resource adaptations prioritizes the start of new jobs and benefits the estimated efficiency of the system. The alternative is to trade estimated efficiency for lower idle node counts and relative increases in job queues.

The elastic backfilling heuristic implementation in the ERS contains a shim that replaces its interaction with the missing EBS. In the design, the expectation is that the ERS will receive new jobs held by the EBS before it produces the final Resource Scaling Vector (RSV) from the Resource Range Section (RRV) produced with the SPMD-Phase model. This will give priority to the start of new jobs over the expansion of running ones.

The elastic backfilling heuristic applies a combination of two new operations to the set of candidate jobs: time balancing and resource filling. The time balancing operation takes a subset of the candidate jobs and attempts to balance their total runtime by adjusting their resources. This operation can be used to minimize the makespan in the current prototype, and to reduce the wait time of jobs with reservations in the future, when the EBS is introduced and its shim removed. The resource filling operation is used to fill any surplus of nodes. This operation can be combined with regular backfilling techniques to minimize idle node counts, again, once the EBS is introduced.

A new programming model is also proposed in this work: Elastic-Phase Oriented Programming (EPOP). Its goal is to simplify the development of elastic distributed memory applications. This model improves the structure and modularity of elastic applications by requiring that their work be defined as a collection of phases and control structures.

The CFG of applications is defined by developers in this model; this eliminates the need of CFG detection and its related overhead. The actual execution of these applications is controlled by a driver program. Driver programs can have different purposes, such as debugging. Different driver programs can be selected at launch time without modifying the applications.

In summary, the contributions in this work add resource-elasticity support for distributed memory applications in current HPC systems. Support for resource-elasticity requires changes to multiple parts of the software stack of a parallel system, such as: programming models, runtime systems, resource managers and schedulers. Because of this, the contributions presented in this document are related to multiple research areas of computer science. More specifically, the contributions of this work can be enumerated as follows:

1. MPI library with an extension for improved dynamic processes support.
2. Resource manager with support for resource-elasticity.
3. MPI library and resource manager integration for continuous interaction.
4. Programming model for distributed memory resource-elastic applications.
5. Measurement and modeling infrastructure for resource-elastic applications.
6. Scheduling heuristic to optimize systems with resource-elasticity support.

This document continues with the motivation and the related work. It then follows with topics related to the message passing programming model. Resource management and scheduling contributions are described afterwards. The document is then completed with the evaluation and closing chapters.

The content is organized in chapters. A set of application types and their scalability with resources are presented as motivation in Chap. 2. In the same chapter, the benefits of resource-elasticity for system-wide performance are summarized and related work cited. This work was done within the transregional Invasive Computing project. The scope and organization of this research project is described briefly in Chap. 3. A detailed discussion on related programming models and resource management research is presented in Chap. 4. The document continues with the chapters related to message passing. Chapter 5 provides an overview of the MPI standard and its implementation within the MPICH library. MPICH was used as basis for the communication library prototype. Afterwards, Chap. 6 describes the proposed MPI extension and its implementation. A new resource-elastic programming model is introduced in Chap. 7: Elastic-Phase Oriented Programming (EPOP). The document continues with the chapters about resource management. Chapter 8 provides a brief introduction to resource management in general. It also provides an overview of the SLURM workload manager, since it is used as basis for the resource manager prototype presented in this document. The new resource manager is presented in Chap. 9. Its interaction with the communication library when performing resource adaptations is described in detail. The design and implementation of the new measurement, modeling and scheduling infrastructure is covered in Chap. 10. In Chapters 11 through 14, the testing methodology is described and evaluation results for the MPI library, the resource manager and a selection of applications are presented. Finally, the document ends with the conclusion in Chap. 15 and a discussion about possible future work in Chap. 16.

2 Motivation

Research and engineering efforts today are conducted in several possible ways. Purely experimental research (using the scientific method) is done when possible. For example, a company can evaluate the response of a new material to changes in temperature in a controlled environment with acceptable accuracy. However, not all research teams have that privilege. In many other cases experiments are impractical, expensive or simply impossible. Consider the case of an environmental team evaluating the impact of some substance in an ecosystem: they could pollute large parts of a forest with a new chemical and then evaluate its impact in the local fauna. Needless to say, this would be unethical and could have permanent detrimental effects in the forest. Another example could be a company in the automotive industry. Such a company may have multiple teams working in on several possible designs for a new engine, but may not afford to create an individual prototype for each possibility. Finally, an experiment may simply be impossible, such as the analysis of the effect of an earthquake of a specific strength in a city where this has never happened.

Computer simulations are an alternative to real experiments. Special care needs to be taken when creating models and simulation code. If successful, simulations can provide accurate results that correlate reliably to reality. Simulations can help companies save on experimental and prototyping costs, by replacing parts of their experiments with simulations. For example, in the automotive company with multiple engine designs scenario mentioned before, the company could simulate all candidate designs before producing physical prototypes. The number of prototypes can then be reduced by discarding less efficient designs, greatly reducing its costs. Simulations can also allow otherwise impossible research to take place, such as the earthquake scenario mentioned before.

The time to solution of a simulation is closely related to its usefulness in many cases. For example, a weather forecasting station needs to produce forecasts before individuals can simply look at the sky. This requirement, together with the processing capabilities of the system and the performance properties of the simulation code, will determine the performance level required to meet deadlines.

Simulations vary greatly in terms of their computational requirements and the required accuracy of their results. For many simulations, a desktop computer or workstation is sufficient. In other cases, when the performance required far exceeds what is available in a single workstation, a distributed memory system is required. In distributed memory systems, the performance of thousands of compute elements can be aggregated. Instead of workstations, the compute elements are divided in nodes and assembled in racks. The definition of a node will vary depending of the vendor that provides the distributed memory system. If the simulation, in addition to large compute requirements, also has strong synchronization requirements, then specialized communication networks are used to interconnect the nodes. A large set of nodes interconnected by a high performance network is today referred to as a supercomputer or a High Performance Computing (HPC) system.

Even small supercomputers can have significant costs, both for the initial purchase and its maintenance during its service time. Costs are one of the reasons why these systems tend to be shared among several users and even several research institutions. A resource manager with a scheduler is necessary to share the resources of these systems fairly and efficiently. Most schedulers today operate in space sharing mode; this means that resources are given exclusively to a job during its complete run, and are only released for other jobs when it completes.

Exclusive access to resources has given applications stable and predictable performance. This solution has been adequate given the scale of current systems and the static nature of most algorithms used in contemporary simulations. This is expected to change as systems continue to grow and applications become more dynamic.

Allocations can often impact the efficiency of simulations negatively. In this work, an allocation is defined as inadequate when it lowers any efficiency metric significantly. The allocation may be inadequate for the entirety of the run of a job, or temporarily.

Elastic execution is proposed as a solution to inadequate allocations, since allocations can be adjusted while applications are running to improve their efficiency metrics. In addition to justifying this research as a fix to inadequate allocations of individual jobs, elastic execution can also improve system-wide efficiency metrics with a mix of elastic and static applications. For example, jobs can be allocated extra nodes from an idle pool; this minimizes the idle node count metric and is only possible with elastic execution.

In the remainder of this chapter, an incomplete set of scenarios where allocations are often inadequate will be introduced. The focus is on parallel efficiency as the main metric, since it is usually negatively affected on inadequate allocations and is often the main objective of optimizations. The relationship between the available parallelism of an application phase and the range of possible adequate resource allocations will be made. This motivates the need for identifying the phases of applications and their available parallelism. Finally, a short discussion about the benefits of resource-elasticity to system-wide efficiency is included, together with references to related work. In summary, resource-elasticity can provide efficiency increases to HPC systems and individual applications.

2.1 Adaptive Mesh Refinement (AMR) Methods

Adaptive Mesh Refinement (AMR) techniques are widely used in scientific applications today. Applications that use these methods exhibit a tendency to generate load imbalances. Additionally, their scalability properties change as a function of the aggregated increase or decrease of primitives in their geometry because of any refinement or coarsening operation applied to their meshes.

Load imbalances are generally well handled today, while changes in scalability properties are not. Load balancing is achieved by application specific techniques. Changes in scalability properties pose a larger challenge because the application, the communication library and the resource manager need to support resource adaptations. A software stack that allows for resource adaptations can more efficiently support AMR applications, since their allocations can be adjusted based on their dynamic scalability to ensure acceptable parallel efficiency throughout their runtime.

Algorithms used to solve linear systems iteratively rely on approximations of their error, usually referred to as the residual, in order to determine the quality of the approximation at each iteration and determine a termination condition. The subset of these algorithms that perform mesh refinements rely on a way to approximate their error at different refinement levels. In most simulations, the main program performs approximations to the solution of a system of partial differential equations per time step. Error estimations can be computed

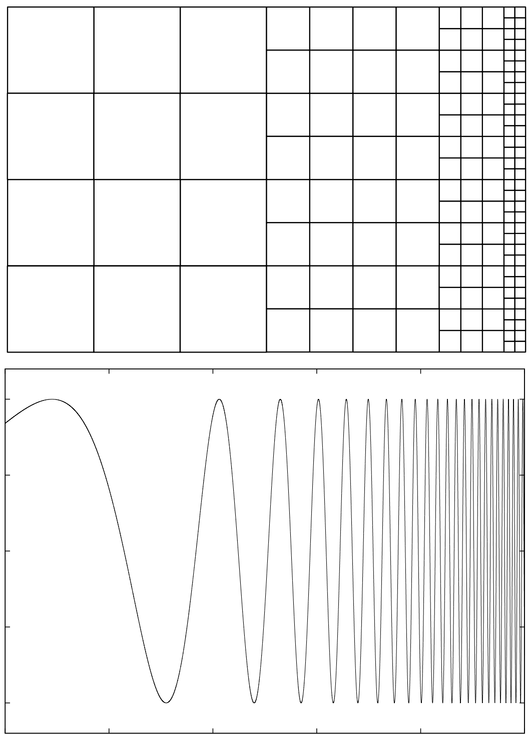

Figure 2.1: Example 2D mesh refinement for the $z = \sin(e^x)$ surface at the $x = y = 0$ plane (top) with plots for the function at $y = 0$ (bottom).

in subdomains, to determine where to refine the mesh. This is much better than refining the full mesh, since finer meshes require more processing and their higher resolution is only of benefit in the submeshes where the error is estimated to be high. The technique can be better understood with an illustration. Figure 2.1 shows a refined mesh produced by a numerical quadrature algorithm for a particular integrand function. As can be seen in the images, the AMR algorithm refines the 2D mesh proportionally to the spatial frequency of the integrand. Note that the refinement divides each element in the mesh into four new elements, although the increased resolution of only one dimension improves the accuracy of the integration in this case.

2.1.1 Challenges of AMR Methods in Distributed Memory Systems

As described before, the use of AMR methods can offer efficiency gains to applications. Unfortunately, there are also some challenges that arise as a result of the use of these methods. In this section, the challenges that are specific to applications that rely on AMR methods and their execution in distributed memory systems will be discussed. These are load imbalance due to process-local refinements and variable scalability due to changes in the total available parallelism of applications.

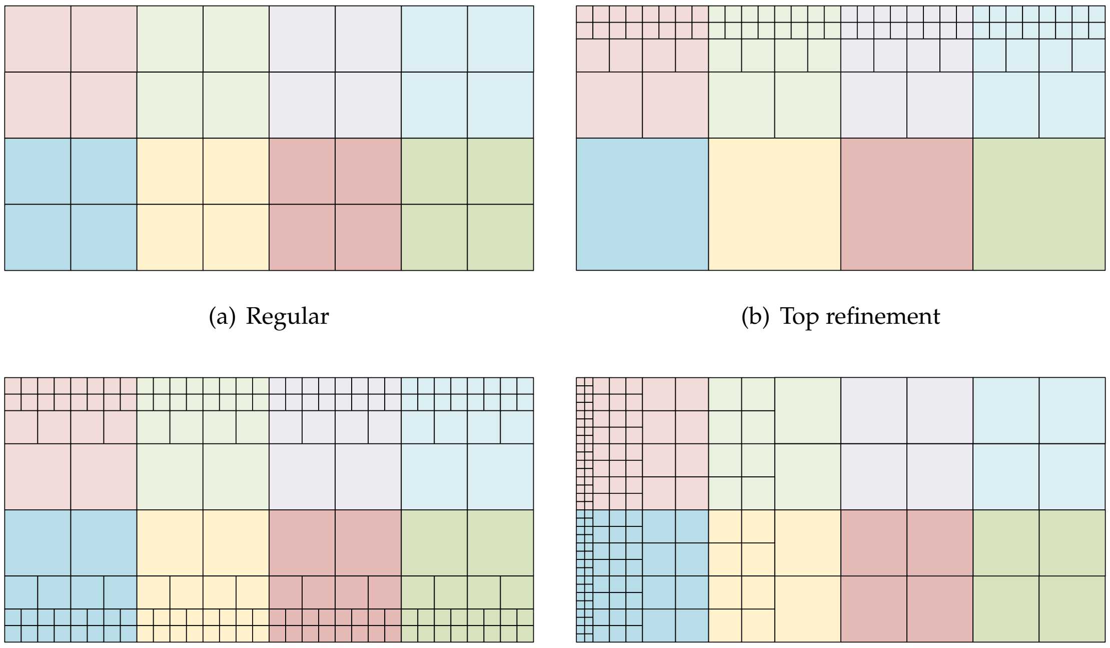
(a) Regular (b) Top refinement (c) Bottom and top refinement (d) Left refinement

Figure 2.2: Meshes with different color for each submesh distributed among 8 processes.

Local Refinements and Load Imbalance

When executing in distributed memory systems, mesh refinements are performed in a partitioned domain. Each participating process owns a different subdomain, and therefore will determine different refinements. The refinements can vary greatly given the amount of symmetry in the domain, initial conditions, the equations involved, among other things.

These differences in the amount of refinement at each of the processes leads to load imbalances. This has been solved in static allocations by having application specific redistribution schemes. An elastic execution model is not a solution to the load balancing problem; however, local refinements also result in variable available parallelism.

Variable Available Parallelism and Scalability

The aggregated number of the refinements performed at each process of a distributed application using AMR changes the scalability of the application. A net increase of the number of points in the mesh will increase the total available parallelism, while a net reduction of the number of points on the distributed mesh will decrease the total available parallelism. In applications that use AMR methods, the available parallelism varies in time.

The available parallelism of an application will determine the amount of resources that it can use at peak or acceptable efficiency, given the relation between its computational and communication requirements. This means that, in order to operate at high efficiency, the resources of AMR applications need to be actively adjusted based on the results of their refinement or coarsening operations. Moreover, in most simulations the refinements produce different results depending on the current time step of the algorithm; therefore, adaptations need to be continuous in time.

A set of examples help illustrate possible situations that an application can run into. Figure 2.2 shows four meshes that are divided into 8 submeshes of equal dimensions. Such a situation can arise on a simulation that is running in a distributed manner on 8 processes. The number of elements in each processor is proportional to the amount of computations its solver must perform. In the figure, meshes `(a)` and `(c)` are load balanced, while `(b)` and `(d)` are not. In addition to this, if `(a)` is the starting point of the algorithm and `(b)`, `(c)` and `(d)` are possible outcomes of the AMR algorithm, the total amount of computation in the distributed application increases and therefore changes its available parallelism. The change in available parallelism may render resource allocations inadequate. Because of this, application specific redistribution schemes developed for elastic execution can benefit greatly from performing load balancing together with resource adaptations.

Elastic execution is proposed as a solution to the loss of parallel efficiency due to the variability of available parallelism produced by AMR methods in distributed memory applications. Elastic execution allows for the adjustment of resources allocated to applications based on their available parallelism as a continuous activity during runs.

2.2 Applications with Multiple Computational Phases

Most applications are seen intuitively by developers as having multiple phases. For example, an application can be seen as having an initialization phase, a computational phase and a finalization phase.

During initialization, an application will most likely perform IO operations to read all of its input files. It will then setup its data structures in a distributed manner, before any computation takes place.

During computation, if the application is a simulation, it will likely perform multiple time steps until the simulation is done. Within each time step, multiple approximations of linear systems of differential equations may be computed as necessary; each of these separate approximations, that could use separate specialized solvers, can be seen themselves as separate phases.

Finally, during a finalization phase, the application may generate any output files as necessary, such as visualization, result files, etc. The application may also free any memory related to its data structures and close any file descriptors.

These different phases of applications tend to have different performance characteristics. In most cases, the initialization and finalization phases do not take significant amounts of time and therefore are not targets for optimization. However, any phases that appear in the computational parts will be very significant to the total run time of the application. Because of their difference in parallel efficiency, these phases may perform better with different allocations of compute resources.

In this section, the variability of available parallelism in phases is described. Overheads related to adaptations of resources are not considered yet; the focus is instead on the performance effects given instantaneous resource adaptations. Most of the common types of phases are classified in terms of their scalability with resources. The benefits that elastic execution can provide to them are stated.

A distinction is made between phases that generate more work as their resources are increased, versus those that keep their amount of work constant regardless. The former type is known as weak scaling, while the later type is known as strong scaling. The performance of strong scaling phases can be optimized by adjusting resources via elastic execution. Their parallel efficiency is a function of the resources that they use for processing. In contrast, weak scaling phases will simply increase the amount of work they do per computational unit, as more resources are provided to them. Because of this, in this work only strong scaling phases are considered.

2.2.1 Phases with Different Scalability Properties

The scalability of distributed compute phases is a measure of efficiency based on the number of resources that are allocated for their computation. The scalability of different phases can vary greatly. A phase that can use more compute resources than another phase is said to be more scalable, comparatively.

Figures 2.3 and 2.4 show the results of the NAS [75] Parallel Benchmarks collected on SuperMUC [13] Phase 1 and Phase 2 nodes respectively. Results are shown for the EP, CG, LU and BT benchmarks, each at sizes W, A and B (where W is the smallest size and B the biggest). The MOPS (millions of operations per second) metric is plotted above, while the efficiency metric (MOPS per process) is plotted below. The EP benchmark stands for Embarrassingly Parallel; as can be seen on the plots, it indeed behaves as having large available parallelism in Phase 1 nodes, while it stops scaling linearly at 32 processes in Phase 2 nodes for size W, and at 64 for the other sizes. In most cases, such as this one, EP applications only scale linearly with resources up to a certain amount that is dependent on their input. All other benchmarks show clearly that they have limited scalability, with LU and BT benchmarks scaling more strongly than the CG benchmark. All of them show input dependent scalability, with CG being the most input dependent.

Phases with limited available parallelism, and therefore with limited scalability, are of special interest since their parallel efficiency is a function of the resources that are allocated to them. Elastic execution allows the adjustment of resources for these types of phases, once their performance has been analyzed. Performance analysis will be discussed together with scheduling in Chap. 10.

2.2.2 Network-, Memory- and Compute-Bound Phases

Phases can also be classified based on which aspects of the parallel system limit their performance. Phases can be limited by the network, memory or compute performance of the system. Phases that are limited by file system performance are considered network-bound, since distributed systems provide their file systems through their network.

It is important to note that this classification only makes sense given specific allocations, since bottlenecks may shift after a resource adaptation. For example, a network-bound phase can become memory-bound after a reduction of resources on its allocation. Such a scenario indicates that the optimal number of resources lies above the new reduced allocation and below the original allocation. This can be observed in the CG benchmark's efficiency plot in Fig. 2.4, evaluated in SuperMUC Phase 2 nodes. In this case, it can be seen that the efficiency and scalability of the application changes after 8 processes only for size B. Its scalability depends on the input, the number of resources allocated to it and the actual type of hardware where it is run, as can be seen when comparing these same results on SuperMUC Phase 1 nodes in Fig. 2.3.

Network-bound phases are limited by the performance of the network of the parallel system. Additionally, they may be sensible to the topology of its allocated resources in

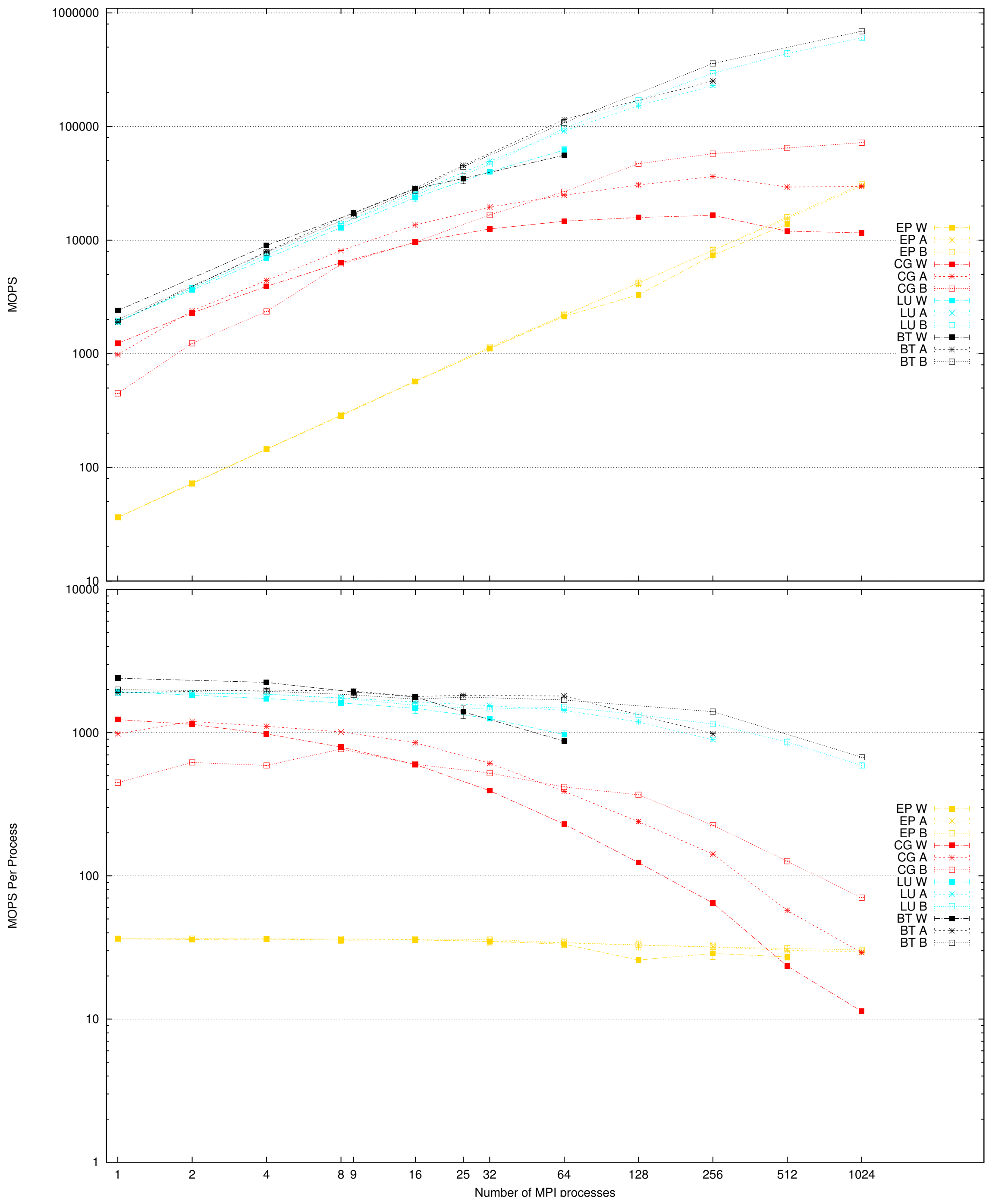

Figure 2.3: Results in MOPS (top) versus MOPS per process (bottom) of the EP, CG, LU and BT benchmarks of the NAS suite on SuperMUC Phase 1 nodes.

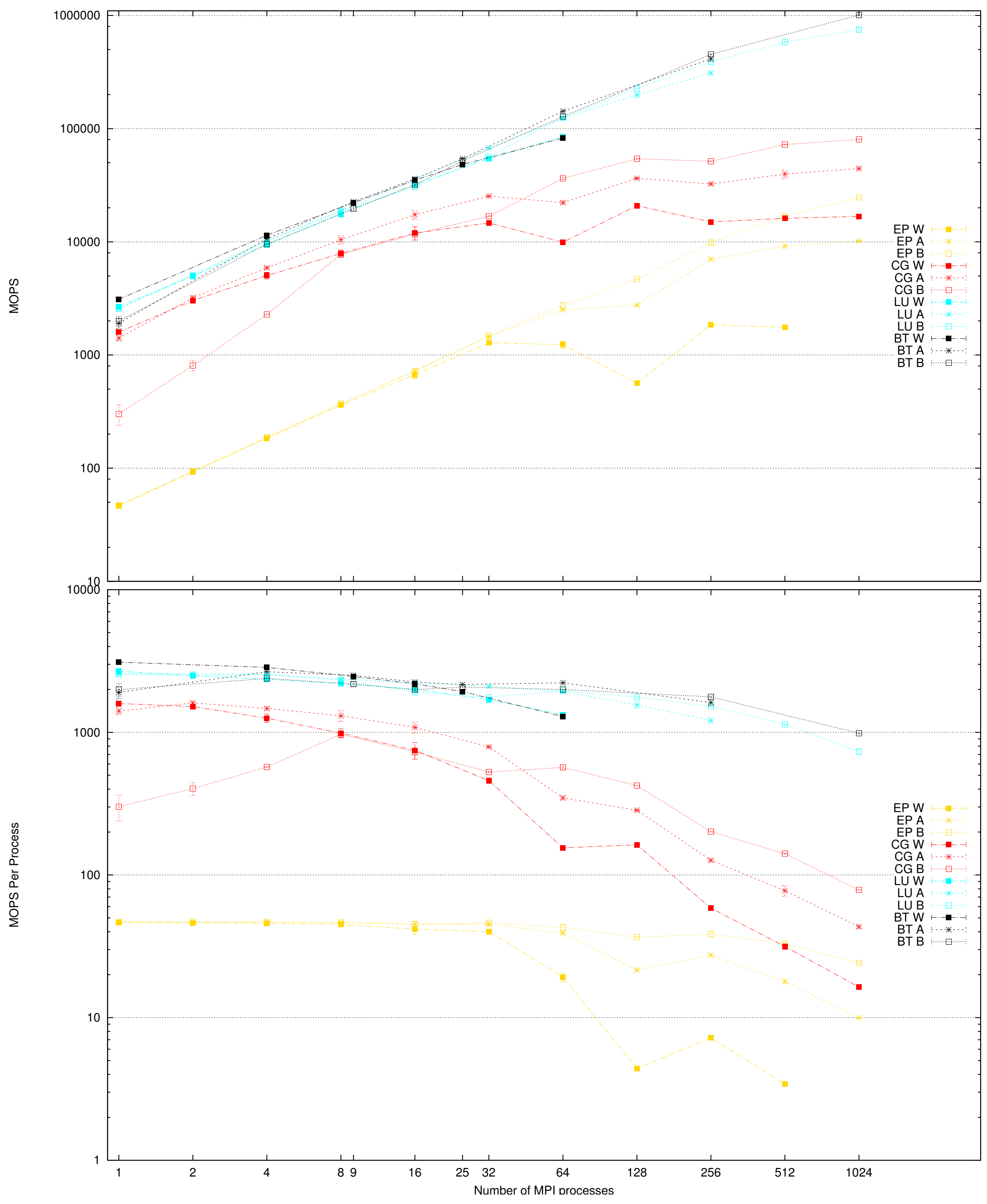

Figure 2.4: Results in MOPS (top) versus MOPS per process (bottom) of the EP, CG, LU and BT benchmarks of the NAS suite on SuperMUC Phase 2 nodes.

the network. Phases of this kind generally do not scale well on large resource allocations, since they tend to lower their parallel efficiency. Setting the number of processes per node to the maximum number of cores per node may provide the best performance in these cases. Parallel efficiency can be improved by reductions of resources in their allocations.

Memory-bound phases are limited by the bandwidth or latency of the memory subsystem of the compute nodes. These may be sensible to the number of processes executing internally at each node, and may benefit by a reduction of these, in contrast to the network-bound case. Scaling with the number of nodes will in most cases still gain significant performance before lowering the parallel efficiency of the application.

Compute-bound phases are limited by the performance of the arithmetic units of the processing elements in its allocation. These phases are targets for expansions of their allocated resources. Compute-bound phases may become network- or memory-bound once given enough resources due to their available parallelism. For example, the EP benchmark in Fig. 2.3 behaves as compute-bound for all allocations and input sizes on SuperMUC Phase 1 nodes, while on Phase 2 hardware its bottleneck is shifted starting at 32 processes (with different severity depending on its input size), as shown in Fig. 2.4.

Compute-bound phases normally scale to larger numbers of resources than other types of phases, such as those that are network- or memory-bound. Compute-bound phases may in some cases scale up to a number of resources that exceeds the available resources of the parallel system. These are not particularly interesting when optimizing an individual application since they perform at near their maximum efficiency independently of the amount of resources that they have allocated. It is important to identify them, since they can be efficient at a wide range of resource allocations and therefore can be used to fill up idle nodes. This can help minimize idle node counts and other system-wide efficiency metrics.

2.2.3 Phases with Different Input Dependent Network and Compute Scaling Proportionalities

It is common to find that the network and compute times of a phase scale with different proportionalities depending on its input. For example, there are distributed kernels and solvers where the computation scales cubically with the size of the subdomain of a process, while the communication scales quadratically with the same size of the subdomain. In this case, the application follows the surface to volume scaling proportionality, due to its computational domain being a volume while its communication boundaries are surfaces. The size of the subdomains depends on the size of the input and number of processes given to the application, as resources. When this situation arises, there are concrete allocation sizes that maximize the efficiency of the computation. Moreover, if overlap of communication and computation is possible, the optimal allocation for parallel efficiency is the one that makes both the computation and communication times match, potentially halving the execution time.

This type of scaling occurs frequently in simulation software due to common domain decompositions and computational kernels. Data partitioning schemes for distributed memory applications split a domain across the processes of an application. When simulating physical phenomena, domains tend to represent a volume. A decomposition scheme slices a specific domain into smaller sub-volumes, where the area of the sides of the sub-volumes determine the proportionality of the communication requirements, while the size of the sub-volumes themselves determine the proportionality of the computational requirements at each process. This situation arises very often in simulations where their solvers are based on stencils that represent sparse diagonal or block diagonal system matrices. In such cases, communication takes place across neighbors in the simulated physical domain; communication can be then optimized by placing processes that are computing in neighboring simulation subdomains close in the real physical network topology.

Even experts will have a hard time guessing the amount of resources a phase can use efficiently before the actual simulation takes place. In addition to this, since applications may have multiple phases, a fixed amount of resources that is efficient for the whole application may be impossible.

This can be better explained with an example. Figure 2.5 presents (from top to bottom) compute time, MPI time, total time (MPI and compute) and efficiency (matrix elements processed per second per process) metrics of a distributed Cannon algorithm implementation for matrix-matrix multiplication evaluated in SuperMUC Phase 1 (left) and Phase 2 (right) nodes. The results for allocations of 4 through 1024 processes are presented. These times were measured at the actual matrix multiplication kernel phase. As can be seen from the plots, as long as the MPI times are only a small fraction of the computation times, the kernel shows relatively constant results for its efficiency metric, and the total time continues to scale well with the number of resources. Once MPI time exceeds computation times, the efficiency and scaling of the application is reduced greatly. For this application, there is a ceiling on the resources that can be used efficiently by it. This ceiling depends on the size of the input (matrices in this case).

2.2.4 Efficient Ranges for Application Phase Scalability

It has been observed that there are ranges of process counts where application phases are efficient. There is only an upper bound on these ranges; applications tend to retain a similar level of efficiency with lower process counts. The upper limit on these ranges correlate to high proportions of MPI time versus compute time in the phases. Ensuring that application phases have resource allocations inside their efficient range is desirable. Exceeding the upper bound of the efficiency range should be avoided.

2.3 System-Wide Parallel Efficiency

The overall efficiency of complete supercomputers is of great importance. Current schedulers try to maximize system-wide efficiency metrics while applying best effort techniques to ensure fairness in terms of the wait times of individual jobs and their performance once started [89, 126, 200, 166, 90].

So far only the efficiency of phases as a function of their allocated resources has been discussed; this efficiency relates only to individual applications and not the efficiency of the complete parallel system. In this section, scenarios where elastic execution can improve the overall efficiency of complete HPC systems are discussed. The assumption here is that a system will have a mix of elastic and static jobs, in contrast to the current systems where jobs are strictly static.

2.3.1 Suboptimal Network Performance due to Fixed Initial Allocations

It has been shown by several researchers that the placement of processes can greatly impact the network performance of distributed applications [24, 162, 116, 202, 133, 171, 129,

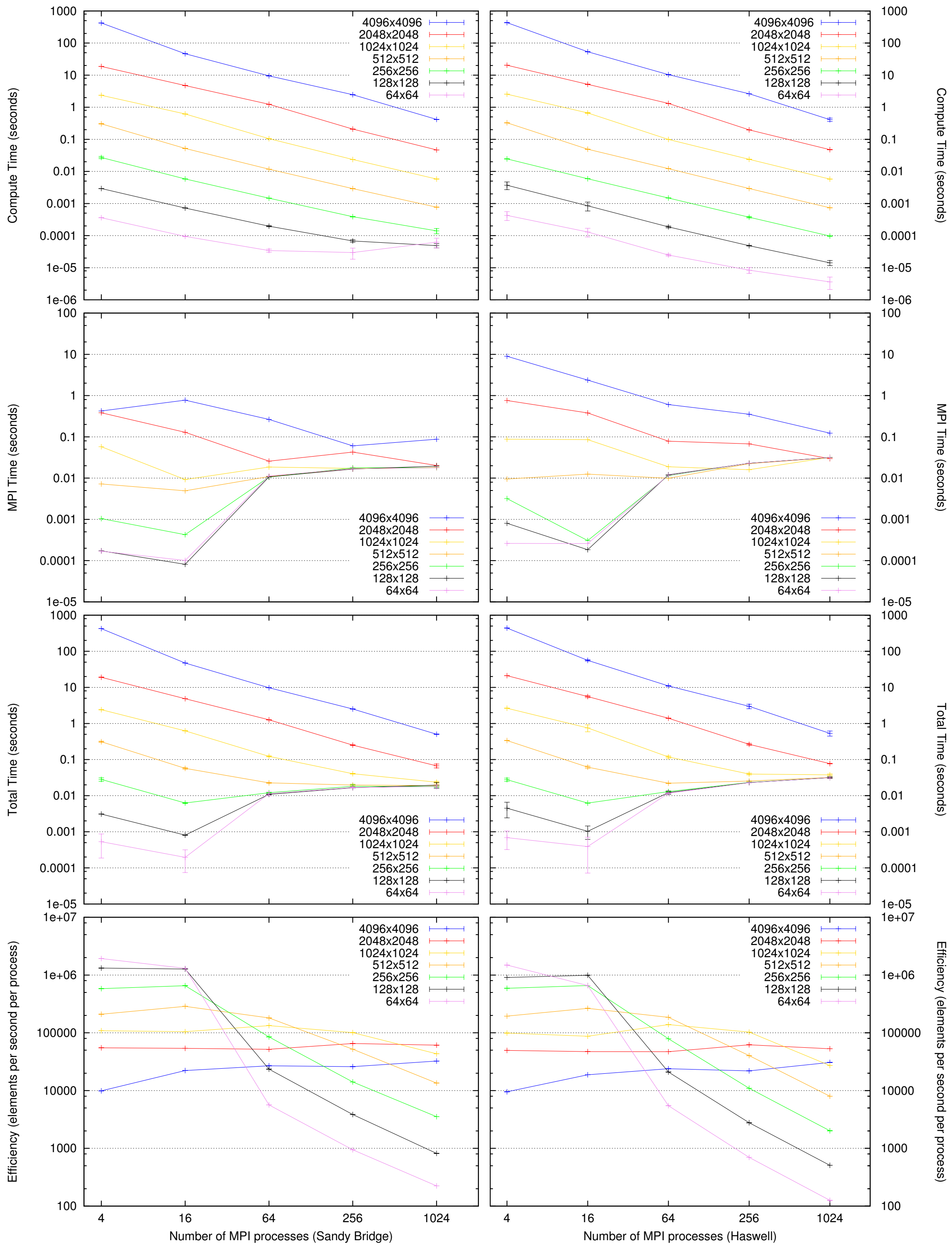

Figure 2.5: Compute, network and total times plus efficiency (top to bottom) of a Cannon's matrix-matrix multiply kernel. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) presented.

156, 187, 155, 134, 196, 157, 220, 211, 182]. Topology aware algorithms already exist that minimize the number of hops between nodes when allocating resources for a job [129, 44, 48, 128, 43, 125, 67, 201, 220, 150, 226, 117]. The goal is to minimize the network latency and maximize the bandwidth between the nodes allocated to a job. However, it is preferable to start a job that is waiting in the queue of a system immediately, instead of waiting for the ideal resources that provide the best network latency and bandwidth. Because of this, very often the allocation of resources is not ideal, depending on the level of fragmentation and node availability on a system when a job is launched. Elastic execution can alleviate this by allowing the reallocation of resources of a job once other resources are made available that minimize the number of hops across the full allocation.

2.3.2 Idle Resources due to Inflexible Resource Requirements in Jobs

The set of available jobs in the queue at a specific time and their static resource requirements can make it impossible for schedulers to fill up the totality of the resources of a parallel system. This becomes more difficult in systems that attempt to ensure fairness based on the arrival time of jobs in the queue. Indeed, the minimization of idle nodes has been the goal of several backfilling techniques developed over the years [200, 166, 212, 199, 206, 143, 233, 193, 144, 136, 151, 227].

In combination with backfilling, elastic execution can further alleviate this problem when there are sufficient elastic jobs in the queue that can make use of any idle nodes. In addition to this, waiting jobs in the queue may start immediately with a lower number of nodes, and later expand as needed once other jobs terminate and release resources. In general, elastic execution can completely eliminate idle nodes without needing the right combination of static jobs in the queue at all times, given sufficient elastic jobs. Jobs with phases that have very high scalability are particularly attractive, since they can be used to fill up idle resources with minimal efficiency penalties.

2.3.3 Energy and Power Optimizations

In addition to parallel efficiency, energy optimizations are increasing in importance in current HPC systems [88, 34, 55, 225, 102, 101, 190]. Energy costs have long overtaken the price of purchase and other maintenance costs, during the lifetime of these systems. Because of this, system administrators today may opt to use schedulers that optimize both performance and energy metrics simultaneously with multi-objective optimization techniques.

As previously discussed, elastic applications can have their resources adjusted to maximize their parallel efficiency and potentially reduce their runtime. In addition to this, resources can be adjusted so that energy efficiency is also optimized. This can be achieved through multi-objective optimizations that find Pareto-optimal solutions or by minimizing metrics such as the energy-delay product. In contrast, static applications cannot be optimized in this manner since the number of resources for the job must remain the same during its run.

Power-level stabilization is of great importance today to some compute centers. There are two common reasons: first, the stability of megawatt power delivery circuits, and second, the way some energy providers set up their billing contracts. Some contracts can have penalties for both not meeting or exceeding certain power levels. Elastic execution makes it easier for scheduling algorithms to meet and stabilize power levels by reducing idle nodes and scaling the resources of elastic jobs with different power characteristics.

3 Invasive Computing

This work is part of the research efforts coordinated by the Transregional Collaborative Research Center 89: Invasive Computing. Invasive Computing focuses on resource-aware computing in parallel systems. It has a wide scope: from embedded devices to distributed memory supercomputers in terms of hardware, and from operating systems to scientific computing applications in terms of software.

The Invasive Computing research project has been divided in three research and funding phases. It is currently in its second phase. There are several research groups divided across three universities. Their organization and goals will be described in this section.

3.1 Invasive Computing Research Groups

Invasive Computing is organized in research groups. Researchers are located across all three participating institutions: University Friedrich-Alexander Erlangen-Nüremberg, Karlsruhe Institute of Technology and the Technical University of Munich.

The research groups are clustered in five sets under the letters A, B, C, D and Z. Each individual group is identified with a unique combination of one of the preceding letters and a number. These are described in detail in this section. Figure 3.1 provides an abstract overview of where projects A through D fit in terms of their hardware and software application areas. The horizontal axis denotes the hardware abstraction level: Arithmetic Logic Unit (ALU), Tightly-Coupled Processor Array (TCPA), Central Processing Unit (CPU) and High Performance Computing (HPC). The only Invasive Computing specific term is the TCPA: these are collections of processing units that are used to accelerate certain workloads. Projects B, C and D are represented on the left vertical axis, with group B representing the lowest level and group D the highest. Projects in the A group interact with the others just mentioned, and are represented vertically on the right side of the figure. Each of the groups of the current research and funding phase will be introduced in the remainder of this chapter.

3.1.1 Group A Projects

Group A projects focus on research related to programming models, algorithm complexity analysis, software performance optimization and hardware-software cooperative design opportunities. The currently active projects in phase two are briefly described below.

A1: Basics of Invasive Computing

This project aims to enhance the programming language and framework developed for Invasive Computing specifically: InvadeX10. As its name suggests, it is the X10 programming language with syntax extensions to enable resource-aware programming. A framework has been developed around it, with a runtime system that allows for resource changes in both shared and distributed memory systems. The project also aims to develop

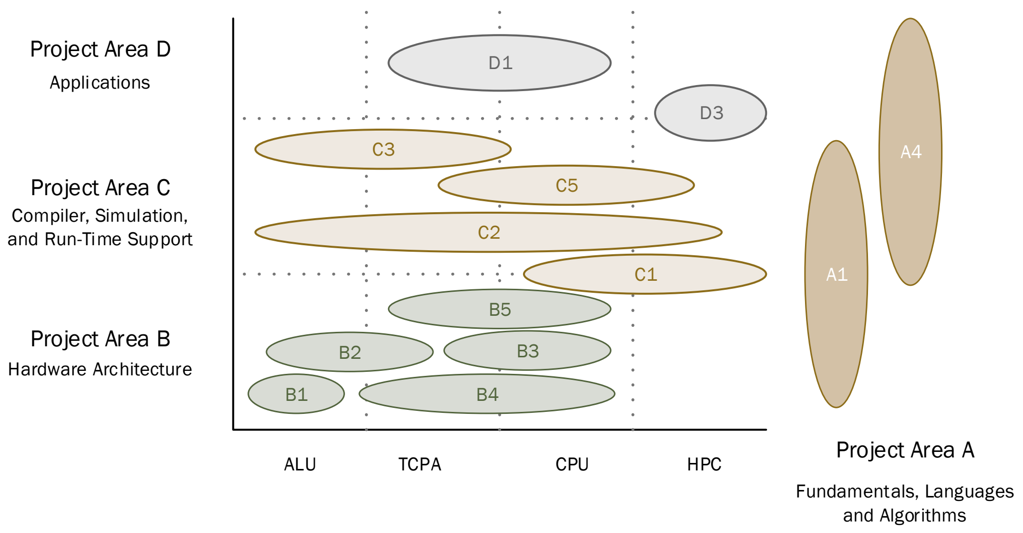

Figure 3.1: Abstract overview of project groups and their application areas.

X10 language extensions for the specification of non-functional guarantees, such as performance, correctness and fault tolerance. Support to any introduced language extensions is added to its runtime system.

A3: Scheduling and Load Balancing

The additional flexibility provided by resource-aware programming enables additional scheduling and load balancing opportunities. This research subgroup focuses on identifying additional scheduling requirements for resource-aware runtime systems, in order to take full advantage of this additional flexibility. The output of this group is a set of techniques for scheduling resource-aware applications; these techniques are designed to improve load balancing and non-functional guarantees related to correctness, performance, energy, among other things.

A4: Design-Time Characterization and Analysis of Invasive Algorithmic Patterns

The added flexibility provided by resource-aware programming also increases the complexity of schedulers and compute algorithms. Because of this, it is necessary to develop new analysis techniques for invasive programs and runtime systems. The output of this group is a set of techniques and tools for the characterization and analysis of invasive software.

3.1.2 Group B Projects

Group B projects research any hardware development opportunities to benefit invasive software and execution models. Specifically, they look at hardware features that benefit embedded systems, where invasive software can provide advantages in terms of predictability and other real time guarantees; the research does not look at performance benefits exclusively. There are five groups currently active and are described below.

B1: Adaptive Application-Specific Invasive Microarchitectures

This project investigates hardware mechanisms to be built into compute microarchitectures that facilitate the adaptation of resources for running applications. The aim is to have hardware resources that are reconfigurable during the run of applications. These resources include communication links, the number and type of computational units, amounts of memory, slices of the cache, even the length of instruction pipelines, among other things. The group aims to design and implement hardware features that enable applications to reserve hardware resources and get exclusive access to them.

B2: Invasive Tightly-Coupled Processor Arrays (TCPAs)

Researchers in this group evaluate design opportunities related to Tightly-Coupled Processor Arrays (TCPAs). Processor arrays can accelerate the processing of vector operations with high energy efficiency and predictable times. These type of workloads can be extracted from code with loops that are common in scientific software and digital signal processing. This group investigates the efficient reconfiguration and allocation of these arrays for resource-aware computing.

B3: Power-Efficient Invasive Loosely-Coupled MPSoCs

Energy efficiency is ever more important for computing hardware designs. Designs for resource-aware computing are no exception. This group investigates opportunities provided by resource-aware hardware design and scheduling for improved energy efficiency. Special attention is given to the Thermal Design Power (TDP) for embedded systems, such that invasive designs do not exceed safe temperature limits while at the same time providing adequate performance and energy efficiency.

B4: Hardware Monitoring System and Design Optimization for Invasive Architectures

This project focuses on the design of monitoring facilities for invasive hardware. These facilities are essential for the evaluation of non-functional properties in resource-aware software, such as performance predictability, fault tolerance, energy efficiency, thermal safety, and others.

B5: Invasive NoCs – Autonomous, Self-Optimizing Communication Infrastructures for MPSoCs

Networks on-Chips (NoCs) are necessary in current hardware designs with many computing elements. Communication bandwidth and latency need to scale well with reasonable amounts of transistors when compared to crossbars. This group designs hardware on-chip networks that can be reconfigured at runtime to provide security and performance guarantees to resource-aware applications.

3.1.3 Group C Projects

Group C projects investigate system level software and programming models for resource-aware computing. Different teams look at operating systems for distributed memory architectures, programming models for distributed algorithms, security for resource-aware applications, among other things. There are four projects that are active currently in phase two; these are introduced and briefly described here.

C1: Invasive Run-Time Support System (iRTSS)

This research group looks mainly at operating system features that help runtime systems support resource-aware software. A Unix-like operating system is part of the output of this group. The operating system has extensions that allow the adaptation of the resources of running applications. Extensions are being added that enable the specification of non-functional requirements (such as performance and power levels) for resource-aware applications.

C2: Simulative Design Space Exploration

In this project, researchers develop simulators based on models of invasive hardware and software. Simulations can then be used to steer the design process for invasive hardware, as well as aid the optimization of resource-aware software.

C3: Compilation and Code Generation for Invasive Programs

Code generation and transformation techniques for resource-aware software are investigated in this project. Code is generated for a variety of hardware resources with different dimensionalities, given the reconfigurable nature of the design of the CPU cores, the Tightly-Coupled Processor Arrays (TCPAs) and network links, among others. The compiler is developed with support for the InvadeX10 language and framework, and has back ends for different CPU architectures.

C5: Security in Invasive Computing Systems

Security in computing systems is of great importance. Because of the highly dynamic nature of resource-aware software, providing security for it is challenging. For example, memory that is reallocated from one application to another, should not be readable at the target, therefore access control alone is not sufficient for the resource-aware case. This research group aims to provide reliable security solutions for invasive software.

3.1.4 Group D Projects

Group D projects focus on potential application areas for Invasive Computing. Most areas of computing are considered, from embedded systems to distributed memory supercomputers. There are currently two active projects and are briefly described here.

D1: Invasive Software-Hardware Architecture for Robotics

This research group investigates the benefits and limitations of resource-aware computing in robotics. In particular, research is aimed at its application on humanoid robots in complex scenarios, where large computations need to take place concurrently and in a timely fashion.

D3: Invasion for High Performance Computing

There are three main areas of research in this group: numerical methods and implementations for resource-aware computing, advantages of resource-aware computing in HPC and support for current HPC hardware and programming models. The work presented in this document belongs to this group of the Invasive Computing project. This work focuses mainly on supporting Invasive Computing in state of the art HPC hardware and software.

3.1.5 Group Z Projects

The general administration tasks of the Invasive Computing project are performed in the Z project groups. Contacts with important individuals, research sites and companies are managed by these groups. These groups also provide the necessary coordination for consolidated results, such as the demonstration platform produced by the Z2 project.

Z2: Validation and Demonstrator

The main goal of this project is to provide an FPGA based hardware platform for validations and demonstrations. Contributions from multiple other projects are integrated in this platform.

4 Related Work

A broad discussion on related work is presented in this chapter. First, an incomplete overview of programming languages, interfaces and resource managers that do not support elastic execution is presented. In spite of the lack of support for elastic execution, meaning that these works are not alternatives to what is proposed in this document, their overview is provided for completeness. Afterwards, closely related works that support resource-elasticity are discussed in detail.

4.1 Programming Languages and Interfaces without Elastic Execution Support

The work presented in this document is related to programming models and resource managers that target applications with high synchronization requirements. In this section, an overview is provided about related works that are not direct replacements to what is proposed in the rest of this document, but that could potentially be extended to support resource-elasticity in the future. First, programming languages and interfaces that target only parallel shared memory systems are discussed. Afterwards, programming languages and interfaces for distributed memory that only support resource-static execution are covered. Finally, solutions for cloud and grid computing systems that support resource-elasticity are described and compared to HPC solutions.

4.1.1 Parallel Shared Memory Systems

Shared memory is a term used to indicate the presence of a single address space across all processing elements. There are several parallel programming languages and interfaces that target shared memory systems exclusively. These have increased in number and importance, due to the increase of parallelism in shared memory systems in recent years. These languages and interfaces are related to this work, since each node that is managed in a distributed memory HPC system is itself a parallel shared memory system. Brief descriptions of the most widely recognized languages and interfaces that target parallel shared memory systems are presented here. Additionally, a brief discussion on hybrid programming models with MPI is provided.

Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) [9, 76, 191] is an Application Programming Interface (API) that can be used by programmers to parallelize serial applications. The API is standardized by the OpenMP Architecture Review Board and is currently at version 4.5. The API is provided as pragma directives to Fortran, C and C++. Additionally, environment variables and compiler extensions are also defined in the specification. OpenMP has enjoyed support from several compilers over the years, both free and commercial, such as: GCC, IBM XL, Intel, PGI, Cray, Clang [4, 17], and others.

The pragmas allow for the annotation of regions that can run in parallel in the source code of programs. Additionally, developers can provide instructions that specify how these parallel regions should be executed. For example, a region can be executed following a fork-join threading model or, since version 3.0 of the specification, following a task-based model [35]. OpenMP is known to allow for relatively simple conversions of serial Fortran and C programs into threaded programs. This makes it an attractive choice for software projects where significant resources have already been dedicated to the development and validation of preexisting source code.

Within the Invasive Computing research project, extensions to OpenMP and a runtime system have been developed to support resource-aware computing [100, 118]. The goal is to adjust the computing resources allocated to OpenMP applications running simultaneously in shared memory systems, such as CPU cores, based on their scalability. This previous research targeted parallel shared memory systems, while the research presented in this document targets distributed memory systems.

POSIX Threads (PThreads)

The POSIX [1, 172] standard defines a thread API and model that is usually referred to as POSIX Threads or PThreads [169] for short. The API is supported by several Unix and Unix-like operating systems, such as: FreeBSD, NetBSD, OpenBSD, Solaris, Linux, Mac OS, etc. There are also implementations available for Microsoft Windows and other operating systems.

The API provides operations for thread management, such as: creation, termination, synchronization, scheduling, etc. For synchronization, there are mutexes, joins and condition variables. Mutexes are used to ensure exclusive access to resources, such as memory or external devices. Joins are operations that are used to wait at a specific thread for other threads to complete. A thread may create multiple other threads and then wait on them with a join operation, following a fork-join pattern. Finally, condition variables can be used to make threads wait until certain conditions are met.

Cilk Plus

Cilk Plus [147, 210, 148] is a parallel programming language based on the C language with extensions for the definition of parallel loops and fork-join patterns. The language and its runtime has been improved over time and has become a commercial product.

The Cilk Plus language is designed to expose parallelism in the source code. Once the parallelism of a program is defined, a runtime system can schedule its work automatically on parallel shared memory systems. The spawn keyword is used for the creation of tasks, while the sync keyword is used to wait for them. These can be used for the creation of fork-join patterns. Cilk Plus also extends C with keywords for the creation of parallel loops, the specification of reduction operations, the creation of arrays and simplifications to array accesses, among other things.

The schedulers found on Cilk Plus runtime systems implement work-stealing [22] techniques that can effectively balance the load across executing units. With work-stealing, threads that are idle due to finishing their own tasks early can take and execute entries from the work queues of other threads, effectively stealing them.

Threading Building Blocks (TBB)

Threading Building Blocks (TBB) [184, 73] is as a C++ template library that provides high level constructs that ease the creation of parallel programs, such as: parallel loops, reductions, pipelines, queues, vectors, maps, memory allocations, mutexes, atomic operations, etc. Similarly to Cilk Plus, it attempts to separate the definition of tasks from their actual execution in a parallel system. Its schedulers also implement work-stealing techniques.

Hybrid Programming Models

Shared memory programming models are numerous and are generally orthogonal to message passing. The passing of messages is unnecessary when data can be accessed directly in the same address space, without the need of copies. The passing of messages is needed across nodes in distributed memory systems. There have been research efforts to bring the benefits of shared memory to MPI applications with minimal modifications to application source code and libraries [94, 168].

The combinations of message passing and shared memory programming models are usually referred to as hybrid programming models. Message passing with MPI can be combined with shared memory APIs and libraries, such as MPI and OpenMP [179, 56, 222, 141], MPI and POSIX Threads [173], etc. The idea is that a shared memory programming model can better abstract the parts of a program that map to the hardware inside of a node, while MPI can better abstract the operations that take place across nodes.

Shared memory programming models are not alternatives to the work presented in this document and are instead complimentary. In this work, resource adaptations are done at the node level, meaning that resource adaptations are done based on whole nodes. Because of this, resource adaptations have no impact on any shared memory language or interface used for intra-node parallel programming in hybrid scenarios.

4.1.2 Distributed Memory Systems

After the discussion about popular languages and interfaces for parallel programming in shared memory systems, a similar discussion is presented here for those that support distributed memory systems. The introduction to MPI is done later in Chap. 5, since it is used as basis for the elastic message passing research presented in this document. The alternatives to MPI presented here follow the Partitioned Global Address Space (PGAS) programming model.

High Performance Fortran (HPF)

High Performance Fortran (HPF) [153, 139, 218, 137] is an extension to the Fortran language. The portability of parallel programs across multiple HPC systems is one of its goals. Its implementations aim to provide high performance while preserving the higher productivity of the language when compared to MPI.

The HPF extensions add support for controlling data distribution. More specifically, the extensions allow for the mapping of arrays to distributed memory with layout control. Data parallel constructs can then be performed on the distributed arrays (e.g., *forall*). The *pure* attribute allows developers to identify functions that do not modify global memory, pointers or perform IO. The *independent* directive allows programmers to identify blocks of code that can be performed in any order. The HPF extensions also include intrinsic functions for distributed array access, among other things.

Coarray Fortran (CAF)

Similarly to HPF, Coarray Fortran (CAF) [161, 130, 85, 113, 224, 192] started as an extension to Fortran that aims to increase productivity while maintaining competitive performance when compared to MPI. It has been added to the official Fortran 2008 standard in 2010, with small changes to its original syntax. Thanks to its inclusion into the Fortran language standard, its implementations have risen in recent years and official support for the PGAS model is now available in Fortran.

The CAF extensions allow programmers to define grids of virtual processors. They also allow for the partitioning of data and its mapping to virtual processors. There are operations for the movement of data and synchronization. Parallel programs are composed of multiple images that can be run in distributed memory systems. These have a simple two-level memory hierarchy, where accesses to memory can be local or remote. Remote memory accesses are performed with put and get operations to global addresses. There are operations for barriers, critical regions, team synchronizations with notify and wait, among other things. Collective operations previously present in CAF were not included in Fortran 2008 and will perhaps appear in future standards [160, 161].

Chapel

Chapel [63, 37, 64, 194, 197] is a programming language that aims to ease the development of parallel software with support for distributed memory systems. Chapel provides a threaded programming model with language constructs that help represent parallelism. There are language grammars that can be used to represent data and task level parallelism. The language also allows the description of data decomposition. The expressiveness of the language aids runtime systems when optimizing the placement of data and computation. This is specially important in distributed memory systems due to the variability of latencies and bandwidth depending on localities.

Unified Parallel C (UPC)

Unified Parallel C [87, 32, 80] is an extension to the C programming language, similarly to how CAF and HPF are extensions to the Fortran language, with the purpose of simplifying the task of producing parallel software with support for distributed memory systems.

Its extensions to the C language allow developers to define data distributions and affinities. Distributed applications consist on multiple threads running independently. These can synchronize with barriers, locks and other memory consistency controls. The memory model consists of pointers in a global address space that can be shared or private. Pointers that are shared can reside in local or remote memory, depending on their affinities, since they are partitioned among threads. There are also language constructs for parallel computation and synchronization.

4.1.3 Cloud and Grid Computing

Both programming models and resource managers exist for cloud computing environments that have achieved the goals of this work: efficient resource-elasticity and scheduling in distributed memory systems. Indeed, there are cloud and grid computing programming models and resource managers that allow for resource-elasticity [50, 39, 223, 217]. Some of these models even support fault-tolerance. In contrast, elastic execution and fault-tolerance research are still in their infancy in the HPC domain. The reason for this is that cloud workloads have lower synchronization requirements, where applications tend to follow client-server or MapReduce patterns. The same can be concluded from embarrassingly parallel workloads, such as parameter sweeps.

For the client-server pattern, a set of load balancers with worker hosts [41, 81, 82, 119, 181, 92] provide a reliable and well performing resource-elastic execution model with the added benefit of fault tolerance. This is all thanks to the low synchronization requirements across user sessions in web server workloads. There are a lot of applications and services on the web that fit this execution model. For this reason, cloud services from several vendors [97] continue to enjoy success in the market place.

Applications [86, 219, 68, 66] that fit well the MapReduce [78, 79, 146] programming model can have a clear separation between their map operations and their reduce operations. Both map and reduce operations, due to their low synchronization requirements, can be allocated effectively to resources so that elastic execution and fault tolerance are achieved. Data is redistributed in an intermediate step between maps and reductions, usually referred to as a shuffle step. Outside of the shuffle step, there are hardly any synchronization requirements in MapReduce applications. There are several data processing algorithms that fit this model well.

Unfortunately, although the before mentioned solutions are mature and quite successful, scientific computing workloads do not fit well [159, 111] in their programming and execution models. The main sources of incompatibility are the large synchronization requirements due to data dependencies. The domain of a typical scientific application is partitioned across processing units. Depending on data dependencies, synchronizations that are typically frequent and periodic are required. Programming models that abstract synchronization operations aid the development and maintenance of scientific applications better.

It is important to note the differences between HPC clusters, Cloud and Grid computing systems [188, 228, 33]. HPC systems are designed with high performance networks to minimize the impact of synchronization and allow applications to scale to larger resource allocations efficiently. In contrast, cloud systems can be designed more economically with commodity networks, thanks to the low synchronization requirements of the workloads they target. The specialized networks used in HPC systems have lower latencies and higher bandwidths, but also higher purchase and maintenance costs when compared to commodity networks. Grid computing can support scientific computing workloads, since a grid system can be composed of multiple HPC systems that are geographically separated. Due to the additional latencies and lower bandwidth across geographic locations, it can be unfeasible to distribute workloads across sites. Grid computing software can be seen as complimentary to a resource manager that manages a single HPC systems at a single geographic location, since it can aggregate multiple of these systems.

4.2 Elastic Programming Languages and Interfaces for HPC

Elastic languages and interfaces are those that have abstractions to represent changes in resources at runtime. These can also be referred to as resource-elastic. In contrast, resource-static are those that have no abstractions for resource changes and operate under the assumption that resources are never modified during the execution of an application.

Elastic programming languages and interfaces can be classified based on whether they support shared memory systems only, or both shared and distributed memory systems. This is an important distinction, since the reconfiguration and movement of memory over a communication network are only necessary in distributed memory systems. In this section, we will only discuss works related to elastic programming models and runtime systems that support distributed memory. In contrast to the previously discussed related works, these are highly relevant and alternatives to the work presented in this document.

There are multiple past and ongoing research efforts related to resource-elasticity in HPC; however, these are not as numerous as those that only support resource-static execution. Developers of both resource-elastic applications and runtime systems need to carefully manage any added overheads related to the reconfiguration of resources and memory, because of their significant performance impact in distributed memory systems. In addition to this, runtime systems and resource managers need to be properly integrated to support resource-elasticity. It is possible that these additional challenges have limited the amount of related works in this area.

4.2.1 Charm++ and Adaptive MPI

Exactly like this work, Charm++[132, 23, 127, 3] and Adaptive MPI [120, 122, 121, 42] are motivated by the dynamic behavior of certain workloads, such as Adaptive Mesh Refinement (AMR) methods (as mentioned in the motivation, Chap. 2) where load imbalances often occur at runtime. Their solution for load imbalances in distributed memory systems is to implement MPI on the Charm++ runtime system. The result is Adaptive MPI, an MPI implementation that supports automatic load balancing given that any preexisting MPI code is converted to meet certain conditions. Load balancing is achieved through thread migration mechanisms [229, 174].

A rank in Adaptive MPI is a user-level thread that is associated with Charm++ objects. Because ranks are threads, additional restrictions need to be applied to global variables when converting preexisting MPI code to Adaptive MPI. Automated tools are available to assist in the conversion of MPI code to Adaptive MPI [168, 170].

Adaptive MPI programs follow a message driven execution model. Its runtime system picks threads that have their messages ready, and therefore can continue doing progress. The system relies on the oversubscription of ranks, where multiple ranks are pinned to each CPU core available.

The ranks in Adaptive MPI can be migrated. There are programming constructs that allow the creation of programs where ranks can be migrated without the need for custom pack and unpack routines. Ranks can be dynamically load balanced by the runtime system. Load balancing techniques can be overloaded by user provided implementations. Some of its runtime systems also provide fault tolerance, through automated checkpointing and restarts [62, 232, 231].

Adaptive MPI currently supports MPI up to version 2.2. The newer features of MPI 3.0 and later, such as non-blocking collectives, are not supported. Adaptive MPI has achieved performance and efficiency comparable to other MPI implementations. Distributed applications that have achieved good strong scaling properties and overall performance have been developed with it. These compare favorably versus their regular MPI versions when load balancing is of increased importance.

Resource-Elasticity with Charm++ and Adaptive MPI

Charm++ and Adaptive MPI are probably the projects that are most closely related to this work. There are several Charm++ and Adaptive MPI resource-elastic works that are highly relevant to what is presented in this document. Support for malleable jobs, that can have the number of nodes allocated to it modified at runtime, has been demonstrated [131, 110]. Some of these rely on the creation of a checkpoint, to then later restart with a modified thread count [110]. The cost of these operations can be mitigated by using shared memory [230]; results have been clearly better than checkpoints that are backed by file systems, although the overheads can still be large depending on the initialization costs of the application. There are limitations in some of the proposed solutions, such as the inability to have larger number of resources than initially allocated [131]. Also, as ranks are abstracted as threads, in some implementations there are processes left running in preexisting resources. These are used for messaging or other Charm++ related support operations [131], and can degrade the performance of other processes in the same node.

Similarly to this work, many of the resource-elastic solutions that are based on Charm++ and Adaptive MPI have been paired with resource management research [110, 177, 178]. This is necessary to enable the adaptation of resources of applications in shared systems; the application programming language or API, the runtime system and the resource manager need to support resource-elasticity and be properly integrated.

The proposed solution presented in this work follows the current MPI execution model of processes with private address spaces and no oversubscription, instead of the threading with oversubscription and a message driven execution model found in Charm++. The current MPI model better prevents interference between applications, but does not provide automatic load balancing.

4.2.2 The X10 Programming Language

X10 [65, 189, 167, 74, 207, 163, 21] is an object oriented programming language with distributed arrays. It follows the Partitioned Global Address Space (PGAS) programming model. It is object-oriented with strong typing. Its runtime system provides a garbage collector. Similarly to other languages that follow the PGAS model, one of its goals is to improve the productivity of application developers when writing applications for distributed memory systems. It differs from other PGAS languages in that many of its constructs are designed to allow asynchronous execution.

As may be expected, X10 shares many similarities with its peer PGAS languages, such as a two-layered memory model with local and remote memory, constructs for parallel execution and synchronization, distributed arrays, etc. The language is supported by a complier, a runtime system and a standard library that are all extensions to their preexisting Java counterparts, with the addition of optional C++ back ends.

Resource-Elasticity with X10

The language is attractive for resource-elasticity support since it abstracts resources (e.g., nodes in a distributed memory system) with the concept of *Places*. There is also the concept of *PlaceGroups*. These are ordered sets of *Places*. Computations and data are distributed across *Places*. Mapping routines attempt to optimize the location of *Places* based on network topologies to optimize performance. Support for elastic execution was added with version 2.5, by allowing applications to execute over dynamically varying sets of *Places* in *PlaceGroups*.

X10 is one of the core programming languages supported by the Invasive Computing project. Researchers have extended the X10 language to support the goals of the project, such as the specification of non-functional requirements (like performance, energy, etc.), support for resource-aware programming, the addition or removal of *Places* at runtime, among other things. The project has produced X10 programs and a full X10 stack [175, 176, 112, 185, 164, 49, 51]: compilers, an operating system, custom runtimes, and even hardware support.

4.2.3 Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) [10, 38, 205, 158, 99, 108] system allows a set of nodes to be viewed as a single parallel computer. The set of nodes is managed by the user and can be modified at runtime. This allows for resource-elasticity and some forms of fault-tolerance. Like MPI, it follows the Message Passing (MP) programming model and supports distributed memory systems.

The main goal of MPI is to provide a message passing interface only, while PVM abstracts a distributed operating system with support for message passing. PVM provides operations to spawn tasks and coordinate them, as well as to modify the parallel machine itself. MPI had spawn operations added in version 2.0 of the standard. The spawn operations in MPI depend on its integration with resource managers while PVM simply spawns new tasks as requested by application processes.

The PVM system is composed of two main parts: a daemon that runs at each node and a runtime library. The daemons need to be started before applications run in the nodes. The library is linked into application binaries and provides the implementation of the PVM API. The typical PVM application is started as a single task that spawns other tasks. Once the tasks are started, they can start exchanging messages. Individual tasks have unique identifiers that are used to send and receive messages.

Resource-elastic behavior can be achieved with PVM within the resources of single jobs. The system provides no coordination with resource managers; this makes PVM inadequate for resource-elasticity in systems with multiple users. Additionally, its message passing features and performance are limited when compared to the current MPI implementations.

4.2.4 Other Related Works

Several research groups have demonstrated the benefits of malleable jobs when optimizing parallel compute systems [91, 83, 123, 215, 57, 204, 165, 46]. Multiple works also describe the need for performance feedback from applications to schedulers, mainly to improve the quality of resource adaptation decisions [26, 203].

Standard MPI applications that rely on spawn operations have been used with customized resource managers to achieve resource-elasticity [59]. The proposed MPI extension provides several advantages over these and are discussed in detail in Chap. 5.

Some other alternatives rely on the creation of check points by the applications, and their ability to restart at different scales [183, 110, 25, 216] with large IO and reinitialization overheads. The solution presented in this document enables resource adaptations with less overhead, thanks memory to memory repartitions over the network in adaptation windows. The trade-off is that adaptation windows need to be developed.

5 The Message Passing Interface (MPI)

Message Passing (MP) is a widely used programming model for distributed memory systems. The Message Passing Interface (MPI) is a standard for message passing that has been of great importance for both communication library implementors and application developers. The standardization efforts of the MPI forum [18] have allowed for compatibility between vendors at the source code level. This means that applications are portable across distributed memory HPC systems.

Portable code was rare in earlier years when systems relied largely on proprietary Application Programming Interfaces (APIs) and libraries for inter-node communication. In past years, many different communication libraries were provided by different vendors for distributed memory computing. Each vendor had its own view on how distributed memory applications should be developed. While this flexibility allowed communication libraries to provide APIs that closely matched the specialized hardware that they abstracted, software had to be ported to each new machine.

A need for standardization was determined and the MPI standard was eventually defined. Its first version was released in the year 1994. The standard defines an API only; it is up to each vendor to decide how to implement it. Two open source MPI implementations are currently the most widely used: Open MPI [96, 105, 103, 154, 104, 115, 7] and MPICH [209, 107, 109, 52, 53, 6]. Some of the current commercial MPI libraries are based on these libraries with the addition of vendor specific customizations for better performance or specific hardware support.

The operations specified in the standard are designed to allow a wide range of communication hardware to be supported efficiently. In some cases, the standard itself has been updated to allow for better efficiency on new communication network hardware. For example, the one-sided communication API was updated to better fit newer RDMA implementations with version 3.0 of the standard.

In this chapter, an incomplete list of the features provided by MPI will be briefly introduced first. Afterwards, limitations in the current specification and implementations of the dynamic processes support of MPI will be identified; the set of extensions described later in Chap. 6 is proposed as a way to overcome these limitations. Finally, an overview of the MPICH library is presented, since it is the basis for the Elastic MPI library presented in Chap. 6.

5.1 MPI Features Overview

In this section, the API specified by the MPI standard in its current 3.1 version is briefly described. MPI offers API sets for: point-to-point communication, collective communication, one-sided communication, parallel IO, derived data types, virtual topologies, group and communicator management, and more.

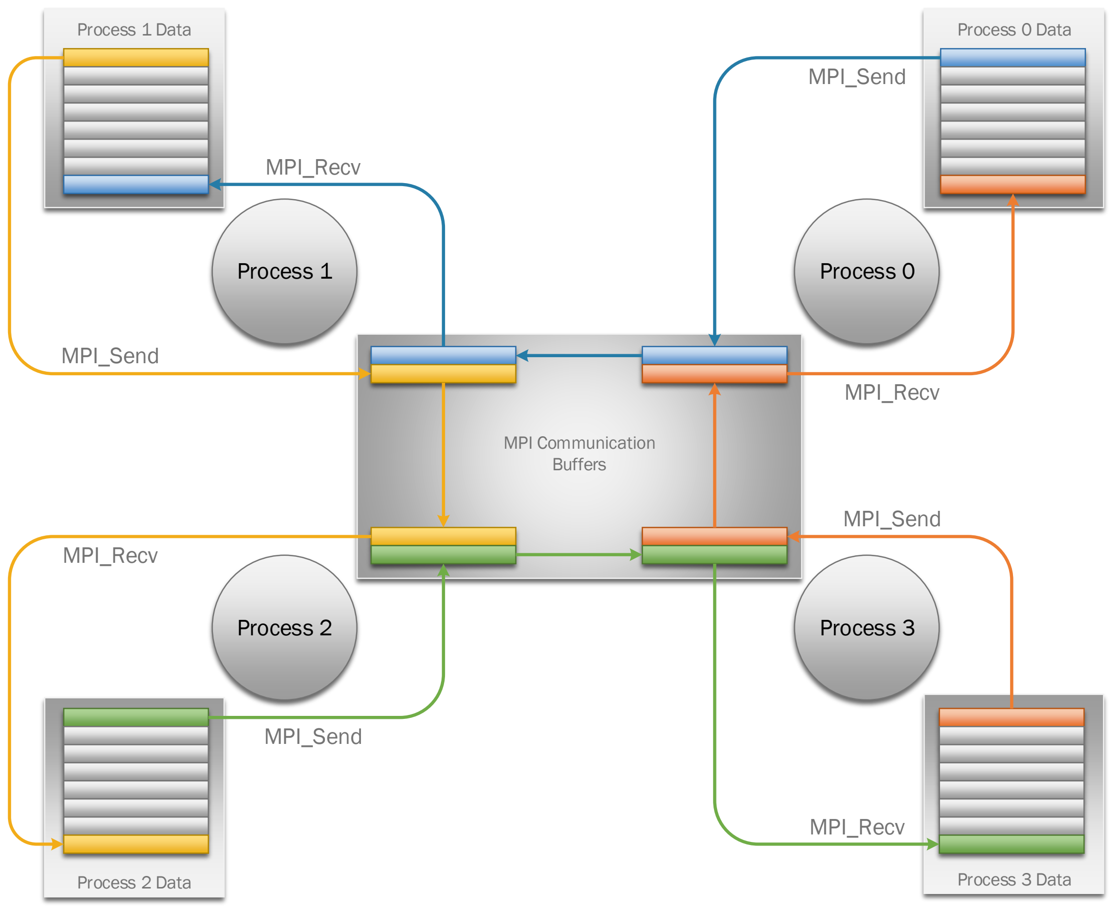

Figure 5.1: Simplified overview of MPI communication and buffering for small and medium buffers (typically smaller than a megabyte) on a four process application with a counterclockwise ring communication pattern.

5.1.1 Data Types

Instead of using types from the C or Fortran programming languages, MPI defines its own data types. This allows MPI libraries to adapt and align memory properly when transferring messages across machines or software stacks with incompatible type representations. MPI provides a set of basic types, such as `MPI_INT` and `MPI_DOUBLE`. In addition to this, users can create their own derived data types, for example, to represent vectors or structures. Data types need to be specified in all MPI operations that perform message transfers or perform distributed arithmetic on buffers.

5.1.2 Groups and Communicators

Groups are ordered sets of processes in MPI. Communicators enable communication among the processes of a group. After initialization, MPI libraries provide a communicator that includes all the processes that were started as part of an application: the `MPI_COMM_WORLD` communicator. This communicator is usually sufficient when managing the communication of a few processes. However, when applications reach a certain level of complexity, it helps to create groups and communicators to modularize the code of MPI applications.

A typical application will first duplicate a communicator, then split it or divide in ways that will benefit the clarity of the algorithms in the distributed application. Communicators can be manipulated directly, or alternatively, groups can be created first and then communicators created from them. The latter approach requires more steps but has flexibility advantages. For example, there are union and intersection operations that can be applied to groups but not to communicators.

5.1.3 Point-to-Point Communication

The point-to-point set of operations in the MPI standard allows for the transmission of bytes from one specific process to another. There are several variants available with different send modes: default, ready, synchronous and buffered. The status of the receiver end is undefined in the default variant. Ready sends complete when the matching receive has been posted. Synchronous sends only complete when the matching receive has also completed. Finally, buffered sends rely on user provided buffers for their operation.

In addition to send modes, these operations also have blocking and non-blocking versions. Blocking versions do not return until the operations have been completed, while non-blocking versions will return immediately. When using non-blocking versions, the application needs to check the status of the operations with the wait or test operations. The non-blocking operations allow for the overlap of communication and computation.

Figure 5.1 provides a simplified visual overview of the interactions of application and MPI communication buffers during point-to-point communication. In the figure, four processes are depicted performing a send and a receive each in a counterclockwise ring pattern. As can be seen, several copies are performed from application buffers to MPI communication buffers and the other way around. This is common in most implementations with internal eager or rendezvous communication protocols at the byte transfer layer; these protocols are used when transferring small to medium buffers (before buffer sizes where DMA transfers become more efficient). Typical MPI implementations, including MPICH (discussed in Sec. 5.3), decompose collective operations into multiple point-to-point messages; therefore, this figure also applies to most types of communication that do not rely on network hardware acceleration (such as RDMA or hardware collective operations).

5.1.4 One-Sided Communication

Network hardware with Remote Direct Memory Access (RDMA) features can improve communication performance by reducing latencies and overheads related to buffering and synchronization. In addition, RDMA allows for better overlap of communication and computation. MPI added its one-sided communication API to allow implementations to efficiently support RDMA hardware. This API was introduced in version 2.0 of the standard, and was updated to match more recent RDMA capable network hardware in version 3.0.

With these operations, MPI implementations can reduce the amount of memory needed for buffering and the number of memory copies performed on typical communication protocols (refer to Fig. 5.1). In cases where hardware RDMA is not available, most MPI libraries fall back to an internal point-to-point based implementation; this way, applications that use one-sided communication remain portable.

In this mode of communication, MPI processes create memory windows that can be accessed by remote processes. They can then read and write data to their own and remote

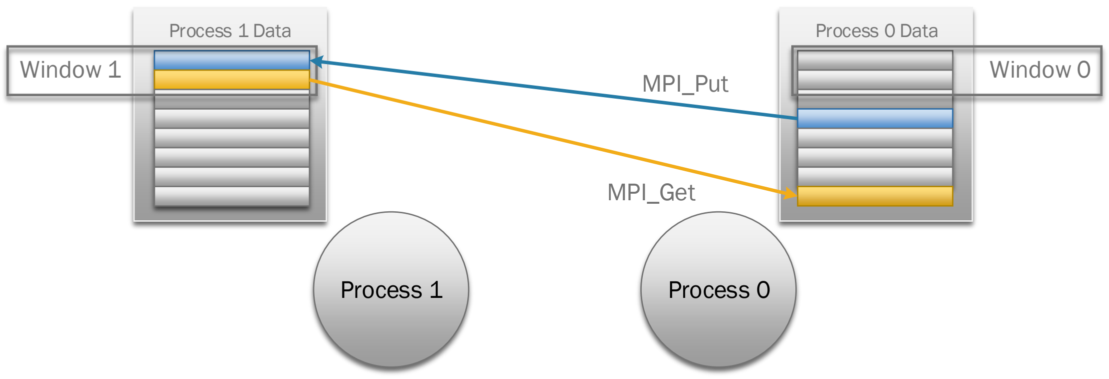

Figure 5.2: Put and get operations initiated both by process 0 using MPI one-sided communication.

buffers. Synchronization operations are also provided, to prevent race conditions. Figure 5.2 provides an illustration of a possible interaction between two processes. In this case, the process with MPI rank 0 transfers some data from its own address space (blue) towards the memory window of the remote process with rank 1. Rank 0 also transfers data from the remote window of rank 1 (yellow) into its own address space. The same operations can be performed by the process with rank 1 on the window created by rank 0.

5.1.5 Collective Communication

MPI also provides operations that work on groups of processes. These can be synchronization operations such as a barrier, data transfer operations such as broadcasts, and collective operations on data such as reductions. With MPI 3.0, non-blocking versions of these operations were introduced. Neighborhood collectives were also introduced with MPI 3.0; these can be more efficient on certain communication patterns, such as those generated by stencil based distributed solvers.

The use of collectives is highly recommended to MPI application developers. The internal collective algorithms implemented within MPI libraries perform well, based on long term research related to their efficiency; in addition to this, MPI implementations can take advantage of hardware network collectives when available. It is very unlikely that users will achieve better performance than the well tuned internal implementations provided by Open MPI or MPICH. Take for example the sequence diagram of a naive algorithm presented in Fig. 5.3: an appli-

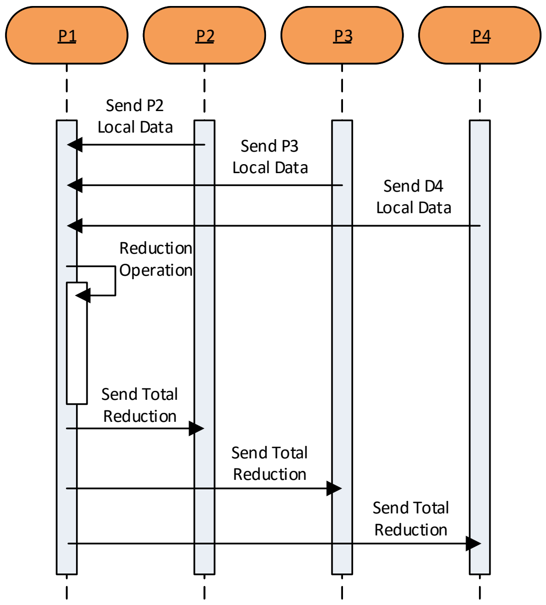

Figure 5.3: Sequence diagram of a naive all-reduce operation implementation.

cation developer may be tempted to perform this sequence of operations with MPI point-to-point operations (a gather, followed by a reduction and finally a broadcast), instead of relying on well researched and optimized internal implementations abstracted by the `MPI_ALLREDUCE` operation.

5.1.6 Parallel IO

MPI offers an abstraction to parallel file systems through its MPI-IO API introduced in version 2.0 of the standard. The Input-Output (IO) API makes applications portable across multiple distributed file system implementations. In contrast to POSIX or proprietary parallel IO APIs, MPI IO benefits from its integration with MPI. For example, its operations can work transparently with MPI data types, including complex derived data types created by MPI application developers.

Similar to other parts of the MPI standard, the IO API is designed to allow for efficient implementations. There are several optimizations possible. For example, since the networks of HPC systems typically have lower latencies and higher bandwidth than their file systems, implementations can rely more on the network and minimize the amount of accesses to the distributed parallel file system. Implementations can streamline the order and reduce the number of IO operations.

5.1.7 Virtual Topologies

The development of distributed memory applications pose additional challenges to computer scientists. Any type of abstraction that can simplify the description of distributed algorithms is a welcome addition to any programming model. MPI virtual topologies is one such feature: developers can define them to simplify the implementation of distributed algorithms, while at the same time exposing communication patterns to the MPI implementation. These patterns can be used by MPI libraries to improve the order of ranks based on the proximity of actual processes in the real network topology of a supercomputer.

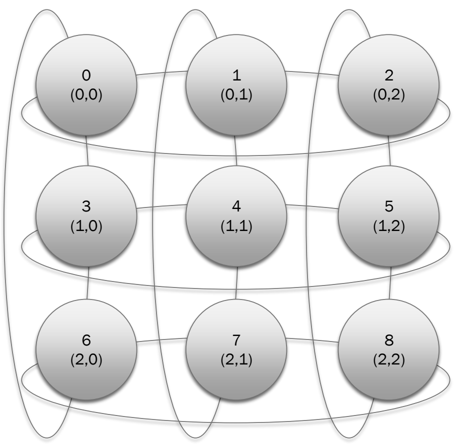

Figure 5.4: MPI processes organized in a 3 by 3 Cartesian grid virtual topology.

With virtual topologies, applications may define a Cartesian grid or a graph, where the nodes are the processes and the edges indicate which processes communicate with each other. The lack of an edge does not impede communication between processes, so arbitrary point-to-point communication is still possible. For example, a Cartesian grid topology may be defined to simplify the communication of an algorithm that operates on a checkered board type of data distribution. Figure 5.4 depicts the organization of 9 MPI processes in a 3 by 3 Cartesian grid virtual topology with wraparound. The top number indicates their ordered ranks in the `MPI_COMM_WORLD` communicator, while the bottom pair indicates their location in the Cartesian grid communicator. The more general case is the graph topology, where any arbitrary relationship can be described.

5.2 Dynamic Processes Support and its Limitations

A large number of scientific applications rely on MPI to achieve scaling with multiple nodes in HPC systems. Scaling these applications to large numbers of nodes is in many cases necessary due to memory and performance requirements.

The original MPI computational model was static in terms of the number of processes. Dynamic processes support was added in version 2.0 of the standard, mainly through the addition of the operations `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE`. Resources can be added to a running MPI application with the use of these operations. In this section, an enumeration of shortcomings in these operations is presented. These shortcomings will be later addressed by the proposed extensions, described in Chap. 6.

The spawn operations create new processes in a separate child process group, where the callers belong to the parent process group. The callers, or parents, block in the spawn operation, while the resource manager creates the new children processes. The new processes are created in a different way than processes created through the launcher command provided by the resource manager (e.g., `mpiexec`, `mpirun`, `srun`, etc.), where during the `MPI_INIT` operation the child process group needs to communicate with the parent process group to collectively generate an intercommunicator. This intercommunicator can then be used by application processes to reach the remote group (the parent group from the children, or the child group from the parents). The difference between the regular spawn and the multiple version, is that the later allows the callers to specify multiple binaries to start as children processes. The flow chart in Fig. 5.5 illustrates the algorithm for `MPI_COMM_SPAWN` in MPICH.

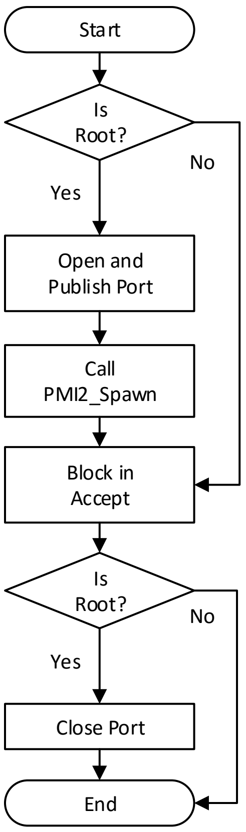

Figure 5.5: Algorithm (flow chart) of the `MPI_COMM_SPAWN` operation as implemented in MPICH.

Additional operations were also introduced so that applications could start independently and later connect their process groups, such as: `MPI_OPEN_PORT`, `MPI_CLOSE_PORT`, `MPI_PUBLISH_NAME`, `MPI_UNPUBLISH_NAME`, `MPI_LOOKUP_NAME`, `MPI_COMM_ACCEPT`, `MPI_COMM_CONNECT` and `MPI_COMM_JOIN`. With these operations, the creation of the new processes is done externally (e.g., with the provided launcher) and the groups establish communication and generate an intercommunicator with a combination of these operations afterwards.

The following shortcomings related to the MPI 2.0 dynamic processes operations have been identified:

1. The spawn operations are synchronous across both the parent and the children process groups.
2. These operations produce intercommunicators based on disjoint process groups.
3. Subsequent creations of processes result in additional process groups.
4. Destruction of processes can only be done on entire process groups.
5. The adaptation of resources can only be initiated by the application.
6. Processes created with spawn are typically run in the same resource allocation.

The first shortcoming affects performance. The spawn operations are collective across both the parent and child process groups. This is particularly taxing for the parent processes, since they block while the children process are being created by the resource manager, then complete their `MPI_INIT` operation and finally establish communication. The process creation delay can be several seconds in current systems. Additionally, the initialization process (the call to `MPI_INIT` after the children process starts) can also be several seconds (depending on the number of processes). A non-blocking version of these operations could be developed to overcome this issue.

The second shortcoming complicates the development of MPI applications and limits the amount of preexisting code that can be reused. Most computational kernels are designed around a single communicator (e.g., `MPI_COMM_WORLD` or a derived communicator) where the rank orders are important for the correctness of the algorithm. For example, many applications derivate communicators with virtual topologies (as explained in Sec. 5.1.7). Having multiple separate process groups complicates the generation of derived communicators. To be fair, an application can use `MPI_INTERCOMM_MERGE` to create a flat intracommunicator from the intercommunicator provided by these operations.

The third shortcoming also complicates the development of MPI applications that require dynamic processes. As mentioned before, an application can indeed use `MPI_INTERCOMM_MERGE` to create a flat intracommunicator from an intercommunicator provided by the current operations; however, subsequent creations of processes will require careful management of merged communicators. The application will have more and more process groups and intercommunicators as a result.

The fourth shortcoming limits what can be achieved with dynamic processes. With the current operations, only whole groups of processes can be destroyed by having them call `MPI_FINALIZE` and manipulating any previously generated intercommunicators or intracommunicators (e.g., through `MPI_INTERCOMM_MERGE`) so that these process groups are no longer included. This limits the granularity and location of the resources that the application can release at runtime.

The fifth shortcoming limits the quality of adaptations. Applications have no information related to the availability of resources or the state of other applications in the system. In contrast, resource managers have access to this information. In addition to this, resource managers have control over running applications, access to the pool of pending jobs, can pick which applications from the queue to start and when, etc. Inverting this control, where the resource adaptation is initiated by the resource manager, is preferable.

Finally, the last shortcoming limits the usefulness of the spawn operation. Most implementations that support `MPI_COMM_SPAWN` make it so that any processes are created in resources already allocated to the job. This is not a limitation of the standard operation itself, but it is found in typical resource manager implementations. The extensions presented in this work (in Chap. 6) are better integrated with the resource manager (described in Chap. 9) such that processes are created on new resources before being added to the allocation of a job; this ensures that new processes can contribute to the performance of the application, due to them running in additional hardware resources.

5.3 MPICH: High-Performance Portable MPI

One of the goals of this work is to overcome the limitations of the current standard dynamic processes support in MPI, as described in Sec. 5.2. For this, it was determined that an extension to MPI was necessary, with an accompanying implementation as proof of concept. For the extended implementation, there were two options: to develop a new MPI library from scratch or to extend an existing open source implementation of MPI. The later was chosen, since the proposed extension to MPI is compact (only four additional operations described in Chap. 6) and developing a completely new MPI library is a long and challenging undertaking.

There are two dominant open source MPI projects today: MPICH [209, 107, 109, 52, 53, 6] and Open MPI [96, 105, 103, 154, 104, 115, 7]. Given the maturity, network hardware support and performance levels of both software projects, both libraries were adequate as the basis for the MPI library prototype. MPICH was selected based in large part to the author's familiarity with this software project in previous work [71, 72, 69, 70, 195]. An overview of the software architecture of MPICH is presented in this section.

5.3.1 Software Architecture

The MPICH library's design contains three main internal software layers: the MPI, device and channel layers. The MPI application interacts with the MPI layer over the standard API. The MPI layer converts MPI operations into their device layer equivalents. Finally, the device decomposes each operation as one or more low level operations on specific network hardware. Network hardware is abstracted behind the channel API. For the prototype presented in this work, the CH3 device and the Nemesis channel implementations were selected, as shown in Fig. 5.6.

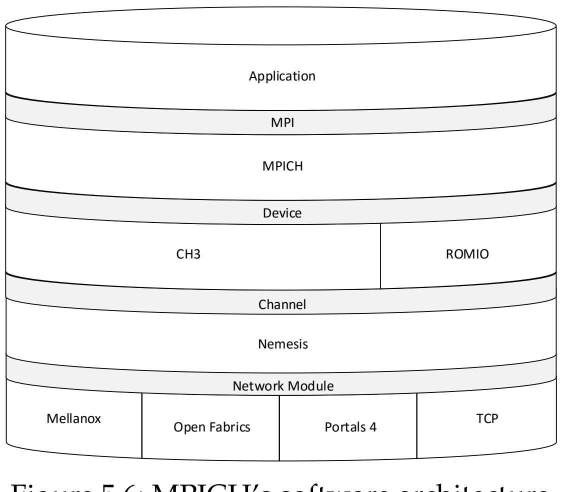

Figure 5.6: MPICH's software architecture.

MPICH's software architecture is preserved in this work and new non-standard MPI operations are added. These operations have their MPI and device layer counterparts, as well as any device and channel extensions that were necessary to support them.

5.3.2 MPI Layer

The MPI layer ensures compliance with the MPI standard. Additionally, parameter validation and errors and handled in this layer and propagated to the calling application. As expected given the tiered architecture, the MPI layer exchanges information exclusively with the device layer of MPICH and the caller application.

5.3.3 Device Layer

The device layer provides operations that are counterparts of every actual MPI operation that is part of the standard. These counterparts do not conform to the standard, and use instead implementation specific data structures and in some cases different numbers of parameters. Often multiple variants of these implementation specific operations are provided with different performance and scaling characteristics; these are selected by the MPI layer depending on the parameters passed by the calling application. For example, for point-to-point operations, the MPI layer may select the eager or rendezvous versions of the operations based on the size of the buffer to be transfered. In the case of collectives, in addition to buffer sizes, the MPI layer may select internal collective implementations based on the number of processes in the specified communicator.

5.3.4 Channel Layer

The lowest layer is the channel layer. This layer is designed mainly to abstract different types of networks. It provides low level byte transfer facilities, as well as network specific operation overloads for hardware accelerated functionality, such as RDMA and hardware collectives.

NEMESIS Channel and its Network Module (NETMOD) System

The Nemesis channel implementation provides a hybrid shared memory and network communication subsystem. This hybrid design allows for fast intra-node communication through shared memory, while preserving support for inter-node communication over network interfaces. The shared memory part of the implementation is essentially the same on all supported platforms; for it, only the type of shared memory model is selected (e.g., POSIX or System V), while the algorithms and data structures remain the same. The network part has multiple implementations available that are specific to different types of network hardware.

A network module (NETMOD) system is implemented within NEMESIS. This system adds an additional layer to the software architecture, as can be seen in Fig. 5.6. Similar to other layers, it has its own API and data structures. Support for different network hardware can be configured by selecting the correct network module.

6 Elastic MPI Library

The Elastic MPI Library is composed of an MPICH base with a set of new operations and their required resource manager integration. The new operations serve as an alternative to the standard dynamic processes support in MPI. The API extension and its implementation were designed with these primary goals in mind: latency hiding, minimal collective latency and ease of programming.

In this chapter, the interfaces of the new operations in the extension are first explained. Afterwards, their implementation and integration with the proposed elastic resource manager (covered in Chap. 9) are described.

6.1 MPI Extension Operations

A set of four operations that overcome the limitations of the dynamic processes support of MPI, discussed before in Sec. 5.2, is proposed. The operations were designed taking into account the needs and goals of developers of MPI applications, resource managers and MPI libraries. The design simplifies the development of elastic MPI applications, while allowing for efficient implementations of both the resource manager and the proposed new MPI operations in communication libraries.

The general flow of an MPI application is different with the use of the proposed extensions. First, the application initializes itself in adaptive mode by calling the new initialization operation. Afterwards, it starts doing its usual work and checking for adaptation instructions from the resource manager with the probe operation. In the event of an adaptation, the application needs to reach a safe location (typically at the beginning or end of a computational loop or phase), where it can begin the adaptation. Once an adaptation is completed, the application resumes doing progress on its computations.

To perform an adaptation, applications rely on the creation of adaptation windows with the new begin and commit adaptation operations. The begin adaptation operation provides the application with helper communicators; the application can use these communicators to redistribute its data. Once an adaptation is complete, the application calls the commit operation, where the `MPI_COMM_WORLD` communicator is transformed permanently. A code snippet of the expected structure of an elastic application that uses the proposed extensions is provided in Listing 6.1. Figure 6.1 shows a simplified sequence diagram of an application that starts with 5 processes and later expands to 7.

There are important differences between these operations and the current ones included with standard MPI:

1. During resource expansions, the new and preexisting process groups operate asynchronously.
2. The operations produce modifications on `MPI_COMM_WORLD` and its process group.
3. Subsequent adaptations result in the same number of process groups.
4. The resource manager can give instructions to destroy processes without group size restrictions.
5. The adaptations are initiated by the resource manager, and not the MPI application.
6. Resource managers can provide extra resources or remove resources of a job together with the adaptation instructions.

These differences allow the new operations to overcome the limitations of the standard dynamic processes API, as described previously (refer to Sec. 5.2). In the following sections, each operation is described in detail with their actual C and Fortran interfaces.

```
int main (int argn, char **argc){
  MPI_Init_adapt(&argn, &argc, &local_status);
  for (...){
    MPI_Probe_adapt(&adapt, ...);
    if(local_status == MPI_ADAPT_STATUS_JOINING
        || adapt == MPI_ADAPT_TRUE){
      MPI_Comm_adapt_begin(...);
      // adaptation window's body with
      // data redistribution code
      MPI_Comm_adapt_commit(...);
    }
    // compute and MPI code
  }
}
```

Listing 6.1: Expected structure of an elastic MPI application.

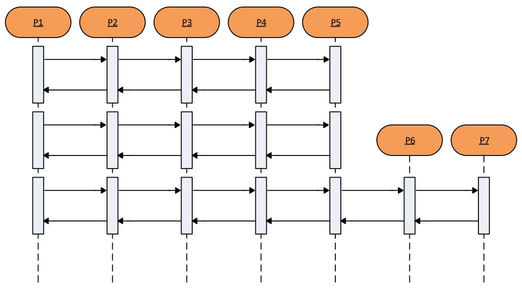

Figure 6.1: Adaptation sequence from 5 to 7 processes.

6.1.1 MPI Initialization in Adaptive Mode

The first operation in the extension allows applications to be initialized in adaptive mode. Its interface is identical to the standard `MPI_INIT` with the exception of an extra output parameter: `local_status`. The C and Fortran versions of this operation can be seen in Listings 6.2 and 6.3.

```
int MPI_Init_adapt(
      int *argc,
      char ***argv,
      int *local_status
      );
```

Listing 6.2: `MPI_INIT` C interface.

```
SUBROUTINE MPI_INIT_ADAPT(&
            local_status,&
            ierror)
      INTEGER local_status, ierror
END SUBROUTINE MPI_INIT_ADAPT
```

Listing 6.3: `MPI_INIT` Fortran interface.

In this operation, the output parameter `local_status` can take two possible values: `MPI_ADAPT_STATUS_NEW` or `MPI_ADAPT_STATUS_JOINING`. It is set to new when the process doing the MPI initialization was created through the launcher command provided by the resource manager (e.g., `mpiexec`, `srun`, etc.). In contrast, when the process doing the initialization was created by the resource manager, as part of an expansion of resources, then the local status is set to joining.

The addition of this extra output parameter at initialization is necessary since joining processes need to call the `MPI_COMM_ADAPT_BEGIN` operation immediately after initialization, so that they can take part in an adaptation window together with preexisting processes and other joining processes.

6.1.2 Probing Adaptation Data

The next proposed operation allows preexisting processes to probe the resource manager for adaptation instructions. As mentioned before, the decision of when and how to do the adaptation comes from the resource manager; this is an inversion of control when compared to the standard spawn operations offered by the dynamic processes support of the MPI standard.

The C and Fortran versions of the probe operation can be seen in Listings 6.4 and 6.5. There are a total of three output parameters in this operation.

```
int MPI_Probe_adapt(int *pending_adaptation,
                    int *local_status, MPI_Info *info );
```

Listing 6.4: `MPI_PROBE_ADAPT` C interface.

```
SUBROUTINE MPI_PROBE_ADAPT(current_operation,local_status,info,ierror)
      INTEGER pending_adaptation
      INTEGER local_status
      INTEGER info
      INTEGER ierror
END SUBROUTINE MPI_PROBE_ADAPT
```

Listing 6.5: `MPI_PROBE_ADAPT` Fortran interface.

The pending_adaptation parameter tells the application whether there is an adaptation pending with the values: MPI_ADAPT_TRUE or MPI_ADAPT_FALSE.

The local_status parameter tells the calling process what its individual status is. This variable has three possible values: MPI_ADAPT_STATUS_JOINING, MPI_ADAPT_STATUS_STAYING or MPI_ADAPT_STATUS_LEAVING. Each process is required to operate based on its local status inside adaptation windows.

A process is joining if it was created by the resource manager in newly allocated resources. In this case, it is redundant information from MPI_INIT_ADAPT, and joining processes can skip calling the probe operation and proceed to call the MPI_COMM_ADAPT_BEGIN operation immediately. Joining processes block in the adapt begin operation before the parents. Preexisting processes are notified about the adaptation only after all joining processes are blocking at the MPI_COMM_ADAPT_BEGIN operation. This allows preexisting processes to continue doing progress without interruption while scheduling decisions are taking place and new processes are being created, effectively hiding those latencies.

Preexisting processes can receive the status staying or leaving. All processes that are required to stay in the process group, after the MPI_COMM_ADAPT_COMMIT operation modifies the MPI_COMM_WORLD communicator, receive the status staying; in contrast, all processes that will be eliminated from the MPI_COMM_WORLD communicator, after the MPI_COMM_ADAPT_COMMIT operation completes, receive the status leaving.

Finally, there is an MPI_INFO object that can be used to provide additional information from resource managers. This parameter is optional: it can be used to provide implementation specific information or alternatively be set to NULL by implementors.

```
int MPI_Probe_adapt(int *pending_adaptation, int *local_status,
                    int *nfailed, int *failed_ranks, MPI_Info *info);
```

Listing 6.6: MPI_PROBE_ADAPT C interface with fault information.

```
SUBROUTINE MPI_PROBE_ADAPT(pending_adaptation,local_status,&
                           nfailed,failed_ranks,info,ierror)
    INTEGER pending_adaptation
    INTEGER local_status
    INTEGER nfailed
    INTEGER failed_ranks (*)
    INTEGER info
    INTEGER ierror
END SUBROUTINE MPI_PROBE_ADAPT
```

Listing 6.7: MPI_PROBE_ADAPT Fortran with fault information.

Potential for Fault Tolerance Support

With the proposed API extensions, there is potential to support adaptations as a result of failures with a modified version of the MPI_PROBE_ADAPT operation. Listings 6.6 and 6.7 show the modified interfaces with two additional output parameters that provide a list of failed ranks in the MPI_COMM_WORLD communicator.

The pending_adaptation output parameter has an additional possible status to indicate that the application needs to adapt its resources due to failures: MPI_ADAPT_FAULT. In such a case, the application is required to read the nfailed and failed_ranks parameters; these values should be used to prevent the application from initiating communication with failed ranks. This additional operation can be complementary to ongoing efforts from members of the Fault Tolerance Working Group of the MPI forum [140, 45, 47, 40]. The addition of this operation alone is not sufficient for fault tolerance. For example, application processes may be blocking waiting for failed processes and may not reach the probe operation.

6.1.3 Beginning an Adaptation Window

The third operation marks the start of an adaptation window. This operation provides two communicators as output: one intercommunicator that is similar to the one provided by standard spawn operations, and one intracommunicator that gives an early view of the future `MPI_COMM_WORLD` communicator. In addition to the two helper communicators, the counts for staying, leaving and joining processes are also provided. These counts can be used by applications to compute any necessary dimensions for their repartitioning schemes. The C and Fortran interfaces for this operation are presented in Listings 6.8 and 6.9.

```
int MPI_Comm_adapt_begin(
        MPI_Comm *intercomm, MPI_Comm *new_comm_world,
        int *staying_count, int *leaving_count, int *joining_count
        );
```

Listing 6.8: `MPI_COMM_ADAPT_BEGIN` C interface.

```
SUBROUTINE MPI_COMM_ADAPT_BEGIN(intercomm,new_comm_world,&
                   staying_count,leaving_count,joining_count,ierror)
    INTEGER intercomm
    INTEGER new_comm_world
    INTEGER staying_count
    INTEGER leaving_count
    INTEGER joining_count
    INTEGER ierror
END SUBROUTINE MPI_COMM_ADAPT_BEGIN
```

Listing 6.9: `MPI_COMM_ADAPT_BEGIN` Fortran interface.

It is up to the application to make calls to this operation in a location where it is safe for it to adapt to new resources. In general, it is expected that it takes place inside of a progress loop. Frequent checks for adaptations are desirable to minimize the idle times of joining processes in newly added resources.

Each process that is staying or joining is required to read its future rank and size from the provided `new_comm_world` helper communicator to perform adaptations consistently.

Processes that are leaving during the adaptation window will not have access to the future `MPI_COMM_WORLD`, since a leaving process will be removed from the process group; leaving processes' `new_comm_world` will be set to `MPI_COMM_NULL`. These processes will need to be reached over the provided `intercomm` from the children, or their current `MPI_COMM_WORLD` from the parents, during an adaptation window.

6.1.4 Committing an Adaptation Window

The last operation commits the adaptation. Its interface takes no parameters. The C and Fortran versions only include the return error code or the error parameter, as presented in Listings 6.10 and 6.11.

```
int MPI_Comm_adapt_commit();
```

Listing 6.10: MPI_COMM_ADAPT_COMMIT C interface.

```
SUBROUTINE MPI_COMM_ADAPT_COMMIT(ierror)
    INTEGER ierror
END SUBROUTINE MPI_COMM_ADAPT_COMMIT
```

Listing 6.11: MPI_COMM_ADAPT_COMMIT Fortran interface.

This operation modifies MPI_COMM_WORLD: any leaving processes are eliminated from it, and any new joining processes are inserted into it. The MPI_COMM_WORLD communicator will match exactly the new_comm_world communicator provided by the MPI_COMM_ADAPT_BEGIN operation after MPI_COMM_ADAPT_COMMIT completes.

This operation also notifies the resource manager that the current adaptation is complete. This is necessary to prevent the resource manager from triggering a new adaptation while one is still ongoing, since adaptation windows are not allowed to be nested.

An application that creates any derivative communicator is required to reconstruct them after this operation. Additionally, any libraries that use MPI will need to be reinitialized as well; this will require support from external libraries such as linear algebra packages, visualization libraries, performance tools, and so forth.

6.2 MPI Extension Implementation

As previously mentioned, the proposed API extensions have been designed so that efficient implementations are possible. The MPICH library has been taken as basis for the initial prototype. The library is largely designed and optimized for static process groups that keep their set of resources fixed during the full runtime of an application. Because of this, a large number of small changes were necessary to support these new operations.

In this section, mainly the high level algorithms of each proposed operation in the extension are described. Important design decisions related to performance will be explained.

6.2.1 MPI_INIT_ADAPT

The addition of the new initialization routine only requires that the Process Management Interface (PMI) [36, 61, 60, 198] provides the extra parameter to the application. This information is provided by the resource manager (covered in Chap. 9), and is simply propagated to each application process.

Currently, internal data structures in MPICH are optimized based on the assumption that the size of the MPI_COMM_WORLD communicator never changes. In the future, this operation could initialize the MPICH library with data structures that are better designed for elastic execution.

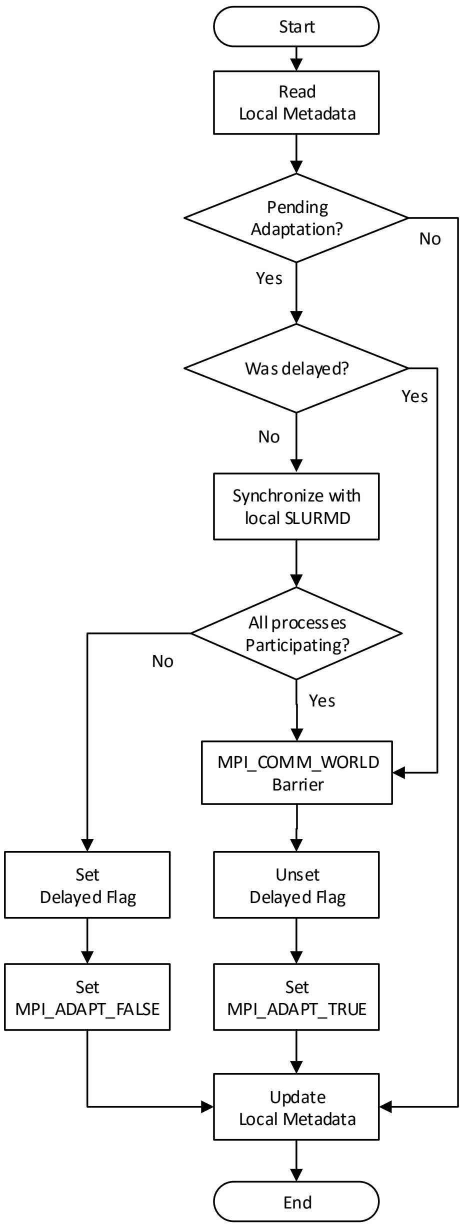

Figure 6.2: Flow chart of the `MPI_PROBE_ADAPT` operation.

6.2.2 MPI_PROBE_ADAPT

As mentioned before in Sec. 6.1.2, this operation provides information to the application. The implementation presented here has been optimized based on two assumptions:

1. This operation will be called periodically and as frequently as possible to minimize the idle times of joining processes in newly allocated resources.

2. The result of the vast majority of calls to this operation will output the value `MPI_ADAPT_FALSE` in the `pending_adaptation` variable.

The operation reads the adaptation metadata set by the resource manager at each node, from a shared memory segment. For this operation, a race condition is only possible when the adaptation flag is set by the resource manager and one or more processes have not read it, while one or more other processes have read it already. This potential race is managed by the probing algorithm.

The flow chart of the probing algorithm is presented in Fig. 6.2. Each process has a counter for entering and another counter for exiting the probe operation. When a process reads that there is an adaptation pending, it proceeds to contact its local SLURMD to synchronize. If the SLURMD daemon detects that not all processes are waiting for instructions and that some have exited the operation (based on the counters and metadata), it releases all local processes and inserts a delay. The network of SLURMD daemons for the application synchronize over the SRUN component, and all insert the delay if one or more of them detected early processes.

In summary, this algorithm depends on interactions between the MPI library and the resource manager over the PMI. It is therefore split between the MPI library and distributed components of the resource manager. The resource manager's side of this algorithm is described in Sec. 9.3.

The performance consequence of this implementation is that the processes simply read and write a few bytes of metadata from a shared memory segment for the case of no adaptation. These reads and writes are performed without synchronization based on a non-blocking access pattern. Each process has exclusive write access to a single memory block; this eliminates the need of locks or any other form of synchronization, and makes the operation very fast for the case of no adaptations. Assuming that this is the most common case, this improves performance greatly at the cost of possible delays in events where there is an adaptation and the race condition is detected. Additionally, when adaptations are required, the aggregated performance penalty of delay insertions depends on the probability that the race condition occurs.

6.2.3 MPI_COMM_ADAPT_BEGIN

The MPI_COMM_ADAPT_BEGIN operation is by far the most important operation in terms of performance. Special attention was given to its design. The perspectives of application, MPI library and resource manager developers were considered when defining its interface.

The main performance benefit of this operation is its ability to hide the latencies related to resource management and MPI process creation. These are:

1. Scheduling and job metadata management at the resource manager.
2. MPI processes creation on new resources.
3. The time of the MPI_INIT_ADAPT operation on newly created processes.
4. Entry on MPI_COMM_ADAPT_BEGIN routine at newly created processes.

The last three latencies only apply when performing resource expansions. These latencies can amount to several seconds, and it is therefore desirable to hide them from the preexisting processes of a job, so that their progress is not interrupted.

These latencies are hidden in part by inverting the order of the MPI_COMM_ACCEPT and MPI_COMM_CONNECT operations, when compared to the MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE operations as implemented in MPICH.

The flow chart of the algorithm is presented in Fig. 6.3. The children processes are created by the resource manager and acquire the status joining from the MPI_INIT_ADAPT

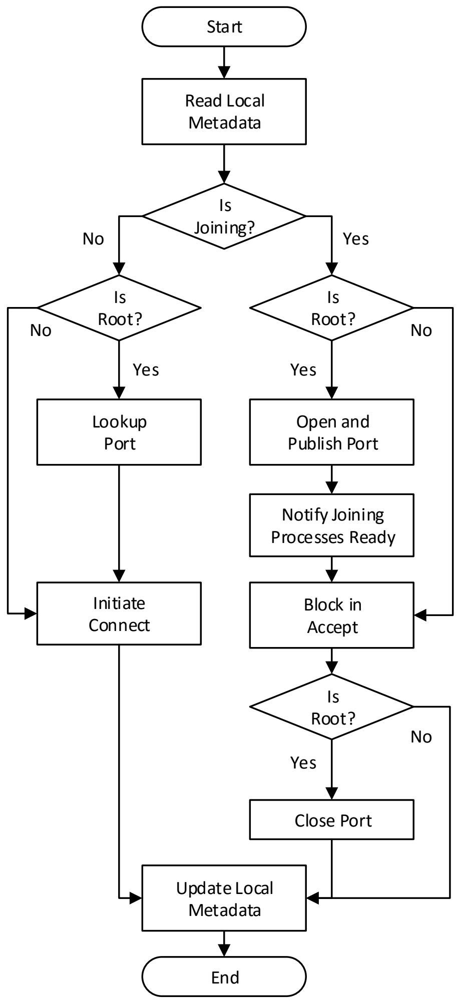

Figure 6.3: Flow chart of the MPI_COMM_ADAPT_BEGIN operation.

operation. They then proceed to call the MPI_COMM_ADAPT_BEGIN operation, open a port for the preexisting processes to find, and perform an accept operation on it. Right before blocking on the accept operation, the root process on the joining process group notifies the resource manager that these processes are ready to join the preexisting processes. The resource manager then notifies all preexisting processes. Once the preexisting processes probe for instructions, they will call the MPI_COMM_ADAPT_BEGIN operation. Inside the operation, they do a lookup on the port name of the newly created processes, and proceed to perform a connect on it. At that point, the MPI_COMM_ADAPT_BEGIN operation's algorithm is very similar to the standard MPI_COMM_SPAWN operation, and it generates an intercommunicator that is equivalent to performing a spawn. Additionally, it computes a process group that contains all staying and joining processes; any leaving processes are excluded from this process group. Based on this process group, the new_world_comm communicator is generated. Both of these communicators are then passed to the application as outputs. Each staying and joining process is expected to read its new rank during the adaptation window, from the new_comm_world communicator. Finally, the counts of staying, leaving and joining processes are computed and provided as output.

6.2.4 MPI_COMM_ADAPT_COMMIT

This operation is relatively fast when compared to the begin adaptation operation, with only an internal implicit barrier as its main communication overhead. It is possible to remove this barrier with some rework on the internals of the MPICH library, specifically on its virtual channel creation algorithm. Unfortunately, in the current implementation the library fails in certain sequences when a virtual channel is being created and the target process is not ready. This situation can arise, although perhaps infrequently, if the barrier is removed from the current implementation.

The commit operation modifies the internal metadata of the local process without the need of extra information: it takes the process group of the generated `new_comm_world` in the `MPI_COMM_ADAPT_BEGIN` operation, and updates the local process' world group and `MPI_COMM_WORLD` communicators based on it. After that, it cleans up any unnecessary metadata, internal process groups and helper communicators generated for the adaptation window. Finally, it destroys any user created communicators and process groups, since they are now invalid given the addition or reduction of MPI ranks in the `MPI_COMM_WORLD` communicator. As mentioned before, applications are expected to recompute any derived communicators and to reinitialize any libraries after this operation completes and before doing progress.

7 Elastic-Phase Oriented Programming (EPOP)

During the development of the Elastic MPI library and supporting applications, difficulties faced by application developers were identified. There were two main difficulties in non-trivial software projects: dealing with the difference between preexisting processes and joining processes, and implementing data redistribution schemes.

Elastic-Phase Oriented Programming (EPOP) attempts to improve the structure of elastic distributed memory applications to help developers cope with these additional challenges of resource-elastic programming. In this chapter, arguments to justify the use of EPOP for the development of resource-elastic applications are first presented. Afterwards, its current implementation is described in detail. Finally, additional benefits that EPOP and its implementation can provide in the future are discussed.

7.1 Motivation for a Resource-Elastic Programming Model

In this section, how EPOP can help both developers and runtime systems is described. It helps developers manage the complexity of resource-elastic distributed memory software by improving code structure with clearly defined phases and redistribution code. Runtime systems can rely on these definitions to better analyze performance and, as a consequence, take better scheduling decisions that optimize individual applications while at the same time maximizing system-wide efficiency metrics.

7.1.1 Identification of Serial and Parallel Phases in the Source Code

As mentioned in the motivation (Chap. 2), the focus of this work is mainly on the analysis and optimization of strong scaling computational phases since their efficiency is largely dependent on resource allocations; the efficiency of weak scaling computational phases is in general independent of allocated resources since they fix the amount of computational work to be done per resource unit.

The efficiency of strong scaling phases follows Amdahl's law. Let p be the number of processes (properly pinned to a hardware computing resource, such as a CPU) and $\tau(p)$ be the time these processes take to finish the computational work of a particular phase. The reduction of time due to an increase from a reference p_r processes to p_n new processes allocated to a phase is defined as $\delta(p_r, p_n)$ and its reciprocal, the speedup, as $\Gamma(p_r, p_n)$. The following formula describes their relationship:

$$\delta(p_r, p_n) = \frac{\tau(p_n)}{\tau(p_r)} = \frac{1}{\Gamma(p_r, p_n)}. \tag{7.1}$$

Amdahl's law separates single process performance, $\tau(1)$, into two parts: a serial and a parallel part. The serial part is defined here as τ_s and the parallel part as τ_p. Since only the parallel part of the execution time of a phase can be reduced with additional processes, it follows that:

$$\tau(p) = \tau_s + \frac{\tau_p}{p} \tag{7.2}$$

Amdahl's law states that the best speedup possible, from a reference of 1 process, is limited by the inequality:

$$\Gamma(1, p_n) = \frac{\tau(1)}{\tau(p_n)} \leq \frac{\tau_s + \tau_p}{\tau_s + \tau_p/p_n}. \tag{7.3}$$

We can conclude from Amdahl's law that resource adjustments will only have an impact on the performance of parallel phases. Therefore, on elastic execution environments all serial phases should be ignored when performing performance analyses and resource adjustments; the identification of these types of phases is therefore of increased importance in elastic execution environments.

These parallel and serial phases are in many cases identifiable by the developers of the application, but are difficult for compilers and profilers to automatically identify. EPOP allows developers to explicitly identify phases that are parallel and elastic, and any other remaining phases of the application as rigid.

Elastic runtime systems can take advantage of the clear definition of parallel elastic phases by only analyzing and optimising the resource allocations of these, and ignoring all others. In reality, there will be some serial blocks in parallel elastic phases and therefore the scalability of these phases will as well follow Amdahl's law; in spite of this, the advantage of having clearly defined phases that are targets for optimization is still great.

7.1.2 Process Entry and Data Redistribution Locations

A performance goal for elastic applications is to minimize the time required to integrate new resources into ongoing simulations. This can be achieved by having entry points be reached as frequently as possible during the run of an application.

Typical scientific simulations are composed of several blocks of code and each of these can run for several minutes, hours and in some cases days. It is therefore beneficial for elastic applications to define multiple entry points for new processes. Furthermore, this also helps schedulers reduce the idle time of resources with minimal need for oversubscription.

Typical MPI programs have several branches in complex control structures. This makes the code hard to statically analyze by performance tools and in many cases even application experts. This situation is even more severe in elastic MPI programs, where additional branching and control structures are introduced due to the difference between preexisting and joining processes.

Given that allowing new processes to join at arbitrary locations of the program is beneficial to the application and system-wide efficiency, it is of great importance to simplify the definition of these locations and to minimize the need for complex control structures.

```
int main (int argn, char **argc){
  MPI_Init_adapt(&argn, &argc, &local_status);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // initialization block
  if(local_status == MPI_ADAPT_STATUS_NEW){
    phase_index = 0;
  } else { // joining process
    MPI_Comm_adapt_begin(&intercomm, &new,
        &staying_count, &leaving_count, &joining_count);
    // redistribution block
    MPI_Comm_adapt_commit();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Bcast(&count, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&phase_index, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
  if(phase_index == 0){
    while (count <= phase0_passes){
      MPI_Probe_adapt(&current_operation, &local_status, &info);
      if(current_operation == MPI_ADAPT_TRUE){
        MPI_Comm_adapt_begin(&intercomm, &new,
            &staying_count, &leaving_count, &joining_count);
        // redistribution block
        MPI_Comm_adapt_commit();
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Bcast(&count, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&phase_index, 1, MPI_INT, 0, MPI_COMM_WORLD);
      }
      // phase 0 computation and communication block
    }
    phase_index++;
  }
  if(phase_index == 1){
    while (count <= phase1_passes){
      MPI_Probe_adapt(&current_operation, &local_status, &info);
      if(current_operation == MPI_ADAPT_TRUE){
        MPI_Comm_adapt_begin(&intercomm, &new,
            &staying_count, &leaving_count, &joining_count);
        // redistribution block
        MPI_Comm_adapt_commit();
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Bcast(&count, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&phase_index, 1, MPI_INT, 0, MPI_COMM_WORLD);
      }
      // phase 1 computation and communication block
    }
    phase_index++;
  }
  MPI_Finalize();
  return 0;
}
```

Listing 7.1: Simple elastic MPI application with 2 phases.

In EPOP, parallel elastic phases that have redistribution code are Elastic-Phases (EPs). The creation of EPs clearly defines process entry and redistribution locations, and allows the runtime system to insert new resources into the application without the need of complex control structures to be developed by the programmers of elastic applications. Listing 7.1 shows a very simple elastic MPI C program with 2 phases. In the code, each process needs to enter the correct redistribution block and then follows into the correct computational and communication block of the phase. As can be observed, the control structure is already quite complex due to the multiple possible paths depending on whether a process is: newly created, preexisting in phase 0, joining into phase 0, preexisting in phase 1, or joining into phase 1.

7.2 The EPOP Programming Model

EPOP is a programming model that has abstractions that help programmers modularize the blocks of computation, the data and the control flow of potentially, but not necessarily, resource-elastic applications. In this section, these abstractions will be described in detail.

7.2.1 Initialization, Rigid and Elastic-Phases (EPs)

Phases are used to represent the computational blocks of an application in the EPOP model. At the core of EPOP is the concept of Elastic-Phases, or EPs for short; these phases are used to represent blocks of code where resource adaptations are possible. There are also two other types of phases. Here is a more detailed description of the phase types:

- **Initialization Phase**: Each application must have one and only one initialization phase. These are called only once at each application process launch. The data of each EPOP application is first allocated and initialized in these phases.
- **Rigid-Phase**: These are phases where resource adaptations are not possible. Applications that have no support for resource-elasticity can be defined with the use of these. Even on resource-elastic applications, these can be used for blocks where it does not make sense to allow adaptations, such as during finalization. These types of phases can define a loop, but this is optional.
- **Elastic-Phase**: These are phases that implement a code block, a redistribution block and a loop definition. In contrast to rigid-phases, the loop definition is mandatory for this type of phases. The computational block of each EP will be executed several times based on the definition of its own loop conditions. The adaptation block is triggered by resource managers when needed.

An illustration of a simple EPOP program with 4 phases is shown in Fig. 7.1. The program consists of one initialization phase (as required by the model), two EPs and a rigid-phase for finalization. As can be seen in the unrolled loop view, each EP is also a loop and the application data is passed to its computational block on each iteration.

7.2.2 EPOP Programs and Branches

A collection of phases can be combined with branches to create an EPOP program. Branches can be if, if-else or similar constructs that operate at the phase level. Branches in EPOP determine the next phase in an execution order based on its conditions.

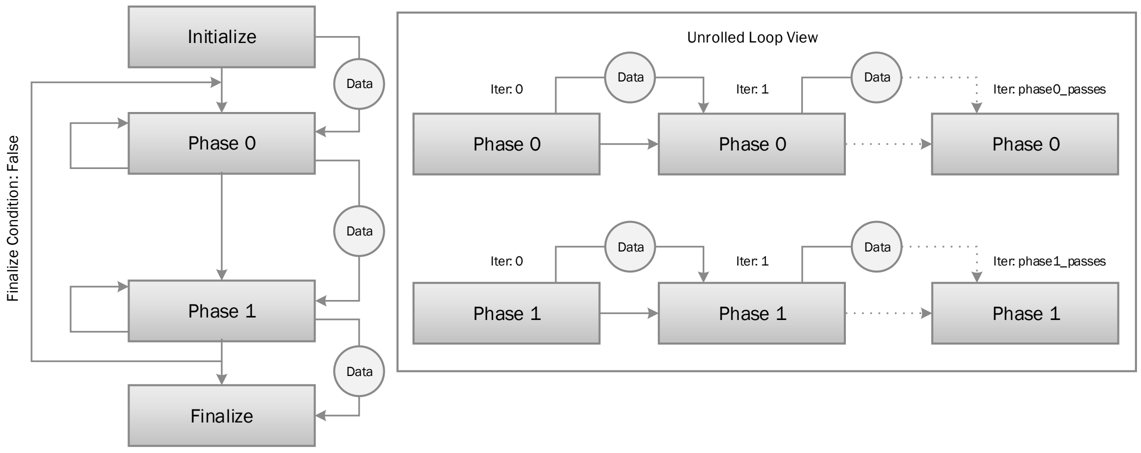

Figure 7.1: Program structure of the simple EPOP example (with source in Listing 7.6).

A branch can select any phase except the initialization phase; the initialization phase is restricted to have a single execution. In the example shown in Fig. 7.1, a branch determines whether to finalize the application after *Phase 1*, or if to continue execution from *Phase 0*.

7.2.3 Application Data

All application state is held outside of the phases in the application data, similarly to functional programming languages. Instead of holding state, all static or allocated application data and file descriptors are required to be created in the data block only and then passed around between phases. Once the computations of a block are applied, the data is passed to the next computational block based on the control flow of the program.

In the example presented in Fig. 7.1, it can be observed that the data of the application is passed from phase to phase. In the case of the EPs, the data is also passed again to its computational block on each iteration. Finally, if the branch is taken, the data is then passed from the EP *Phase 1* to the EP *Phase 0*.

7.3 Current Implementation

A minimalistic implementation of EPOP is presented in this section. It is currently implemented in the C programming language. The implementation will be used to show more concretely how EPOP solves the original problem of improving the structure of elastic MPI programs, while also providing some additional benefits, like simplifying the creation of performance models to assist runtime schedulers.

7.3.1 Driver Program

EPOP programs are not compiled as executable binaries. Instead, they are built into archives that are loaded by a driver program. Currently only one driver is provided with EPOP, but multiple drivers can be provided later since driver programs are independent of EPOP applications. Profiling and automatic tuning drivers could be added to the implementation in the future.

Each EPOP program implements a `get_program()` routine that returns a program definition. The program definition includes its size, a program structure and some other metadata that describes it.

```
mpiexec -n 1024 epop ./double_phase_example <parameter1> <parameter2>
```

Listing 7.2: EPOP + MPI application started with an MPI launcher. Multiple EPOP drivers are started (each with its own instance of the EPOP application and common parameters).

EPOP applications are not necessarily also MPI applications. In fact, these applications don't even need to be parallel to be implemented with the EPOP framework. For EPOP + MPI applications, multiple driver programs are launched. Each driver instance created by the MPI launcher will load the EPOP + MPI application in a typical SPMD style launch. Listing 7.2 shows a sample launch command with a typical `mpiexec` launcher provided by some resource manager and MPI library combinations.

```
// the initialization phase is called before this code block
// the initial pc value is provided by the resource manager
while(pc < program->size){
  switch(program->elements[pc].type){
    case EP:
      do {
        // probing for adaptations is done here
        if(resource_adaptation)
          program->elements[pc].phase_adapt(program->data);
        program->elements[pc].phase_exec(program->data);
      } while(program->elements[pc].loop_condition(program->data));
      pc++;
      break;
    case RP:
      do{
        program->elements[pc].phase_exec(program->data);
      } while(program->elements[pc].loop_condition(program->data));
      pc++;
      break;
    case BRANCH:
      pc = program->elements[pc].branch_condition(pc, program->data);
      if(pc == 0 || pc > program->data) return -2;
      continue;
    default:
      return -1;
  }
}
```

Listing 7.3: Simplified example control loop of a driver program.

Listing 7.3 shows a simplified example of a control loop used to implement a driver program. The program counter (`pc` variable) is used as an index in an array of EPs, rigid-phases and branches, and each is called according to its type with the program's data.

7.3.2 Program Element

As seen in the sample code in Listing 7.3, the driver program traverses an array of both phases and branches. This is possible with the use of the program element structure as the type for the elements of the array used by the implementation. A program element can be one of the three phase types or a branch. In the sample driver code, there is no case for the initialization program element type, since these are called only once before entering the driver loop. The current C language implementation of the program element type is shown in code Listing 7.4.

```
typedef struct {
  int type;
  void (*init)(int*, char***, void**);
  void (*phase_adapt)(void*);
  void (*phase_exec)(void*);
  int (*loop_condition)(void*);
  int (*branch_condition)(int, void*);
} program_element_t;
```

Listing 7.4: C structure of the program element.

The compute blocks of phases resemble kernels in other programming models. Each compute block defined in nearly the same way as a PThread [169, 1, 172], with a single void pointer as input and an integer return type for error handling by the driver program. The adaptation block has the same interface, but has a different purpose. When the driver program interacts with the resource manager and determines that there is an adaptation to be performed, this routine is called on preexisting processes. On joining processes, the driver program proceeds to get the program counter value from the resource manager first, and then calls the adaptation block of the appropriate phase. This design eliminates the need for complex control structures found in regular resource-elastic MPI applications.

The branch type can be used for arbitrary jumps. This is achieved by modifying the program counter passed to the branch implementation by the driver program. The branch operation takes both the application data and the program counter. Differential or absolute jumps can be implemented by either computing a concrete value for the program counter, or by adding or subtracting a differential to its current value. An example code snippet of an `if-else` block based on differential jumps can be seen in Listing 7.5.

```
int branch(int pc, void *appdata){
  if (appdata->branch_condition == 0) return (pc+1);
  else return (pc+2);
}
```

Listing 7.5: C code for a differential branch performing an `if-else` operation. The driver continues to the program element at `pc+1` or `pc+2` depending on the `appdata->branch_condition` variable. The program structure must be assembled with the correct order in the `get_program()` routine.

```
int init(int *argn, char ***argc, void **data){
  // initialization block
  MPI_Init_adapt(NULL, NULL, &((*appdata).status));
  MPI_Comm_rank(MPI_COMM_WORLD, &((*appdata).rank));
  MPI_Comm_size(MPI_COMM_WORLD, &((*appdata).size));
  *data = (void*) appdata;
}
int adapt(void* data){
  MPI_Comm_adapt_begin(&intercomm, &new,
      &staying_count, &leaving_count, &joining_count);
  // redistribution block
  MPI_Comm_adapt_commit();
}
int phase0(void* data){
  // phase 0 computation and communication block
}
int phase0_condition (void *data){
  if (appdata.count <= appdata.phase0_passes) return 1;
  else return 0;
}
int phase1(void* data){
  // phase 1 computation and communication block
}
int phase1_condition (void *data){
  if (appdata.count <= appdata.phase1_passes) return 1;
  else return 0;
}
int fini(void* data){
  MPI_Finalize();
}
program_t *get_program(void){
  program_t *program = malloc( sizeof(program_t) );
  epop_alloc_phases(&program, 4);
  program->elements[0].type            = INIT;
  program->elements[0].phase_exec      = init;
  program->elements[1].type            = EP;
  program->elements[1].phase_adapt     = adapt;
  program->elements[1].phase_exec      = phase0;
  program->elements[1].loop_condition  = phase0_condition;
  program->elements[2].type            = EP;
  program->elements[2].phase_adapt     = adapt;
  program->elements[2].phase_exec      = phase1;
  program->elements[2].loop_condition  = phase1_condition;
  program->elements[3].type            = RP;
  program->elements[3].phase_exec      = fini;
  program->elements[3].loop_condition  = NULL;
  return program;
}
```

Listing 7.6: Simple elastic EPOP application (for comparison with MPI in Listing 7.1).

Examples of the initialization, EP and rigid-phase types can be seen in the double phase example in Listing 7.6; this example matches the code in the MPI example shown in Listing 7.1. When comparing the original MPI code to the EPOP conversion, it can be seen that once the program structure is created in the `get_program()` routine, the programmer no longer needs to track the branching structure of the application and can instead focus on the body of the computation and adaptation blocks of both elastic phases. In addition to this, the structure is exposed at the beginning of the application to the driver program; this structure can then be shared with other components such as the resource manager, a performance analyzer, etc. Finally, the driver program can report application progress to the resource manager or a tracker without extra development effort by the application programmer.

7.3.3 Program Structure

The program structure is a collection of program element instances and the application data. The array of program elements included in the structure is traversed by the driver program. The data is passed to each computational block.

This array is created in the `get_program()` routine of an EPOP application, as seen in the code snippet presented in Listing 7.6. The structure of an EPOP program is overlaid on the flat array of program elements. This array is not allowed to be modified at runtime. The driver program traverses it by modifying the program counter and calling each program element based on its type, as seen in the example code of the driver program in Listing 7.3. When a computational block completes, the program counter is incremented by one except when it is computed by a branch program element. Drivers are in complete control of the execution of EPOP programs through their program counters.

```
typedef struct {
  void *data;
  int size;
  program_element_t *elements;
} program_t;
```

Listing 7.7: C code of the program structure.

7.4 Additional Benefits of the EPOP Model and Driver Programs

The definition of phases in EPOP programs can help improve the quality of performance modeling techniques. The structure of programs is known before the application starts. A driver program can serialize the program structure of an EPOP application and transfer it to a resource manager or a monitoring program. A performance profile can then be populated by annotating the program structure with performance data. This can be done on completion or partially with updates at runtime. The types of the phases of an application are also known; because of this, performance modeling routines can be enabled selectively for EPs, while rigid-phases may be ignored especially when they do not loop.

Driver programs can also implement progress reporting transparently. A driver program can track phase changes and the iterations of phases that loop, and then transfer progress information to a resource manager. This can help increase the quality of resource adaptation and scheduling decisions, since the resource manager can better predict the future time when an application will complete. The rate of progress can be evaluated at different application processes to evaluate their load balance. The collective rate of progress can also be used to evaluate the quality of resource adaptations, since they correlate well with efficiency metrics.

A pausing mechanism can be added to driver programs. This can make preemption possible. It can also allow for the timing of applications or specific phases to be controlled. For example, the start time of an application can be delayed. Timing control can be useful for schedulers that try to minimize the idle times of nodes, since application processes may be created early if there is sufficient free memory in the nodes of an allocation. These processes can then be released immediately after the preceding application completes.

8 Resource Management in High Performance Computing

Resource managers for distributed memory systems have traditionally differed significantly from their shared memory counterparts. In general, implementations have tried to solve very different problems, and therefore relied on entirely different approaches and scheduling algorithms.

In this chapter, a brief introduction to resource management in shared memory systems is provided. Afterwards, resource management is covered more specifically in the context of distributed systems. Towards the end of the chapter, the state of the art of resource management in HPC is summarized. Additionally, necessary changes needed to support of elastic execution in distributed memory HPC systems are identified. These are used as motivation for the development of the resource manager presented in Chap. 9. Finally, an overview of the implementation of the SLURM resource manager is provided, since it was used as basis for the elastic resource manager presented in this work.

8.1 Resource Management in Shared Memory Systems

Computers had no operating systems in earlier times. Early compute systems used to run a single program until it completed. There was no need for resource management since the program had exclusive access to all the resources in the machine. Subsequent developments in computer hardware and software made it possible to share resources among multiple programs in a single computer. Once resources could be shared, there was the need for arbitration in cases of contention. Operating systems provide this needed arbitration and are the resource managers found in shared memory systems today.

Shared memory schedulers have been developed as part of operating systems. These execute tasks immediately without queuing, with some exceptions. For example, a system may create tasks periodically based on its time-based job scheduler; however, when the task is started it runs immediately. Work is not postponed or queued. Compute and other resources are shared in time as necessary. There are some exceptions, such as in the case of real-time systems, where space-sharing is still favored over time-sharing. Granting exclusive access to resources with space-sharing helps ensure that any required deadlines are met in real-time systems.

Processing units first come to mind when thinking of hardware resources that need management. In addition, an operating system needs to arbitrate access to IO (such as hard drives and other long term storage devices), network, random access memory, audio and video devices, keyboard and mouse, etc. Current systems support the concurrent execution of tasks. While having large numbers of tasks running concurrently has the benefit of increasing resource utilization, it also has the disadvantage of creating resource contention.

The scheduler of an operating systems controls the placement of tasks in processing resources and their timing to minimize contention. Different trade-offs are made by real-time, desktop and server oriented schedulers in shared memory systems. Schedulers can be optimized for real-time guarantees or for throughput. Real-time schedulers support the exclusive allocation of resources in space-sharing mode, while throughput optimized operating systems tend to operate in time-sharing mode exclusively. Desktop and server schedulers are throughput optimized. These differ in that resource utilization is maximized for servers, while a compromise between response time and resource utilization is targeted for desktop systems.

As mentioned before, HPC systems today are a collection of nodes interconnected with a high-performance network. Each of these nodes is a shared-memory system with its own local operating system. Their schedulers are throughput optimized and operate in time-sharing mode. Time-sharing is achieved through preemption on these schedulers: started tasks use a processing element, such as a CPU, for a time slot before being interrupted by the operating system to give a time slot to another task. Most algorithms employ a round robin strategy, where the time of computing resources is distributed equally among tasks of equal priority.

8.2 Resource Management in Distributed Memory Systems

Distributed memory schedulers have, for the most part, been developed separately from operating systems. These schedulers are implemented as separate software packages that run is user space (although usually under a super user). This is a consequence of the fact that there is no real distributed operating system to date; there have been some prototypes such as GNU Hurd [5] or Microsoft's Barrelfish [14], but no production solution to this date. More specifically, the major open source operating systems today (such as FreeBSD [15] and Linux [16]) do not support distributed memory. This is also true for proprietary operating systems such as Microsoft Windows or closed source Unix systems. Instead of being developed for operating systems, these schedulers have been developed for distributed resource managers; these operate in space-sharing mode, and are commonly referred to as batch schedulers.

As mentioned before, current distributed memory systems are collections of shared memory systems. Each shared memory systems that is part of a distributed system is typically referred to as a node. Current distributed systems are considered supercomputers if these nodes are interconnected by a high performance network, with optimized latencies and bandwidth. Each node, being a shared memory system itself, runs an operating system with a scheduler in time-sharing mode and optimized for throughput, as described in the previous section. One of the typical assumptions is that each computing element will run a single task, and special effort is made to minimize system noise.

Distributed memory resource managers have centralized and distributed components. Usually a resource management daemon is run at each node available for computation. Most, if not all, of current resource managers also require an additional dedicated node to run their centralized components. Utilities related to performance tracking, node status trackers, user reports, accounting, etc., run in a centralized manner together with the batch scheduler.

Batch schedulers are fundamentally different than time-sharing schedulers found in operating systems. One of the main differences is that jobs are not started immediately after they are created. Instead, these schedule jobs after they have been submitted by users and placed in a queue. Jobs are run in the background at some time in the future without any user interaction. Resources are managed at a coarser level, where full nodes are typically the resource unit instead of individual processing elements. These run in space-sharing mode, where each node is given exclusively to a job during its run. Some resource managers do allow for time-sharing as well, but this is not commonly enabled even when available, since the minimization of interference between jobs is of higher importance in the domain of supercomputing. The starting times of jobs is generally not guaranteed; this differs from time-sharing schedulers, where tasks are started immediately.

Batch scheduling in space-sharing mode with rigid allocations has been adequate up to the present date. For elastic execution, where applications can adjust to changes in computational resources at runtime, additional scheduling strategies need to be developed.

8.2.1 Additional Requirements for the Scheduling of Elastic Jobs

In addition to the exclusive access to resources and the minimization of interference required by HPC jobs, elastic jobs need continuous performance tracking. Since the resources of elastic jobs can be adjusted at runtime, additional scheduling strategies need to be developed that will maximize or minimize performance metrics in HPC systems (such as throughput, idle node count, power level stabilization, etc.) based on continuous resource adaptations.

Schedulers for elastic jobs require that their resource management infrastructure supports the modification of resource allocations. The number of resources of individual elastic jobs needs to be adjusted during their execution, based on their efficiency. Once a scheduler has determined that a resource adaptation is needed, it needs to be applied to the running system in some manner. When operating in distributed memory, an implementation needs to reconfigure the running system while ensuring consistency between its distributed components and the application processes of running jobs. Interactions between the distributed resource manager components need to be coordinated though communication protocols over the management network of the HPC system.

No resource managers for HPC systems today meet these additional requirements necessary to support elastic jobs. It is for this reason, that as part of this work, a new resource manager or an extension of an existing one was needed. After a long survey of available options, it was decided that extending the SLURM [12] workload manager to support elastic jobs was a good option.

8.3 Simple Linux Utility for Resource Management (SLURM)

The Simple Linux Utility For Resource Management (SLURM) [12] is a scalable workload manager currently being developed by SchedMD [11] and is released under the GNU General Public License (GPL). It is a highly scalable solution and quite popular in current HPC systems, including several of the fastest computers in the world today [19, 11].

From an implementation's perspective, SLURM can be seen as a collection of binaries that share a single configuration file. There is a significant amount of shared source code among these binaries, for purposes such as configuration parsing, plugin loading and management, communication protocols, among others things.

The collection of binaries that make up SLURM include: the controller (`SLURMCTLD`), the node daemons (`SLURMD`), the launcher (`SRUN`), the step daemons (`SLURMSTEPD`), as well as several account management and information utilities. These binaries are highly threaded and have a collection of plugin interfaces for several extensible functions, such as:

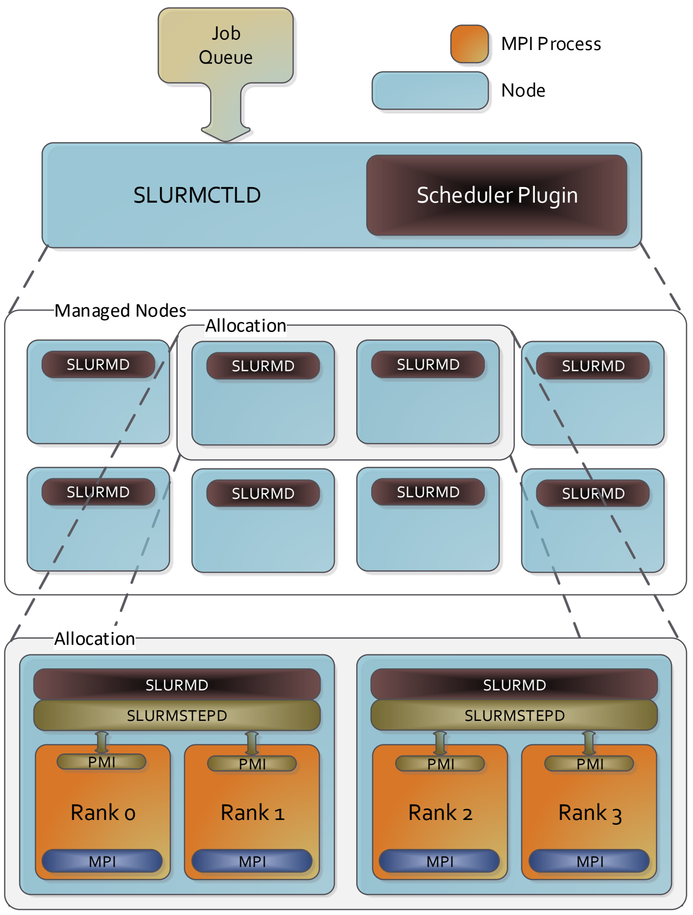

Figure 8.1: Abstract organization of a cluster based on SLURM and its main programs: SLURMCTLD, SLURMD and SLURMSTEPD. SRUN runs in the first node of an allocation (not shown).

scheduling, topology, MPI/PMI support, accounting, database support, and many more features. Figure 8.1 shows how these binaries interact and where they reside on an HPC system. The PMI and MPI libraries linked to MPI processes are also illustrated.

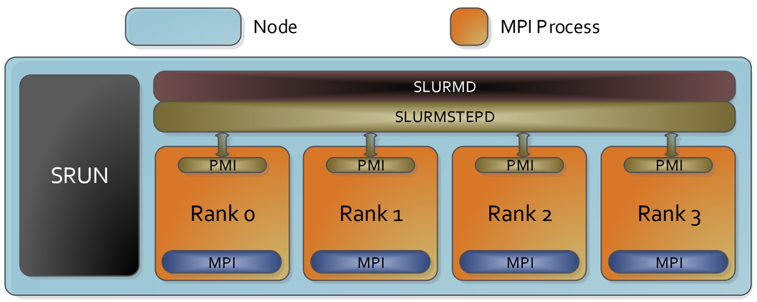

Figure 8.2: SLURMD, SLURMSTEPD, MPI processes and SRUN in the master node of an allocation.

8.3.1 Controller Daemon (SLURMCTLD)

The controller daemon (SLURMCTLD) is the only centralized component of SLURM. This component includes the batch scheduling algorithm, manages security, user accounts, tracks individual nodes (by reaching each distributed daemon), and so forth. Each of the user control and information binaries interact over the network with this component, such as:

- SRUN: Launch a parallel job (interactively or inside a batch).
- SBATCH: Submit a job defined in a batch script.
- SQUEUE: View information about submitted jobs that reside in the queue.
- SSTAT: Display status information about running jobs.
- SCANCEL: Used to cancel submitted jobs (running or queued).
- SACCT: Display accounting data about jobs in the system.
- SCONTROL: Used to view and modify the configuration of SLURM online.

There are many parts of SLURM that have been made modular and loadable as plugins. One of them is the batch scheduler algorithm. There are multiple implementations that can be selected though the configuration file.

8.3.2 Node Daemon (SLURMD)

The SLURMD binaries run as daemons on each node that is part of a partition. Each daemon monitors its individual node. They are also responsible for setting up and starting an additional type of daemon: the SLURMSTEPD. Daemons of this type track the node local part of a job step: the subset of application processes that run on its node. A job step in SLURM is an application started with SRUN and its allocated resources. Figure 8.2 illustrates the interaction between the SLURMD daemon of a node, the local SLURMSTEPD daemon, and the MPI processes through the PMI. The figure also illustrates how the SRUN binary runs independently of MPI processes in the master node of the allocation of a job step.

9 Elastic Resource Manager

The SLURM workload manager was selected as the basis for the development of the elastic resource manager presented in this document. It has been modified extensively for elastic resource management and integration with the Elastic MPI library.

Similarly to MPICH, SLURM has been designed for static resource allocations. There is some form of resource-elasticity, where a job can have an extra allocation as an expansion to its resources. This was not sufficient for the use cases considered in this work, since both increases and reductions to the resources given to MPI applications are needed. In addition, expansions through adaptation windows require that new processes be created on newly allocated resources; the creation of processes was only possible through the launcher in SLURM (e.g., `mpiexec` or `srun`).

The creation of a new resource manager may be required to support the elastic execution of MPI applications more elegantly and perhaps also more efficiently. Indeed, such efforts have been already initiated with the development of the Flux [31] resource manager. The Flux resource manager was still under development and was not mature enough to be used as the basis for the prototype presented in this document. Additionally, the creation of a new resource manager was not realistic within the time scheduled for this work. For these reasons, the extension of an existing and mature resource manager was determined as the most feasible option. SLURM was selected based on its compatibility, performance and widespread adoption.

Close integration with the MPI runtime is necessary for elastic applications. It is desirable that each created process is pinned to a hardware CPU core. In order to ensure that, the resource manager needs to create processes on new resources, and destroy processes (in coordination with MPI) when resources are taken from an application. New processes could be created in the same resources, allowing for oversubscription of CPU cores, but that is of little benefit to the performance and scalability of most HPC applications.

Multiple SLURM binaries and shared subsystems were modified to add support for resource-elasticity and integrate the new MPI operations. The most important changes will be described in this section.

9.1 Overview of the Integration with the Elastic MPI Library

An overview of the integration of both the Elastic MPI library (based on MPICH) and the Elastic Resource Manager (based on SLURM) is presented in this section. The interactions between the MPI library and the different resource manager components are explained in detail. These interactions were introduced to support the proposed extensions to MPI.

Figure 9.1 presents a high level view of nodes, resource manager components and MPI processes. The following resource manager components are labeled in the figure: `Elastic Batch Scheduler`, `Elastic Runtime Scheduler`, `SRUN`, `SLURMD` and `SLURMSTEPD`. Finally, inside the MPI processes there are two linked libraries: the `PMI` library and the `MPI` library.

From the proposed extensions introduced in Chap. 6, the MPI_INIT_ADAPT operation is essentially the same as the standard MPI_INIT operation. The launcher command, in this case SRUN, receives an allocation and credentials from the Elastic Runtime Scheduler (ERS). It then proceeds to create the necessary MPI processes through interactions with each SLURMD daemon running at each node of its allocation. After that, it waits until the MPI application completes its execution. The extra value of local_status needs to be propagated to each process together with its initialization metadata.

With the MPI_PROBE_ADAPT operation, information is forwarded from the scheduler of the resource manager all the way to each preexisting MPI process of an application. The SRUN binary coordinates how this information is forwarded. Preexisting process groups are only notified after the newly created expansion processes are ready.

It takes six steps to complete an adaptation window in the current implementation. Figure 9.1 contains arrows to indicate which components participate during each step. The arrows point from the component that initiates the action towards the components that performs it. Each step is enumerated and described here:

1. **Reallocation Message:** The scheduler makes a decision and the resource manager applies it by sending a reallocation message to the job step's SRUN instance.

2. **Create New Processes in Expansion Nodes:** If there is one or more processes to be created in one or more expansion nodes, SRUN sends a launch command with the required instructions to the SLURMD at each participating expansion node.

3. **New Processes Ready:** After the processes created in the expansion allocation are ready at the MPI_COMM_ADAPT_BEGIN operation, the SLURMD at the leader node notifies SRUN.

4. **Notify Preexisting Processes:** SRUN sends instructions to each preexisting node's SLURMD. Each SLURMD then updates the metadata to be provided to each of its local processes though the MPI_PROBE_ADAPT operation. This will cause the parent processes to enter the adaptation window and interact with the newly created children processes in the expansion.

5. **Adaptation Commit:** Once the adaptation window is completed (all staying and joining processes are ready and leaving processes are terminating) the leader node notifies SRUN that the adaptation is complete.

6. **Reallocation Complete:** SRUN then notifies the scheduler that the reallocation was applied to the job step. Finally, SRUN receives an updated credential from resource manager with leaving nodes removed from its allocation.

It should be noted that steps 2 and 3 are only necessary when there is a resource expansion (i.e., these steps are unnecessary when only a reduction of resources is performed) as part of the adaptation. Additionally, step 4 only contains instructions for processes to leave the process group if there is a resource reduction.

The design does allow for simultaneous resource expansions and reductions. The only current limitation is that the node where SRUN is located needs to remain in the allocation, since SRUN cannot be migrated given the inherited design from SLURM. Adding a migration feature to SRUN would enable arbitrary migrations of full MPI applications with the current design.

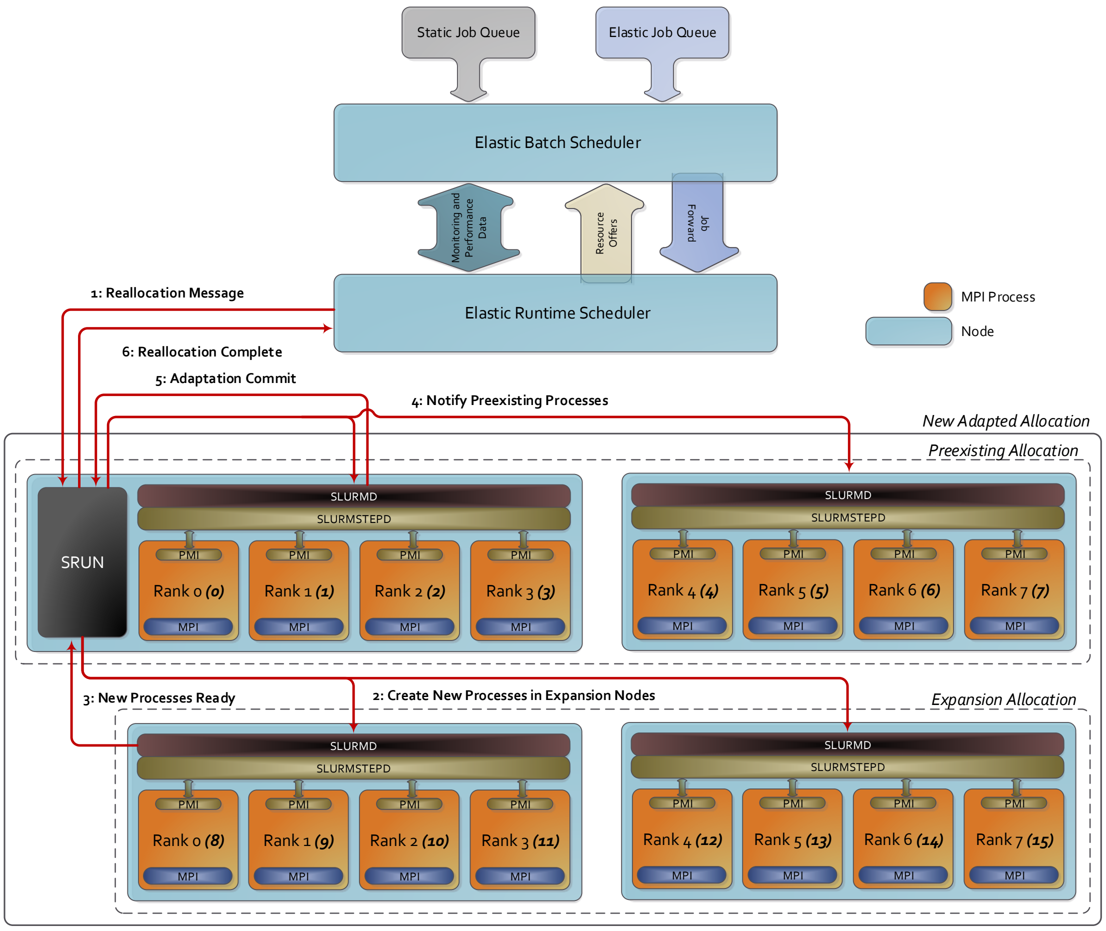

Figure 9.1: Overview of interactions between MPICH and SLURM components during adaptations.

9.1.1 Rank to Process Mapping Strategy

In Fig. 9.1, it can be seen that each MPI process in the scenario presented has two rank numbers, with one in parenthesis. The first number is its rank before the commit operation, while the one in parenthesis is its rank afterwards in the adapted `MPI_COMM_WORLD` communicator.

The current rank to process mapping algorithm minimizes rank changes on preexisting staying processes. This is achieved by selecting leaving processes with the maximum ranks possible, and giving joining processes ranks bigger than the greatest staying rank. Avoiding rank changes on staying ranks helps minimize the movement of data during redistributions in adaptation windows.

9.1.2 Support for Arbitrary Node Identification Orders

During the normal operation of the resource-elastic system, multiple resource adaptations per job may occur. These adaptations may be expansions or reductions of resources of arbitrary amounts of nodes. Each node in these sets is unique and is identified with a number. These identifiers are set based on the order the nodes appear in the `slurm.conf` configuration file.

Each resource allocation consists of a set of unique nodes that are specified with their identifiers in the metadata. These are ordered from the lowest identifier to the highest. To support MPI and other distributed programming models, the system provides vectors that specify the way application processes are to be overlaid in a set of nodes. These vectors have the following format:

$(vector, (identifier, count, processes), (identifier, count, processes), ...)$

Each tuple in the specified vector is a subset of the mapping. The $identifier$ value specifies the node in the allocation to start with. The $count$ value indicates for how many nodes to apply the mapping. The $processes$ value determines how many processes to create in each node of the mapping. For example, if the system has 8 physical CPU cores per node, and the user launched an application that requires 20 processes in an allocation with 3 nodes, a dense mapping would look as follows:

$(vector, (0, 2, 8), (2, 1, 4))$

This concretely specifies that 8 processes should be created in each of the first two nodes of the allocation, while in the third and last node only 4 processes are to be created, for a total of 20 as required by the launch.

As may be deduced from the overview, there are multiple process groups during adaptation operations (refer to Fig. 9.1). In the case of an expansion, the first one is the preexisting group, followed by the expansion group. In the case of a reduction of resources, there is only the preexisting group. On the completion of an adaptation, there is the new resulting group. This means that the resource manager needs to specify three vector mappings in the expansion case and two vector mappings in the reduction case, so that the system knows how to overlay the processes in the changing node allocation.

This vector data is also used by the MPI library. It uses the vector mapping to order the ranks of the processes. As explained before, the rank ordering technique clips higher ranks on reductions, and appends new ranks on expansions.

The correctness of this scheme depends on the incremental order of the node identification numbers. Consider the adaptation sequence presented in Fig. 9.2. Each square represents a node, with its unique identifier in the system as the first number, followed by its identifier in the allocation for the job. Each color represents a different node allocation. In adaptations 1 and 2, the node identifiers of each application are ordered incrementally, although in the second adaptation job 2 has its nodes fragmented. Fragmentation does not affect the correctness of the mapping; however, if the node identifiers are not ordered incrementally, the original scheme does not work. In adaptation 3, job 1 has an allocation with the sequence: $2, 2; 3, 3; 4, 0; 5, 1$. The second digits of each pair indicate that its internal node identifiers are not incremental: $2, 3, 0, 1$. This occurred because an expansion to its resources was performed and its expansion nodes were of lower global identifiers than its preexisting nodes. A similar situation arises in adaptation 4 for job 3.

The issue is that the processes will be overlaid incorrectly in the original system, with the vector specification, because the node identifiers were always assumed to be in incremental order. Additional data was added to the launcher infrastructure and the algorithm was adjusted so that the vector specifications can be applied to arbitrary orders of nodes. The format of the vectors was preserved, since this ensures compatibility with the elastic MPI library without any changes necessary. Preserving the vector format will also ensure that integrating other programming models in the future is less challenging.

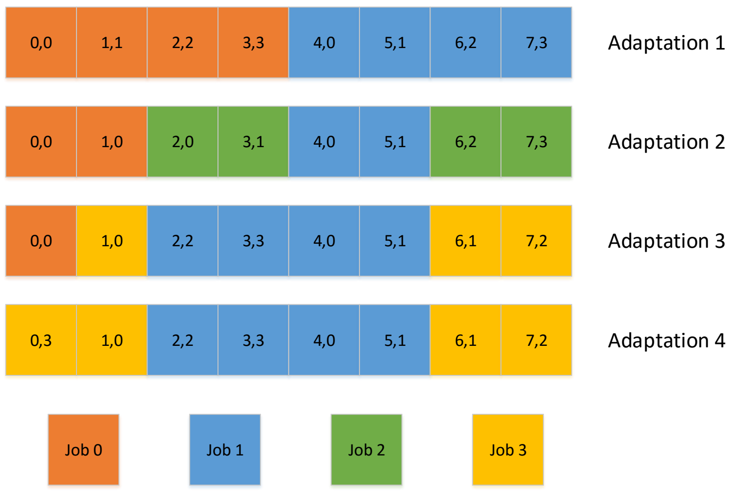

Figure 9.2: Sequence of adaptations on 8 nodes that lead to node identifier orders that are not incremental in some of the presented allocations.

9.2 Elastic Batch and Runtime Scheduler

As can be seen in Fig. 9.1, the controller daemon (`SLURMCTLD`) from the original SLURM design is no longer present. Instead, it has been replaced by two separate components: the Elastic Batch Scheduler (EBS) and the Elastic Runtime Scheduler (ERS). The new resource-elastic scheduling algorithm that is split into these two components is described separately in Chap. 10.

The functions that were originally provided by the controller in SLURM, such as batch scheduling and security, are now managed by the EBS. The more dynamic behavior needed for the support of elastic MPI applications is implemented in the ERS, such as node monitoring, performance modeling, as well as the management of adaptation windows through the `SRUN` component.

Here is a list with the most important changes that were introduced with the addition of the EBS and ERS components:

1. A new operation that initiates an adaptation through the `SRUN` instance of a job step: `srun_realloc_message`.
2. Supporting protocol messages and handlers for the new `srun_realloc_message` operation.
3. A new elastic scheduling algorithm that relies on performance modeling (covered in Chap. 10).
4. Extra metadata for elastic job entries in the elastic job queue. For example, a new job status that indicates that jobs are performing an adaptation: `JOB_ADAPTING`.

The `srun_realloc_message` is used to initialize the adaptation process of a single application in a job. A job may have multiple applications running simultaneously, in what is called a job step. Each job step has its own instance of the `SRUN` binary and a set of `SLURMSTEPD` daemons. The number of `SLURMD` daemons remains at one per node in all situations.

The `srun_realloc_message` provides `SRUN` the following information:

- New nodes allocated to the application and the number of processes to create on them.
- Preexisting nodes from where processes need to be destroyed and how many processes to destroy in them.
- The full list of nodes that compose the new allocation.
- Updated SLURM credential object that is necessary for communication with the new expanded nodes.
- The set of vector process mappings mentioned earlier. These are used by the MPI library to generate the communicators of each process group, as well as the future `MPI_COMM_WORLD` communicator correctly.

Only whole nodes are added or removed from a job during adaptations in the current implementation. Future developments may add the ability to schedule intra-node resources.

9.3 Node Daemons

The resource manager side of the algorithm used to support the `MPI_PROBE_ADAPT` operation is implemented in these daemons, based on the instructions forwarded to each participating node.

Given the design inherited from SLURM, when MPI processes need to be created at a node, its `SLURMD` forks first a `SLURMSTEPD`. This daemon then receives all required information to create the subgroup of processes that will run in the node.

The PMI plugin is loaded by the `SLURMSTEPD` daemon. The PMI has been extended with additional operations that allow MPI processes to notify their `SLURMSTEPD` when:

1. Processes that are joining have opened a port and are waiting in the internal accept operation of `MPI_COMM_ADAPT_BEGIN`.
2. Both the joining and preexisting processes have completed their adaptation and will shortly exit `MPI_COMM_ADAPT_COMMIT`.

The first extension to the PMI is used by leader processes of joining groups. A leader process notifies its local `SLURMSTEPD`. This daemon then notifies the `SRUN` instance of the job step. The `SRUN` instance then proceeds to notify each of the `SLURMD` daemons running at preexisting nodes. These daemons then proceed to update their local `MPI_PROBE_ADAPT` metadata and start their side of the probe algorithm. The flowchart of the probe algorithm from the side of the daemons is presented in Fig. 9.3. Each daemon starts by setting the adaptation flag to true and then reading its local metadata. It then determines whether it is locally consistent, meaning that all local processes read the flag correctly and are now blocking waiting for confirmation. The local status can be inconsistent when some processes are not blocking and incremented their counter, indicating that they exited the probe operation too early. Each local consistency result is communicated to `SRUN`. All nodes are notified if one or more nodes reached an inconsistent state. If the preexisting processes are

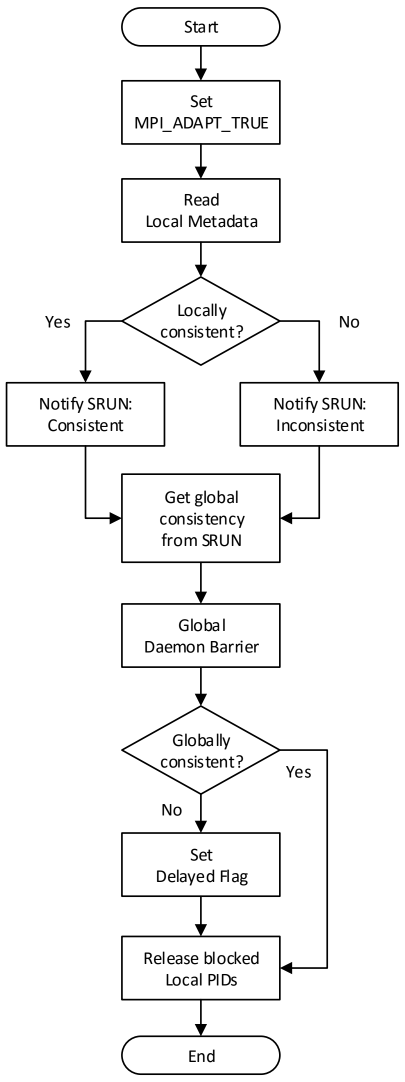

Figure 9.3: Probe operation at the SLURMSTEPD daemon.

globally consistent, then all processes are released and they can then enter the adaptation window; otherwise, a delay is inserted and all processes will be consistent the next time they call the probe operation.

The second extension to the PMI is used by the leader processes of adapted process groups. These groups are created as a result of successful completions of adaptation windows through calls to the MPI_COMM_ADAPT_COMMIT operation. Leader daemons (those where the MPI process with rank 0 resides) send a notification message to the SRUN instance of its parallel application. The SRUN instance then forwards it to the scheduler. The resource manager handles these messages by updating the status of the adapting job from JOB_ADAPTING to JOB_RUNNING, making the application eligible again for resource adaptations. Additionally, the resource manager provides an updated job credential to the SRUN instance as a response.

9.4 Launcher for Elastic Jobs

The final component of the resource manager that has been extended is the SRUN program. Most of the operations that are initiated by either the scheduler or any application process (via the PMI) are handled partially by SRUN. Here are some of the new protocol messages that SRUN can now handle:

1. Reallocation message received from the controller daemon.
2. Notification that joining processes are ready and waiting in the MPI_COMM_ADAPT_BEGIN operation.
3. Notification that an ongoing adaptation window was completed though a successful MPI_COMM_ADAPT_COMMIT.

In addition to these handlers, SRUN has also been extended to manage the IO redirection of joining processes. In the original implementation, these were setup only at the beginning when the user launched the application; SRUN can now manage redirections dynamically as processes are created and destroyed.

Another important change to SRUN is a new set of operations that enable the reconfiguration of its Tree Based Overlay Network (TBON) with the daemons of its current allocation. All communication between SRUN and these daemons is done through this network to ensure scalability. However, with each resource adaptation, the number of nodes in an allocation may be reduced or increased; this requires a reconfiguration of the TBON. The reconfiguration is performed in a distributed manner between SRUN and its daemons on each adaptation.

The inherited design from SLURM, where SRUN needs to run in the master node of an allocation, is a limitation to elastic execution models such as the one presented in this work: the SRUN binary has to remain in the same node throughout the execution of a job. This means that migrating applications to a completely new set of nodes is currently impossible. It may be desirable to add migration functionality to SRUN in the future, since the communication performance of MPI can be improved by relocating applications to nodes that are more closely clustered in the network of an HPC system, as they become available.

10 Monitoring and Scheduling Infrastructure

Distributed computing systems are expected to deliver performance that is commensurate to their available hardware resources. This is achieved by the optimization of system-wide performance metrics. The optimization of these performance metrics is a task usually delegated to schedulers. In the case of distributed systems, schedulers take as input the jobs to be performed and the set of available compute resources. They produce as output the job startup order and the resources where they will be executed. These orders are referred to as schedules. These schedules affect the performance of individual applications and whole systems, and therefore determine the quality of schedulers.

The terms resource manager and scheduler are sometimes used interchangeably. In reality, these are different software components that are often bundled together due to their equal importance. Distributed systems need both a resource manager and a scheduler in order to share its resources with its users in a fair and efficient manner.

In previous chapters, the resource manager and its unique features that allow support for elastic jobs were introduced. In this chapter, the scheduler and its unique features for optimizing application and system-wide efficiency metrics are discussed. This scheduler takes advantage of the support for elastic jobs provided by the resource manager.

As seen previously in Fig. 9.1, the scheduler is divided in two smaller sub-schedulers that closely interact: the Elastic Batch Scheduler (EBS) and the Elastic Runtime Scheduler (ERS). Unfortunately, the EBS was not developed in time to be described in this document. Because of this, only the ERS and the measurement infrastructure will be described in detail. The role of the EBS and its expected interaction with the ERS will be covered briefly.

This chapter begins with an introduction to the general multiprocessor scheduling, the batch scheduling and the runtime scheduling problems. The additional features of the resource manager that provide performance information to the Elastic Runtime Scheduler (ERS) are described afterwards. Finally, the ERS and its current scheduling heuristic are described.

10.1 Theoretical Background on Multiprocessor Scheduling

The general multiprocessor scheduling problem is stated in an abstract manner in this section. The problem statement for batch scheduling with static resource allocations is presented after that, together with a short discussion on the taxonomy of schedulers and how it is classified. This problem statement is then extended to fit the more specific elastic scheduling problem addressed in this work. New requirements are identified from the new problem statement.

10.1.1 Problem Statement

Multiprocessor scheduling is an optimization problem that can be stated verbally as follows: given a set of tasks to be completed and a set of resources that can complete them by some means, find an assignment of tasks to resources that optimizes a set of objective functions. The tasks are bounded in time and may require collectively more resources than are available simultaneously; therefore, the assignment of tasks to resources may also require an order. Different orders can produce different outputs of the objective functions.

We can define the problem of scheduling more rigorously. Let T be a set of tasks t_i where the subscript $i \in \mathbb{N}$ identifies each task uniquely; this set may or may not be finite. Similarly, let R be a set of m resources r_j where the subscript $\{j \in \mathbb{N} \mid j < m\}$ identifies each resource uniquely. One or more resources in R can perform the tasks in T in some manner. If $\tau(t_i) \in \mathbb{R}$ is the maximum execution time and $\rho(t_i) \in \mathbb{N}$ the number of resources required to perform a task t_i, then we can define multiprocessor scheduling as the following optimization problem:

$$\begin{array}{ll} \text{given inputs} & T = \{t_i \mid \tau(t_i) < \infty \wedge \rho(t_i) \leq m\}, \\ & R = \{r_j \mid j < m\} \\ \text{compute a} & S = \{t_i \mapsto \varrho_i\} \\ \text{that optimizes} & \sum_{k=0}^{w} O_k \end{array} \tag{10.1}$$

The result of this optimization is a schedule S. The schedule is a set of mappings from individual tasks t_i into specific subsets of resources ϱ_i of size $\rho(t_i)$, where $\varrho_i \subset R$. Tasks where $\rho(t_i) > m$ are impossible to schedule and therefore not considered.

Objective functions typically produce single scalar values in $\mathbb{R}$ within the range $[0, \infty)$. By optimizing (either minimizing or maximizing) the sum of the output of each O_k objective function, where $\{k \in \mathbb{N} \mid k < w\}$, the quality of the produced schedule can be improved. Different objective functions can evaluate the quality of full schedules S or individual mappings $t_i \mapsto \varrho_i$. This allows schedulers to optimize based on system-wide metrics, performance metrics of individual applications, or both.

The sum of all required resources of the tasks in T may exceed the total number of resources m in R:

$$\sum \rho(t_i) > m \tag{10.2}$$

In such a condition all tasks cannot be started simultaneously at the earliest time of the schedule $\{\delta_0 \in \mathbb{R} \mid \delta_0 > 0\}$. Because of this, both a starting time and duration need to be added as part of each mapping in the schedule when resource sharing is not allowed. Each mapping then becomes a reservation of resources with a starting time $\delta_i \geq \delta_0$ and the duration of its task $\tau(t_i)$, in addition to its set of unique resources ϱ_i. A schedule S then becomes:

$$S = \{t_i \mapsto \langle \varrho_i, \delta_i, \tau(t_i) \rangle\} \tag{10.3}$$

This modification to S can be inserted in the initial optimization problem definition (equations 10.1) to indicate that schedules need to be produced with these additional timing specifications.

10.1.2 Computational Complexity

The theoretical complexity of the multiprocessor scheduling problem can be determined with the aid of complexity theory. The goal is to determine the asymptotic complexity of the optimization problem based on its inputs. A bound to the number of steps of possible algorithms, based on the number of steps required to reach a solution, should be determined. Thankfully, this topic has been of great interest to researchers and results from previous analyses [135, 98, 149, 142, 93] are available.

The multiprocessor scheduling problem belongs to a family of problems that have no known solutions of polynomial or better complexity [77, 84, 138, 214]. It is for this reason that current schedulers rely on approximation algorithms that are based on heuristics. These algorithms settle for solutions that are feasible but not necessarily optimal; the assumption is that in most cases adequate heuristics guide the approximations so that produced schedules approach optimal results, based on a set of objective functions.

10.1.3 Resource-Static Scheduling in Distributed Memory HPC Systems

A scheduling problem for specific compute systems, in a more concrete way, can be classified by several characteristics related to its set of tasks, its set of resources and its method used to generate the output schedule. There have been several efforts to create a taxonomy of scheduling problems [152, 106, 186, 58, 149]. The scheduling problem in distributed HPC systems is clearly defined [89, 126, 200, 166, 90] for current resource-static execution models. Current solutions consist generally of First-Come First-Serve (FCFS) batch scheduling with static allocations and backfilling.

Current supercomputing systems are usually shared among several researchers across multiple institutions. Individual tasks are submitted to these systems by its users, in the form of batch job definitions. The arrival rate of these job definitions can be modeled with the aid of traffic theory. Batch job definitions include their number of resources required, their priority and their maximum execution time, among several other aspects that may not be as important to schedulers. Batch job definitions are entered in a queue. This queue represents the input task set T of the optimization problem 10.1.

The resources of current supercomputing systems tend to be similar. In most systems, the hardware on each node is identical. There may be cases where the nodes have heterogeneity internally (e.g., in the form of accelerators). In general, it can be assumed that all resources can handle all tasks similarly. A node is abstracted as a single resource in most cases. This means that in spite of the growing amount of parallelism internally at each node, schedulers only consider as a resource a full node, instead of subsets of cores or even accelerators where available.

The operation of schedulers is currently divided in two steps: batch scheduling and backfilling. The batch scheduling step scans a window of the job queue and attempts to start as many jobs as possible based on their priority. When a job cannot be started immediately, it may instead get a resource reservation in the future. Once this first step is done, the scheduler proceeds to the backfilling step: it attempts to start jobs that fit in the gaps of remaining idle resources. Jobs that are started during this second step should not delay the start of higher priority jobs that have reservations.

The general strategy is illustrated in Fig. 10.1. It presents a scenario with four nodes, a job queue of six jobs with a priority based order. In the illustration, a schedule is computed where job 4 receives a reservation later than jobs 5 and 6 due to the availability of resources. In the same schedule, job 6 is scheduled early to minimize idle nodes through a backfilling operation.

In summary, static batch scheduling with backfilling on current distributed systems has the following task set, resource set and algorithm properties:

- Task set:
  - Set properties:
    - Multiple users submit tasks
    - Tasks submitted randomly
    - Unbounded task capacity
    - Best effort First-In, First-Out (FIFO)
    - Tasks are removed on completion
  - Task properties:
    - Set of one or more tasks as jobs
    - Jobs are time bounded
    - Jobs and tasks are not periodic
    - Fixed number of resources specified
    - Jobs receive exclusive access to resources
    - No Service Level Agreements (SLAs)
- Resource Set
  - Symmetric Multiprocessing (SMP) nodes as resources
  - Nodes have identical hardware (homogeneous)
  - Nodes may have attached accelerators
  - No quality of service (QoS) support
  - Resources are finite and cannot be scaled on demand
  - Resources are located in a single building
  - Power and energy scaling features available
  - No job or task migration support
  - No fault tolerance support
- Algorithm
  - Nodes as the units of resources
  - Job level scheduling (no task level scheduling)
  - Objective functions for mainly system-wide performance metrics
  - Two step *resource-static* scheduling
    - Batch scheduling with priority based FIFO
    - Backfilling to minimize idle nodes
  - Scheduling without performance guarantees
  - Scheduling without reactive adjustments
  - Jobs cannot be preempted

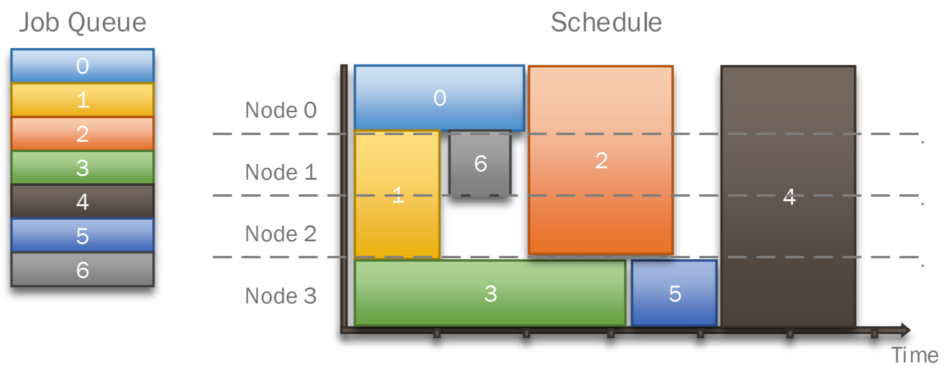

Figure 10.1: Possible schedule of a set of static jobs ordered by priority in a queue.

10.1.4 Modified Scheduling Problem for Resource-Elastic Execution

The scheduling problem described so far applies to cases where only static allocations are possible. Static allocations mean that the resource reservation of a job stays constant throughout its execution. The scheduling problem needs to be updated if the resource allocation of a job can change during the runtime of its tasks; resources may increase (expansion), decrease (reduction) or the unique nodes allocated to a job may change while their total stays the same (migration).

The current scheduling problem, solved with batch scheduling and backfilling, needs to be modified to include the added flexibility of resource-elastic execution. Only the properties of the jobs in the task set need to be modified:

- Jobs have a range of feasible resource counts.
- Jobs have a time bound that is a function of its resources.

This modified scheduling problem remains very similar to the preexisting one due to only these two differences. All other mentioned properties in the previous section remain. Jobs still retain exclusive access to the resources on its resource allocations, although some resources may be added or removed from this allocation at runtime. Due to this, the time required for the job to complete becomes dependent of the number of resources in time. In general, jobs will still provide a worst case time bound as part of its description.

Although similar to the preexisting scheduling problem, these two differences in the properties of jobs add new requirements to the algorithm of a potential scheduler. In addition to the previous batch and backfilling steps, a scheduler for HPC systems with resource-elastic execution capabilities must also:

1. Continuously monitor the performance of the tasks of running jobs.
2. Adjust the resource allocations of jobs based on their observed performance.

In the proposed design, the first activity is delegated to the previously described infrastructure, while the second activity is delegated to the new Elastic Runtime Scheduler (ERS). Most of the traditional batch-scheduling activities are still handled by a more traditional scheduler. The design will be covered in the next sections of this chapter.

10.2 Performance Monitoring Infrastructure

The performance of individual jobs is monitored by the infrastructure. The infrastructure is composed of the MPI library described in Chap. 6 and the resource manager components described in Chap. 9. Performance data is captured and a performance model is built. The performance model is then used to drive scheduling decisions.

The collection of data is performed in a hierarchical manner. At the lower level, each MPI library linked into each application process detects the structure of the computation in the local process and collects performance data. This structure is then reduced to a node-local representation by the `SLURMD` daemon at each node. Finally, the Elastic Runtime Scheduler (ERS) (described later is Sec. 10.3) performs a final reduction to create the individual performance model of the distributed application. The set of models of all running applications are used to drive scheduling decisions.

10.2.1 Process-Local Pattern Detection and Performance Measurements

At the process-local view, the MPI library linked to the process performs pattern detection and performance metrics evaluations. The pattern of computation is detected before any performance metric is determined, since these metrics will be attached to specific control flow locations only after they are detected. Process local operations are kept to a minimum once the pattern is detected.

Pattern Detection

Since the pattern detection is intended to occur during the actual production run of applications, the minimization of its performance impact is of great importance. Because of this, the structure of computation is detected based on markers introduced by the compilation wrappers provided by the MPI library (`mpicc` and `mpifc` in this case). There have been previous works that rely on backtracing the sequence of calls in a program to determine unique locations during execution. These are then used as identifiers for pattern detection [95, 27, 28, 29, 30, 124], such as loops, in MPI applications. The introduction of these markers at compilation time eliminates the overhead related to backtracing, although the technique is limited to binaries generated within a single software project.

The markers are inserted by splitting the compilation of objects into the emission of assembler and the final assembly step. Thankfully, most modern compilers have support for these operations. In the current implementation, the compiler wrapper works with Intel and GNU compilers. Versions 10.0 and later of the Intel compilers were tested, while versions 4.9 and later were tested for the GNU compilers. Other compilers were not tested, since those are the ones available in the SuperMUC system where this work was evaluated.

The current wrapper based technique relies on the way these compilers generate library calls in the emitted assembler. The actual API names of library calls are preserved, when linking C based libraries. Fortunately, MPI is a pure C based library and its calls can be easily identified with text processing in the intermediate assembler of each target object of the compilation. Additionally, since the MPI standard requires that any operation with the `MPI_` prefix be provided only by the MPI library in compliant programs, it is guaranteed that only MPI operations will be intercepted. Additionally, the `PMPI_` pattern can be selected to preserve support for any PMPI based profilers and tools.

Once the MPI calls are identified in the assembler, a unique ID is computed and inserted before the MPI call through an additional operation available in the Elastic MPI library. This operation is called `MPI_T_set_call_site_identifier`, and as its prefix `MPI_T` suggests, it belongs to the MPI tools interface. This tooling call sets the identifier for the device layer of the layered software architecture inherited from MPICH (as described is Chap. 5). This operation sets an integer identifier that is later read by the library at each individual MPI operation. This identifier establishes the uniqueness of the call site without the need of backtracing.

The MPI library relies on these markers to detect the structure of the computation at runtime. There have been several algorithms developed to detect patterns in sequences [145, 213, 208, 114, 180]. A pattern detection algorithm, that was originally designed to analyze programs from decompilation, fits well this pattern detection use case [221]; this algorithm is also used in other recent related works [28].

The pattern detection algorithm was implemented within the Elastic MPI library. In the current implementation, the algorithm provides the following output information to the runtime system, based on the current partial sequence of call site identifiers provided to it as input:

1. The detected Control Flow Graph (CFG).
2. Each node of the CFG is annotated with its number of revisits.
3. Nodes that are the heads of unique loops are marked.
4. Nodes that are the tails of unique loops are marked.
5. Nodes that are reentry points from nested loops are marked.

The detection logic is only available when the MPI application has been initialized with the `MPI_INIT_ADAPT` operation. In addition to this, the detection algorithms is disabled at the start of applications. It is only enabled after a running application performs its first `MPI_PROBE_ADAPT` call. It should be noted that EPOP applications disable this feature entirely through an `MPI_T` extension, since these provide the structure of applications at launch time and therefore do not need to rely on any form of pattern detection.

The CFG update routine is called at relevant MPI operations with their unique identifiers and types. There are different operation types for point-to-point, one-sided, collectives, MPI-IO, etc. The system does not perform CFG updates inside adaptation windows. The MPI library has an operation that serializes its local CFG to a shared memory segment, where it can then be read directly by the local daemon. Unique blocks of shared memory are dedicated to each MPI rank in the node.

An example can be used to better explain the algorithm's behavior. Listing 10.1 shows the log output of a single MPI process given the sequence of identifiers:

```
2 0 6 3 1 6 3 1 6 3 1 9 7 9 7 3 1 6 3 1 6 3 1 9 7 9 7
```

The detector can produce a text representation of its current CFG, in tabular form, as logging output. Listing 10.2 shows the detected CFG that matches the previous sequence. Each output row represents a node in the CFG. The first column is the address in the local memory of the process. The second columns is the identifier number. After that, the loop head flag (H), the loop body flag (B), the reentry counter (Re) and the revisit counter (Rv) are provided. The final two columns provide the tail data of loop heads, and the head data of loop body nodes. As seen in listing 10.1, there is also a time differential (TD) computed at each step. In the current implementation, the time resolution of this differential is in nanoseconds. The time of creation is set each time a new node is added to the CFG. Total differential times from head nodes are accumulated on node revisits. The average distance in time from the head node of a loop to any node in the body can therefore be computed by dividing the accumulated differential by its total number of revisits.

Figure 10.2 presents a graphical depiction of the text based CFG output. Reverse arrows on the left side of the figure represent loops, while the reverse arrow on the right represents a reentry. The time taken at each MPI block is represented as its vertical length. The time of the compute blocks can be computed by subtracting the MPI times from the differential from preceding MPI operations. Their time is also represented by their vertical length in the figure. In summary, all necessary data is included so that such a graph can be computed by the local daemon from the serialized CFG data.

```
 0: root id: 2
 1: id: 0; detected: 0; -> NOT in a loop; (TD: 4638)
 2: id: 6; detected: 0; -> NOT in a loop; (TD: 10243)
 3: id: 3; detected: 0; -> NOT in a loop; (TD: 14440)
 4: id: 1; detected: 0; -> NOT in a loop; (TD: 17938)
 5: id: 6; detected: 1; -> head: 6; (TD: 22178)
 6: id: 3; detected: 1; -> head: 6; (TD: 26174)
 7: id: 1; detected: 1; -> head: 6; (TD: 30090)
 8: id: 6; detected: 1; -> head: 6; (TD: 33756)
 9: id: 3; detected: 1; -> head: 6; (TD: 37407)
10: id: 1; detected: 1; -> head: 6; (TD: 41180)
11: id: 9; detected: 0; -> NOT in a loop; (TD: 44758)
12: id: 7; detected: 0; -> NOT in a loop; (TD: 48493)
13: id: 9; detected: 1; -> head: 9; (TD: 52336)
14: id: 7; detected: 1; -> head: 9; (TD: 56155)
15: id: 3; detected: 1; -> body re-entry; head: 6; (TD: 60054)
16: id: 1; detected: 1; -> head: 6; (TD: 63853)
17: id: 6; detected: 1; -> head: 6; (TD: 67418)
18: id: 3; detected: 1; -> head: 6; (TD: 70916)
19: id: 1; detected: 1; -> head: 6; (TD: 74361)
20: id: 6; detected: 1; -> head: 6; (TD: 77798)
21: id: 3; detected: 1; -> head: 6; (TD: 81239)
22: id: 1; detected: 1; -> head: 6; (TD: 84788)
23: id: 9; detected: 1; -> head re-entry; head: 9; (TD: 88710)
24: id: 7; detected: 1; -> head: 9; (TD: 92452)
25: id: 9; detected: 1; -> head: 9; (TD: 96131)
26: id: 7; detected: 1; -> head: 9; (TD: 99669)
```

Listing 10.1: Step by step updates based on the specified ID sequence.

```
Current detected Control Flow Graph (CFG):
0x030; id: 2; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x2b0; id: 0; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x310; id: 6; H: 1; B: 0; Re: 0; Rv: 4; tail: 0x3d0; head:
0x370; id: 3; H: 0; B: 1; Re: 1; Rv: 5; tail: 0x550; head: 0x310
0x3d0; id: 1; H: 0; B: 1; Re: 0; Rv: 5; tail:      ; head: 0x310
0x4f0; id: 9; H: 1; B: 0; Re: 0; Rv: 3; tail: 0x550; head:
0x550; id: 7; H: 0; B: 1; Re: 0; Rv: 3; tail:      ; head: 0x4f0
```

Listing 10.2: Example CFG detected based on the specified ID sequence.

Performance Measurements

The MPI library starts to record performance data once the heads and tails of one or more loops are detected. Currently two performance metrics are recorded:

1. Total Loop Time (TLT)
2. Total MPI Time (TMT)

The TLT metric is the total time spent on the detected loop. The TLT metric can be computed at each loop, including nested loops. The TLT metric is computed from two real numbers. The first one is its creation time. This time is set for each node in the CFG structure regardless of its type. The second one is the last visit time. The MPI library does not perform any more operations for this metric. Instead, the data is provided as it is to the local daemon once requested. The daemon is expected to perform the subtraction of these values for the total accumulated time, and to divide this value by the number of visits (revisits plus one) to get the average.

The second metric is the Total MPI Time (TMT). The TLT is inclusive of this time. This time is the difference between the entry and the exit times of each MPI call. In contrast to the TLT, these times are not stored in the CFG nodes where they are computed; instead, this metric is aggregated in the loop head of the node. There is no recursive search for the loop head in nested loops. The average can be computed by dividing the aggregated times by the total number of visits to their loop heads.

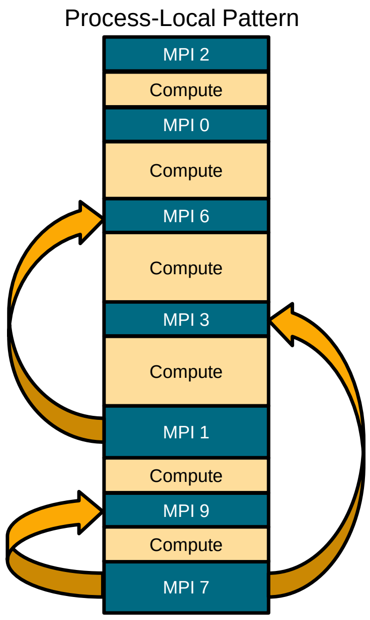

Figure 10.2: Process-local Control Flow Graph (CFG) representation.

10.2.2 Node-Local Reductions and Performance Data Updates

Once a loop is detected, the library switches to a mode of CFG verification and performance data collection. As mentioned before, each process serializes its CFG data on its own shared memory segment. Each process notifies its local daemon on the following events:

- Loop detected
- Unexpected Loop exit
- Unexpected loop reentry

These events occur in the sequence presented in Listing 10.1: a loop detection occurs in steps 5 and 13, in step 11 an unexpected loop exit occurs, and in step 15 an unexpected loop reentry is encountered. All of these create changes in the CFG and therefore need to be communicated to the local daemon. These events tend to be more common during the initialization of MPI applications, and settle after a while. Expected loop reentries in the

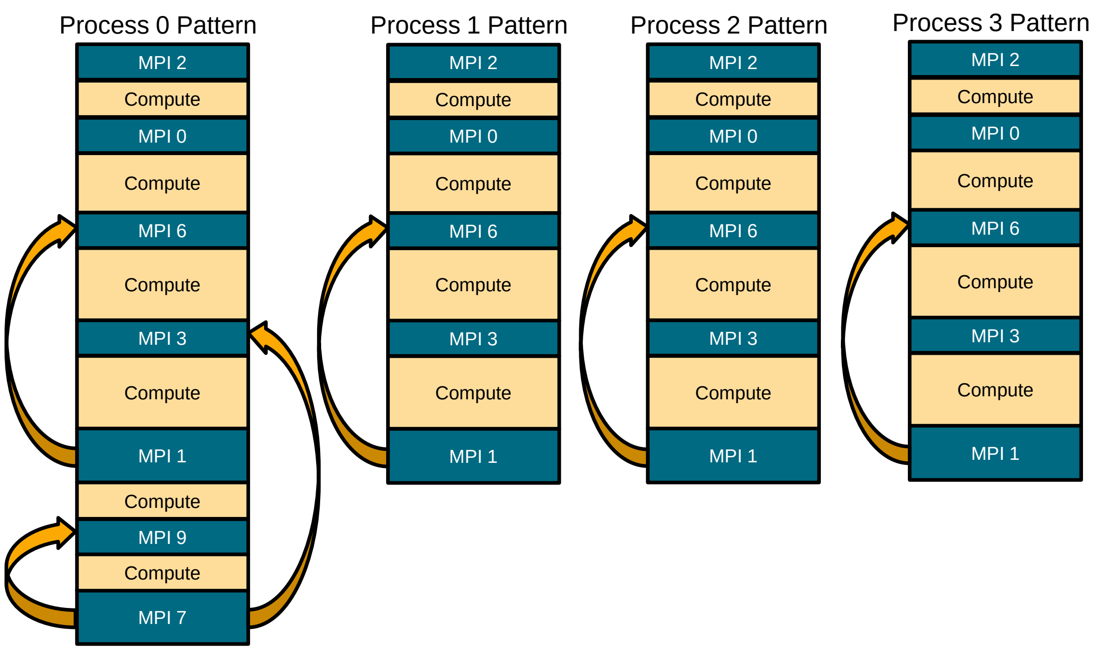

Figure 10.3: Set of four CFGs at a node before reduction.

body or loop heads do not generate any events, since they do not trigger changes in the CFG. The library instead continues updating performance data without notifying its local daemon, if there are no changes to the CFG.

The number of notifications to the local daemon is limited by the sampling timer that currently defaults to one minute. This minimizes synchronization overheads, especially during the initialization of an application. If one or more loop detection or break events occur between timers, the local daemon is notified only once.

Performance data is updated separately from the CFG. These are updated periodically on each expiration of the sampling timer. These are only produced at the next loop head reentry, and not in any arbitrary MPI operation. Each metric specifies the identifier of its loop head, since more than one loop may be detected.

The local daemons do not read the performance data periodically. Instead, the latest data is read on demand when requests from the Elastic Runtime Scheduler (ERS) are received. These requests also have a field that optionally specifies a new value for the sampling timer. This enables the ERS to adjust the frequency of data collection per application, based on previous performance data and trends.

Node-Local CFG Reduction

The daemon of a node keeps track of the notifications generated by each of its MPI process. When any of its local processes have notified that their CFGs have been updated, it proceeds to read them and to perform a CFG reduction operation. The reduction operation depends on the order and type of the operations in it.

The following rules are followed on the collection of CFGs to produce a reduction:

1. Nodes outside of loops are ignored.
2. Consecutive point-to-point or one-sided operations are collapsed.
3. Identical loops are combined into one with a process range.

The reduced CFG is then stored in the memory of the local daemon. It is populated with performance data before it is sent to the Elastic Runtime Scheduler (ERS) on each request. If a request is received from the ERS, but the CFG data is still unavailable, the response to the request has a field to indicates this.

An example set of four CFGs is presented in Fig. 10.3. All processes contain the loop from 6 to 1, but miss the nested loop with head 9 and tail 7. Rule 1 ignores the nodes 2 and 0. MPI operations with identifier 6 and 3 are of the type point-to-point. This means that they will be collapsed according to rule 2. All other operations are in loops. Finally, given rule 3, the loop from 6 to 1 will be clustered for ranks 0 through 3, while the loop from 9 to 7 will be separated for only rank 0. The information on its reentry is preserved. This indicates that it is nested within the common loop, but only at rank 0. The result is presented in Fig. 10.4.

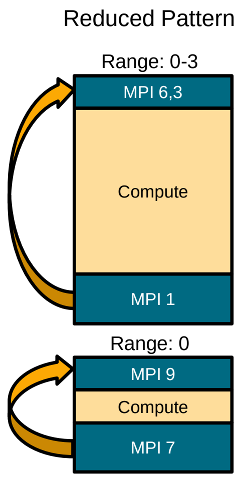

Figure 10.4: Reduced CFG from Fig. 10.3.

The three rules in the reduction algorithm can be justified. The first rule is justified by the fact that code that occurs outside of loops is not relevant to elastic execution. The creation of adaptation windows is only performed inside of loops. The second rule comes from the observation that MPI applications that use multiple point-to-point and one-sided operations match logically across ranks. For example, it is common to observe branching based on the rank number of the local process in an MPI program to determine is the process will perform a send or a receive. These sends and receives can be matched as a single block of communication in a distributed view of the program, greatly simplifying the loop matching algorithm. The final rule produces the reduction based on similarity. It is essentially a form of compression.

Node-Local Performance Data Reductions

The sum of all the TLT and TMT metrics of each process in a loop are added to the data of the reduced loop head nodes. In contrast, the mode (the value that occurs the most) of the loop revisit counts are set. It is expected that with enough revisits a small difference in the number of measurements will not affect the mean of the metrics significantly.

10.2.3 Distributed Reductions and Performance Models

The Elastic Runtime Scheduler (ERS) generates requests for performance data that reach all the daemons of an application. The requests and responses are routed through the `SRUN` binary of the application, over the Tree Based Overlay Network (TBON) that it creates with the nodes of its application. In the response to these request, each daemon sends the reduced CFGs with the TLT and TMT metrics attached to each loop head. The final distributed view of the CFG of the application is then generated from these at the ERS.

Matching loops are reduced by combining all of their TLT and TMT metrics. The union of the process sets is set as the final range. The final distributed representation of the earlier example is presented in Fig. 10.5.

Finally, the average loop time and MPI time metrics are computed based on the number of iterations of the loop heads and the TLT and TMT metrics provided. Additional memory is dedicated to store the mean, variance, minimum and maximum values of these final metrics. Additionally, a vector of their recent values is stored, to detect performance trends.

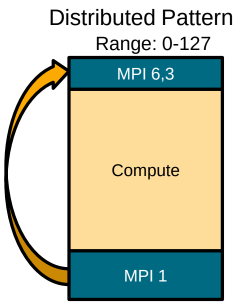

Figure 10.5: Final reduced CFG at the ERS from Fig. 10.4.

SPMD-Phase Performance Model

Currently only one type of performance model has been implemented: the SPMD-Phase model. When the system detects one or more distributed loops, it creates an SPMD-Phase performance model instance for the application. Applications that do not fit this model (i.e., that have no distributed loop) are currently ignored. SPMD-Phase models consist of a set of distributed loops and their performance metadata. In general, models are used by the scheduling heuristic to try to ensure that application phases remain in their efficient range of resources. This design decision comes from the observations first summarized in the motivation (Sec. 2.2.4).

10.2.4 EPOP Integration

The capture of the CFG and the performance data is supported in the EPOP programming model through the driver programs and a special mode of operation without pattern detection for the MPI library and the infrastructure components. This mode of operation is set by an additional MPI tools operation: `MPI_T_enable_EPOP_mode()`. This operation is set by driver programs at launch, before calling the initialization routine of the EPOP program loaded.

Driver programs in EPOP generate CFGs from the EPOP program definitions. This eliminates the overhead of the pattern detection techniques and the reductions. In addition to this, the added certainty allows the MPI library to operate in verification mode from the start of the application.

Loop head TLT and TMT metrics are still computed by the MPI library. This means that the sampling timer is still controlled by it, and not the driver program, based on the instructions received from the Elastic Runtime Scheduler (ERS). All of the techniques are otherwise used in the same manner for EPOP programs.

10.3 Elastic Schedulers

Instead of completely replacing the current solution for systems with static execution models, batch scheduling with backfilling, the preservation of several aspects of these successful techniques is proposed. The extra flexibility of resource-elastic execution models is addressed separately. For this purpose, the scheduler has been separated into two: the Elastic Batch Scheduler (EBS) and the Elastic Runtime Scheduler (ERS).

The EBS has not been implemented in this work, and will instead be developed as part of future work. The expected functionality of this scheduler will be mentioned where relevant. In its expected design, the EBS replaces the batch and backfilling techniques with moldable batch scheduling. Moldable batch scheduling is performed on resource offers provided by the ERS, instead of on the global view of resources. Resource offers depend on the availability of nodes due to job terminations and resource adaptations on running jobs. That means that any job start or reservation decision depends on the evolution of resource allocations on the set of running jobs. In contrast, previous systems performed scheduling decisions that depended only on job completions and reservations.

The lack of a batch scheduler means that MPI applications are started directly with the `SRUN` launcher, and not the `SBATCH` command. It also means that instead of queues, each individual `SRUN` command blocks until the Elastic Runtime Scheduler (ERS) has enough resources to launch its job step. In summary, `SRUN` will be the source of jobs in the evaluation and jobs will consist of only one job step.

The ERS adds the two additional capabilities that were previously identified for elastic execution support (Sec. 10.1.4): it continuously monitors application performance and periodically adjusts the resources of running jobs as necessary. This scheduler relies on the additional performance data collection facilities in the infrastructure (described earlier in this chapter) to create performance models that drive the resource adaptations.

In the remainder of this section, the current performance model and the Elastic Runtime Scheduler (ERS) are described in detail.

10.3.1 Elastic Runtime Scheduler (ERS)

As mentioned before, the Elastic Runtime Scheduler (ERS) fulfills the extra requirements of resource-elasticity (Sec. 10.1.4). These extra requirements are a consequence of the added flexibility of malleable jobs. The design decisions made when developing this scheduler were motivated by the following observations:

1. The scalability properties of distributed applications based on its allocated resources is input dependent.
2. Empirical and historical based methods for performance prediction require performance measurements at multiple resource allocations sizes.
3. Machine learning research that is applicable to job scheduling is still in its early stages; additionally, these techniques require additional storage and databases. These requirements would add premature complications to the current prototype.
4. It is desirable to be able to optimize applications that are running for the first time, as well as applications with new input sets.
5. In addition to backfilling, expansions of running jobs can be performed to minimize the count of idle nodes.
6. Both system-wide and individual application performance metrics must be optimized.

The first observation comes from the experience of running simulation codes that perform differently with different sized inputs, or even inputs of similar size but with different geometries. In addition to affecting their overall performance, the input size also determines the available parallelism of the application. Its available parallelism determines the amount of resources it can use efficiently.

The second observation is that history based performance predictors require large collections of data. In the case of distributed systems, resource adaptations can involve the movement of large amounts of data and require expensive repartitioning sequences. This makes the collection of empirical data a lot more expensive computationally than on shared memory systems, where a reconfiguration does not involve the movement of memory over a network. A history of empirical data can be created as applications are executed. Given enough time a predictor may collect adequate amounts of data.

The third observation is that machine learning research, as it relates to scheduling on distributed memory systems, is at the moment limited. These techniques have great potential and may be applied in this domain in the near future. The storage requirement is related to the second observation; once it is in place and enough samples are collected, this methods will become feasible. Data collection may need to be repeated on new system installations, depending on the method.

The next observation is that the system should be able to handle new elastic jobs that have no history with acceptable efficiency. It should also be possible to efficiently execute jobs that execute only once. Applications are often run multiple times but each time with different input sets.

Another observation is that there is potential to further improve the minimization of idle nodes in distributed systems due to the added flexibility of elastic execution. With the addition of support for malleable jobs, it is easier to perform the backfilling operation by filling up idle nodes with resource expansions. Additionally, jobs can be started at different node counts. These are advantages over systems with support for only rigid jobs.

The final observation is that the overall efficiency of an HPC system depends on the efficiency of the individual applications running, and not only idle nodes. System-wide and application efficiency metrics can be further improved with resource-elastic execution models.

The ERS performs scheduling decisions at a configurable rate. On each evaluation, the elastic scheduling algorithm follows the following steps:

1. Iterate the list of jobs and make a list of selected running jobs that are elastic and have performance data available.
2. Process the performance data to generate the performance model of the selected jobs.
3. Compute a range of optional and mandatory resource adaptations on the set of jobs.
4. Provide a resource offer based on the ranges to the Elastic Batch Scheduler (EBS).
5. Perform Elastic Backfilling (described later in this section).
6. Use the `srun_realloc_message` to apply individual resource adaptations.
7. Start any new jobs based on the batch definitions received from the EBS.
8. Wait until the system reaches a steady state before applying further resource transformations.

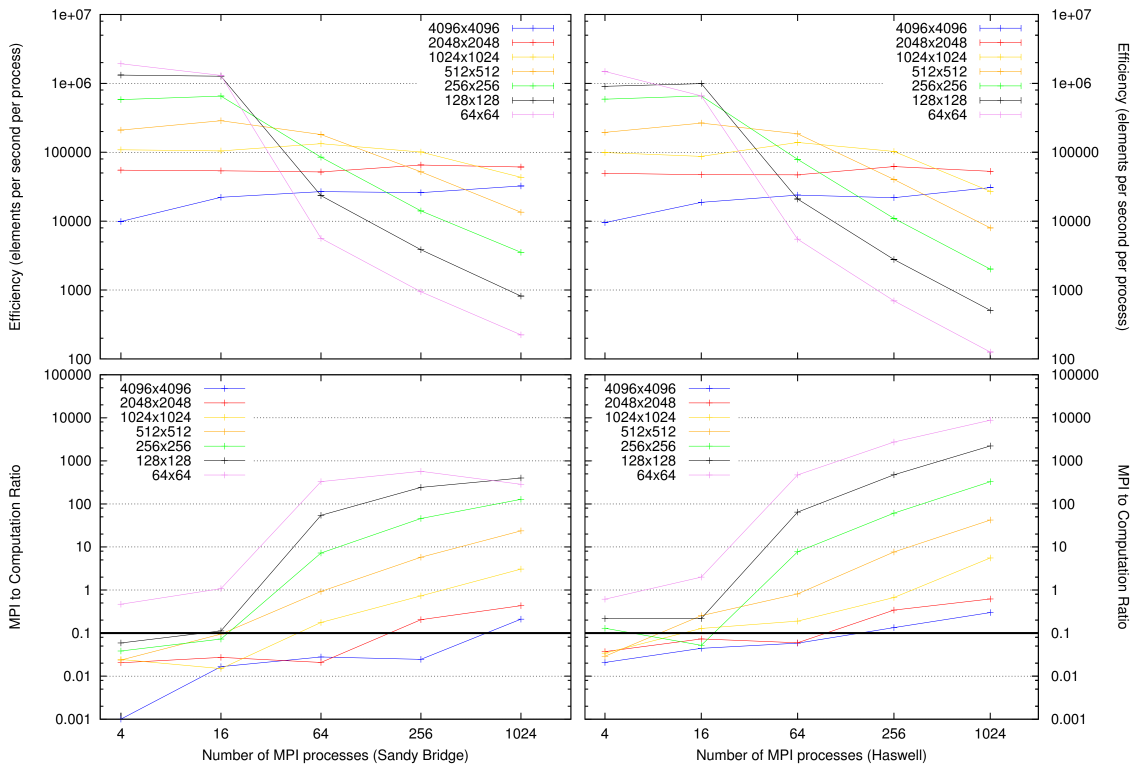

Figure 10.6: Efficiency (top) and MPI time to compute time ratio (bottom) of a Cannon's matrix-matrix multiply kernel. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) presented. A line is added for the constant 0.1 boundary of the ratio.

The remainder of this section will be dedicated to the description of steps 2, 3, and 5. Step 1 is trivial, since the scheduler simply iterates the list of running jobs and makes an additional list of those that are marked as elastic and have their performance data populated. Steps 4 and 7 are not yet implemented, since they depend on the availability of the EBS. As mentioned before, the EBS has not been implemented as part of this work, and is instead postponed as future work. Step 6 has been described extensively already in Chap. 9, where the design of the elastic resource manager and its interaction with the elastic MPI library have been documented. The final step has also been described in that chapter. When a transformation is triggered on a job via the `srun_realloc_message` command, its status changes from `JOB_RUNNING` to `JOB_ADAPTING`. With the `MPI_COMM_ADAPT_COMMIT` operation, each application notifies the resource manager when its adaptation is completed. Its job record is then updated from the status `JOB_ADAPTING` to `JOB_RUNNING`; this state change marks the application as eligible for adaptations again and its released resources available for other jobs. At that moment, the resource manager updates the credentials for `SRUN` based on its new allocation.

10.3.2 Performance Model and Resource Range Vector (RRV)

As mentioned before, in step 2 of the scheduling algorithm, a performance model is generated based on the performance data collected. In step 3, the ranges of optional and mandatory resource adaptations are produced. In this section, both of these steps are explained based on the performance model and heuristic implemented. The performance model is used to conservatively predict the maximum amount of nodes that can be allocated to the application while preserving an acceptable level of efficiency. Efficiency is evaluated individually for each application, and not based on general expectations.

A heuristic was developed based on empirical data collected from different distributed computational kernels used in simulations. These include: block-tridiagonal iterative solvers, fast-Fourier transform (FFT), embarrassingly parallel problems, matrix-matrix multiplication, matrix-vector multiplication, dense linear system solvers, sorting algorithms, conjugate-gradient iterative solvers, multi-grid iterative solvers, matrix decomposition, and others. The data collected shows a correlation between the proportion of MPI time in computational kernels and their individual efficiency metrics.

Figure 10.6 shows concrete values for a cannon matrix-matrix multiplication kernel. As can be observed, regardless of the size of the input size, the efficiency metric (matrix elements per second per process in this case) drops drastically when the proportion of MPI time to compute time exceeds a certain upper threshold. It can also be observed that a wide range of applications remain efficient below a certain lower threshold. Based on empirical data, a value of 0.1 has been set for the upper threshold and a value of 0.01 has been set for the lower threshold, as defaults. These have been found to manage well the behavior of the evaluated applications in the SuperMUC HPC system.

A Resource Range Vector (RRV) is generated based on the average and trend values of the MPI to compute time (MTCT) metric of the current distributed loop of an application. The average and trends for this metric are reset after each resource adaptation of the application. Recent MTCT values are stored to compute the recent trend of the metric (tMTCT) within a configurable time window. The RRV is generated as follows:

- If any MTCT is above the upper threshold, set the resource range to half of itself.
- If one of the MTCTs is between the upper and lower threshold while the other is below, remove the application from the set of candidates.
- If both MTCTs are below the lower threshold, then set the range to double of itself.

The difference between the upper and lower thresholds creates a band that helps prevent resource adaptation oscillations. The doubling and halving of resources can be replaced with more precise values once better performance models are developed in the future. The algorithm tries to keep the values of the MTCT metric below the upper threshold for all applications. The reduction of resources is mandatory, while resource expansions are optional. This prioritizes the startup of applications in the queue over expanding running ones: resources are eagerly reduced and conservatively expanded. The resources of an application are not lowered below their specified minimum, since a minimum node count may be due to memory requirements and not performance.

10.3.3 Elastic Backfilling

The elastic backfilling algorithm is designed to reduce the number of idle nodes by taking advantage of the extra flexibility provided by resource-elasticity. Elastic backfilling is performed in step 5 of the Elastic Runtime Scheduler (ERS) algorithm.

The algorithm takes the following parameters of each job description as input:

1. Current node count.
2. Estimated time of completion.
3. Its resource range (from its entry in the RRV).
4. Average adaptation time.

The estimated time of completion is estimated linearly based on the provided completion time by the user and the starting node count. This value is currently set with the `-t` or `--time` option of the `SRUN` launcher. The minimum number of nodes is set through an extension that reads an environment variable. Both these values will be specified in additional options in batch descriptions once the Elastic Batch Scheduler (EBS) is developed. The third input is produced by the heuristic explained previously. It can have a reduction of nodes or an optional expansion. The average adaptation time is computed from previous measured adaptations. It is not available if the application has not adapted before. This time is measured from the moment the `srun_realloc_message` is sent to when the status flag of the application is changed back to `JOB_RUNNING`.

The algorithm performs two basic operations: time balancing and resource filling. These operations are applied to sets of jobs and are described in the remainder of this section.

Time Balancing

Time balancing is an operation that transforms the number of nodes of each job in a job set such that their completion times become as close as possible. This can lower the makespan of the schedule and reduce wait times. It is a transformation that can be described with linear algebra. In this subsection, the transformation will be described for cases with two, three and four applications. Extrapolating from these, the technique can be understood for an arbitrary number of applications. This operation is only applied if applications are expected to retain their efficiency levels with the new resource allocations.

Consider the two jobs presented in Fig. 10.7. If $t0$ and $t1$ are the estimated time completions of jobs 0 and 1, and their node counts are $n0$ and $n1$ respectively, then the following linear system can be solved to find a vector of $x0$ and $x1$, such that the expected completions times of both jobs match. This transformation assumes linear scaling; this can be assumed to be approximately true within efficient node ranges, if the performance model determines these accurately. Defining $x0' = 1/x0$ and $x1' = 1/x1$, the following linear system can be solved to get the scaling factors of the time balancing operation:

$$\begin{pmatrix} 1/t0 & -1/t1 \\ n0 & n1 \end{pmatrix} \begin{pmatrix} x0' \\ x1' \end{pmatrix} = \begin{pmatrix} 0 \\ n0 + n1 \end{pmatrix} \tag{10.4}$$

After that we can directly multiply $x0' * n0$ and $x1' * n1$ to get our scaled node counts and apply the time balancing operation. In the same matter, time balancing can be applied to a set of three applications.

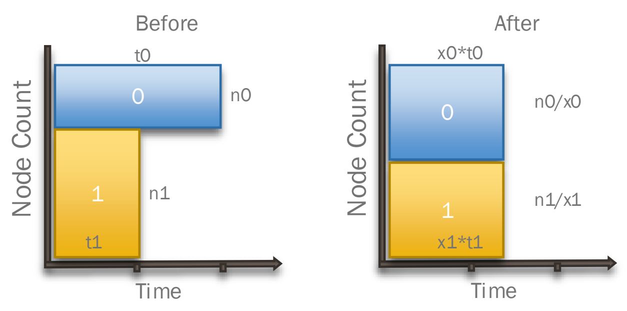

Figure 10.7: Time balancing applied to two jobs.

Figure 10.8 presents a similar scenario as before, but with one extra application. The following linear system can be solved to obtain the three scaling factors $x0'$, $x1'$ and $x2'$:

$$\begin{pmatrix} 1/t0 & -1/t1 & \\ & 1/t1 & -1/t2 \\ n0 & n1 & n2 \end{pmatrix} \begin{pmatrix} x0' \\ x1' \\ x2' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ n0+n1+n2 \end{pmatrix} \tag{10.5}$$

Similarly, the following linear system can be solved to obtain the four scaling factors $x0'$, $x1'$, $x2'$ and $x3'$ for a set of 4 applications:

$$\begin{pmatrix} 1/t0 & -1/t1 & & \\ & 1/t1 & -1/t2 & \\ & & 1/t2 & -1/t3 \\ n0 & n1 & n2 & n3 \end{pmatrix} \begin{pmatrix} x0' \\ x1' \\ x2' \\ x3' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ n0+n1+n2+n3 \end{pmatrix} \tag{10.6}$$

As can be deduced, these linear systems follow a simple pattern while increasing the number of jobs to be transformed. Its current implementation creates matrices in augmented form based on the selected jobs and solves the linear system with Gaussian elimination. These are solved quickly even for transformations with thousands of jobs. The current number of idle nodes (N_{idle}) can be added to the total of nodes to the last equation. For example, the last equation in the 4 job example becomes:
$x0' * n0 + x1' * n1 + x2' * n2 + x3' * n3 = n0 + n1 + n2 + n3 + N_{idle}$
Solving the modified linear system makes the time balancing operation produce a resource scaling vector that can fill these idle nodes as well.

Since the scaling factors are real numbers, while node counts are natural numbers, a floor operation is applied to each of the results: $x0' * n0$, $x1' * n1$, $x2' * n2$, etc. This means that the operation may indeed produce a surplus of nodes, instead of zero, in some cases. Any remaining nodes can be filled with the resource filling operation described next.

Resource Filling

This operation takes the number of nodes that are idle and expands applications according to their estimated efficient maximum number of nodes determined by the performance model and heuristic. The resource filling operation is much simpler than time balancing. The algorithm starts by creating a list of jobs based on their remaining runtime, from highest to lowest. This reordering helps lower the makespan of the schedule. It then expands

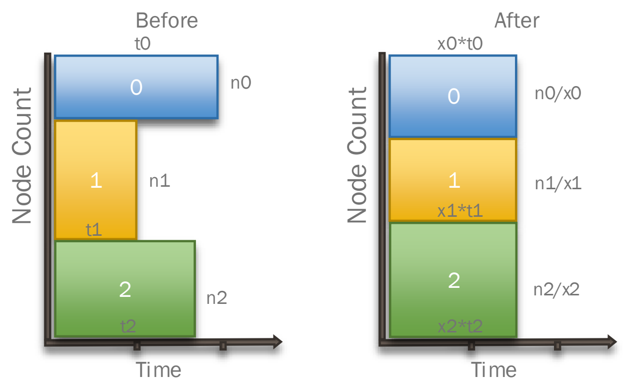

Figure 10.8: Time balancing applied to three jobs.

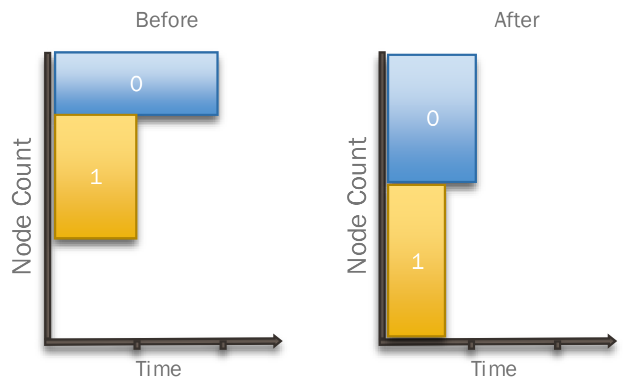

Figure 10.9: Resource filling applied to two jobs.

the jobs one by one until all idle nodes are filled or all candidate jobs reach their maximum node count.

Figure 10.9 provides an illustration of this operation being applied to two jobs. In this case the completion times do not need to match. This operation is performed so that the idle node count is minimized, and generally produces a reduction on the available number of nodes for potential application starts. Job priorities are not taken into consideration.

Resource Scaling Vector (RSV)

As mentioned earlier, currently the Elastic Batch Scheduler (EBS) is not implemented. The Elastic Runtime Scheduler (ERS) produces schedules for jobs that are started with the `SRUN` command. Multiple applications can be launched simultaneously and the `SRUN` command blocks until available resources are available given the number of nodes required by the application. The priority of the jobs is assigned based on their arrival time, where earlier jobs have higher priority following a First-Come First-Serve (FCFS) policy.

The ERS needs to produce a resource scaling vector (RSV) to minimize the makespan of all the jobs, running or blocking with a reservation. The RSV specifies new node counts for a subset of the running jobs. This vector is applied to the running system through an `srun_realloc_message` per application. If the node count is smaller than the current

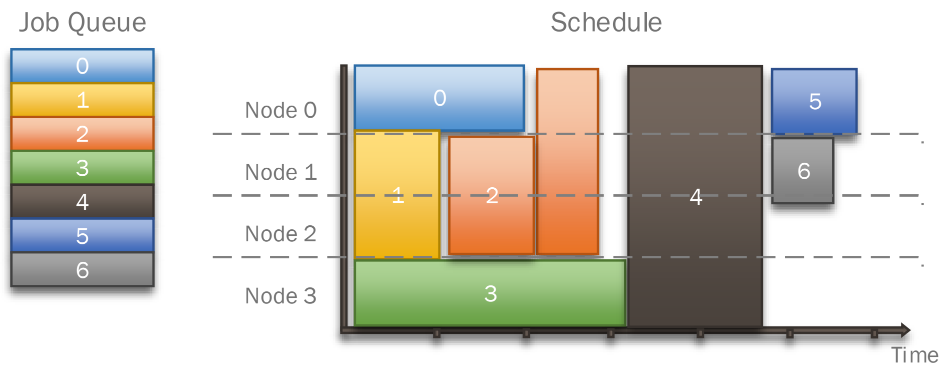

Figure 10.10: Possible schedule of a set of elastic jobs ordered by priority in the queue.

value, then an expansion is started. In contrast, if the node count is greater a reduction is started. If the new value is the same, then the message is not generated. This can occur due to rounding during any of the transformations, although the individual application was a candidate for adaptations.

The elastic backfilling algorithm applies traditional backfilling together with the previously described time balancing and resource filling techniques. The following rules summarize the heuristic used to apply these three techniques.

1. If one or more higher priority jobs were delayed due to lack of resources:
   a) Select a set of running jobs with resources that add up to the number of required resources to start the high priority jobs. Include any idle nodes based on availability.
   b) If a set is found, apply time balancing to it.
   c) Create reservations on the time balanced nodes for the high priority jobs.
2. If after starting new jobs and applying time balancing there are still idle nodes:
   a) Select jobs that fit in the gaps and apply traditional backfilling.
   b) Apply resource filling to fill any remaining jobs.

The first rule is an attempt to minimize the wait times of high priority jobs. The second rule shares the same goal as traditional backfilling techniques: to try to minimize the number of idle nodes. It is better than backfilling only when resource-elastic jobs are available to further fill the gaps. As mentioned before, because the EBS is not available, its shim always returns that the queue is empty. This means that currently new jobs are never started with backfilling.

Figure 10.10 illustrates an alternative schedule produced thanks to the support of resource-elasticity in the presented prototype. The job queue matches that of Fig. 10.1. As can be observed, in this case job 2 has been started at an initial smaller node count and later expanded. This allows the start of the jobs based on their priorities, therefore ensuring fairness. Additionally, the makespan is shorter. These are clear improvements to fairness and performance when compared to the previous static schedule. Ensuring per application efficient operation, with the techniques described in this chapter, is an additional benefit.

11 Evaluation Setup

The evaluation has been performed in the SuperMUC [13] petascale system. This supercomputer is managed by the Leibniz Supercomputing Center (LRZ) and is located in Garching, Germany. The resources of this HPC system are managed by an IBM Load Leveler resource manager.

There were some challenges encountered when testing the custom resource manager and communication library. As may be expected, it is not possible to replace the resource manager from the HPC system in production. Additionally, the new resource manager is composed of multiple daemons in a distributed memory setup. This system is shared among many users; this means that jobs need to wait for undetermined amounts of time in a queue. To overcome these challenges a set of scripts and custom binaries were developed.

11.1 Elastic Resource Manager Nesting in SuperMUC

A set of scripts have been written to allow the presented resource manager to be bootstrapped inside a job allocation. The scripts parse the set of host names that were allocated with each Load Leveler job that is submitted for testing the infrastructure. A `slurm.conf` file is generated dynamically for each job. With the configuration in place, the set of daemons are started at each host of the allocation. A few seconds are allowed for the resource manager to bootstrap itself and become capable of supporting new application starts. An HPC cluster of the size of the Load Leveler resource allocation is emulated this way in SuperMUC. In summary, the set of hosts allocated for a Load Leveler job becomes a test parallel system. Applications built with the custom MPI implementation can be launched inside Load Leveler jobs once the custom resource manager has finished bootstrapping itself inside an allocation.

This setup has the disadvantage that different performance is observed with each different job allocation, due to the different subset of nodes allocated by the Load Leveler in each run. There is not much control over the selection of the nodes. The job description may ask for the nodes to be allocated inside of a single island (set of racks with lower latency across nodes), and not much else. In general, a different set of nodes is expected with each new test run. This requires that multiple tests be performed to smooth out the variability of measurements due to different node sets.

Another disadvantage is that, as a normal job submission to the Load Leveler, the test jobs need to wait in the job queue. Jobs will have variable wait times depending on the size of the queue and the number of resources. It is common to observe wait times of multiple days for test jobs with more than a hundred node allocations. Because of this, only modest node counts (64 maximum) were requested with test jobs.

11.1.1 Phase 1 and Phase 2 Nodes

The SuperMUC system has multiple types of nodes divided in two sets: Phase 1 and Phase 2. There are three types of Phase 1 nodes: Fat Nodes, Thin Nodes and Many Core nodes. In this work, tests were performed only on Phase 1 Thin Nodes, and not on Fat or Many Core Nodes. Phase 1 thin nodes are based on Intel's Sandy Bridge architecture. Phase 2 nodes only have one type. These are all based on Intel's Haswell architecture.

Phase 1 nodes are based on a dual socket board with two Sandy Bridge-EP, Xeon E5-2680, Central Processing Units (CPUs). Each of these CPUs has 8 physical cores each, for a total of 16 per node, running at 2.7 GHz. These have support for Simultaneous Multi-Threading (SMT) and the feature is enabled. This must be taken into account in the configuration since the operating system sees the extra threads as additional physical cores. Each of these cores has a peak performance of 21.6 billion double precision floating point operations per second (gigaFLOPS). SuperMUC has a total of 9216 nodes of this kind. These have provided 2.897 petaFLOPS of performance under the High-Performance Linpack (HPL) benchmark used for the top 500 supercomputers [19] ranking.

Phase 2 nodes are also based on a dual socket board but with two Haswell-EP, Xeon E5-2697, Central Processing Units (CPUs). These have a higher CPU count of 14 physical cores each, for a total of 28 per node, running at lower 2.6 GHz. From a feature set, these are very similar to their Phase 1 counterparts. The peak floating point performance rating of these newer cores is higher at 41.6 gigaFLOPS. SuperMUC has a total of 3072 nodes of this kind. The higher peak gigaFLOPS as well as improvements in efficiency allow these nodes to reach 2.814 petaFLOPS of performance under the HPL benchmark.

11.1.2 MPI Library and Compilers Setup

All components (SLURM, MPICH and test applications) have been compiled with the GCC version 6 module provided in the SuperMUC system. The SuperMUC interconnect is based on Mellanox Infiniband network interfaces. Because of this, MPICH was configured to use its OpenFabrics Interfaces (OFI) [8] network module (with the Verbs provider). There is currently an open issue that results in incorrect reconfigurations when using this module. Because of this, resource adaptation tests were performed with the TCP/IP module in the Elastic MPI performance evaluation (next chapter). This means that significant performance improvements are possible in the near future. All other tests were performed with the OFI module. MPICH was configured with the `--enable_fast=all` option, and all binaries (application, MPI library and resource manager) were compiled with the `-O2` optimization level.

11.2 Testing and Measurement Binaries

Due to the complexity of the system, several special modes of operation were added to the resource manager daemons and the MPI library. These allow for quick isolated and precise testing of the different pattern detection, reduction, and scheduling algorithms. Testing these separate aspects of the infrastructure would have been much more time consuming and less precise during regular application runs.

12 Elastic MPI Performance

In this chapter, the performance and scaling of the new MPI operations is evaluated. For the measurements presented here, a simple test application that performs redundant MPI communication was used. The application runs indefinitely and adapts to new resources based on a precomputed schedule. Performance data was collected every time the test application adapted. Sweeps from 16 to 512 or 1024 processes were performed. Measurements were accumulated from 10 separate runs (with different allocations) for each type of SuperMUC node.

12.1 MPI_INIT_ADAPT

Figure 12.1 presents the mean time and standard deviation for the `MPI_INIT_ADAPT` operation. The times observed are indistinguishable from those with standard MPICH and SLURM with the provided PMI2 implementation. Poor scaling with increased numbers of processes is observed; this may become a target for optimization in the future. These times are observed from both the original processes in an application launch and those created by the resource manager on an expansion. The latency of this operation is hidden from preexisting processes thanks to the design of the `MPI_COMM_ADAPT_BEGIN` operation, as described in Sec. 6.2.3.

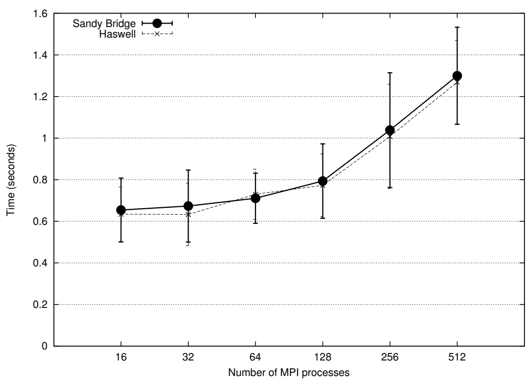

Figure 12.1: `MPI_INIT_ADAPT` latency.

12.2 MPI_PROBE_ADAPT

As mentioned in Sec. 6.2.2, this operation has been designed such that the general case is very fast. As can be seen in Fig. 12.2, it is the fastest operation in the MPI extension. When the adaptation flag is false, the latency of this operation is about 1 millisecond at 512 processes. The performance is much slower when the adaptation flag is set to true. As explained before, the expectation is that resource adaptations will be infrequent; therefore, the low latency of this operation when no adaptations need to take

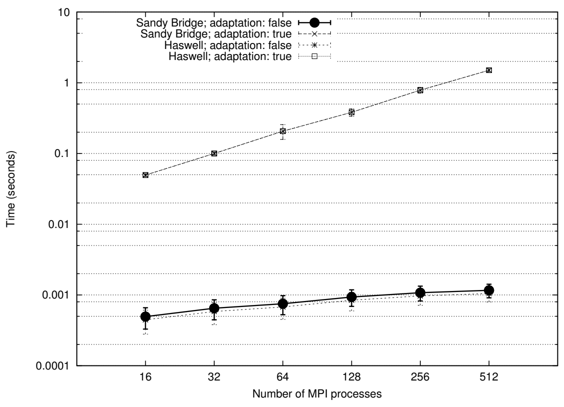

Figure 12.2: `MPI_PROBE_ADAPT` latency.

place is more important. The latency on the true case is dominated by the TBON protocol between the SRUN program and the daemons.

12.3 MPI_COMM_ADAPT_BEGIN

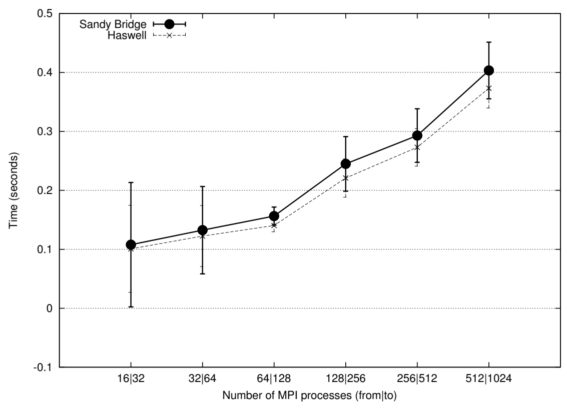

Figure 12.3: `MPI_COMM_ADAPT_BEGIN` latency from a number of staying processes to a new total.

A full sweep of all possible combinations of preexisting processes and expansion processes is not presented for this operation, since its latency is dominated by the size of the biggest process group. Because of this, balanced cases are presented, where preexisting process groups are of the same size as expansion process groups, for resulting process groups of double the size of the preexisting ones. It is also worth mentioning that a reduction of resources does not impact the performance of this operation, since preexisting leaving processes participate in the same way as preexisting staying processes during adaptation windows, due to their required participation during data repartitions.

As can be seen in Fig. 12.3, the implementation is successful in hiding the latencies related to the creation of new processes on new resources from preexisting processes. The measured times are significantly lower than the initialization times required by the children processes. Unfortunately, linear scaling has been observed due to the inherited implementation of the accept and connect routines from MPICH. These operations are reused in the current implementation and could be targets for optimization in the future.

12.4 MPI_COMM_ADAPT_COMMIT

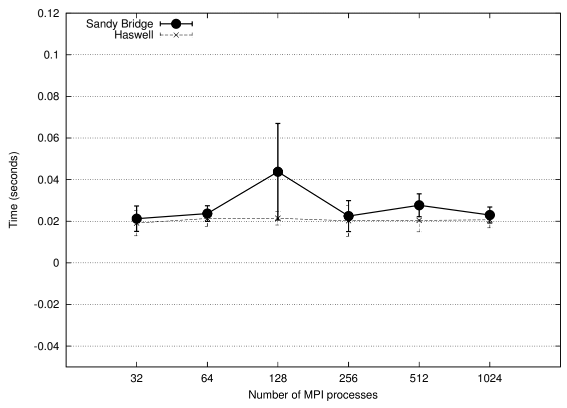

Figure 12.4: `MPI_COMM_ADAPT_COMMIT` latency.

The last operation to be evaluated is `MPI_COMM_ADAPT_COMMIT`. This operation is evaluated on the total number of processes, since it operates on the consolidated process group after an adaptation. This operation is in general very fast and has good scalability properties. It has not been a target for optimization. The reason for this is that all required synchronization takes place in the `MPI_COMM_ADAPT_BEGIN` operation and is stored in the MPI library. When this operation is called, the process group and communicator metadata is updated locally in the memory of the process.

13 Elastic Resource Manager Performance

A selection of resource manager operations is evaluated in this chapter. This selection contains all operations that impact the performance of MPI operations during normal computations. The operations that were not included are very numerous, but are either performed locally by one of the resource manager components, or do not impact the performance of preexisting MPI processes thanks to the latency hiding features described in previous chapters.

13.1 Tree Based Overlay Network (TBON) Latency

The communication between `SRUN` and the `SLURMD` daemons that manage the execution of an MPI application is important for the `MPI_PROBE_ADAPT` operation when the adaptation flag is set to true. The algorithm for probing has two sides: the side at each MPI process and the side at each `SLURMD` daemon. When the adaptation flag is set to true, multiple synchronization operations between the `SRUN` program and each daemon take place. These synchronization operations are performed over the Tree Based Overlay Network that connects `SRUN` to each `SLURMD` daemon. Because of this, the latency of messages over the TBON can impact the overhead of MPI processes when they are required to adapt.

Figure 13.1 presents the latency of a single message and its confirmation from each participating node. In the figure, its scalability based on process count is presented. This means that the results for the Sandy Bridge and Haswell nodes will differ mainly due to the different core counts in the nodes. In the case of Haswell, only 20 nodes are needed to run 512 processes, while 32 nodes are needed in the Sandy Bridge nodes. As expected of a TBON network, the latency of messages scale logarithmically.

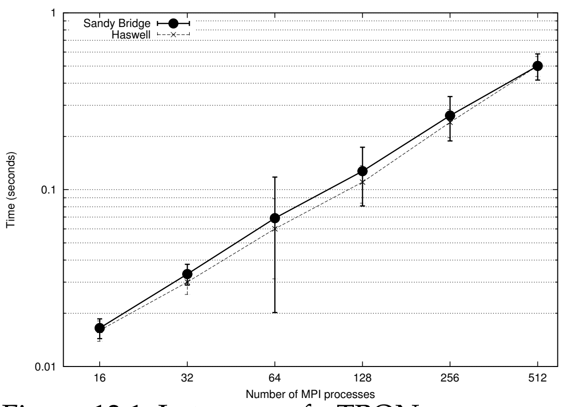

Figure 13.1: Latency of TBON messages from `SRUN` to daemons.

13.2 Control Flow Graph (CFG) Detection Overhead

In this section, the overhead of the set of operations that perform Control Flow Graph (CFG) detection is measured. Some of these operations impact the performance of MPI processes directly, while some can have a small impact since they are performed in the core where the `SLURMD` daemon of the node runs. These operations are: insertion, reduction, packing, unpacking and collapse.

The reduction, packing, unpacking and collapse operations are not as significant to the performance of MPI application processes due to their infrequent executions, as mentioned. That leaves the insertion operation as the only one that can impact the performance of application processes. In the remainder of this section, the latency of these operations will be presented. The measurements are presented based on their scalability with respect to the size of the CFG graph, the total number of processes at each node, and finally the number of iterations of the loop in the application.

13.2.1 Scaling with Control Flow Graph (CFG) Size

It is important to understand how the detection overheads scale with increased CFG complexity. Figure 13.2 presents the scalability of all of the operations for CFG sizes between 8 and 1024 entries. Results for Phase 1 and Phase 2 nodes are included side by side for comparison. The sizes of CFGs are typically less than 100 entries, so the wide range of up to 1024 entries is pessimistic.

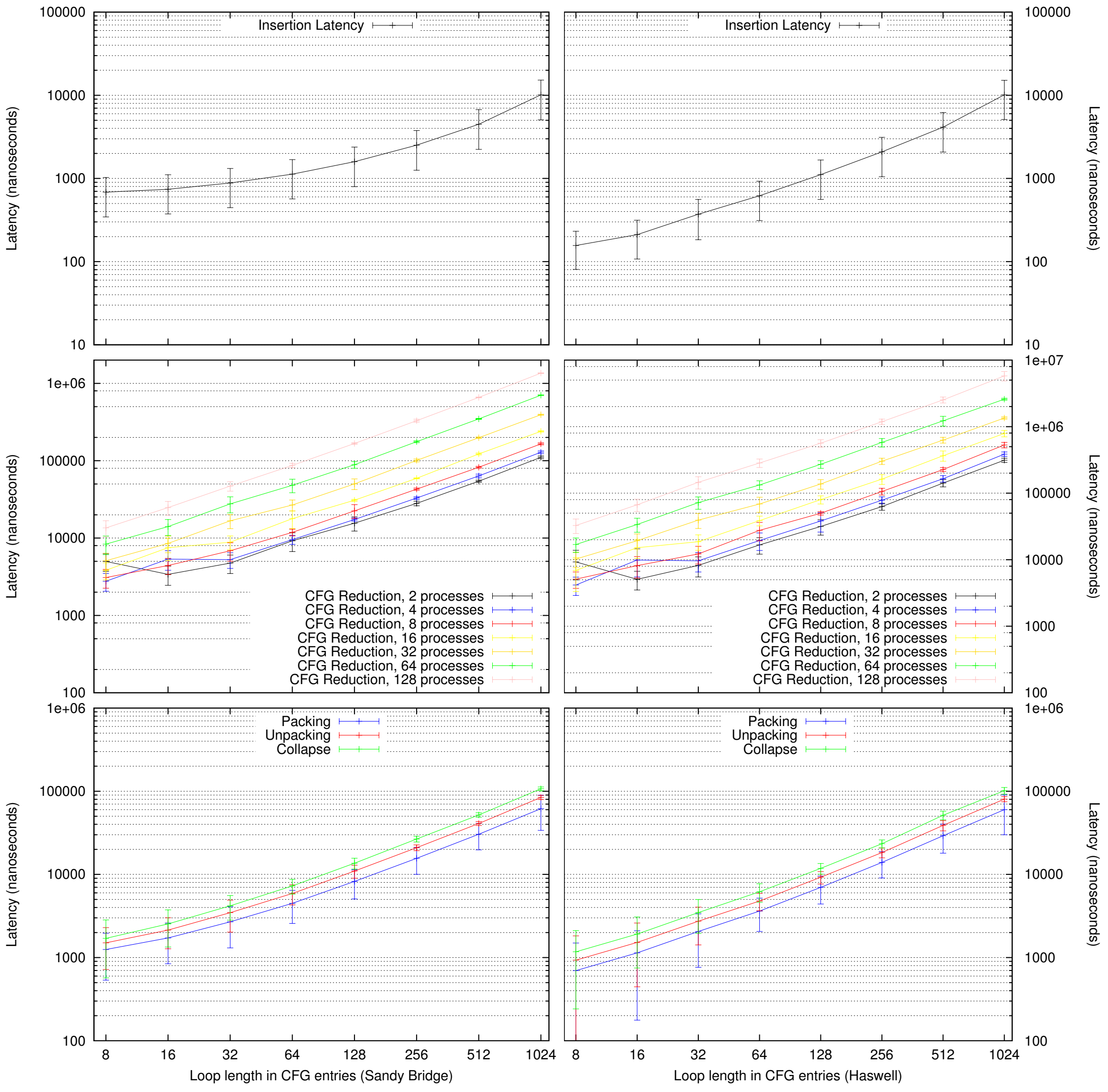

Figure 13.2: CFG size performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

As mentioned before, the insertion latency is the most significant overhead. Unfortunately, the insertion latency scales exponentially with the number of entries in the CFG. Fortunately, although with bad scalability, the actual cost of the operation is small. A typical MPI operation runs for multiple milliseconds, while the insertion overhead is of around 700 nanoseconds for a 8 entry CFG, up to 10 microseconds for the extreme case of 1024 CFG entries. For the typical case of 128 CFG entries, the overhead of insertion is less than 2 microseconds.

The CFG reduction operation scales exponentially with the number of entries in the CFG. The overhead of 5 microseconds for 8 entries up to about 500 microseconds in the extreme 1024 entry case are acceptable, given the infrequency of this operation. The packing, unpacking and collapse operations scale exponentially, but their actual costs is much lower than the reduction operation, since these are performed in parallel with the participation of each MPI process. Their maximum cost of 100 microseconds at the extreme case of 1024 entries is also acceptable given the infrequency of these operations.

13.2.2 Scaling with Process Counts

In addition to scaling with the size of the CFG, it is also important to evaluate how the overheads scale with increasing numbers of processes at each node. These are intra-node operations, so only process counts that are expected to be possible, without oversubscription, in near future HPC nodes are considered: from 2 to 128 processes.

Figure 13.3 presents scalability data for the detection operations based on process counts. Results for the larger CFG sizes 256, 512 and 1024 are presented for Phase 1 (left) and Phase 2 (right) nodes. As can be seen, the overheads for the insertion, packing, unpacking and collapse operations do not depend on the process counts, while the reduction operation does. Their latencies vary between a few hundred nanoseconds to a few hundred microseconds.

Not scaling with the number of processes is desirable, since it means that an arbitrary number of processes can be added at each node and these overheads will not increase. This is specially important in the case of the insertion latency, since this overhead is added to each MPI operation while the CFG detection mechanism is enabled. Once the CFG logic switches to verification, this overhead is removed. The packing, unpacking and collapse overheads are not as impact full to application performance, as mentioned before, since these occur infrequently.

The situation for the reduction operation is not so fortunate, where its overhead increases with the number of processes per node of an application. As measured before, the overhead of this operation also increases with larger CFG sizes. Because of this, this operation has the worse scaling properties of the measurement infrastructure. Fortunately, these operations do not occur frequently and the absolute latency numbers it reaches are still not large.

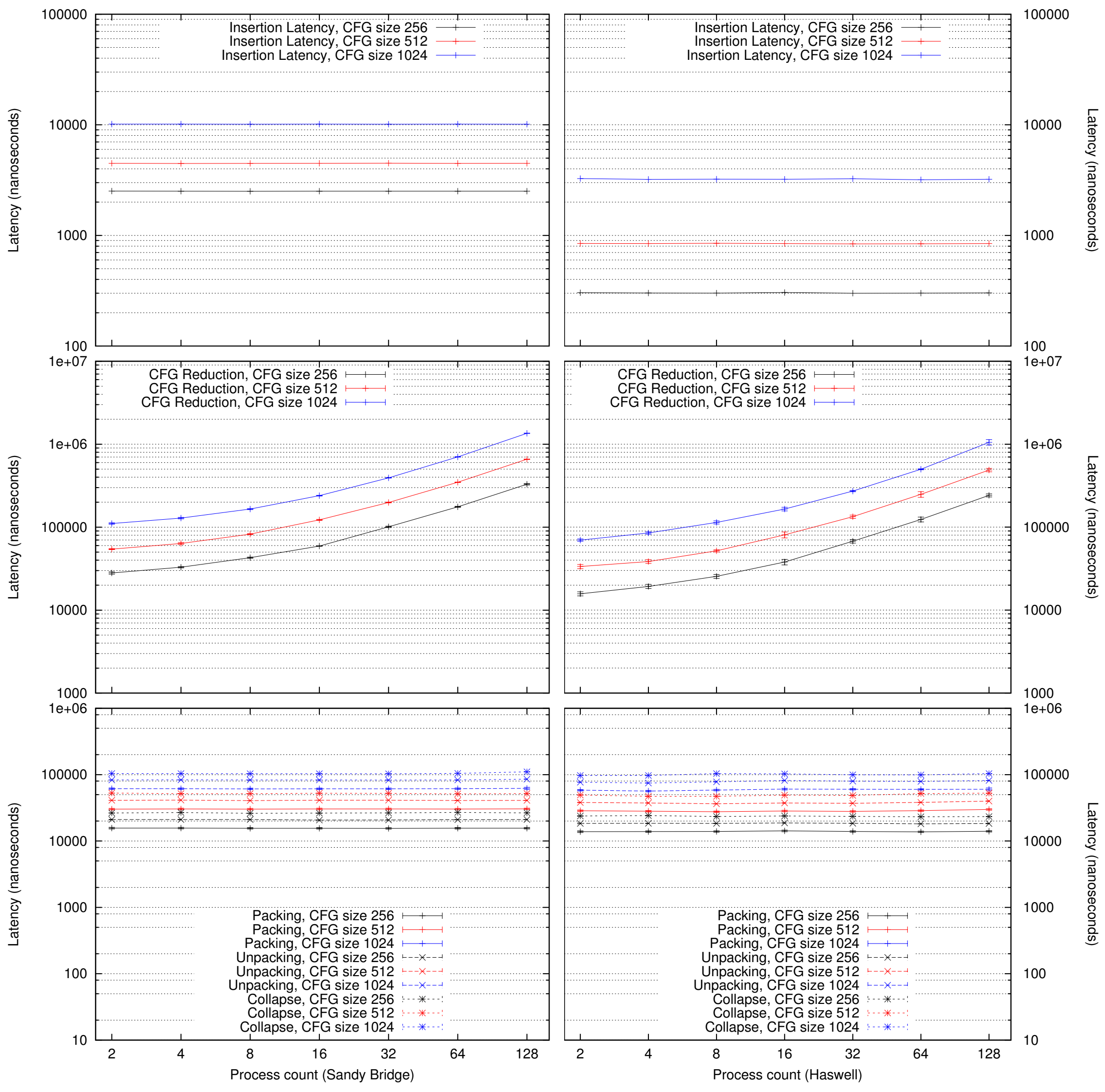

Figure 13.3: Process count performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

13.3 MPI Performance Impact of the CFG Detection Overhead

Additional measurements were performed to evaluate the impact of these operations in actual MPI operations. MPI operations can run from a few microseconds to multiple seconds, depending on the type of operation, the number of processes and the size of the buffers.

In Fig. 13.4, results for the `MPI_SEND` and `MPI_BCAST` operations are presented. These two operations were selected since they have the lowest latencies among the set of point-to-point and collective operations, respectively. The figure presents the latency for the `MPI_SEND` operation at the top and the `MPI_BCAST` operation at the bottom. Results for Phase 1 (left) and Phase 2 (right) nodes are presented side by side for comparison. Results for 16 and 1024 processes are presented with buffer sizes from 16 bytes up to a megabyte. The size of the CFG was set to 32 for these tests. Most applications and benchmarks that have been evaluated generate less CFG entries by the time they terminate.

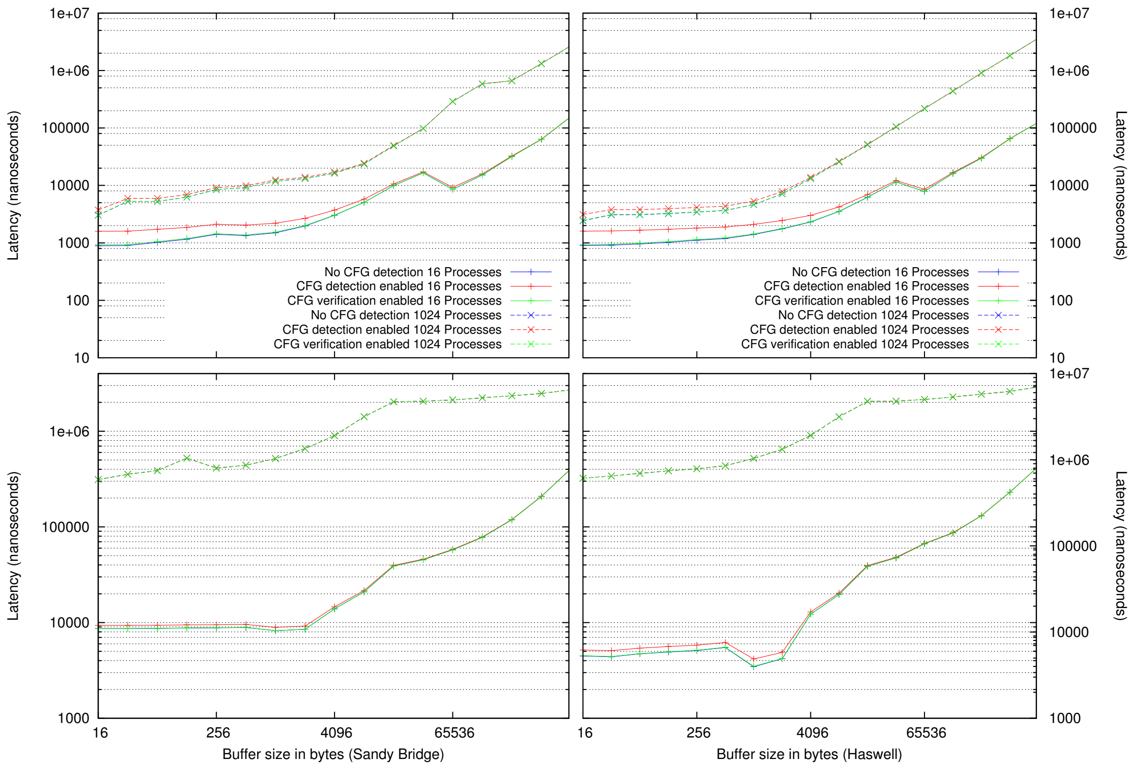

Figure 13.4: MPI_SEND (top) and MPI_BCAST (bottom) performance examples with detection enabled and disabled on a 32 entry CFG loop. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

As can be seen in the plots, the performance of MPI_SEND is only impacted significantly for message sizes of up to 4096 bytes, but only at lower process counts. For the case of 1024 processes, the overhead of the CFG detection algorithm is insignificant even for very small messages of 16 bytes. Additionally, the overhead of detection is not measurable on verification mode. This means that its overhead will only be observed when the detection algorithm has not encountered a loop, or when it exits a loop and resumes its detection.

A smaller performance impact can be observed for the MPI_BCAST operation. As mentioned before, the latency of this operation is the lowest among MPI collectives; therefore, the impact of CFG detection can be expected to be almost negligible when collectives are being used. Although the detection overhead is lower in terms of absolute latency, the percentage impact is higher in the case of Phase 2 nodes.

14 Case Studies with Distributed Memory Applications

Two computational kernels are evaluated in detail in this chapter: a matrix-matrix multiplication kernel and a Gaussian elimination kernel. The matrix-matrix kernel is based on the Cannon algorithm, while the Gaussian elimination kernel is a naive row-block implementation.

These have been selected due to their simplicity: these have well understood scalability and efficiency properties and execute fast enough. Because of this, a full sweep of possible resource combinations with them can be done in a timely manner. More complex applications are already developed or under development, such as Computational Fluid Dynamics (CFD) simulations with AMR [70].

14.1 Cannon Matrix-Matrix Multiplication

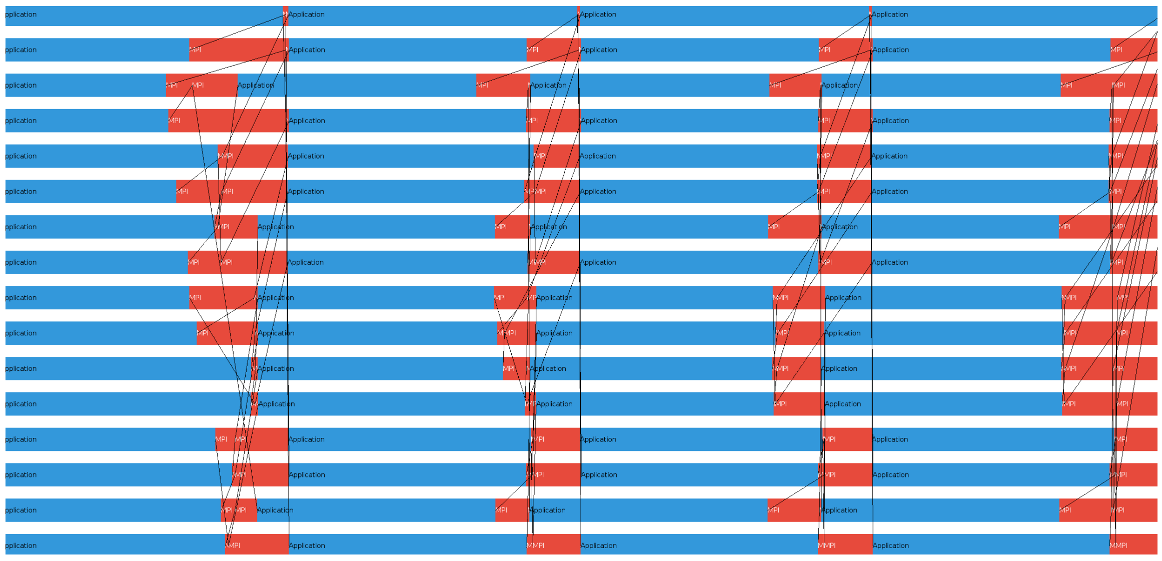

Figure 14.1: Cannon matrix-matrix multiplication trace for 16 processes. MPI time in red and application time in blue.

In this section, a matrix-matrix distributed multiplication kernel based on the Cannon [54] algorithm is analyzed. The response of the CFG detection and scheduling algorithms of the infrastructure to its performance and scalability properties are discussed.

14.1.1 Basic and EPOP Implementations

The original implementation was a small single C source file with the MPI based Cannon algorithm. The new implementation uses MPI topologies to simplify the communication

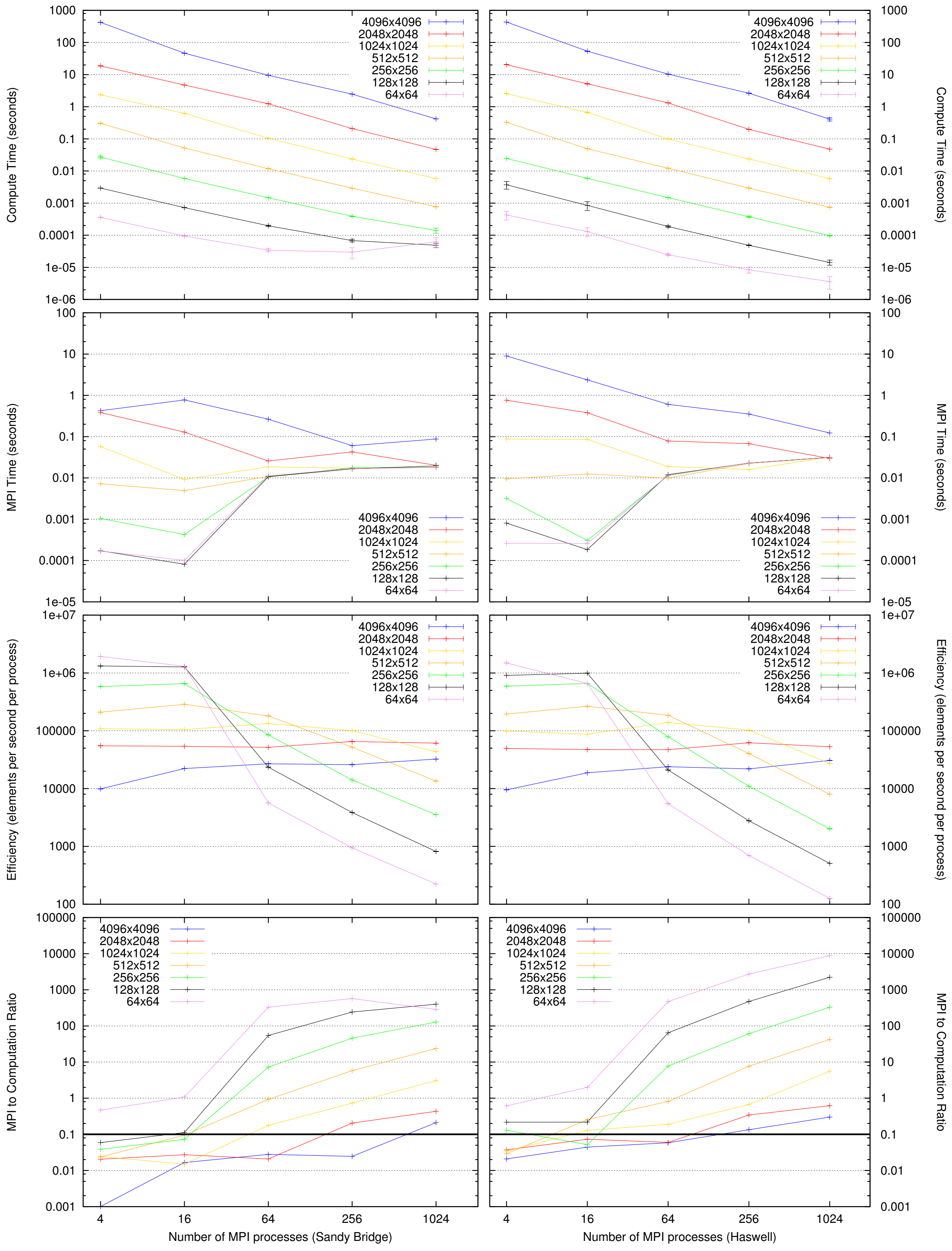

Figure 14.2: Compute, MPI, efficiency and MTCT ratio (top to bottom) of a Cannon Matrix-Matrix multiplication kernel. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

with neighbor processes during computation. This is particularly helpful with the Cannon algorithm given its block wise exchanges in the main kernel. The kernel remains the same in both EPOP and basic versions of the code. This kernel is presented in Listing 14.1.

```
for(cannon_block_cycle = 0; cannon_block_cycle < sqrt_size; cannon_block_cycle++){
  for(C_index = 0, A_row = 0; A_row < A_local_block_rows; A_row++){
    for(B_column = 0; B_column < B_local_block_columns; B_column++, C_index++){
      for(A_column = 0; A_column < A_local_block_columns; A_column++){
        C_local_block[C_index] +=
          A_local_block[A_row * A_local_block_columns + A_column] *
          B_local_block[A_column * B_local_block_columns + B_column];
      }
    }
  }
  // rotate blocks horizontally
  MPI_Sendrecv_replace(A_local_block, A_local_block_size, MPI_DOUBLE,
      (coordinates[1] + sqrt_size - 1) % sqrt_size, 0,
      (coordinates[1] + 1) % sqrt_size, 0, row_communicator, &status);
  // rotate blocks vertically
  MPI_Sendrecv_replace(B_local_block, B_local_block_size, MPI_DOUBLE,
      (coordinates[0] + sqrt_size - 1) % sqrt_size, 0,
      (coordinates[0] + 1) % sqrt_size, 0, column_communicator, &status);
}
```

Listing 14.1: Cannon kernel.

The adaptation window was inserted in the main kernel loop. No proper adaptation code was implemented. Instead, the root process of the application redistributes the matrix data on each adaptation. A better solution will be to add an MPI based collaborative repartitioning scheme where all processes participate.

For testing, long running applications are needed to observe the behavior of the scheduler. Because of this, an additional loop was added that effectively repeats the number of matrix-matrix multiplications that are performed by the application. The source matrices are not modified, therefore no changes were necessary to ensure correctness.

Although it is a very simple application, it suffers from the difficulties described in the EPOP chapter. An EPOP version of the application was also developed. Figure 14.3 illustrates its design based on EPOP blocks. It is a single EP application, with its required initialization block and a single rigid phase used for finalization. Because EPOP operates at a very coarse level, the performance of the application in both versions is indistinguishable. Because of this, the performance data presented in this evaluation are relevant to both implementations.

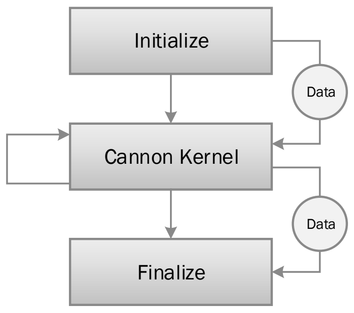

Figure 14.3: Cannon application with EPOP blocks.

14.1.2 Pattern Detection

The Cannon application was also used to verify the correctness of the pattern detection functionality presented in the scheduling chapter. Figure 14.4 illustrates what occurs when the system detects the CFG of the non-EPOP implementation. At the beginning, each application process starts the detection process. Fortunately, the application is simple enough that the detected CFG of a full execution can be illustrated. The CFGs of the root process

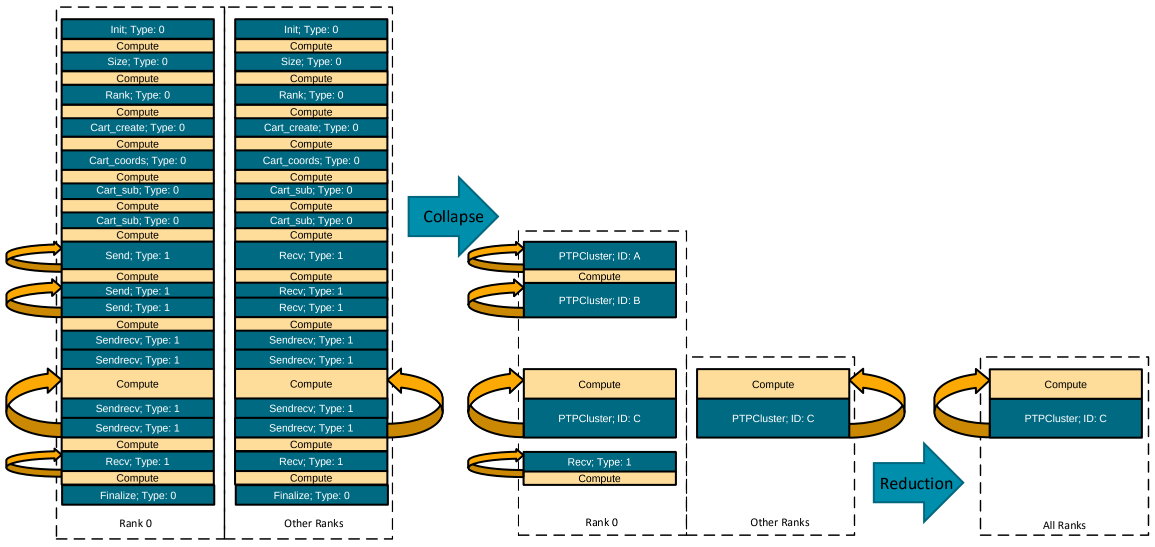

Figure 14.4: Cannon CFG detection process illustrated.

and other processes are illustrated on the left side of the figure. These differ in that root has more loops than the rest of the processes. There is a loop where matrix dimensions are broadcasted, and another loop where the operand matrix sub-blocks are distributed. There is an additional loop where the final results are gathered. For each of these three loops, the rest of the processes have a matching receive (one for each of the first two loops at root) and a matching send (for the final gather loop at root). The CFG collapse and reduction operations for this application are illustrated from left to right, respectively. The collapse operation simplifies the loops at each process. The reduction operation detects the loops that are present at all processes and produces the distributed loop metadata.

14.1.3 Performance Analysis

A trace with an allocation of 16 processes showing the MPI and application times for this application is presented in Fig. 14.1. As can be seen, the proportion of MPI to compute time is low. Figure 14.2 shows a detailed sweep of the performance and efficiency properties of this application based on the number of processes. It helps to remember that, in the presented infrastructure, the number of processes of an application is ensured to match the number of CPU cores that are allocated to it. In the figure, the different times for the iteration of the detected loop in the CFG are presented. From top to bottom: total time, MPI time, efficiency and MPI to compute time ratio (the MTCT metric described in the scheduling chapter).

As can be seen in the bottom plots, the number of processes for MTCT metric values below 0.1 correlate well with the number of processes where the efficiency metric of the application is near the maximum possible for each input size. The heuristic described in the scheduling chapter halves the number of process in all cases where the average or trend MTCT values are above 0.1. The quality of the decisions can be verified for this application, since its performance and efficiency has been evaluated before for a wide range of input matrices and process counts. In this case, the algorithm makes resource adaptation decisions that do not lower the application's parallel efficiency significantly.

14.2 Gaussian Elimination

In this section, a distributed Gaussian elimination kernel is analyzed. The approach to its analysis is very similar to that of the previously discussed Cannon matrix-matrix multiplication implementation. This kernel has very different performance properties when compared to the previous matrix-matrix example.

14.2.1 Basic and EPOP Implementations

The original distributed Gaussian elimination implementation was even simpler than the previous matrix-matrix multiplication example, with a single C source file of less than 300 lines of code. This is a very minimalistic Gaussian elimination implementation with row-blocking.

This base implementation was extended for resource-elastic execution in a similar manner to the Cannon application, with the root process redistributing the matrix and right hand side vector, instead of a collaborative repartitioning scheme. Also similarly, this otherwise short running application was made to run longer with an outer loop; this is necessary to properly observe the response of the scheduler.

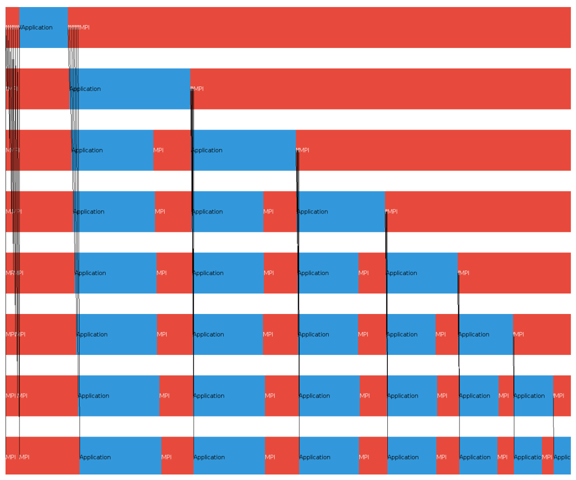
Figure 14.5: Gaussian elimination trace for 8 processes. MPI time in red and application time in blue.

The EPOP version of this application is identical in structure to that of the Cannon application. A single initialization block, a single EP block and a single rigid block for finalization. Refer to the EPOP illustration in the previous section. Again, in this case the benefits of EPOP were more related to the elegance and cleanliness of the implementation, with no extra branching code to enter adaptation windows or for locating joining processes. The performance is indistinguishable in both implementations.

14.2.2 Pattern Detection

The pattern detection produces a peculiar result that is too complex to illustrate it compactly in this document. The implementation has multiple loops, but only the inner loops get detected at all processes. In the current implementation, only these loops are then tracked. This creates a situation where a lot of relevant performance data gets clipped out of the model. The measured MTCT metric allows the heuristic to make the correct decisions, but the results show a lower ratio than that found with tracing. This type of communication pattern could be one of the worst cases for the detection algorithm presented in this work.

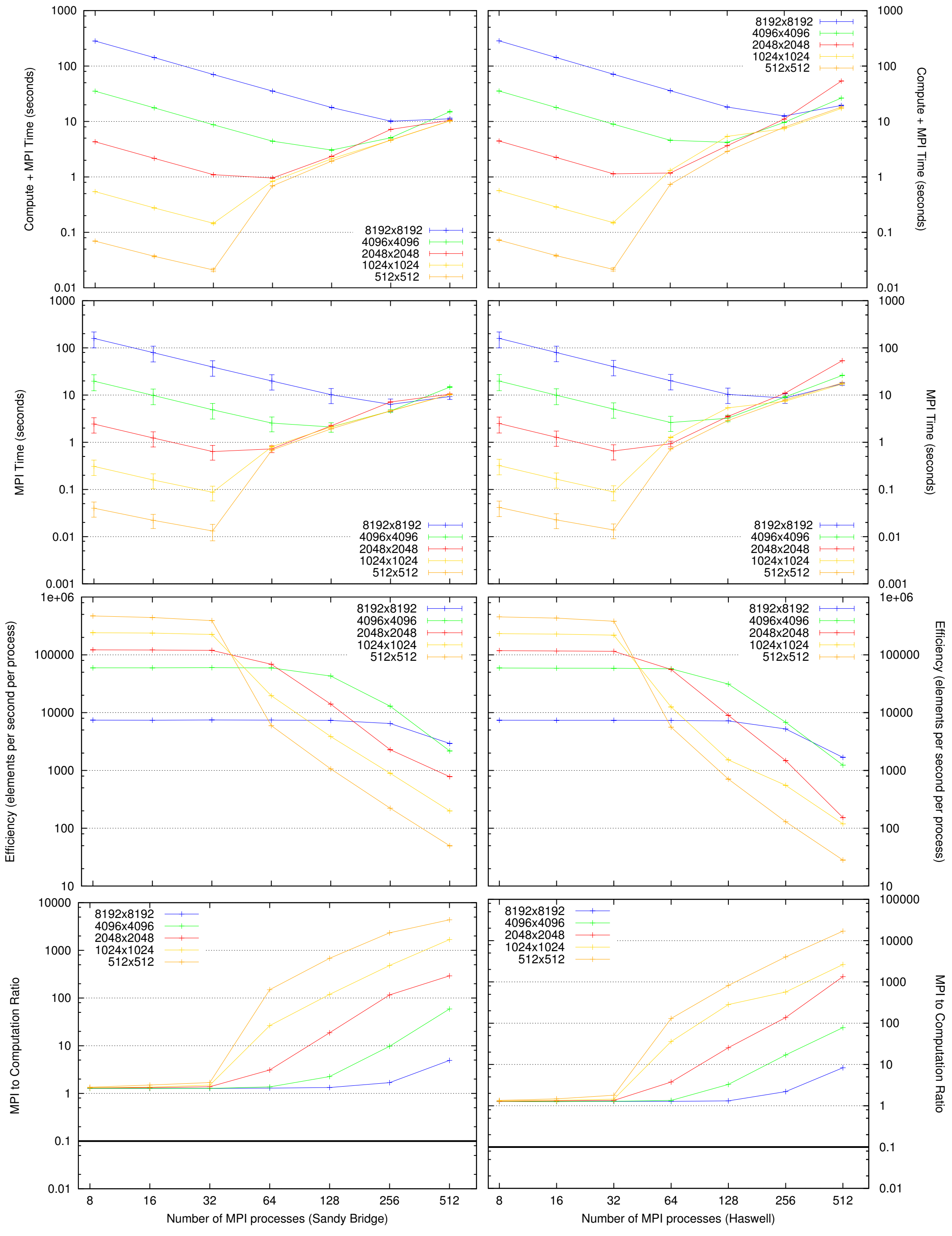

Figure 14.6: Compute, MPI, efficiency and MTCT ratio (top to bottom) of a Gaussian elimination kernel. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

14.2.3 Performance Analysis

Figure 14.5 shows a trace for this application with 8 processes, with compute time in blue and communication time in red. As can be seen, most of the time is spent in MPI operations. This can only result in very low efficiency metrics for this application. Indeed, this is a naive row-blocking distributed implementation of the Gaussian elimination algorithm and is known to have low efficiency and poor scaling properties with increased numbers of processes.

Figure 14.6 shows a detailed sweep of the performance and efficiency properties of this application based on the number of processes. Again, the infrastructure ensures that the number of processes match available physical cores in all cases. Similarly to the previous analyzed kernel, the figure presents from top to bottom: compute, MPI time, efficiency and MPI to compute time ratio. Again, the ratio in the bottom plots is the MTCT metric used by the performance model described in the scheduling chapter. In both this case and the matrix-matrix multiplication before, these are only the averages and not the trend values for the MTCT metric.

The heuristic reacts very differently with this kernel, when compared to the Cannon kernel. In this case, the MTCT is never below the 0.1 threshold, as can be seen. Indeed, the scheduler always determines that this application is operating at an inefficient scale and will halve its resources in each scheduling iteration until it reaches its minimum number of processes, as specified by the user.

14.3 Cannon Matrix-Matrix Multiplication and Gaussian Elimination Interaction

The interaction between the Cannon matrix-matrix and the Gaussian elimination (GE) applications can be well understood after their characteristics have been determined individually. In this section, these applications are run together and the response of the system, through its performance modeling features and its scheduler, is observed.

These observations are made currently in the log output of the Elastic Runtime Scheduler (ERS). Its logs record when the `srun_realloc_message` is sent and when the commit message is received, for each individual application. It also logs the preexisting allocation, the expansion allocation and the final allocation of each individual resource adaptation (in both node counts and process counts). In the plots presented in this section, only the completion times are illustrated, since these are the most relevant. The resource adaptation decisions are done within milliseconds of each periodic scheduler decision time. It is important to note that the ERS schedules job steps and not jobs. Job steps are the combination of an `SRUN` instance, and the MPI processes of an application with their `SLURMSTEPD` per node. These applications adapt quickly: the difference between the time when the `srun_realloc_message` is sent and the commit is confirmed is in the millisecond scale. Future work needs to include the adaptation times of applications in the scheduling decision; adaptation times are currently ignored.

First, combinations of both applications are run to evaluate the effect of the resource filling operation. Afterwards, two instances of the Cannon matrix-matrix multiplication application are used to observe the effect of the time balancing operation. It is important to note again that these applications have been modified to be long-running, by adding an outer loop around their core kernels. This allows the system to properly collect perfor-

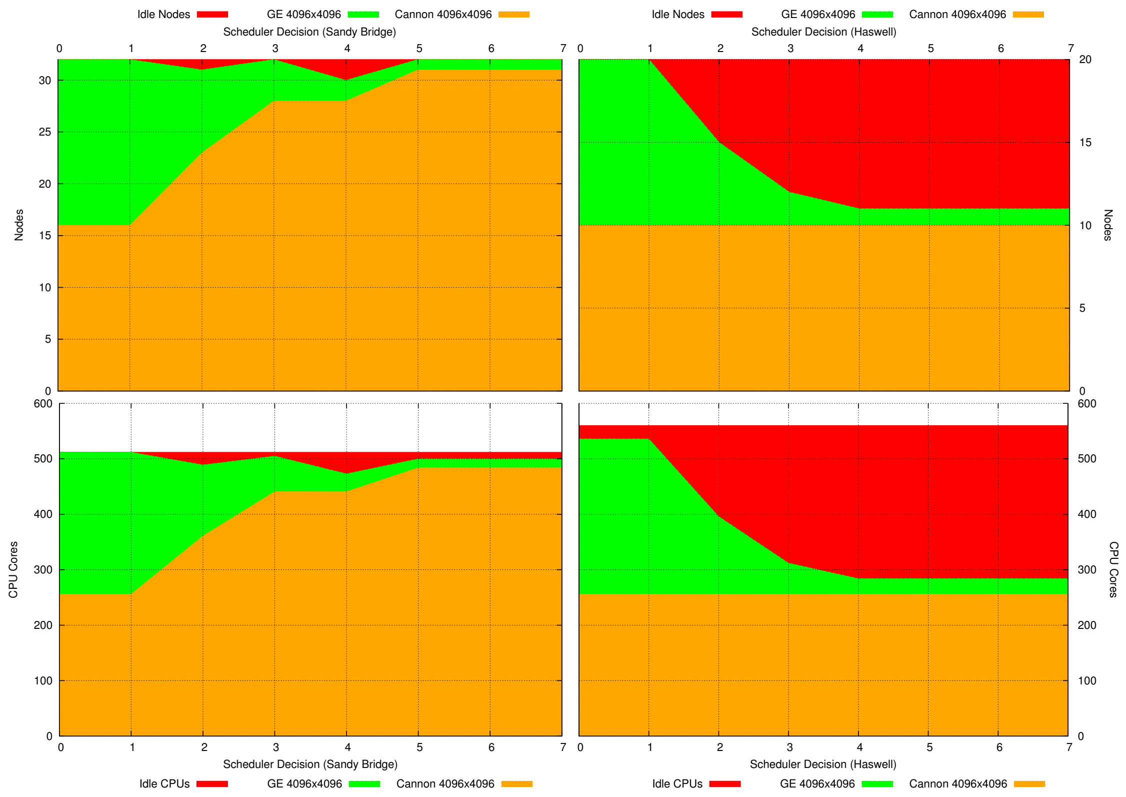

Figure 14.7: Nodes (top) and MPI processes (bottom) during the interaction between the Cannon's matrix-matrix multiply kernel with 4096x4096 matrices and the Gaussian elimination application with 4096x4096 matrices. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

mance data and react by modifying their resources. Finally, a discussion of the effect of the upper and lower MTCT thresholds is included.

14.3.1 Gaussian Elimination and Cannon Matrix-Matrix with 4096x4096 Matrices

Figure 14.7 shows the node counts of the allocations of the Gaussian elimination and Cannon applications. The horizontal axis represents the times where the scheduler makes a decision. The frequency of these decisions can be configured, and has been set to one minute for these experiments. As can be seen in the plots, in the first few iterations the scheduler does nothing. This is to be expected since the performance data is requested in the first step of the scheduler and both applications have been started simultaneously. The performance data is available only after the second scheduler step, and this is where the first resource adaptations can take place.

Node counts are presented in the top, and CPU cores in the bottom. Results for both types of SuperMUC nodes are included, with Phase 1 results on the left and Phase 2 nodes on the right, in the same arrangement as previous figures. As can be seen in the plots, the results vary greatly depending on the node type.

For Phase 1 nodes, the scheduler manages to keep the idle node and CPU counts low. This is thanks to the high efficiency estimation for the Cannon application with 4096x4096 matrices. This application has its resources increased from 16 nodes to 28 nodes in 2 steps. Afterwards in step 5, its resources are increased to 31 nodes by an application of the re-

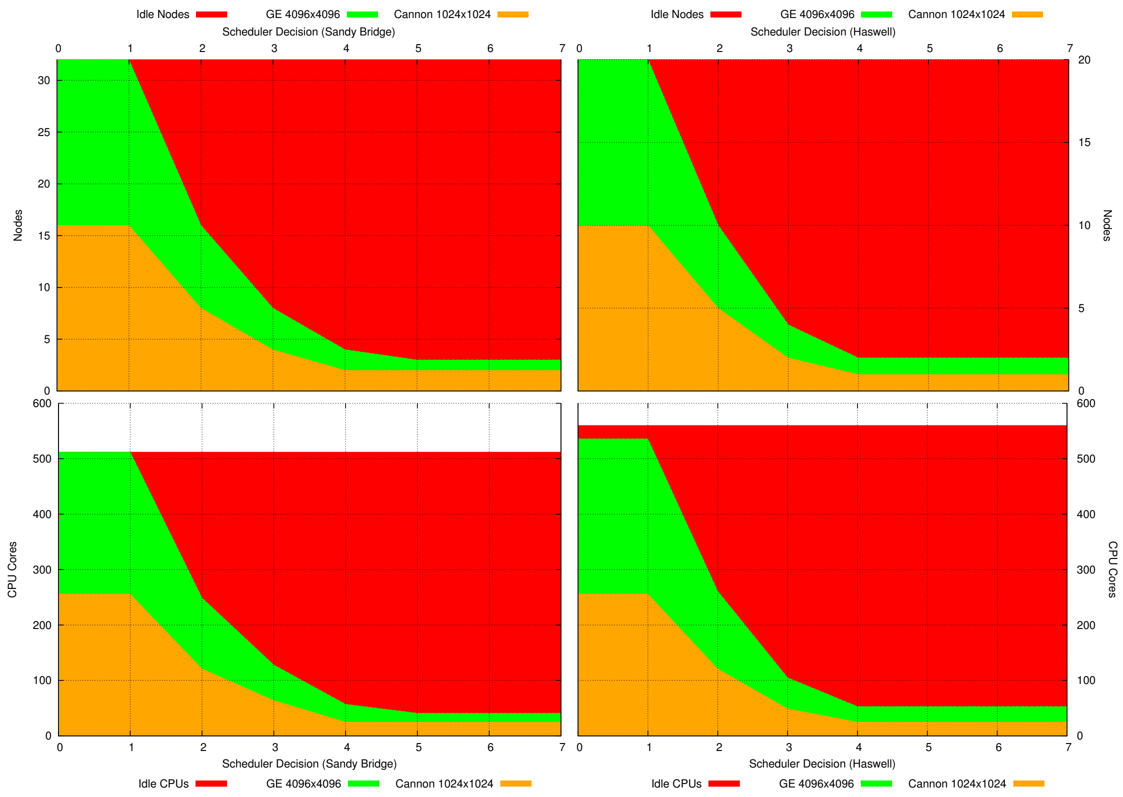

Figure 14.8: Nodes (top) and MPI processes (bottom) during the interaction between the Cannon's matrix-matrix multiply kernel with 1024x1024 matrices and the Gaussian elimination application with 4096x4096 matrices. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented.

source filling operation; this successfully lowers the idle node count to zero. The sequence of adaptations for the Cannon application is due to its implementation being limited to square process counts only. This is also the reason why there are still idle CPU cores although all nodes have been filled. In contrast, the efficiency of the GE application is estimated to be low. This application is reduced from 16 nodes to 1 node in 4 steps.

The sequence of adaptations and the idle node counts are very different for Phase 2 nodes. The Cannon application is estimated to lose efficiency at a lower node count by the performance model. The band created by the upper and lower MTCT thresholds, with values 0.1 and 0.01, result on the application being removed from the candidate set at each scheduler iteration. The application is not perturbed and it is left to run with a 16 node allocation for the entirety of its run. The GE application has the same resource reduction as in the Phase 1 case. Its resources are reduced to one node in the same 3 reduction steps.

14.3.2 Gaussian Elimination and Cannon Matrix-Matrix with 1024x1024 Matrices

An additional test is performed with the same applications but with the input matrices changed to the 1024x1024 size for the Cannon application. The changes in the resources for each application are presented in Fig. 14.8. This figure follows the same organization as before. The difference in the resource adaptations is significant.

The resource adaptations for both types of nodes are indistinguishable in this case. The resources of the GE application are the same as before, since the input of this application was not modified. Additionally, the efficiency of this application is estimated to be low for all possible input sets, so a difference on the resource scheduling decision would be unexpected. In contrast, the response for the Cannon application is very different with the 1024x1024 input size. The application has its resources lowered from 16 nodes to 2 nodes in 3 steps. The upper and lower band for the MTCT metric keep the application fixed at 2 nodes. The systems reaches a steady state with 1 node for the GE application and 2 nodes for the Cannon application in Phase 1 nodes, and one each in Phase 2 nodes. The number of idle nodes and idle CPU cores is much larger for both types of nodes in this case.

14.3.3 Cannon Matrix-Matrix with 4096x4096 Matrices and Different Time Limits

The next and final scenario presented is a set of three Cannon application instances that each has different numbers of iterations on their outer loops. These iterations are set so that they approximate 5, 10 and 15 minutes of runtime. These times are set through the `SRUN` command's `--time` option in these experiments, since the Elastic Batch Scheduler (EBS) is not yet available. The third instance of this application blocks until resources are available for it. This is achieved by specifying the `--immediate` option; this option forces the `SRUN` instance of the application to block until its required resources are available.

As can be observed in Fig. 14.9, the number of idle nodes in the schedule for Phase 1 differs significantly for the cases with and without time balancing. Time balancing provides three benefits in this case: it manages to reduce the number of idle nodes, it reduces the wait time for the third application instance start, and it also reduces the makespan of the schedule.

For Phase 2 nodes, no time balancing was possible since the scheduler removes both instances of the Cannon algorithm from the candidate list for adaptations. This is a consequence of the efficiency estimation given the upper and lower thresholds. The idle node and CPU counts are a lot higher in this case. The wait time for the third instance of the application is not lowered. Finally, the makespan of the schedule is also not improved.

14.3.4 Upper and Lower MTCT Threshold Effects Summary

The MTCT upper and lower bands influence the generation of the Resource Range Vector (RRV) for the list of candidate jobs at each scheduling step. Each entry in the final Resource Scaling Vector (RSV) that is applied to the running system is within its range in the RRV. By modifying the upper and lower thresholds, the RRV and therefore the possible values of the RSV can be influenced. A wider range for these thresholds means that applications are more likely to be left running uninterrupted instead of their resources being adapted. Higher values for both of these thresholds increase the tolerance of the system to the estimated inefficiencies of individual applications. A large difference in these values help prevent oscillations, where the system may attempt to increase and decrease the resources of an application in an endless cycle. This can be very detrimental to performance, especially of applications with large adaptation times.

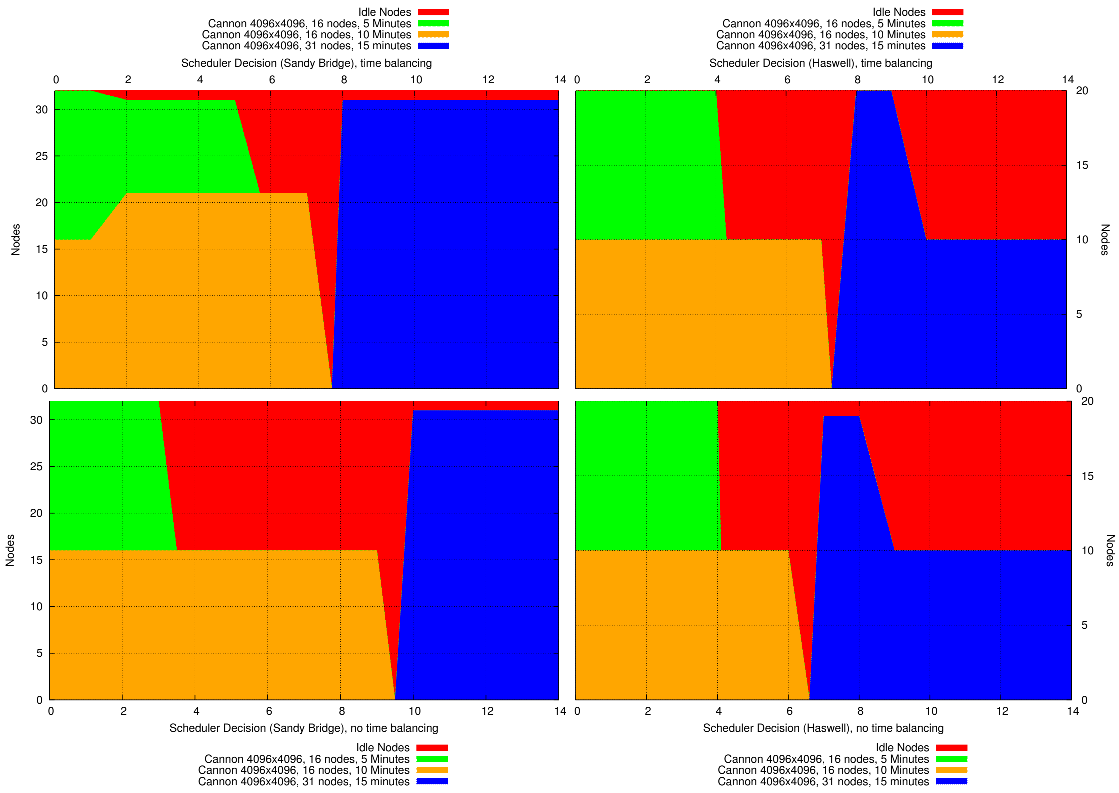

Figure 14.9: Time balancing enabled (top) versus disabled (bottom) with different run times on the same Cannon Matrix-Matrix multiplication application.

Different trade-offs can be configured between idle node counts and total system efficiency with the adjustment of these thresholds. A higher idle node count favors the start of new jobs from the queue, while a lower idle node count favors the completion of running jobs. The current setting of 0.1 for the upper threshold and 0.01 for the lower threshold appears to favor queued job starts and higher overall estimated system efficiency in the types of nodes evaluated. These parameters need to be adjusted for each HPC system based on performance data and expected job queue lengths.

Results with the upper threshold for the MTCT metric modified from 0.1 to 0.5 are presented in Fig. 14.10. As can be observed, the schedule is the same for the Phase 1 nodes case. The same cannot be stated for the schedule in Phase 2 nodes. The increase in the threshold enable the two instances of the Cannon application to run with lower estimated efficiencies. This allows the system to generate an RRV where both applications can be expanded, and therefore time balancing applied. The result is that the idle node count is reduced, the wait time for the third Cannon instance is reduced, and the makespan is reduced. This comes at a trade-off of overall lower estimated efficiency. If jobs could be started from a queue, this also has the effect of delaying their start since idle nodes are minimized.

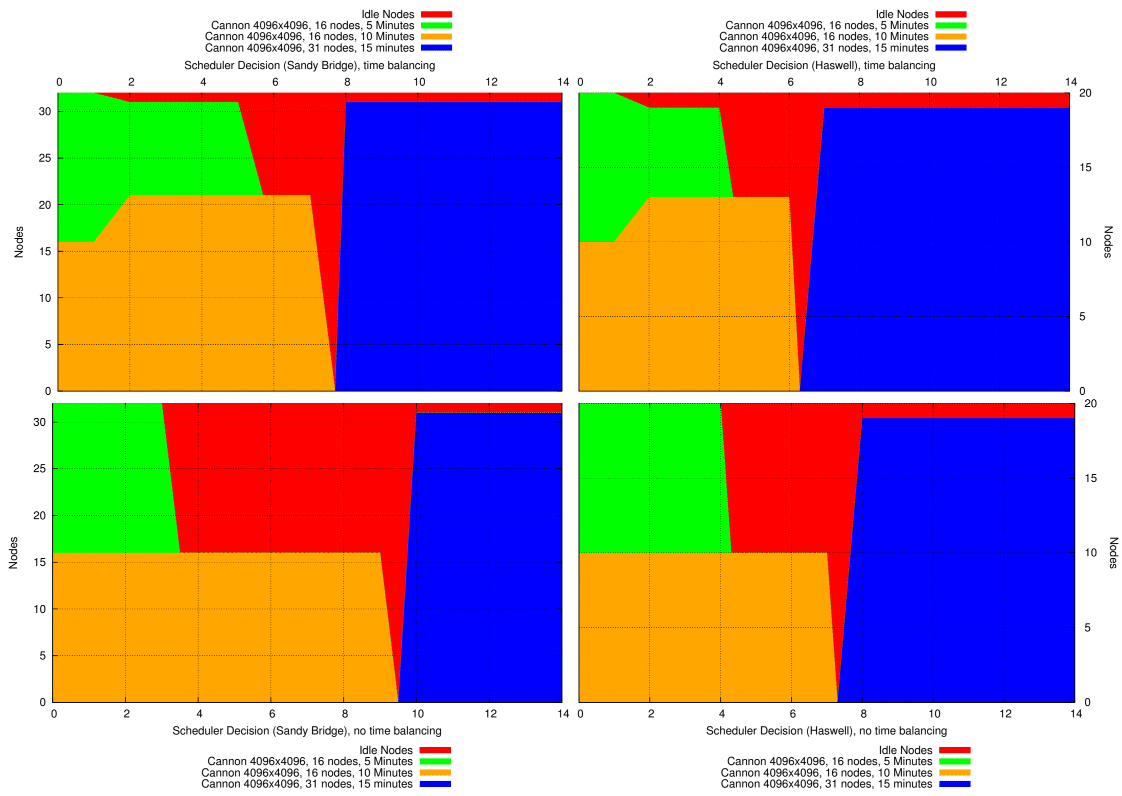

Figure 14.10: Repeat of the time balance test with upper MTCT threshold increased to 0.5 for comparison with Fig. 14.9.

14.4 Summary and Discussion

The doubling and halving of resources is an underestimation of the needed adaptations in our evaluations. More precise individual ranges in the RRV are desirable. This could be achieved with performance model improvements in the future. Underestimating the ranges has the negative effect of requiring more adaptation steps before a steady state is reached for an application. The overhead of unnecessary adaptations can be very expensive depending on the adaptation costs of the application being adapted. Overestimating an expansion can also have negative effects, since the efficient maximum allocation of an application may be exceeded. Modifying the resources of an application to an amount that lowers its parallel efficiency should be avoided.

Applications with a lot of available parallelism and that can adapt to arbitrary numbers of resources are the most beneficial to system-wide efficiency metrics, such as node utilization. Even better are applications that can also utilize all the CPU cores available in the nodes of the system.

Scheduling trade-offs can be configured through the adjustment of the MTCT thresholds of the presented SPMD-Phase model. Node utilization potential can be traded for better estimated parallel efficiency and lower job wait times.

The current defaults favor job starts over expansions. Favoring application starts can be negative when queues are low or empty. It may be beneficial to run at lower estimated efficiency levels and minimize idle node counts in these cases. Alternatively, idle nodes may be turned off; this is indeed supported already by the design inherited from SLURM. This will increase job start latencies when nodes that are turned off need to be booted.

15 Conclusion

Support for resource-elasticity was successfully demonstrated on the SuperMUC HPC system with the software prototype presented in this document. The prototype is composed of two main software components: an MPI library and a resource manager.

The MPI library was extended with a set of new operations that improve the support for dynamic processes in MPI. These new operations allow for initialization, probing for adaptation instructions and the creation of adaptation windows. The needs and goals of application, MPI library and resource manager developers were considered in their design.

The creation of resource-elastic applications is more flexible with the use of adaptation windows, when compared to using the current standard spawn operations of MPI. MPI application developers can insert adaptation windows in locations that can be reached periodically. Applications can support resource adaptations of arbitrary sizes with these extensions. Resources are abstracted as MPI processes in the world communicators of applications.

Preexisting MPI applications can be converted to support resource-elasticity. Conversions require the insertion of control flow statements to differentiate between processes that are part of a normal application launch and processes that were created by the resource manager as part of an expansion of resources. All processes must meet at the correct location where the adaptation window begins.

The performance of the new dynamic processes operations in the proposed MPI extension was evaluated. The performance of the initialization operation is identical to the standard one. The probe operation is very fast for the common case where no adaptations need to be made, while performing well when adaptation instructions are received. The highlight of the implementation is the adaptation window. The split design with a begin and a commit operation allowed for the demonstrated latency hiding design. This was verified by measurements at the begin operation of preexisting processes. The commit operation that is used to complete adaptation windows was shown to be very fast when compared to the begin operation. In summary, the latency hiding properties and general performance goals of the design were achieved.

The MPI library and the resource manager prototypes are well integrated. The resource adaptations are initiated by the resource manager, and not the application. This inversion of control, when compared to standard spawn operations, allows schedulers to optimize for both application and system-wide efficiency metrics. The resource manager has additional features for the gathering of performance data through continuous interaction with MPI processes.

A CFG detection algorithm was implemented without the need of backtracing, in the MPI library. These CFGs are detected at each process and shared with the local resource manager daemons at compute nodes. These are eventually transfered to the scheduler running at a remote node through the TBON and the `SRUN` instance of applications. The overhead was shown to depend on the length of the CFG of applications. Because most applications produce CFGs that are in the order of hundreds of elements and the detection does not rely on backtracing, the overhead of detection was kept in the order of nanoseconds in most cases. The library switches to a verification only mechanism when a partial CFG remains stable. The overhead of verification cannot be measured even on single byte MPI messages with latencies in the order of microseconds.

A performance model is produced at the scheduler for each MPI application with its CFG and performance data. Currently only one model is implemented: the SPMD-Phase model. This model relies on the detection of distributed loops. It provides the scheduler with average and trend MPI to Compute Time (MTCT) ratios. These ratios are then used to generate Resource Range Vectors (RRV) for sets of applications that are candidates for resource adaptations.

The generation of these range vectors can be influenced by the setting of two parameters in the heuristic: the upper and lower MTCT thresholds. The modification of these thresholds have multiple effects on the quality of the schedules produced. A wide margin between them prevents potential resource adaptation oscillations. These parameters can also be calibrated for different trade-offs between estimated efficiencies and idle node counts. This was demonstrated in the evaluation.

An experimental scheduler was also presented. This scheduler has a split design, composed of two separate schedulers with a clear separation of concerns: the Elastic Batch Scheduler (EBS) and the Elastic Runtime Scheduler (ERS). Unfortunately, the EBS was not implemented in time to be demonstrated together with the ERS in this document. The ERS was described in detail, and its interaction with the eventual EBS discussed. The ERS produces a Resource Scaling Vector (RSV) from the RRV at each scheduling interval. This vector contains the final resource count for the allocation of each application in the list of candidates for resource adaptations. Although incomplete tests, given the missing EBS, the evaluation of the ERS with two test applications illustrated the benefits of resource-elasticity for HPC systems: it can reduce the makespan of schedules, wait times of jobs and idle node counts. These are only initial results and further analyses are needed.

The importance of integrating resource managers and programming models for resource-elasticity support was illustrated. The message passing model was integrated through the extension to MPI. An additional model that targets resource-elastic execution was presented: Elastic-Phase Oriented Programming. This model provided important abstractions that further simplified the development of resource-elastic applications, but also its integration with resource managers and performance modeling techniques.

It is expected that the integration of programming models and resource managers will increase in importance as exascale levels of performance are reached in HPC systems. Programming models that support resource-elastic execution and bring computational and energy efficiency benefits, while at the same time allowing for fault-tolerance, are expected to increase in importance in the near future.

In summary, the research presented in this document is related to multiple areas of computer science: programming models, resource management, performance modeling and scheduling. A prototype that is a combination of a communication library and a resource manager was presented and evaluated. The prototype currently supports SPMD type MPI applications with resource-elasticity. Its scheduling heuristic can provide system-wide and individual application parallel efficiency improvements, in some cases. The results presented in the evaluation are limited, but also promising given the early stage of development of the prototype.

16 Future Work

A large amount of research on resource-elasticity in HPC is still left to be done. The research and prototypes presented in this work are only the beginning. In this chapter, an incomplete discussion on future research opportunities related to message passing, the missing Elastic Batch Scheduler (EBS), and finally resource management, is presented.

16.1 Elastic Message Passing

More applications need to be developed with support for resource-elasticity. The benefits of resource-elasticity are only possible when sufficient elastic jobs are submitted to compute systems. Options should be explored for the simplification of the conversion of existing codes to resource-elasticity, with the elastic MPI library or the EPOP model.

To support exascale, fault tolerance needs to be an important part of future development efforts. Several options are being evaluated at the Fault Tolerance Working Group of the MPI forum [140, 45, 47, 40]. These research efforts should be followed closely. Fault-tolerance should be added as soon as possible to potential future resource-elastic software stacks such as the one presented in this work.

Automatic tuning needs to be added to the elastic MPI library. MPI has many configuration parameters that have a performance impact. These parameters tend to be set when the application initializes the MPI library, and remain the same through its execution. Because of the expected changes in resource allocations and the number of processes in MPI applications due to resource-elastic behavior, the MPI library should update these parameters periodically, or at least once per resource adaptation.

A more sophisticated implementation of the EPOP model may be worth developing. Its current implementation is a minimalistic C library with a single driver program. While this is sufficient to illustrate the benefits of the model, a more elaborate solution may be better for the development and performance of EPOP applications. For example, a new programming language or an extension to an existing one may simplify the development of applications. Additionally, there may be optimizations possible to the patterns of EPOP programs based on a global view of their structure. Driver programs for automatic tuning and other purposes can also be added.

The current insertion of markers for automatic pattern detection can be improved. The insertion of markers can be made better through the use of compilers, such as Clang [4] from the LLVM [17] project. This would allow for more data to be included with the markers, such as the location of loops or branches. The detection algorithms could be simplified as a consequence.

Elastic programming models may be split into shared memory and distributed memory techniques. In this work, only distributed memory techniques were considered. A combination with shared memory adaptation and load balancing techniques can prove beneficial. These techniques can be treated as orthogonal, and later integrated in a complete software stack for distributed memory systems.

Integration with external visualization (such as Vampir [20]), performance modeling and reporting tools (such as Caliper [2]) will also be of great importance in the future. Visualization techniques that aid the understanding of the resource-elastic executions of these distributed applications should be developed. The dynamic changes in resources make the understanding of these applications even more challenging than typical distributed memory applications.

The extension to MPI should be adapted based on the new developments of the future MPI 4.0 standard. Any changes to the way the communicators are handled, such as the sessions [18] proposal, should be considered in updates to the proposed adaptation window creation operations.

16.2 Elastic Batch Scheduler (EBS)

The immediate next development goal is to implement and integrate the Elastic Batch Scheduler (EBS) into the infrastructure. The necessary operations to support efficient moldable job starts are already available in the Elastic Runtime Scheduler (ERS). Given the current design, the batch scheduler is expected to interact with the ERS on each situation that results in the availability of nodes or more jobs, such as:

- Typical job completion or termination events.
- Mandatory or optional reductions determined based on the performance model.
- New job arrivals at the queue of the Elastic Batch Scheduler (EBS) from users.

A moldable batch scheduling technique is to be developed, where the starting node allocation of jobs is based on a range instead of the fixed amounts found in current systems. The technique should preserve the FCFS policy by attempting to start early jobs first. In cases where only lower priority jobs can be started due to resource availability, higher priority jobs should still be forwarded to the Elastic Runtime Scheduler (ERS). The ERS should be then forced to provide reservations for these higher priority jobs and minimize their waiting times, with elastic resource transformations such as time balancing.

16.3 Elastic Resource Management

The creation of a more flexible infrastructure for resource management may be the best long term option. New resource management research such as that of the Flux [31] project may benefit resource-elasticity greatly. A more scalable, modular and hierarchical approach to resource management may be necessary to better support resource-elasticity at exascale.

Better performance models and analysis techniques can be added that produce the necessary ranges used as input by the Elastic Runtime Scheduler (ERS). These could consider additional performance metrics, progress reports and adaptation measurements. Since the design is modular, multiple performance models may be implemented in the future.

Energy optimization will also be of greater importance at exascale. Energy metrics need to be measured by the infrastructure. New performance models that consider energy metrics need to be added. Multi-objective optimizations that optimize performance and energy need to be developed in the future. Additionally, strategies for system-wide power-level stabilization and power capping will be required.

Machine learning and other history based techniques may be added to the scheduler and its performance model. These techniques have great potential in optimization problems such as scheduling. Given the highly dynamic nature of the presented system and the high cost of adaptations due to distributed memory, any technique that improves the quality of predictions is of great importance.

The pattern detection technique should continue to be improved. Currently it is effective at the detection of SPMD patterns, but it should be extended to handle other patterns as well. Master-worker patterns can be supported by representing them as separate SPMD blocks that are coupled. Additionally, support for arbitrary MPMD patterns is desirable.

Resource management may be performed on intra-node resources, such as cores and memory. Memory should be tracked as a resource by the scheduler and the resource manager. If memory usage is known, then the minimal number of nodes specified by users can be taken as a hint, instead of as a fixed constraint. This can further benefit the minimization of idle node counts by the scheduler.

Topology optimizations should be explored in the future. For this, first the `SRUN` program should be extended with a migration function. After that, full migrations from fragmented allocations to dense allocations can be performed on jobs so that their network performance is efficient. The MPI library may be extended with communication pattern detection mechanisms to support this.

The quality of the time balancing operation in the runtime scheduler depends on the accuracy of the remaining time estimation of the job. This estimation is currently provided by users and is very unreliable. New modeling techniques that predict the remaining run times of jobs should be developed. Additionally, progress reporting APIs can be added to allow applications to report their progress and an estimation of their remaining time. The EPOP model can include a way to report the current iteration number and bound of the loops in Elastic-Phases (EPs).

Bibliography

[1] Standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pages 1–3957, Sept 2016.

[2] Caliper: Application Introspection System. `http://computation.llnl.gov/projects/caliper`, 2017. [Online].

[3] Charm++: Parallel Programming with Migratable Objects. `http://charm.cs.illinois.edu/research/charm`, 2017. [Online].

[4] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`, 2017. [Online].

[5] GNU Hurd. `https://www.gnu.org/software/hurd/hurd.html`, 2017. [Online].

[6] MPICH: High-Performance Portable MPI. `http://www.mpich.org`, 2017. [Online].

[7] Open MPI: Open Source High Performance Computing. `https://www.open-mpi.org/`, 2017. [Online].

[8] OpenFabrics Alliance. `http://openfabrics.org/`, 2017. [Online].

[9] OpenMP: An API for multi-platform shared-memory parallel programming in C/C++ and Fortran. `http://www.openmp.org`, 2017. [Online].

[10] Parallel Virtual Machine (PVM). `http://www.csm.ornl.gov/pvm/`, 2017. [Online].

[11] SchedMD. `http://www.schedmd.com/`, 2017. [Online].

[12] Simple Linux Utility For Resource Management. `http://slurm.schedmd.com/`, 2017. [Online].

[13] SuperMUC Petascale System. `https://www.lrz.de/services/compute/supermuc/`, 2017. [Online].

[14] The Barrelfish Operating System. `http://www.barrelfish.org/`, 2017. [Online].

[15] The FreeBSD Project. `http://www.freebsd.org`, 2017. [Online].

[16] The Linux Kernel. `http://www.linux.org/`, 2017. [Online].

[17] The LLVM Compiler Infrastructure. `http://llvm.org/`, 2017. [Online].

[18] The MPI 4.0 standardization efforts. `http://mpi-forum.org/mpi-40/`, 2017. [Online].

[19] Top 500 Supercomputers. `http://www.top500.org/`, 2017. [Online].

[20] Vampir - Performance Optimization. `https://www.vampir.eu/`, 2017. [Online].

[21] X10: Performance and Productivity at Scale. `http://x10-lang.org/`, 2017. [Online].

[22] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

[23] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kalé. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.

[24] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10–20, April 2006.

[25] A. M. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*, pages 167–176, 1999.

[26] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 100–109, New York, NY, USA, 2006. ACM.

[27] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. MPI trace compression using event flow graphs. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 1–12. Springer International Publishing, 2014.

[28] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. Automatic on-line detection of MPI application structure with event flow graphs. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 70–81, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[29] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. Visual MPI performance analysis using event flow graphs. *Procedia Computer Science*, 51:1353 – 1362, 2015.

[30] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. Event flow graphs for MPI performance monitoring and analysis. In *Tools for High Performance Computing 2015:*

Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany, pages 103–115, Cham, 2016. Springer International Publishing.

[31] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. Flux: A next-generation resource management framework for large HPC centers. In *10th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*. IEEE Computer Society, 2014.

[32] Michail Alvanos, Montse Farreras, Ettore Tiotto, and Xavier Martorell. Automatic communication coalescing for irregular computations in UPC language. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 220–234, Riverton, NJ, USA, 2012. IBM Corp.

[33] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[34] Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde. A case study of energy aware scheduling on SuperMUC. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*, pages 394–409, Cham, 2014. Springer International Publishing.

[35] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[36] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. PMI: A scalable parallel process-management interface for extreme-scale systems. In *Recent Advances in the Message Passing Interface: 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*, pages 31–41, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[37] Richard F Barrett, Philip C Roth, and Stephen W Poole. Finite difference stencils implemented using Chapel. *Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122*, 2007.

[38] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, and Vaidy Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In *Software for Parallel Computation*, pages 91–99, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[39] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, April 2003.

[40] Maciej Besta and Torsten Hoefler. Fault tolerance for remote memory access programming models. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 37–48. ACM, 2014.

[41] Abhay Bhadani and Sanjay Chaudhary. Performance evaluation of web servers using central load balancing policy over virtual machines on cloud. In *Proceedings of the Third Annual ACM Bangalore Conference*, COMPUTE '10, pages 16:1–16:4, New York, NY, USA, 2010. ACM.

[42] Milind Bhandarkar, L. V. Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *Computational Science - ICCS 2001: International Conference San Francisco, CA, USA, May 28—30, 2001 Proceedings, Part II*, pages 108–117, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[43] Abhinav Bhatel e, Eric Bohm, and Laxmikant V. Kalé. Optimizing communication for Charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 23(2):211–222, 2011.

[44] A. Bhatele and L. V. Kalé. Application-specific topology-aware mapping for three dimensional topologies. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

[45] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in MPI. In *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, pages 193–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings*, pages 191–197, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[47] Aurelien Bouteiller, George Bosilca, and Jack J Dongarra. Plan b: Interruption of ongoing MPI operations to support failure recovery. In *Proceedings of the 22nd European MPI Users' Group Meeting*, page 11. ACM, 2015.

[48] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers and Fluids*, 80:372 – 380, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[49] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Dynamic X10: Resource-aware programming for higher efficiency. Technical Report 8, Karlsruhe Institute of Technology, 2014. X10 '14.

[50] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *2007 IEEE International Conference on Cluster Computing*, pages 372–381, Sept 2007.

[51] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. Invasive computing in HPC with X10. In *Proceedings of the Third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19, New York, NY, USA, 2013. ACM.

[52] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10 pp.–530, May 2006.

[53] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634 – 644, 2007. Selected Papers from EuroPVM/MPI 2006.

[54] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[55] T. Cao, Y. He, and M. Kondo. Demand-aware power management for power-constrained HPC systems. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 21–31, May 2016.

[56] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12, Nov 2000.

[57] T. E. Carroll and D. Grosu. Incentive compatible online scheduling of malleable parallel jobs with individual deadlines. In *2010 39th International Conference on Parallel Processing*, pages 516–524, Sept 2010.

[58] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Feb 1988.

[59] Márcia C. Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*, ICDCN'10, pages 242–257, Berlin, Heidelberg, 2010. Springer-Verlag.

[60] S. Chakraborty, H. Subramoni, A. Moody, A. Venkatesh, J. Perkins, and D. K. Panda. Non-blocking PMI extensions for fast MPI startup. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 131–140, May 2015.

[61] S. Chakraborty, H. Subramoni, J. Perkins, A. Moody, M. Arnold, and D. K. Panda. PMI extensions for scalable MPI startup. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 21:21–21:26, New York, NY, USA, 2014. ACM.

[62] Sayantan Chakravorty, Celso L. Mendes, and Laxmikant V. Kalé. Proactive fault tolerance in MPI applications via task migration. In *High Performance Computing - HiPC 2006: 13th International Conference, Bangalore, India, December 18-21, 2006. Proceedings*, pages 485–496, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[63] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[64] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in Chapel: Philosophy and framework. In *Proceedings of the second USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[65] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[66] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[67] I-Hsin Chung, Che-Rung Lee, Jiazheng Zhou, and Yeh-Ching Chung. Hierarchical mapping for HPC applications. *Parallel Processing Letters*, 21(03):279–299, 2011.

[68] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science Engineering*, 11(4):29–41, July 2009.

[69] Isaías A. Comprés Ureña, Michael Gerndt, and Carsten Trinitis. Wait-free message passing protocol for non-coherent shared memory architectures. In *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, pages 142–152, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[70] Isaías A. Comprés Ureña, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. Infrastructure and API extensions for elastic execution of MPI applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 82–97, New York, NY, USA, 2016. ACM.

[71] Isaías A. Comprés Ureña, Michael Riepen, Michael Konow, and Michael Gerndt. RCKMPI - lightweight MPI implementation for Intel's single-chip cloud computer (SCC). In *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011.

[72] Isaías A. Comprés Ureña, Michael Riepen, Michael Konow, and Michael Gerndt. Invasive MPI on Intel's Single-Chip Cloud Computer. In *Architecture of Computing Systems – ARCS 2012: 25th International Conference, Munich, Germany, February 28 - March 2, 2012. Proceedings*, pages 74–85, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[73] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008.

[74] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: Efficient failure-aware programming. *SIGPLAN Not.*, 49(8):67–80, February 2014.

[75] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, January 1991.

[76] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[77] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[78] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[79] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[80] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with MPI and Unified Parallel C. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 177–186, New York, NY, USA, 2010. ACM.

[81] S. G. Domanal and G. R. M. Reddy. Load balancing in cloud computingusing modified throttled algorithm. In *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–5, Oct 2013.

[82] S. G. Domanal and G. R. M. Reddy. Optimal load balancing in cloud computing by efficient utilization of virtual machines. In *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–4, Jan 2014.

[83] Richard A. Dutton and Weizhen Mao. Online scheduling of malleable parallel jobs. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '07, pages 136–141, Anaheim, CA, USA, 2007. ACTA Press.

[84] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[85] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An open-source compiler and runtime implementation for Coarray Fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[86] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *2008 IEEE Fourth International Conference on eScience*, pages 277–284, Dec 2008.

[87] Tarek El-Ghazawi and Lauren Smith. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[88] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Parallel job scheduling for power constrained HPC systems. *Parallel Computing*, 38(12):615 – 630, 2012.

[89] Yoav Etsion and Dan Tsafrir. A short survey of commercial cluster batch schedulers. *School of Computer Science and Engineering, The Hebrew University of Jerusalem*, 44221:2005–13, 2005.

[90] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling — a status report. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'04, pages 1–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[91] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS '97 Processing Workshop Geneva, Switzerland, April 5, 1997 Proceedings*, pages 1–34, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[92] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, March 2012.

[93] Lance Fortnow. The status of the P versus NP problem. *Commun. ACM*, 52(9):78–86, September 2009.

[94] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid MPI: Efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 18:1–18:11, New York, NY, USA, 2013. ACM.

[95] Karl Fürlinger and David Skinner. Capturing and visualizing event flow graphs of MPI applications. In *Euro-Par 2009 – Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25-28, 2009, Revised Selected Papers*, pages 218–227, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[96] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19 - 22, 2004. Proceedings*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[97] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 263–270, Nov 2012.

[98] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[99] GA Geist, James A Kohl, and Phil M Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[100] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. Invasive computing with iOMP. In *Proceeding of the 2012 Forum on Specification and Design Languages*, pages 225–231, Sept 2012.

[101] Neha Gholkar, Frank Mueller, and Barry Rountree. Power tuning HPC jobs on power-constrained systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 179–191, New York, NY, USA, 2016. ACM.

[102] César Gómez-Martín, Miguel A. Vega-Rodríguez, and José-Luis González-Sánchez. Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods. *Concurrency and Computation: Practice and Experience*, 27(17):5436–5459, 2015. cpe.3607.

[103] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, Sept 2006.

[104] Richard L. Graham, Brian W. Barrett, Galen M. Shipman, Timothy S. Woodall, and George Bosilca. Open MPI: A high performance, flexible implementation of MPI point-to-point communications. *Parallel Processing Letters*, 17(01):79–88, 2007.

[105] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics: 6th International Conference, PPAM 2005, Poznań, Poland, September 11-14, 2005, Revised Selected Papers*, pages 228–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[106] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[107] William Gropp. MPICH2: A new start for MPI implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[108] William Gropp and Ewing Lusk. Why are PVM and MPI so different? In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 4th European PVM/MPI Users' Group Meeting Cracow, Poland, November 3–5, 1997 Proceedings*, pages 1–10, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[109] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[110] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé. Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[111] Abhishek Gupta and Dejan Milojicic. Evaluation of HPC applications on cloud. In *Proceedings of the 2011 Sixth Open Cirrus Summit*, OCS '11, pages 22–26, Washington, DC, USA, 2011. IEEE Computer Society.

[112] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau. Resource-aware programming and simulation of MPSoC architectures through extension of X10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '11, pages 48–55, New York, NY, USA, 2011. ACM.

[113] Manuel Hasert, Harald Klimach, and Sabine Roller. CAF versus MPI - applicability of Coarray Fortran to a flow solver. In *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, pages 228–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[114] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, July 1997.

[115] Nathan Hjelm. Optimizing one-sided operations in Open MPI. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 123:123–123:124, New York, NY, USA, 2014. ACM.

[116] T. Hoefler and J. L. Traff. Sparse collective operations for MPI. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

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

[118] Andreas Hollmann and Michael Gerndt. Invasive computing: An application assisted resource management approach. In *Multicore Software Engineering, Performance, and Tools: International Conference, MSEPT 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, pages 82–85, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[119] J. Hu, J. Gu, G. Sun, and T. Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, pages 89–96, Dec 2010.

[120] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*, pages 306–322, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[121] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. Performance evaluation of adaptive MPI. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 12–21, New York, NY, USA, 2006. ACM.

[122] Chao Huang, Gengbin Zheng, and Laxmikant V Kalé. Supporting adaptivity in MPI for dynamic parallel applications. *Technical Report; Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign*, 2007.

[123] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213, Oct 2004.

[124] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra. Phase-based application-driven hierarchical power management on the single-chip cloud computer. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 131–142, Oct 2011.

[125] Satoshi Ito, Kazuya Goto, and Kenji Ono. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. *Computers and Fluids*, 80:88 – 93, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[126] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the Maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, pages 87–102, London, UK, UK, 2001. Springer-Verlag.

[127] N. Jain, A. Bhatele, J. S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kalé. Charm++ and MPI: Combining the best of both worlds. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 655–664, May 2015.

[128] E. Jeannot, G. Mercier, and F. Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, April 2014.

[129] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, pages 199–210, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[130] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and C. Yang. Implementation and performance evaluation of the HPC challenge benchmarks in Coarray Fortran 2.0. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 1089–1100, May 2011.

[131] L. V. Kalé, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 230–230, May 2002.

[132] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[133] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[134] C. Karlsson, T. Davies, and Z. Chen. Optimizing process-to-core mappings for application level multi-dimensional MPI communications. In *2012 IEEE International Conference on Cluster Computing*, pages 486–494, Sept 2012.

[135] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103, Boston, MA, 1972. Springer US.

[136] Peter J. Keleher, Dmitry Zotkin, and Dejan Perkovic. Attacking the bottlenecks of backfilling schedulers. *Cluster Computing*, 3(4):245–254, 2000.

[137] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[138] A. A. Khan, C. L. Mccreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 243–250, Aug 1994.

[139] Charles H Koelbel. *The high performance Fortran handbook*. MIT press, 1994.

[140] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in MPI applications. *International Journal of High Performance Computing Applications*, page 1094342015623623, 2016.

[141] Michael Lange, Gerard Gorman, Michèle Weiland, Lawrence Mitchell, and James Southern. Achieving efficient strong scaling with PETSc using hybrid MPI/OpenMP optimisation. In *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, pages 97–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[142] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. Chapter 9 sequencing and scheduling: Algorithms and complexity. In *Logistics of Production and Inventory*, volume 4 of *Handbooks in Operations Research and Management Science*, pages 445 – 522. Elsevier, 1993.

[143] Barry G. Lawson and Evgenia Smirni. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers*, pages 72–87, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[144] Barry G. Lawson, Evgenia Smirni, and Daniela Puiu. Self-adapting backfilling scheduling for parallel systems. In *Proceedings of the 2002 International Conference on Parallel Processing*, ICPP '02, pages 583–593, Washington, DC, USA, 2002. IEEE Computer Society.

[145] Inbok Lee, Costas S. Iliopoulos, and Kunsoo Park. Linear time algorithm for the longest common repeat problem. *Journal of Discrete Algorithms*, 5(2):243 – 249, 2007. 2004 Symposium on String Processing and Information Retrieval.

[146] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with MapReduce: A survey. *SIGMOD Rec.*, 40(4):11–20, January 2012.

[147] Charles E. Leiserson. Programming irregular parallel applications in Cilk. In *Solving Irregularly Structured Problems in Parallel: 4th International Symposium, IRREGULAR'97 Paderborn, Germany, June 12–13, 1997 Proceedings*, pages 61–71, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[148] Charles E. Leiserson. Cilk. In *Encyclopedia of Parallel Computing*, pages 273–288, Boston, MA, 2011. Springer US.

[149] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 343 – 362. Elsevier, 1977.

[150] D. Li, Y. Wang, and W. Zhu. Topology-aware process mapping on clusters featuring NUMA and hierarchical network. In *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, pages 74–81, June 2013.

[151] A. M. Lindsay, M. Galloway-Carson, C. R. Johnson, D. P. Bunde, and V. J. Leung. Backfilling with guarantees made as jobs arrive. *Concurrency and Computation: Practice and Experience*, 25(4):513–523, 2013.

[152] R. V. Lopes and D. Menascé. A taxonomy of job scheduling on distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3412–3428, Dec 2016.

[153] David B Loveman. High performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[154] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of Open MPI: Enabling third-party collective algorithms. In *Proceedings of the Workshop on Component Models and Systems for Grid Applications.*, pages 167–185, Boston, MA, 2005. Springer US.

[155] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra. Hierknem: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 970–982, May 2012.

[156] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra. Process distance-aware adaptive MPI collective communications. In *2011 IEEE International Conference on Cluster Computing*, pages 196–204, Sept 2011.

[157] T. Malik, V. Rychkov, A. Lastovetsky, and J. N. Quintin. Topology-aware optimization of communications for parallel matrix multiplication on hierarchical heterogeneous HPC platform. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 39–47, May 2014.

[158] Timothy G. Mattson. Programming environments for parallel and distributed computing: A comparison of P4, PVM, Linda, and Tcgmsg. *Int. J. High Perform. Comput. Appl.*, 9(2):138–161, June 1995.

[159] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 41–50, New York, NY, USA, 2012. ACM.

[160] J Mellor-Crummey, L Adhianto, and WN Scherer III. A critique of co-array features in Fortran 2008. *Fortran Standards Technical Committee Document J*, 3:08–126, 2008.

[161] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A new vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[162] Guillaume Mercier and Emmanuel Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, pages 39–49, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[163] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a parallel language for scientific computation: Practice and experience. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 1080–1088, May 2011.

[164] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. Cutting out the middleman: OS-level support for X10 activities. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10 2015, pages 13–18, New York, NY, USA, 2015. ACM.

[165] Gregory Mounie, Christophe Rapine, and Dennis Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '99, pages 23–32, New York, NY, USA, 1999. ACM.

[166] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, Jun 2001.

[167] PVR Murthy. Parallel computing with X10. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, pages 5–6, New York, NY, USA, 2008. ACM.

[168] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. Automatic MPI to AMPI program transformation using Photran. In *Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31–September 3, 2010, Revised Selected Papers*, pages 531–539, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[169] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[170] J. L. Overbey, S. Negara, and R. E. Johnson. Refactoring and the evolution of Fortran. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 28–34, May 2009.

[171] Antonio J. Peña, Ralf G. Correa Carvalho, James Dinan, Pavan Balaji, Rajeev Thakur, and William Gropp. Analysis of topology-dependent MPI performance on Gemini networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 61–66, New York, NY, USA, 2013. ACM.

[172] Frederic Petrot and Pascal Gomez. Lightweight implementation of the POSIX threads API for an on-chip MIPS multiprocessor with VCI interconnect. In *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2*, DATE '03, Washington, DC, USA, 2003. IEEE Computer Society.

[173] W. Pfeiffer and A. Stamatakis. Hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[174] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. A. Navaux, F. Broquedis, J. F. Méhaut, and L. V. Kalé. A hierarchical approach for load balancing on parallel multi-core systems. In *2012 41st International Conference on Parallel Processing*, pages 118–127, Sept 2012.

[175] Alexander Pöppl and Michael Bader. SWE-X10: An actor-based and locally coordinated solver for the shallow water equations. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 30–31, New York, NY, USA, 2016. ACM.

[176] Alexander Pöppl, Michael Bader, Tobias Schwarzer, and Michael Glaß. SWE-X10: Simulating shallow water waves with lazy activation of patches using ActorX10. In *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ESPM2, pages 32–39, Piscataway, NJ, USA, 2016. IEEE Press.

[177] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf. A batch system with fair scheduling for evolving applications. In *2014 43rd International Conference on Parallel Processing*, pages 351–360, Sept 2014.

[178] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kalé. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 429–438, May 2015.

[179] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009.

[180] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, March 1999.

[181] M. Randles, D. Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 551–556, April 2010.

[182] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. Multi-core and network aware MPI topology functions. In *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, pages 50–60, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[183] A. Raveendran, T. Bicer, and G. Agrawal. A framework for elastic execution of existing MPI programs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 940–947, May 2011.

[184] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[185] Sascha Roloff, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich. ActorX10: An actor library for X10. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 24–29, New York, NY, USA, 2016. ACM.

[186] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings - Computers and Digital Techniques*, 141(1):1–10, Jan 1994.

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

[188] N. Sadashiv and S. M. D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *2011 6th International Conference on Computer Science Education (ICCSE)*, pages 477–482, Aug 2011.

[189] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: Concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 271–271, New York, NY, USA, 2007. ACM.

[190] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kalé. Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 807–818, Piscataway, NJ, USA, 2014. IEEE Press.

[191] M. Sato. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *15th International Symposium on System Synthesis, 2002.*, pages 109–111, Oct 2002.

[192] William N. Scherer, III, Laksono Adhianto, Guohua Jin, John Mellor-Crummey, and Chaoran Yang. Hiding latency in Coarray Fortran 2.0. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 14:1–14:9, New York, NY, USA, 2010. ACM.

[193] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing:*

9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper, pages 228–251, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[194] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua. Performance portability with the Chapel language. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 582–594, May 2012.

[195] Anna Sikora, Eduardo César, Isaías A. Comprés Ureña, and Michael Gerndt. Autotuning of MPI Applications Using PTF. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, pages 31–38, New York, NY, USA, 2016. ACM.

[196] Edgar Solomonik, Abhinav Bhatele, and James Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 77:1–77:11, New York, NY, USA, 2011. ACM.

[197] S. Spetka, H. Hadzimujic, S. Peek, and C. Flynn. High productivity languages for parallel programming compared to MPI. In *2008 DoD HPCMP Users Group Conference*, pages 413–417, July 2008.

[198] Jaidev K. Sridhar and Dhabaleswar K. Panda. Impact of node level caching in MPI job launch mechanisms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*, pages 230–239, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[199] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings. International Conference on Parallel Processing Workshop*, pages 514–519, 2002.

[200] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. Selective reservation strategies for backfill job scheduling. In *Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers*, pages 55–71, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[201] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold, and D. K. Panda. Design of network topology aware scheduling services for large infiniband clusters. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Sept 2013.

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

[203] H. Sun, Y. Cao, and W. J. Hsu. Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):594–607, April 2011.

[204] H. Sun, Y. Cao, and W. J. Hsu. Fair and efficient online adaptive scheduling for multiple sets of parallel applications. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 64–71, Dec 2011.

[205] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, November 1990.

[206] D. Talby and D. G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 513–517, Apr 1999.

[207] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and APGAS at petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, New York, NY, USA, 2014. ACM.

[208] Robert Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 96–107, New York, NY, USA, 1973. ACM.

[209] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[210] S. Tham and J. Morris. Cilk vs MPI: comparing two very different parallel programming styles. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 143–152, Oct 2003.

[211] J. L. Traff. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28, Nov 2002.

[212] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, June 2007.

[213] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[214] J.D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[215] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on MPI jobs. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 215–224, Sept 2004.

[216] SATHISH S. VADHIYAR and JACK J. DONGARRA. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.

[217] Sathish S. Vadhiyar and Jack J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[218] G Matthijs van Waveren. High performance Fortran. In *International Conference on High-Performance Computing and Networking*, pages 429–433. Springer, 1994.

[219] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[220] Sebastian von Alfthan, Ilja Honkonen, and Minna Palmroth. Topology aware process mapping. In *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*, pages 297–308, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[221] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 170–183, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[222] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pages 13–22, Feb 2003.

[223] Gosia Wrzesinska, Jason Maassen, and Henri E. Bal. Self-adaptive applications on the grid. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 121–129, New York, NY, USA, 2007. ACM.

[224] Chaoran Yang, Wesley Bland, John Mellor-Crummey, and Pavan Balaji. Portable, MPI-interoperable Coarray Fortran. *SIGPLAN Not.*, 49(8):81–92, February 2014.

[225] Xu Yang, Zhou Zhou, Sean Wallace, Zhiling Lan, Wei Tang, Susan Coghlan, and Michael E. Papka. Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 60:1–60:11, New York, NY, USA, 2013. ACM.

[226] H. Yu, I. h. Chung, and J. Moreira. Topology mapping for blue Gene/L supercomputer. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 52–52, Nov 2006.

[227] Y. Yuan, Y. Wu, W. Zheng, and K. Li. Guarantee strict fairness and utilization prediction better in parallel job scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):971–981, April 2014.

[228] Yan Zhai, Mingliang Liu, Jidong Zhai, Xiaosong Ma, and Wenguang Chen. Cloud versus in-house cluster: Evaluating amazon cluster compute instances for running MPI applications. In *State of the Practice Reports*, SC '11, pages 11:1–11:10, New York, NY, USA, 2011. ACM.

[229] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kalé. Hierarchical load balancing for Charm++ applications on large supercomputers. In *2010 39th International Conference on Parallel Processing Workshops*, pages 436–444, Sept 2010.

[230] G. Zheng, Xiang Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.

[231] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.

[232] Gengbin Zheng, Lixia Shi, and L. V. Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 93–103, Sept 2004.

[233] D. Zotkin and P. J. Keleher. Job-length estimation and performance in backfilling schedulers. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*, pages 236–243, 1999.