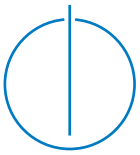
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Job Scheduling for Adaptive Applications in Future HPC Systems

Nishanth Nagendra



DEPARTMENT OF INFORMATICS

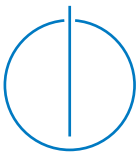
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Job Scheduling for Adaptive Applications in Future HPC Systems

Job Scheduling für Adaptive Anwendungen auf Zukünftigen HPC Systemen

Author:	Nishanth Nagendra
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Isaias Alberto Compres Urena
Submission Date:	Jul 15, 2015



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Jul 15, 2015 Nishanth Nagendra

Acknowledgments

Abstract

Invasive Computing is a novel paradigm for the design and resource-aware programming of future parallel computing systems. It enables the programmer to write resource aware programs and the goal is to optimize the program for the available resources. Traditionally, parallel applications implemented using MPI are submitted with a fixed number of MPI processes to execute on a HPC(High Performance Computing) system. This results in a fixed allocation of resources for the job. Modern techniques in scientific computing such as AMR(Adaptive Mesh Refinement) result in applications exhibiting complex behaviors where their resource requirements change during execution. Invasive MPI is an ongoing research effort to provide MPI extensions for the development of Invasive MPI applications that will result in jobs which are resource-aware for the HPC systems and can utilize such AMR techniques. Unfortunately, using only static allocations result in these applications being forced to execute using their maximum resource requirements that may lead to inefficient resource utilization. In order to support such kind of parallel applications at HPC centers, there is an urgent need to investigate and implement extensions to existing resource management systems or develop a new system. This thesis will extend the work done over the last few months during which an early prototype was implemented by developing a protocol for the intgeration of invasive resource management into existing batch systems. Specifically, This thesis will now investigate and implement a job scheduling algorithm in accordance with the new protocol developed earlier for supporting such an invasive resource management.

Contents

1 Introduction

Over the last two decades, the landscape of Computer Architecture has changed radically from sequential to parallel . Due to the limiting factors of technology we have moved from single core processors to multi core processors having a network interconnecting them. Traditionally, the approach of designing algorithms has been sequential, but designing algorithms in parallel is gaining more importance now to better utilize the computing power available at our disposal. Another important trend that has changed the face of computing is an enormous increase in the capabilities of the networks that connect computers with regards to speed, reliability etc. These trends make it feasible to develop applications that use physically distributed resources as if they were part of the same computer. A typical application of this sort may utilize processors on multiple remote computers, access a selection of remote databases, perform rendering on one or more graphics computers, and provide real-time output and control on a workstation. Computing on networked computers ("Distributed Computing") is not just a subfield of parallel computing as the basic task of developing programs that can run on many computers at once is a parallel computing problem. In this respect, the previously distinct worlds of parallel and distributed computing are converging.

As technology advances, we have newer problems or applications that demand larger computing capabilities which push the limits of technology giving rise to newer advancements. The performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently. A metric used to quantify the performance of a computer is FLOPS (floating point operations per second). The time to perform a basic operation is ultimately limited by the "clock cycle" of the processor, that is, the time required to perform the most primitive operation. The term ***High Performance Computing (HPC)*** refers to the practice of aggregating computing power (multiple nodes with processing units interconnected by a network in a certain toplogy) or the use of parallel processing for running advanced application programs efficiently, reliably and quickly. The term applies especially to systems that function above a ***teraflop*** or 10^{12} floating-point operations per second. The term HPC is occasionally used as a synonym for Supercomputer that works at more than a ***petaflop*** or 10^{15} floating-point operations per second. The most common users of HPC systems are scientific researchers, engineers, government

agencies including the military, and academic institutions. In general, HPC systems can refer to Clusters, Supercomputers, Grid Computing etc. and they are usually used for running complex applications.

A ***Batch System*** is used to manage the resources in a HPC System. It is a middleware that comprises of two major components namely the ***Resource Manager*** and ***Scheduler***. The role of a Resource Manager is to act like a glue for a parallel computer to execute parallel jobs. It should make a parallel computer as easy to use as a Personal Computer (PC). A programming model such as ***Message Passing Interface (MPI)*** for programming on distributed memory systems would typically be used to manage communications within a parallel program by using the MPI library functions. A Resource Manager allocates resources within a HPC system, launches and otherwise manages Jobs. Some of the examples of widely used open source as well as commercial resource managers are ***SLURM, TORQUE, OMEGA, IBM Platform LSF*** etc. Together with a scheduler it is termed as a Batch System. The role of a job scheduler is to manage queue(s) of work when there is more work than resources. It supports complex scheduling algorithms which are optimized for network topology, energy efficiency, fair share scheduling, advanced reservations, preemption, gang scheduling (time-slicing jobs) etc. It also supports resource limits (by queue, user, group, etc.). Many batch systems provide both resource management and job scheduling within a single product (e.g. LSF) while others use distinct products(e.g. Torque Resource Manager and Moab Job Scheduler). Some other examples of Job Scheduling Systems are ***LoadLeveler, OAR, Maui, SLURM etc.***

Existing Batch Systems usually support only static allocation of resources to an application before they start which means the resources once allocated are fixed for the lifetime of the application. The complexity of applications have been growing, However, especially when we consider advanced techniques in Scientific Computing like ***Adaptive Mesh Refinement (AMR)*** where applications exhibit complex behavior by changing their resource requirements during execution. The Batch Systems of today are not equipped to deal with such kind of complex applications in an intelligent manner apart from giving them the maximum number of resources before it starts that will result in a sheer wastage of resources leading to a poor resource utilization. In order to support such adaptive applications at HPC centers there is an urgent need to investigate and implement extensions to existing resource management systems or develop an entirely new system. These supporting infrastructures must be able to handle the new kind of applications and the legacy ones intelligently keeping in mind that they should now be able to achieve much higher system utilization, throughput, energy efficiency etc. compared to their predecessors due to the elasticity of the applications.

1.1 Invasive Computing

The throughput of HPC Systems depends not only on efficient job scheduling but also on the type of jobs forming the workload. As defined by Feitelson, and Rudolph, Jobs can be classified into four categories based on their flexibility:

- ***Rigid Job:*** Requires a fixed number of resources throughout its execution.
- ***Moldable Job:*** The resource requirement of the job can be molded or modified by the batch system before starting the job(e.g. to effectively fit alongside other rigid jobs). Once started its resource set cannot be changed anymore.
- ***Evolving Job:*** These kind of jobs request for resource expansion or shrinkage during their execution. Applications that use Multi-Scale Analysis or Adaptive Mesh Refinement (AMR) exhibit this kind of behavior typically due to unexpected increases in computations or having reached hardware limits (e.g. memory) on a node.
- ***Malleable Job:*** The expansion and shrinkage of resources are initiated by the batch system in contrast to the evolving jobs. The application adapts itself to the changing resource set.

The first two types fall into the category of what is called as the static allocation since the allocation of rigid and moldable jobs must be finalized before the job starts. Whereas, the last two types fall under the category of dynamic allocation since this property of expanding or shrinking evolving and malleable jobs (together termed adaptive jobs) happens at runtime. Adaptive Jobs hold a strong potential to obtain high system performance. Batch systems can substantially improve the system utilization, throughput and response times with efficient shrink/expand strategies for running jobs that are adaptive. Similarly, applications also profit when expanded with additional resources as this can increase application speedup and improve load balance across the job's resource set.

Invasive Computing is a novel paradigm for the design and resource-aware programming of future parallel computing systems. It enables the programmer to write efficient resource aware programs. This approach can be used to allocate, execute on and free resources during execution of the program. The result is an adaptive application which can expand and shrink in the number of its resources at runtime. HPC infrastructures like clusters, supercomputers execute a vast variety of jobs, majority of which are parallel applications. These centers use intelligent resource management systems that should not only perform tasks of job management, resource management and scheduling but also satisfy important metrics like higher system utilization, job throughput

and responsiveness. Traditionally, MPI applications are executed with a fixed number of MPI processes but with Invasive MPI they can evolve dynamically at runtime in the number of their MPI processes. This in turn supports advanced techniques like AMR where the working set size of applications change at runtime. Such kind of adaptive programming paradigms need to be complemented with intelligent resource management systems that can achieve much higher system utilization, energy efficiency, throughput etc. compared to their predecessors due to elasticity of the applications.

Under the collaborative research project funded by the **German Research Foundation (DFG)** in the **Transregional Collaborative Research Centre 89 (TRR89)**, research efforts are being made to investigate this Invasive Computing approach at different levels of abstraction right from the hardware up to the programming model and its applications. **Invasive MPI** is an effort towards invasive programming with MPI where the application programmer has MPI extensions available for specifying at certain safe points in the program, the possibility of a changing the resource set of the application during runtime.

1.2 Dynamic Resource Management

Two of the most widely used resource managers on HPC systems are **SLURM** and **TORQUE**. The two major components in general of any sophisticated resource manager are the batch scheduler and the process manager. The Process Manager is responsible for launching the jobs on the allocated resources and managing them throughout their lifetime. Examples of process manager are ***Hydra, SLURM Daemon (slurmd)*** etc. The process managers interact with the processes of a parallel application via the **Process Management Interface (PMI)**. In order to support Invasive Resource Management, The following components will be implemented: ***iScheduler*** (Batch Scheduler for Invasic Jobs) built as an extension into an existing batch system and ***iDRScheduler*** (Invasive Distributed Run TIme Scheduler) similar to a controller daemon which will sit between the batch scheduler and the process manager. **SLURM** is the choice of an existing batch system on which this prototype will be implemented for demonstrating Invasive Computing and 3.2 shows a high level illustration of the architecture for such an Invasive Resource Management.

The above figure illustrates the proposed invasive resource management architecture. In addition to a job queue for legacy static jobs, we now have an additional job queue for invasic jobs. The existing batch scheduler needs to be extended in order to schedule

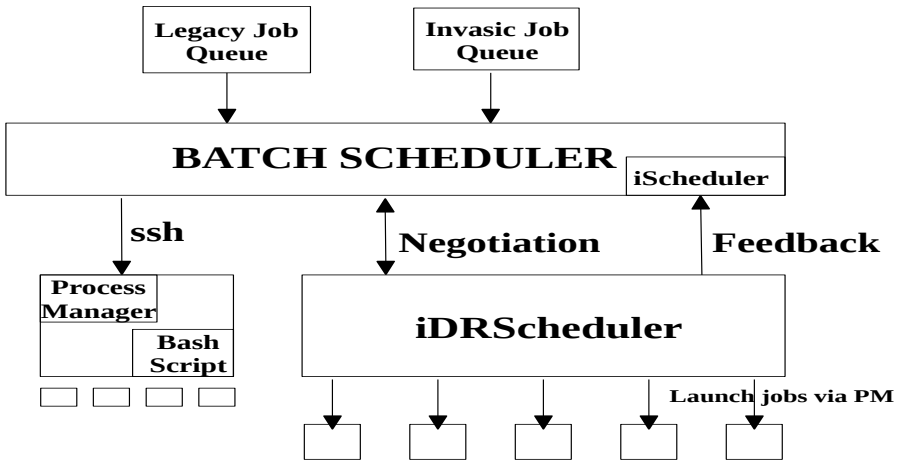


Figure 1.1: Invasive Resource Management Architecture

these new type of Invasic Jobs and a new component called ***Invasic Scheduler (iScheduler)*** is responsible for this. In contrast to modifying an existing system to support Invasive Resource Management, a new component called ***Invasive Distributed Run Time Scheduler (iDRScheduler)*** is proposed which sits between the Batch Scheduler and Process Manager. The objective of such a multi-level approach is to avoid modifying the existing system which will be a substantially large effort and rather have an independent component that caters specifically to Invasic Jobs. It is responsible for managing the resources present in the Invasic paritition used specifically for running Invasic Jobs. With this approach, the existing Legacy Jobs can be served via the existing batch scheduler and the new Invasic Jobs can be served by via iScheduler and iDRScheduler. iDRScheduler talks to the iScheduler via a protocol called the Negotiation protocol to receive Invasic Jobs dispatched from iScheduler which it then launches for execution by performing some run time scheduling like pinning of jobs, expand/shrink etc.

The new components proposed in the architecture help achieve the objective of supporting dynamic resource management for invasic applications. iScheduler is responsible for scheduling Invasic Jobs. The scheduling decisions are communicated via negotiation protocol to iDRScheduler and these decisions are basically job(s) selected via a scheduling algorithm to be submitted for execution. The decisions will be made on the basis of Resource Offers sent by iDRScheduler which creates these resource offers based on the state of the partition. Upon receiving a resource offer, iScheduler will either accept it by selecting jobs from the queue that can be mapped to this offer or reject it. A Resource Offer can represent real or virtual resources because the iDRScheduler can also present a virtual view of resources in the hope of getting a mapping of jobs to offer that is

more suitable to satisfy its local metrics such as resource utilization, power stability, energy efficiency etc. It can either accept or reject the mapping received from iScheduler. Similar to iDRScheduler, the iScheduler makes its decisions to optimize for certain local metrics such as high job throughput, reduced job waiting times, deadlines, priorities etc. This highlights the mismatching policies/metrics for which both the iDRScheduler and iScheduler make their decisions on and hence both will be involved in some kind of a negotiation via the protocol to reach a common agreement. iDRScheduler is an independent entity introduced with the purpose of inter-operating with existing batch systems rather than replacing them with an entirely new one. It may be possible that in the future this component will not be a separate entitiy but will be built into the batch system itself.

Shrink/Expand Strategies: There are several strategies one may employ to tackle adaptive applications during runtime and take decisions that can lead to higher system utilisation, energy efficiency etc.:

- Let us consider that at any given point of time there are some Invasic Jobs running in the system. If the parallel runtime environment is able to anticipate that in the near future, there may be a large window of free resources available because some applications may shrink according to a prediction of their scalability behavior with the help of run time profiling and collected performance data then iDRScheduler can provide a resource offer to iScheduler. This offer can specify a virtual list of nodes more than what is available in order to get a mapping of jobs. These jobs can then be shrunk and fit into the existing space of resources with the knowledge that they may be able to expand later.

- Another scenario is where there is an anticipation of a smaller window of resources in the future because some of the currently running applications may expand. In such a case iDRScheduler will provide a resource offer to iScheduler with a virtual list of nodes smaller than what is currently available in order to get a mapping of jobs. These jobs can be then be expanded and fit into the existing space of resources with the knowledge that they may have to shrink later.

- The third variation could be the case where the runtime anticipates the state of resources to remain the same as the current state for the near future since none of the running applications may expand or shrink. In such a scenario iDRScheduler will send a resource offer that is a list of nodes which is exactly the same as the available resources to iScheduler in order to get a mapping of jobs. These jobs can then be fit into the space of resources available as it is without expanding/shrinking them.

1.3 Master Thesis

The focus of this Master Thesis is to extend the early prototype developed as a part of the guided research in the last few months. It will now give concrete meanings to the negotiation protocol, defining the format of the invasic job records, its constraints, definition of resource offers, feedback reports and most important of all to investigate and develop an efficient job scheduling algorithm at the iScheduler level. Following this closely from the Guided Research would be to continue having an automated testing in place that will help in simulating a workload of jobs, testing the job scheduling algorithm for its correctness, evaluating and analysing the performance of such a prototype for various metrics. Given below is a tentative plan of the activities proposed monthwise starting from November 15 till May 15 to be carried out for the duration of 6 months in this Master Thesis.

1.3.1 Tentative Plan

- Literature survey for current/recent related work on batch job scheduling from research groups focusing on problems in the areas of batch scheduling, resource management, middleware etc. In addition to this, defining resource offers, invasic job records and job constraints that can come from a pre-computed performance model of an application during its runtime or could be user defined. Once these tasks have been completed, it will follow up with extending the early prototype by the implementation of these ideas and testing them for correctness using the automated testing feature.

- Integration of the fake iHypervisor with the real iHypervisor by making it as its plugin. Test this integration for all the earlier development using the automated testing feature for correctness. Review, analyse, fix any errors and repeate the process till the integration is stable. Follow this up by understanding the SLURM's existing scheduling algorithms including the recent Machine Learning approach for a possible reuse or enhancement. Define the format of feedback reports sent by iHypervisor and how they can be processed on the iScheduler side including its storage mechanism (possibly using SLURM's existing database functionalities). Implement these ideas in the prototype and test it thoroughly.

- Investigate and experiment with basic scheduling algorithms first to later proceed with complex ones. Design and implement a sophisticated scheduling algorithm for mapping jobs from Invasic job queue to the resource offers sent by iHypervisor. This algorithm must perform its decision making by also using the collected

history/feedback data about many statistical measures provided by iHypervisor periodically.

- Implementation of the Job Scheduling Algorithm continues for realizing all the necessary requirements as proposed earlier till it has been successfully implemented and later followed by a lot of functional testing.
- Test the full prototype along with the other component iHypervisor (Including its Run Time scheduling and Invasive Resource Management) for simple to complex workloads (emulating a queue of invasic jobs possibly including legacy static jobs). Testing also needs to be done with real Invasic applications developed by the *Chair Of Scientific Computing*. Find issues/errors, review, analyse, correct and repeat the process till it is stable and then collect all the results necessary.
- Coming up with the draft version of the Master Thesis Report followed by reviews, corrections. This process is repeated till the final version is decided. Also prepare the slides for the Master thesis to present them at a later stage.

The above timeline highlights a tentative plan for the activities to be taken up during the Master Thesis and the 6 items above correspond to these 6 months of the Thesis in a chronological order. The steps may overlap or shift depending on the progress but the same overall structure will be followed for the Thesis. It will also include in parallel small amounts of documentation in the report as and when necessary during the 6 month period and not necessarily everything at the end.

1.4 Motivation

1.5 Document Structure

This is end of the first section which gave an introduction to this Master Thesis and the kind of problem it deals with. The rest of this report is organized as follows:

- ***Modeling:*** This section will explain in brief on how the DNDP will be mathematically formulated using a bilevel linear program with all its contraints, decision variables and objective function. It will explain the approach to approximate the non-linear objective function of the inner problem of DNDP which is TAP and an introduction to metaheuristics that will be used to solve such kind of a combinatorial optimization problem.

- ***Genetic Algorithm:*** This section will dive into the details of the genetic algorithm that includes all the important aspects which fall under GA that needs to be tackled in order to come up with a correct implementation yielding good performance. It will explain in detail some of the many choices one has for implementation during every step of GA.
- ***Implementation:*** This section will illustrate with the help of a flow chart the high level view of the GA implementation followed by some pseudo codes to demonstrate in a simple language the implementation details of the inner workings of GA.
- ***Experiments and Visualization:*** This section will present all the results of the experiments conducted on different types(sizes) of datasets using the implemented GA in the form of tables. It will also mention in brief some of the observations and inferences that have been drawn by looking at these results.
- ***Conclusion:*** This section concludes the report on this project with a highlight of what was successfully achieved along with the possible scope of what can be done as a part of future research work. This is followed by a list of some useful references that played an important role in the understanding of many of the concepts towards the realization of this project.

2 Related Work

2.1 Batch Scheduling

2.2 Runtime Scheduling

2.3 Adaptive Programming Paradigms

3 Invasive Computing

3.1 Job Classification

The throughput of HPC Systems depends not only on efficient job scheduling but also on the type of jobs forming the workload. As defined by Feitelson, and Rudolph, Jobs can be classified into four categories based on their flexibility:

- ***Rigid Job:*** Requires a fixed number of resources throughout its execution.
- ***Moldable Job:*** The resource requirement of the job can be molded or modified by the batch system before starting the job(e.g. to effectively fit alongside other rigid jobs). Once started its resource set cannot be changed anymore.
- ***Evolving Job:*** These kind of jobs request for resource expansion or shrinkage during their execution. Applications that use Multi-Scale Analysis or Adaptive Mesh Refinement (AMR) exhibit this kind of behavior typically due to unexpected increases in computations or having reached hardware limits (e.g. memory) on a node.
- ***Malleable Job:*** The expansion and shrinkage of resources are initiated by the batch system in contrast to the evolving jobs. The application adapts itself to the changing resource set.

3.2 Resource Aware Programming

In this section, we will briefly look into the details of the earlier invasive extensions to OpenMP and MPI done as a part of this ongoing research project. This will give us an insight into the earlier approach taken towards realizing such resource-aware programming models and also serve as a prelude to the approach currently undertaken.

iOMP
Earlier efforts were made in the ongoing INVASIC project to extend the OpenMP parallel programming model for shared memory systems. This was called as invasive OpenMP or iOMP and it allows the programming of resource aware applications.

Resource awareness could mean either that the program can allocate or free resources according to the amount of available parallelism / the dynamic size of the data or it could mean that it can adapt to the available resources for execution.

OpenMP is implemented as a set of compiler directives, library routines and environment variables. Parallelization is based on directives inserted into the application's source code to define parallel regions that are executed in parallel using a fork-join parallelization model. The parallel region would then be executed by a team of threads whereas the sequential region would be executed by a single master thread. There are different ways to control the number of threads in a parallel region, most common approach is through the environment variable, or through OpenMP library call or as an additional clause in its directives. iOMP is implemented as a library in C++ using an object oriented approach. It provides two important methods available in its class ***Claim***:

- ***Invade***: This operation allocates additional resources / PE's. A constraint parameter passed as an argument to this operation specifies the details such as which resources and how many of them [range] are additionally required from the resource manager.
- ***Retreat***: This operations deallocates resources / PE's. A constraint parameter passed as an argument to this operation specifies the details such as which resources and how many of them must be freed to the resource manager.

iOMP follows a similar terminology as mentioned in section x.x.x for X10 in invasive computing. A *Claim* in iOMP referes to all the resources / PE's allocated to the application. This means that an iOMP program will always have a single claim. Initially, the claim size is 1 but it will increase and decrease during the runtime of the application. The constraint parameter mentioned before also allows the programmer to specify several other constraints such as memory, pinning strategy, architecture specific optimizations etc. Below is a small snippet of code from the paper that shows an example of iOMP program.

```
int main()
{
       Claim claim;
       int sum = 0;
       /* Acquire resources according to the given constraints */
       claim.invade(PEQuantity(1, 3));

       /* Executing a parallel for loop on the given resources */
```

```
        #pragma omp for reduce reduction(+:sum)
        for (int i=0; i < 100000; i++)
                sum += i;

        /* Free resources and delete pinning */
        claim.retreat();
}
```

As another part of iOMP, A resource manager has also been implemented. This has a global view of the resources in the shared memory system and acts like a server to every other running application that are its clients. Every client-server communication happens over a message queue. The resource manager handles the redistribution of the resources over time to all running applications based on their invade / retreat operations.

3.3 Traditional Resource Management

The role of a resource manager is to acts like a *glue* for a parallel computer to execute parallel jobs. It should make a parallel computer as easy to use as almost a PC. MPI would typically be used to manage communications within the parallel program. A resource manager allocates resources within a cluster, launches and otherwise manages jobs. Some of the examples of widely used open source as well as commercial resource managers are **SLURM, TORQUE, OMEGA, IBM Platform LSF** etc. Together with a scheduler it is termed as a ***Batch System***. The Batch System serves as a middleware for managing supercomputing resources. The combination of *Scheduler+Resource Manager* makes it possible to run parallel jobs.

The role of a job scheduler is to manage queue(s) of work when there is more work than resources. It supports complex scheduling algorithms which are optimized for network topology, energy efficiency, fair share scheduling, advanced reservations, preemption, gang scheduling(time-slicing jobs) etc. It also supports resource limits(by queue, user, group, etc.). Many batch systems provide both resource management and job scheduling within a single product (e.g. LSF) while others use distinct products(e.g. Torque resource manager and Moab job scheduler). Some other examples of Job scheduling systems are **LoadLeveler, OAR, Maui, SLURM** etc.

3.3.1 Classification

Before the classification begins we define some terms that are used in the following.

- The term *scheduling* stands for the process of computing a schedule. This may be done by a queuing or planning based scheduler.
- A resource request contains two information fields: the number of requested resources and a duration for how long the resources are requested for.
- A job consists of a resource request plus additional information about the associated application. Examples are: information about the processing environment (e. g. MPI or PVM), file I/O and redirection of stdout and stderr streams, the path and executable of the application, or startup parameters for the application. We neglect the fact that some of these extra job data may indeed be needed by the scheduler, e. g. to check the number of available licenses.
- A reservation request is a resource request starting at a specified time for a given duration. Once the scheduler accepted such a request, it is a reservation.

We call a reservation Fix-Time request to emphasize that it can not be shifted on the time axis during scheduling. Accordingly we call a resource request Var-Time request, as the scheduler can move the request on the time axis to an earlier or later time according to the used scheduling policy. In the following we focus on resource management systems that use space-sharing.

The criterion for the differentiation of resource management systems is the planned time frame [42]. Queuing systems try to utilize currently free resources with waiting resource requests. Future resource planning for all waiting requests is not done. Hence, waiting resource requests have no proposed start time. Planning systems in contrast, plan for the present and future. Planned start times are assigned to all requests and a complete schedule about the future resource usage is computed and made available to the users. A comprehensive overview is given in Table 3.2 at the end of this section.

Queuing Systems Today almost all resource management systems fall into the category of queuing systems. Several queues with different limits on the number of requested resources and the duration exist for the submission of resource requests. Jobs within a queue are ordered according to a scheduling policy, e. g. FCFS (first come, first serve). Queues might be activated only for specific times (e. g. prime time, non prime time, or weekend). The task of a queuing system is to assign free resources to waiting requests. The highest prioritized request is always the queue head. If it is

possible to start more than one queue head, further criteria like queue priority or best fit (e. g. leaving less resources idle) are used to select a request. There might also exist a high priority queue whose jobs are preferred at any time. If not enough resources are available to start any of the queue heads, the system waits until enough resources become available. These idle resources may be utilized with less prioritized requests by backfilling mechanisms.

As the queuing systems time frame is the present, no planning of the future is done. Hence, no information about future job starts are available. Consequently guarantees can not be given and resources can not be reserved in advance. However, if participating in grid environments this functionality is desirable. Using reservations eases the way of starting a multi-site application which synchronously runs on different sites. By reserving the appropriate resources at all sites, it is guaranteed that all requested resources are available at the requested start time. With queuing systems this still has to be done manually by the administrative staff. Usually high priority queues combined with dummy jobs for delaying other jobs are used.

Users do not necessarily have to specify run time estimates for their jobs, as a queuing system might let jobs run to completion. Obviously users would exploit this by starting very long running jobs which then block parts of the system for a long time. Hence, run time limits were added to the queues (Table 3.1). A longer run time than the limit of the queue is not allowed and the resource management system usually kills such jobs. If the associated application still needs more CPU time, the application has to be checkpointed and later restarted by the user.

Planning Systems Planning systems schedule for the present and future. With assigning start times to all requests a full schedule is generated. It is possible to query start and end times of requests from the system and a graphical representation of the schedule is alsopossible (Figure 3.1). Obviously duration estimates are mandatory for this planning. With this knowledge advanced reservations are easily made possible. There are no queues in planning systems. Every incoming request is planned immediately.

The re-planning process is the key element of a planning system. Each time a new request is submitted or a running request ends before it was estimated to end, a new schedule has to be computed and this function is invoked. At the beginning of a re-plan all non-running requests are deleted from the schedule and sorted according to the scheduling policy. Then all requests are re-inserted at the earliest possible start time in the schedule. After this step each request is assigned a planned start and end time. The

non-running requests are usually stored in a list structure and different sorting criteria are applied. They define the scheduling policy of the system.

As planning systems work with a full schedule and assign start times to all requests, resource usage is guaranteed and advanced reservations are possible. A reservation usually comes with a given start time or if the end time is given the start time is computed with the estimated run time. When the reservation request is submitted the scheduler checks with the current schedule, whether the reservation can be made or not. That is the amount of requested resources is available from the start time and throughout the complete estimated duration. If the reservation is accepted it is stored in an extra list for accepted reservations. During the re-planning process this list is processed before the list of variable requests. It does not have to be sorted as all reservations are accepted and therefore generate no conflicts in the schedule. Furthermore, additional types of job lists are thinkable which are then integrated in the re-planning process according to their priority (reservations should have the highest and variable requests the lowest priority).

Controlling the usage of the machine as it is done with activating different queues for e. g. prime and non prime time in a queuing system has to be done differently in a planning system. One way is to use time dependent constraints for the planning process, e. g. "during prime time no requests with more than 75 percent of the machines resources are placed". Also project or user specific limits are possible so that the machine size is virtually decreased. Examples for such limitations are:

- Jobs requesting more than two thirds of the machine are not started during daytime, only at night and on weekends.
- Jobs that are estimated to run for more than two days are only started at weekends.
- It is not possible that three jobs are scheduled at the same time where each job uses one third of the machine.

If an already running request interferes with limits during its run time, it is not prematurely killed. It runs until the estimated end is reached. With the examples from above one could think of a job that requests all resources of a machine, is started on Sunday and runs until Tuesday. Such a job would then block the whole machine on Monday which contradicts the limits for Monday.

Planning systems also have drawbacks. The cost of scheduling is higher than in queuing systems. And as users can view the current schedule and know when their requests are planned, questions like "Why is my request not planned earlier? Look, it

would fit in here." are most likely to occur [50]. Besides the pure and easily measurable performance of the schedule (e. g. utilization or slowdown), other more social and psychologic criteria might also be considered. It might be beneficial to generate a less optimized schedule in favor of having a more understandable schedule. Furthermore, the usage of reservations should be observed, especially if made reservations are really used. Again, users tend to simply reserve resources without really needing and using them [50]. This can be avoided by automatically releasing unused reservations after a specific idle time.

Table 3.2 shows a summary of the previously described differences between queuing and planning based resource management systems.

3.3.2 Job Scheduling

Scheduling Policies Typical resource management systems store requests in list-like structures. Therefore, a scheduling policy consists of two parts: inserting a new request in the data structure at its submission and taking requests out during the scheduling. Different sorting criteria are used for inserting new requests and some examples are (either in increasing or decreasing order):

- by arrival time: FCFS (first come first serve) uses an increasing order. FCFS is probably the most known and used scheduling policy as it simply appends new requests at the end of the data structure. This requires very little computational effort and the scheduling results are easy to understand: jobs that arrive later are started later. With this example the term fairness is described [79]. In contrast, sorting by decreasing arrival time is not commonly used, as 'first come last served' makes no sense in an on-line scenario with the potential risk of waiting forever (this is also called starvation). However, a stack works with decreasing order of arrival time.

- by duration: Both increasing and decreasing orders are used. Sorting by increasing order leads to SJF (shortest job first) respectively FFIH (first fit increasing height2). Accordingly LJF (longest job first) and FFDH (first fit decreasing height) sort by decreasing run time. In an on-line scenario this requires duration estimates, as the actual duration of jobs are not known at submission time. SJF and LJF are both not fair, as very long (SJF) and short (LJF) jobs potentially wait forever. LJF is commonly known for improving the utilization of a machine.

- by area: The jobs area is the product of the width (requested resources) and

length (estimated duration). FFIA (first fit increasing area) is used in the SMART algorithm (Scheduling to Minimize Average Response Time) [91, 76].

- by given job weights: Jobs may come with weights which are used for sorting. Job weights consist of user or system given weights or a combination of both. For example: all jobs receive default weights of one and only very important jobs receive higher weights, i. e. they are scheduled prior to other jobs.
- by the Smith ratio: The Smith ratio of a job is defined by weight area and is used in the PSRS (Preemptive Smith Ratio Scheduling) algorithm [75].
- by many others: e. g. number of requested resources, current slowdown, ...

In the scheduling process jobs are taken out of the ordered data structure for either a direct start in queuing systems or for placing the job in a full schedule (planning system):

- front: The first job in the data structure is always processed. Most scheduling policies use this approach as only with this a sorting policy makes sense. In queuing systems jobs might have to wait until enough resources are available. Planning systems also process the front of the data structure while placing requests as soon as possible. FCFS, SJF, and LJF use this approach.
- first fit: The data structure is traversed from the beginning and always the first job is taken, that matches the search constraints, i. e. requests equal or less resources than currently free.
- best fit: All jobs are tested to see whether they can be scheduled. According to a quality criterion the best suited job is chosen. Commonly the job which leaves the least resources idle in order to increase the utilization is chosen. Of course this approach is more compute intensive as the complete data structure is traversed and tested. If more than one job is best suited an additional rule is required, e. g. always take the first, the longest/shortest job, or the job with the most weight.
- next fit: The SMART algorithm uses this approach in a special case (NFIW) [91, 76].

In general, all combinations are possible but only a few are applicable in practice. Figure 3.2 shows example schedules for FCFS, SJF, LJF, and FFIA. Sorting requests in any order while using first or best fit is not necessary, as the best job is always chosen regardless of its position in the sorted structure. However, a sorting policy could be used to choose one job, if many jobs are equal. Furthermore, best fit comes with the

risk of making schedules unfair and opaque for users.

If fairness in common sense has to be met, i. e. the starting order equals the arrival order, only the combination of sorting by increasing arrival time and always processing the front of the job structure can be used. All other combinations do not generate fair schedules. However, such a fair scheduler is not very efficient, as jobs usually have to wait until enough free resources are available. Therefore, basic scheduling policies are extended by backfilling, a method to avoid excessive idleness of resources. Backfilling became standard in modern resource management systems today. If requests are scheduled out of their sorting order by first or best fit, some form of backfilling is carried out.

EASY Backfilling The default algorithms used by current job schedulers for parallel supercomputers are all rather simple and similar to each other [37], employing a straightforward version of variable partitioning. (Recall that this means space-slicing with static-partitioning, where users specify the number of processors required by their jobs upon submittal.) In essence, schedulers select jobs for execution in first-come first-served (FCFS) order, and run each job to completion, in batch mode. The problem with this simplistic approach is that it causes significant fragmentation, as jobs with arbitrary sizes/arrivals do not pack perfectly. Specifically, if the first queued job requires many processors, it may have to wait a long time until enough are freed. During this time, processors stand idle as they accumulate, despite the fact there may very well be enough of them to accommodate the requirements of other, smaller, waiting jobs.

To solve the problem, most schedulers therefore employ the following algorithm. Whenever the system status changes (job arrivals or terminations), the scheduler scans the queue of waiting jobs in order of arrival (FCFS) and starts the traversed jobs if enough processors are available. Upon reaching the first queued job that cannot be started immediately, the scheduler makes a reservation on its behalf for the earliest future-time at which enough free processors would accumulate to allow it to run. This time is also called the shadow time. The scheduler then continues to scan the queue for smaller jobs (require fewer processors) that have been waiting less, but can be started immediately without interfering with the reservation. In other words, a job is started out of FCFS order only if it terminates before the shadow time and therefore does not delay the first queued job, or if it uses extra processes that would not be needed by the first queued job. The action of selecting smaller jobs for execution before their time provided they do not violate the reservation constraint is called backfilling, and is illustrated in Fig. 1.1 (see detailed description in Section 2.3).

This approach was initially developed for the IBM SP1 supercomputer installed at the Argonne National Laboratory as part of EASY (Extensible Argonne Scheduling sYstem), which was the first backfilling scheduler [98].1 The term "EASY" later became a synonym for FCFS with backfilling against a reservation associated with the first queued job. (Other backfill variants are described below.) While the basic concept is extremely simple, a comprehensive study involving 5 supercomputers over a period of 11 years has shown that consistent figures of 40–60 percent average utilization have gone up to around 70 percent, once backfilling was introduced [79]. Further, in terms of performance, backfilling was shown to be a close second to more sophisticated algorithms that involve preemption (time slicing), migration, and dynamic partitioning [19, 170].

User Runtime Estimates The down side of backfilling is that it requires the scheduler to know in advance how long each job will run. This is needed for two reasons:

- to compute the shadow time for the longest-waiting job (e.g. in the example given in Fig. 1.1, we need to know the runtimes of job 1 and job 2 to determine when their processors will be freed in favor of job 3), and
- to know if smaller jobs positioned beyond the head of the wait-queue are short enough to be backfilled (we need to make sure backfilling job 4 will not delay job 3, namely, that job 4 will terminate before the shadow time of job 3).

Therefore, EASY required users to provide a runtime estimate for all submitted jobs [98], and the practice continues to this day. Importantly, jobs that exceed their estimates are killed, so as not to violate subsequent commitments (the reservation). This strict policy has the additional benefit that it supplies an inherent and clear motivation for users to provide high quality estimates , as short enough values increase the chances for backfilling, but too-short values will get jobs prematurely killed.

Popularity of EASY The burden placed on users to provide estimates has not been detrimental. Rather, the combination of simplicity, effectiveness, and FCFS semantics (often perceived as most fair [123]) has made EASY a very attractive and a very popular job scheduling strategy. Nowadays, virtually all major commercial and open-source production schedulers support EASY backfilling [37], including

- IBM's LoadLeveler [60, 82],
- Cluster Resources' commercial Moab [118] and open-source Maui [75] (which is probably the most popular scheduler used within the academia),
- Platforms' LSF (Load Sharing Facility) [172, 24],

- Altair's PBS (Portable Batch System) [68] in its two flavors: commercial PBS-Pro [33] and open-source OpenPBS [10], and
- Sun's GridEngine [61, 106]

The default configuration of all these schedulers, except PBS, is either EASY or plain FCFS (with FCFS, however, the schedulers' behavior becomes EASY if backfilling is nevertheless enabled). The CTO of Cluster Resources has estimated that 90-95 percent of Maui/Moab installations do not change their default (EASY) settings [74]. Being the exception that implies the rule, the PBS variants use Shortest-Job First (SJF) as their basic default policy. However, even with PBS, when a job is "starved" (a situation defined by PBS to occur if the job is waiting for 24 hours or more) then the scheduling reverts to EASY until this job is started. As a testament for its immense popularity, a survey about the top 50 machines within the top-500 list revealed that, out of the 25 machines for which relevant information was available, 15 (= 60 percent) were operating with backfilling enabled [36].

Variations on Backfilling Despite the simplicity of the concept, backfilling has nevertheless been the focus of dozens of research papers attempting to evaluate and improve the basic idea.3 We do not list them all here, but rather, cite many of them (and more) when appropriate, within their respective contexts later on. The remainder of this section only briefly mentions some of the various tunable knobs of backfilling algorithms.

One tunable parameter is the number of reservations. As mentioned above, in EASY, only the first queued job receives a reservation. Thus, backfilling may cause delays in the execution of other waiting jobs which are not the first and therefore do not get a reservation [47]. The obvious alternative is to allocate reservation to all the jobs. This approach has been named "conservative backfilling" as opposed to the "aggressive" approach taken by EASY [108]. However, it has been shown that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison to the aggressive alternative. The MAUI scheduler includes a parameter that allows system administrators to set up the number of reservations [75]. It has been suggested that allocating up to four reservations is a good compromise [15].

A second parameter is the looseness of reservations. For example, an intriguing suggestion is a "selective reservation" strategy depending on the extent different jobs have been delayed by previous backfilling decisions. If some job is delayed by too much, a reservation is made for this job [141]. This is somewhat similar to the "flexible backfilling" strategy, in which backfilling is allowed to violate the reservation(s) up to a certain slack [150, 166]. (Setting the slack in the latter strategy to be the threshold used

for allocating selective reservations in the former strategy, is more or less equivalent.)

A third parameter is the order of queued jobs. EASY, as well as many other system designs, use FCFS order [98]. A general alternative is to prioritize jobs in a certain way, and select jobs for scheduling (including as candidates of backfilling) according to this priority order. For example, flexible backfilling combines three types of priories: an administrative priority to favor certain users or projects, a user priority used to differentiate between the jobs of the same user, and a scheduler priority used to guarantee that no job is starved [150]. The Maui scheduler has a priority function that includes even more components [75]. Another approach is to prioritize based on various job characteristics. In particular, a set of criteria related to the current queueing time and expected resource consumption of jobs has been proposed, which generalizes the well-known SJF algorithm for improved performance [174, 115] as well as combines it with fairness notions [19, 15]. The queuing order and the timing of reservations can also be determined by economic models [35] or various quality of service assurances [72].

A fourth parameter (related to the previous one) is the partitioning of reservations. The processors of a machine can be partitioned into several disjoint sets (free processors can dynamically move around between them based on current needs). Each set is associated with its own wait-queue and reservation. Lawson and Smirni divided the machine such that different sets serve different jobs classes, characterized by their estimated runtime (e.g. short, medium, and long) [90, 92]. A backfilling candidate is chosen in a round-robin fashion, each time from a different set, and must respect all reservations. By separating short from long jobs, this multiple queue policy reduces the likelihood that a short job is overly delayed in the queue behind a very long job, and therefore improves average performance metrics.

A fifth parameter is the adaptiveness of backfilling. An adaptive backfill scheduler continuously simulates the execution of recently submitted jobs under various scheduling disciplines, compares the hypothetical resulting performance, and periodically switches the scheduling algorithm to be the one that scored the highest. In the face of different workload conditions, this adaptiveness has the effect of both improving and stabilizing the observed performance results [144, 149].

A sixth parameter is the amount of lookahead into the queue. Most backfilling algorithms consider the queued jobs one at a time when trying to backfill them, which often leads to loss of resources to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize the uti-

lization while at the same time respecting the allocated reservation(s). This appears to be a NP-hard problem, but due to the fact machine sizes are relatively small, this can be done in polynomial time (in the complexity of the machine size) using dynamic programming, leading to optimal packing [132, 131].

A seventh and final parameter is related to speculative backfilling, where the scheduler is allowed to exploit gaps in the schedule for backfilling, even if the backfilled job interferes with the reservation. By doing so, the scheduler speculates that the backfilled job would terminate sooner than its estimate suggests, and in any case before the shadow time. Successful speculations obviously improve performance and utilization, and have no negative side effects. But unsuccessful speculations must somehow be dealt with. Unfortunately, all previously suggested solution resulted in a scheduling algorithm that lies outside the attractive variable partitioning domain: The simplest alternative is to kill the offending backfilled job and restart it later on [90]. A similar idea is to employ "short test runs", during which jobs either manage to terminate, or are reinserted to the wait-queue with a tighter estimate deduced from the test [115, 15]. Unfortunately, both ideas assume jobs are restartable, which is often not the case. A possible workaround is to employ preemption (time sharing), such that instead of killing and restarting the job, it is suspended and kept in memory, only to be resumed later on from the point in which it was stopped [139]. However, if preemption comes into play, it may be preferable to instead combine backfilling with a full fledged gang scheduler altogether [169]. This combination has even been further extended by adding migration capabilities [85, 170]. A recent study suggested preemption is actually redundant if migration or dynamic partitioning are available. The idea is to reduce the backfilled job's processor allocation by folding it over itself. This frees most of its processors, and limits the performance degradation to the offending job [162].

Finally, a new direction in job scheduling research is to try and minimize the electric power demand, which is rapidly becoming a problem in the context of supercomputing [129]. It has been suggested to integrate the concept of scheduling and power management within EASY [91]. The proposed scheduler continuously monitors load in the system and selectively puts certain nodes in "sleep mode" (makes them unavailable for execution), after estimating the effect of fewer nodes on the projected job slowdown. Using online simulation, the system adaptively selects the minimal number of processors that are required to meet certain negotiated service level agreements.

3.3.3 SLURM

The prime focus of this work will be on **SLURM(Simple Linux Utility For Resource Management)** which will be the choice of batch system upon which the support for Invasive Computing will be demonstrated. SLURM is a sophisticated open source batch system with about 500,000 lines of C code whose development started in the year 2002 at Lawrence Livermore National Laboratory as a simple resource manager for Linux Clusters and a few years ago spawned into an independent firm under the name SchedMD. SLURM has since its inception also evolved into a very capable job scheduler through the use of optional plugins. It is used on many of the world's largest supercomputers and is used by a large fraction of the world's TOP500 Supercomputer list. It supports many UNIX falvors like AIX, Linux, Solaris and is also fault tolerant, highly scalable, and portable.

Place the original SLURM architecture diagram here

Plugins are dynamically linked objects loaded at run time based upon configuration file and/or user options. 3.1 shows where these plugins fit inside SLURM. Approximately 80 plugins of different varities are currently available. Some of them are listed below:

- ***Accounting storage:*** MySQL, PostgreSQL, textfile.
- ***Network Topology:*** 3D-Torus, tree.
- ***MPI:*** OpenMPI, MPICH1, MVAPICH, MPICH2, etc.

PLugins are typically loaded when the daemon or command starts and persist indefinitely. They provide a level of indirection to a configurable underlying function.

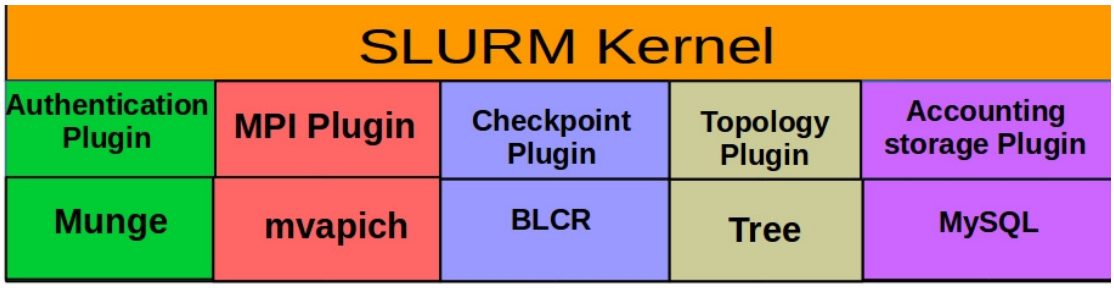


Figure 3.1: SLURM with optional Plugins

The collection of binaries that make up SLURM include: the controller (SLURMCTLD), the node daemons (SLURMD), the launcher (SRUN), as well as several account

management and information utilities. Each of these binaries is highly threaded and have a collection of plugin interfaces for several extendible functions, such as: scheduling, topology, MPI/PMI support, and so forth. Figure 5 shows how these binaries interact and where they reside on an HPC system. In the following subsections, the extensions made to SLURM binaries: SLURMCTLD, SLURMD and SRUN and any related plugins where applicable will be covered.

SLURMCTLD

The SLURM controller is the component where centralized decisions are made. This component runs the batch scheduling thread, manages security, tracks each individual node (by reaching each *slurmd*), and so forth.

SLURMD

The SLURMD binaries are daemons that are started at 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. These second type of daemons track the node local part of a job step, which are the subset of processes that run on the local node. A job step in SLURM is an application started with SRUN, and its allocated resources. Figure 6 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 in the master node of an allocation.

SRUN

3.4 Invasive Resource Management

3.4.1 Invasive MPI

Message Passing has for long remained the dominant programming model for distributed memory systems. MPI stands for Message Passing Interface. It is a standardized and portable message passing system designed to function on a wide variety of parallel computers. It defines the syntax and semantics of a core of library routines for writing portable programs in C, C++ and Fortran. It implements a message passing type of parallel programming model where the application consists of a set of processes with separate address spaces. The processes exchange messages by explicit send/receive operations. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These have fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications. Some of the most popular and widely used open-source implementations of MPI standard are MPICH and OpenMPI. LAM

/ MPI was the predecessor of OpenMPI and was another early MPI implementation. There are also commercial implementations from HP, Intel and Microsoft.

In the context of invasive computing, parallel applications are resource aware and will invade or retreat from resources depending on their availability and on the load imbalances encountered during their runtime. To support this, some form of dynamic process management of the parallel application is necessary. And, in order to realize this in practice, the most basic requirement would be the need for a library that will serve as an application programming interface for programmers to implement such invasic applications that are capable of adapting to a changing set of resources. This requirement needs to be complemented by the extension of the resource management systems which would need to allocate / deallocate resources and coordinate with the MPI library to allow for such expand / shrink operations of an invasive parallel application.

Traditionally MPI applications are static in nature which means that they are executed with a fixed number of processes. Dynamic process support is available through the MPI spawn and its related operations. The spawn operation creates new processes in a separate child process group, whereas the callers belong to the parent process group. This is a blocking operation where the processes in the parent process group block till the operation is completed. The two process groups are connected to each other via an intercommunicator which is generated during this spawn operation and returned to the application. This intercommunicator can then be used to reach children processes from the parent's process group, or parent processes from the children's process group. Another form of spawn is the multiple version which allows the callers to specify multiple binaries to start as children processes and few other related operations with their own usefulness.

Although the MPI standard provides support for dynamic processes, it suffers from many drawbacks as mentioned below:

- The spawn operations are collective operations and are synchronous across both the parent and child process groups. This effects performance and can induce delays of several seconds.
- These operations produce intercommunicators based on disjoint process groups. Subsequent creation of processes result in multiple disjoint process groups. These factors can complicate the development of an MPI application.
- Destruction of processes can be done only on entire process groups. This short-

coming limits the granularity of operations that can be carried out as only entire process groups can be destroyed. This also limits the location of the resources that the application can release at runtime.

- The processes created with spawn are typically run in the same unmodified resource allocation. Although, not a limitation of the standard itself, but, lack of support from resource manager will result in limiting the usefulness of this spawn operation by not changing the physical resource set of the application.

To overcome the above mentioned shortcomings of the standard dynamic process operations, Invasive MPI is being developed as a part of an ongoing research effort. Invasive MPI is an extended version of the MPI library that provides new API calls in order to allow the programmer to create an invasive application. These new API extensions are necessary to make the application resource aware and to adapt according to a change in the resource set by performing data / load redistributions. Following are the proposed extensions being implemented in MPICH:

MPI Initialization in Adaptive Mode This allows the application to be initialized in adaptive mode. It is an extension of the standard MPI_Init operation and is now called as MPI_Init_adapt. The difference now is that a new parameter called local status is being passed. Upon the return of this Init function, local status will hold a value of *new* if the process doing this MPI initialization was created using the mpiexec command or it will hold a value of *joining* if the process was created by the resource manager as a part of the expansion of an already running invasive MPI application. The *joining* processes will then begin the adaptation window after completing the initialization.

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

Probing Adaptation Data The resource manager decides when and how the adaptation of a running application will be initiated. This operation will allow the application to probe the resource manager for adaptation instructions. This operation is called MPI_Probe_adapt and instructs the application on whether there is an adaptation pending.

```
int
MPI_Probe_adapt( int *current_operation,
                 int *local_status,
```

```
    int *nfailed,
    int *failed_ranks,
    MPI_Info *info
);
```

A value of false returned from the current operation parameter will simply tell the application to continue doing progress normally. A true or fault value indicates that there is an adaptation to be done. In the case of a fault, the application will receive information of the failed MPI ranks, since failed processes may no longer be reachable; these values can be used to prevent the application from initiating communication with the failed ranks and thus prevent deadlocks. This operation will also provide the application with additional information on whether it is a joining process if the process was created by the resource manager to represent newly allocated resources as a part of an expansion operation. Joining processes can skip calling the probe operation. If the information returned is staying then it means it should remain in the process group after the adaptation, otherwise it is retreating.

Beginning an Adaptation Window This operation marks the start of an adaptation window. It provides two communicators as output: one intercommunicator that is equivalent to what is provided by standard spawn operations, and one intracommunicator that gives an early view of how the MPI_COMM_WORLD communicator will look like after the adaptation is committed. It is up to the application to make calls to this operation in a safe location. In general, it is expected that it takes place inside of a progress loop, so that the application may be able to adapt to resource changes during its lifetime. There are no requirements in terms of how often the application should be able to react to resource changes; however, frequent checks for adaptations are desirable to reduce idle times in newly created resources, as well as to minimize interference with other applications running concurrently in the HPC system. Each process is required to read its future rank and the future size of the process group from the helper new_comm_world communicator to perform an adaptation consistently. This new size and local rank of the process will persist after the MPI_COMM_ADAPT_COMMIT operation. Processes that are retreating during the adaptation window will not have access to the future MPI_COMM_WORLD, since a retreating process will be removed from the process group, their 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 the adaptation window. MPI_COMM_ADAPT_BEGIN.

```
int
MPI_Comm_adapt_begin(
```

```
        MPI_Comm *intercomm,
        MPI_Comm *new_comm_world,
        );
```

Commiting an Adaptation Window This operation commits the adaptation. This operation affects MPI_COMM_WORLD: any process that has retired is eliminated from it, and any new joining process is inserted into it. After the commit, the MPI_COMM_WORLD communicator will match exactly the new_comm_world intracommunicator provided by the previously mentioned MPI_COMM_ADAPT_BEGIN operation. 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.

```
int
MPI_Comm_adapt_commit();
```

3.4.2 Resource Management Extensions

Existing batch systems usually support only static allocation of resources to applications before they start. We need to integrate invasive resource management into these existing batch systems in order to change the allocated resources dynamically at runtime. This will allow for both an elastic and fault tolerant execution of MPI applications. Such efforts have already been initiated in the Flux project. Existing systems like SLURM allow a job to have extra resources by expanding its allocation. But, this does not fully satisfy the use case here as we need to either grow or shrink the application. Another important factor is the support needed from a programming model that would allow applications to be adaptive to such allocations. The extensions needed on the MPI library side have already been mentioned in the previous section. In order to achieve the extensions on the resource manager side the following components are proposed / extended:

Invasive Runtime Scheduler (alias iRTSched): An independent component which can talk to the existing batch systems via a protocol to obtain invasive jobs submitted specifically to a partition that can support invasive computing. The iRTSched will then take these jobs and perform some kind of runtime scheduling for pinning these jobs to the resources in the partition. The decisions taken by the iRTSched will be towards optimizing certain local metrics such as resource utilization, throughput, topology, power stability, energy efficiency etc. The scheduling here is done at the granularity of cores and sockets. iRTSched is the one that has the complete information of the

resources in the partition and also manages them. This component in an independent entity with the purpose of inter-operating with existing batch systems rather than replacing them with an entirely new system. It may be possible that in the future this component will not be a separate entitiy but will be built into the batch system itself.

Invasive Batch Scheduler (alias iBSched): This component will be a new extension built into the existing batch systems for performing batch job scheduling. The scheduling decisions are communicated via the protocol used to speak to iRTSched and these decisions are basically job(s) selected via a scheduling algorithm to be submitted to the iRTSched for execution. The scheduling decisions will be made on the basis of available resources in the partition and iRTSched communicates this to iBSched in the form of resource offers (Real/Virtual). It can be a virtual resource offer because the iRTSched can hide the real resources and present a rather fake view of them to iScheduler in the hope of getting a mapping of jobs to offer that is more suitable to satisfy its local metrics. Similar to iRTSched, the iScheduler makes its decisions to optimize for certain local metrics such as reduced job waiting times, fairness, deadlines, priorities etc. This highlights the mismatch in the policies/metrics for which both the iRTSched and iBSched make their decisions on. This is the reason why both will be involved in some kind of a negotiation via the protocol to reach a common agreement.

Negotiation Protocol: This protocol forms the core of the interaction between the iBSched and iRTSched. It allows for iRTSched to make one or a set of resource offers to iBSched which then needs to select jobs from its job queue to be mapped to these resource offers and send back the mapping to the iRTSched. The iRTSched will then decide whether to accept/reject this mapping based on whether it satisfies its local metrics. If it accepts it will launch them based on some run time scheduling and if it rejects then it informs this to iBSched in addition to sending it a new resource offer that will also contain possible future start times for pending jobs. The iBSched can also reject the resource offer in which case it will forward the previous job mapping(if any) again with relaxed resource constraints for jobs that could not be mapped. On accepting an offer, the iBSched will again send back a mapping to iRTSched. This exchange of messages continue until both reach an agreement. If the number of such exchanges reach a threshold then iBSched will just accept whatever offer it receives and iRTSched will also accept the final mapping it received and try its best to satisfy all the jobs forwarded. This will close the transaction.

SLURM
This is choice of the existing batch system upon which this proof of concept to support the paradigm of resource-aware programming will be demonstrated. In the near future,

This can further motivate such supporting infrastructures with other batch systems.

The resource manager needs to closely coordinate with the invasive MPI library to support invasive applications. It needs to fork processes on new resources with the right pinning when an application is expanding or destroy them in case it is shrinking. Both of which needs to be done in coordination with MPI. New processes could be created in the existing resource allocation of the running application, possibly allowing for oversubscription of CPU cores, but that would be of little benefit to most HPC application's performance and scalability. As a part of the ongoing research efforts in the INVASIC project alongside the development of invasive MPI library, The following are the extensions being implemented specifically to SLURM in order to support dynamic resource management:

Slurmctld Extensions A new operation that initiates an adaptation through the *srun* command: *srun_realloc_message* is introduced. The *srun_realloc_message* provides *srun* the following information: the list of new nodes allocated to the application and the number of processes to create on them, the list of 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, and an updated SLURM credential object that is necessary for communication with the new expanded nodes. Currently adaptations are based on full nodes, but this operation is ready for future developments where fine grain scheduling may be implemented. When a transformation is triggered on a job, its status changes from *RUNNING* to *ADAPTING*. Each application notifies the resource manager when its adaptation is completed and its job record will get updated from the status *ADAPTING* to *RUNNING*; this state change marks the application as eligible for adaptations again and its released resources available for other jobs.
Slurmd Extensions 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. The PMI plugin is loaded by the SLURMD daemon. We have extended the PMI2 plugin to support these operations. Notify the local daemon that joining processes have opened a port and are waiting in the internal accept operation of MPI_COMM_ADAPT_BEGIN. Notify the local daemon that both the joining and preexisting processes have completed their adaptation and exited MPI_COMM_ADAPT_COMMIT. The first extension to the PMI is used by the leader process of the joining group. It notifies its local SLURMD daemon, which then notifies the SRUN instance of the job step. The SRUN instance then proceeds to notify each of the SLURMD daemons running in the preexisting nodes. These daemons then proceed to update their local MPI_PROBE_ADAPT metadata and start their side of the algorithm (as described in section 3.2). The second extension to the PMI is used by the leader

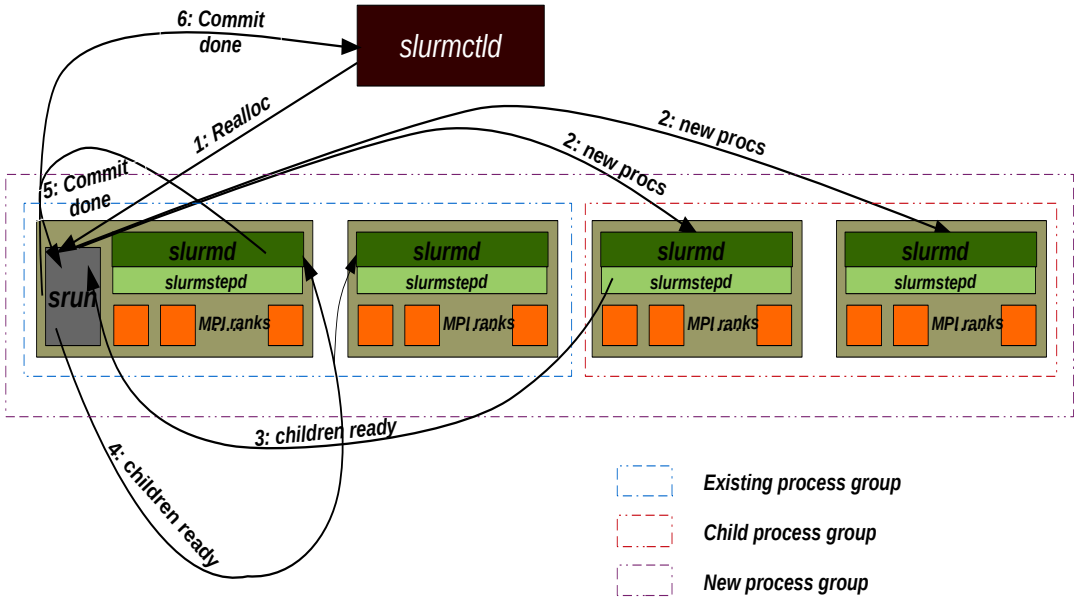


Figure 3.2: SLURM Extension

process of the new adapted process group, that was created as a result of a successful completion of MPI_COMM_ADAPT_COMMIT. The local daemon sends a notification message to SRUN, which then forwards it to SLURMCTLD. The controller handles this message by updating the status of the job from *JOB_ADAPTING* to *JOB_RUNNING*.

Srun Extensions Most of the operations that are initiated by either the controller or any application process (via the PMI and the SLURMD daemons) is handled partially by SRUN. Reallocation message received from the controller (through srun_realloc_message) will be handled by *srun*. Notification that joining processes are ready and waiting in the MPI_COMM_ADAPT_BEGIN operation. Notification that the adaptation 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 launch time; it can now manage redirections dynamically as processes are created and destroyed. The current design of SLURM, where SRUN needs to run in the master node of an allocation, is a limitations 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, which

prevents its migration to a completely new set of nodes. Therefore, SLURM's current design prevents us from ensuring optimal bisection bandwidth in all reallocations.

4 Architecture["What we are building". Abstraction of the complete system at a high level showing all the components and how they will interact with each other like a skeleton. It deals with what is being done and where is it being done but not how. The "how" is tackled in the design

4.1 Dynamic Resource Management

This section illustrates and describes a high level design of the software implemented with the help of protocol sequence diagrams and state machine diagrams. It will help to understand at a high level as to how the system has been designed to support this new approach of invasive computing and how will many of its components in the software hierarchy interact with each other with new protocols or extensions of existing protocols to integrate such an invasive resource management into current batch systems.

The following page shows the software architecture of how Invasive Resource Management can be supported with a traditional resource manager and how exactly the new software components will fit in the existing software hierarchy. The 4.1 relates closely to how SLURM is organized since the intention of this work would be to demonstrate the support for Invasive Computing with the help of SLURM as a resource manager.

- The top layer is that of the core resource management component which has access to job queues. In this architecture, it will now have access to not only the queue for the legacy(static) jobs but also invasive job queue(jobs submittted to invasic partition that supports invasive computing).

- In a traditional setup the top layer will perform the task of job scheduling as well. This means that it will select a job(s) from the queue of jobs based on the current state of resources and many other factors to dispatch it to the traditional process manager below in the hierarchy. The process manager then takes the responsibility of launching these jobs on the allocated resources in the partition and managing them for their full lifetime. In case of parallel jobs, it will manage the job in a parallel environment along with facilitating the communication amongst the parallel tasks/processes with the help of a PMI(Process Manager Interface) server. The process manager may also spawn slave daemons on each of the nodes which are a part of the resource allocation for a single job to manage them more effectively.

- As discussed in the previous chapter, an independent Invasive resource management component by the name "iHypervisor" will be implemented which needs to communicate with a new scheduling component iScheduler and influence the scheduling decisions taken by it. The iHypervisor sits between the top layer and the process manager.

- A new job scheduler specifically for invasive jobs needs to be integrated into the existing batch system. This is due to the reason that the scheduler for invasive jobs will work in a different manner based on the approach described earlier in comparison to the legacy job scheduler for static jobs. In case of SLURM which has a modular design with several optional plugins, a new plugin by name "iScheduler" will be implemented for SLURM to handle job scheduling specifically for invasic jobs.

- Communication between iHypervisor and iScheduler will involve the negotiation protocol as explained in the previous chapter but will also include periodic feedbacks being sent by iHypervisor to iScheduler having some useful statistical measures about current state of resources, resource utilization, job throughput etc. that may help influence the decision making of iScheduler. This communication will also additionally support a means to service urgent jobs immediately.

Communication Phases

- ***Protocol Initialization:*** This phase basically establishes the initial environment between the communicating parties (iScheduler and iHypervisor) for proper communication later on. Successful initialization of this phase prepares both the parties to start negotiating based on the negotiation protocol described in the following points. During this protocol initialization various parameters such as protocl version, maximum attempts for negotiation, timer intervals and several

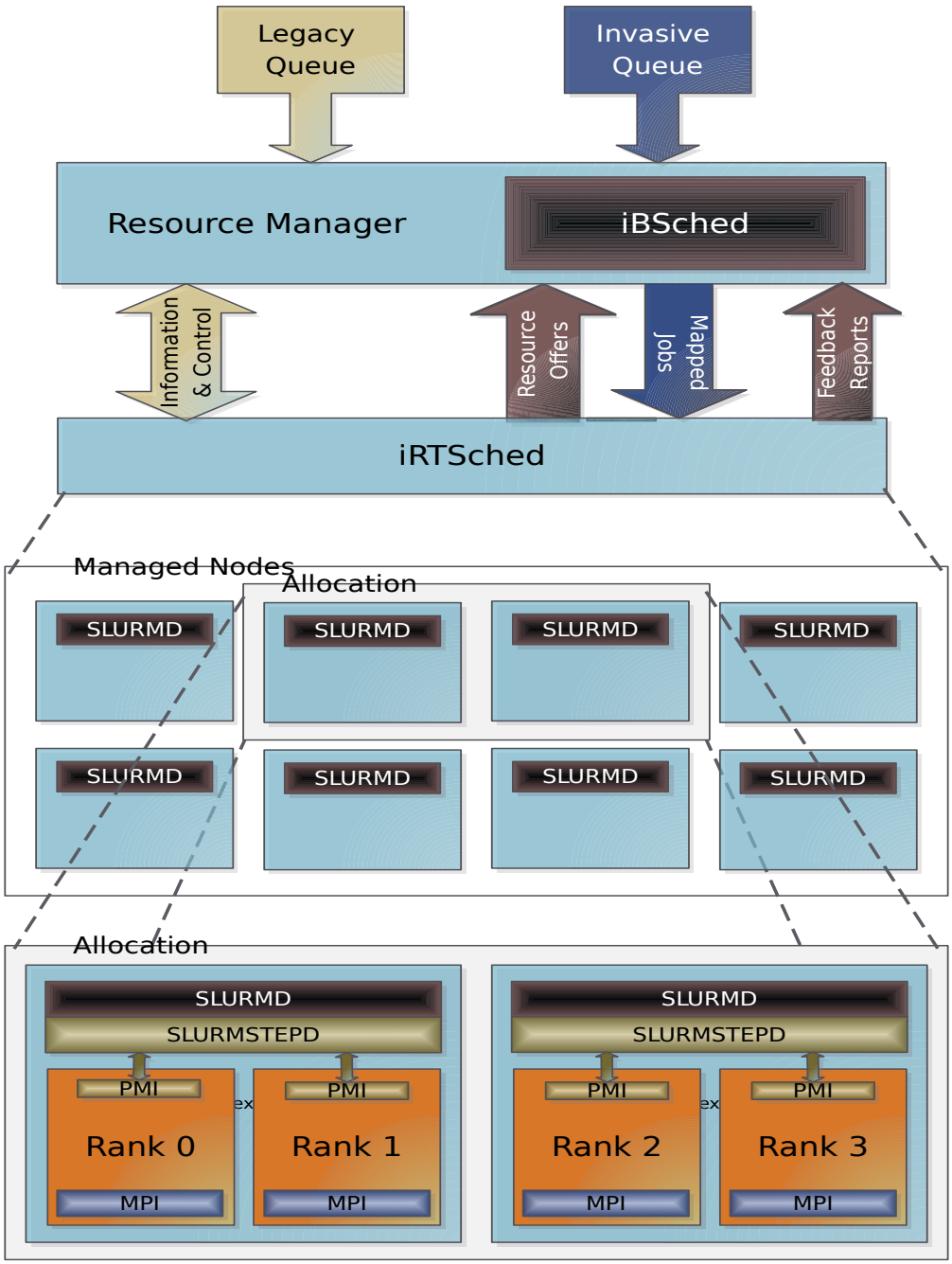


Figure 4.1: Invasive Resource Management Architecture

<EVENT> <=> <PACKET>	DIRECTION OF COMMUNICATION
<REQUEST_RESOURCE_OFFER>	iScheduler → iHypervisor
<RESOURCE_OFFER>	iHypervisor → iScheduler
<RESPONSE_RESOURCE_OFFER>	iScheduler → iHypervisor
<NEGOTIATION_START>	iScheduler → iHypervisor
<RESPONSE_NEGOTIATION_START>	iHypervisor → iScheduler
<NEGOTIATION_END>	iScheduler → iHypervisor iHypervisor → iScheduler
<RESPONSE_NEGOTIATION_END>	iScheduler → iHypervisor iHypervisor → iScheduler
<STATUS_REPORT>	iHypervisor → iScheduler
<URGENT_JOB>	iScheduler → iHypervisor
<RESPONSE_URGENT_JOB>	iHypervisor → iScheduler

Figure 4.2: Message Types

others could be exchanged to set up the internal data structures and configuration tables for both the communicating parties. This protocol is a bi-directional communication.

- ***Protocol Finalization:*** This phases signals the end of communication between iHypervisor and iScheduler using negotiation protocol. It leads to a safe termination of this communication followed by the release of any internal data structures allocated earlier along with configuration parameters. This results in consistent behaviour of both the communicating parties which can then proceed to safely terminate and exit. This protocol is a bi-directional communication.

- ***Negotiation:*** This is the most important phase in this whole approach to support invasive computing as discussed in the previous chapter. It is the phase during which both iHypervisor and iScheduler are negotiating with each other till they reach an agreement. If they do not then they continue till a certain limit to the number of negotiating attempts are reached after which both of them just agree in their final attempt closing the current negotiation. After this a new transaction of negotiation begins.

- ***Feedback:*** This concerns the periodic feedback sent by the iHypervisor to the iScheduler containing useful information such as the job states, latest snapshot

of the resources in the invasic partition and many other statistical measures not limited to system utilization, job throughput, waiting times of jobs to help and influence the iScheduler in its decision making for scheduling jobs during its future transactions of negotiation. This protocol is a uni-directional communication.

- ***Urgent Jobs:*** This protocol concerns the support for urgent jobs. At any given point of time a cluster or supercomputing center may want to support very high priority jobs immediately without any further delay. By introducing support for invasive computing, it makes it all the more feasible to help run these urgent jobs immediately by either shrinking the resources of other jobs or suspending/Killing them.

Separation of Concerns: In this thesis, The idea of separating the batch and runtime scheduling components of a Job Scheduler is explored.

4.1.1 Invasive Batch Scheduler

Today almost all resource management systems fall into the category of queuing systems. Several queues with different limits on the number of requested resources and the duration exist for the submission of resource requests. Jobs within a queue are ordered according to a scheduling policy, e. g. FCFS (first come, first serve). Queues might be activated only for specific times (e. g. prime time, non prime time, or weekend). The task of a queuing system is to assign free resources to waiting requests. The highest prioritized request is always the queue head. If it is possible to start more than one queue head, further criteria like queue priority or best fit (e. g. leaving less resources idle) are used to select a request. There might also exist a high priority queue whose jobs are preferred at any time. If not enough resources are available to start any of the queue heads, the system waits until enough resources become available. These idle resources may be utilized with less prioritized requests by backfilling mechanisms.

4.1.2 Invasive Run Time Scheduler

4.1.3 iMPI Process Manager

4.2 Negotiation Protocol

This section focuses on iScheduler and a thread iRM_AGENT that it spawns which is the one responsible for all the communication with the iHypervisor including spawning other agent threads for handling feedbacks and urgent jobs.

4.2.1 Protocol Sequence Diagrams

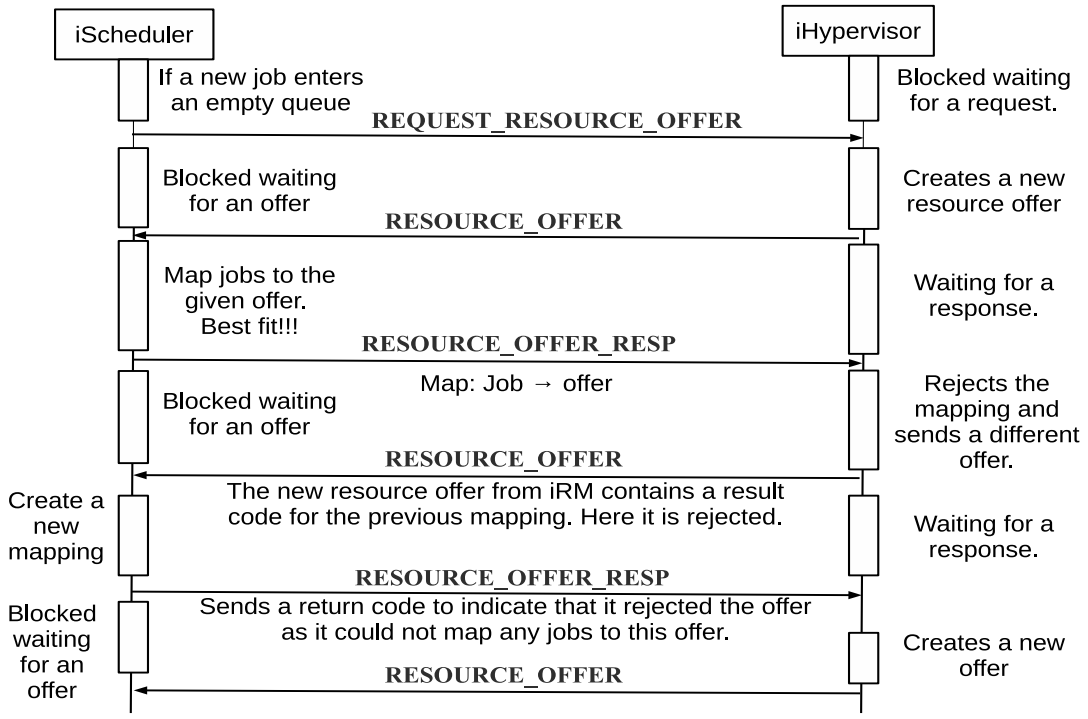


Figure 4.3: Scenario 1

- Above diagram illustrates a scenario where both iScheduler and iHpervisor are negotiating with each other. The scenario is continued in the next page. 4.5 illustrates another scenario where negotiations may stop when job queue becomes empty and iHypevisor then will wait for a request from iScheduler for a resource offer that will happen when new jobs arrive.
- iScheduler makes scheduling decisions at a coarser level of granularity which is nodes whereas iHypervisor does at the granularity of cores and sockets. Both will negotiate with each other till they reach an agreement.
- It is an event based scheduling which means iScheduler makes a scheduling decision only when it is triggered by receiving a resource offer from iHypervisor. It is only at the start when there are no jobs in the queue and during the operations when the queue may become empty that the iScheduler will have to explicitly send a request message to iHypervisor for a resource offer otherwise at all other times scheduling is event based.

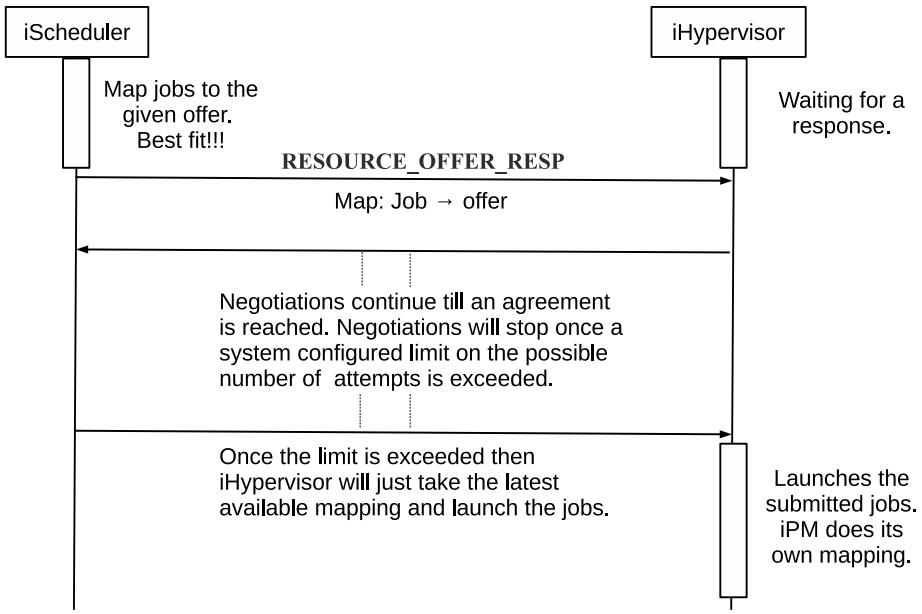


Figure 4.4: Scenario 1 contd.

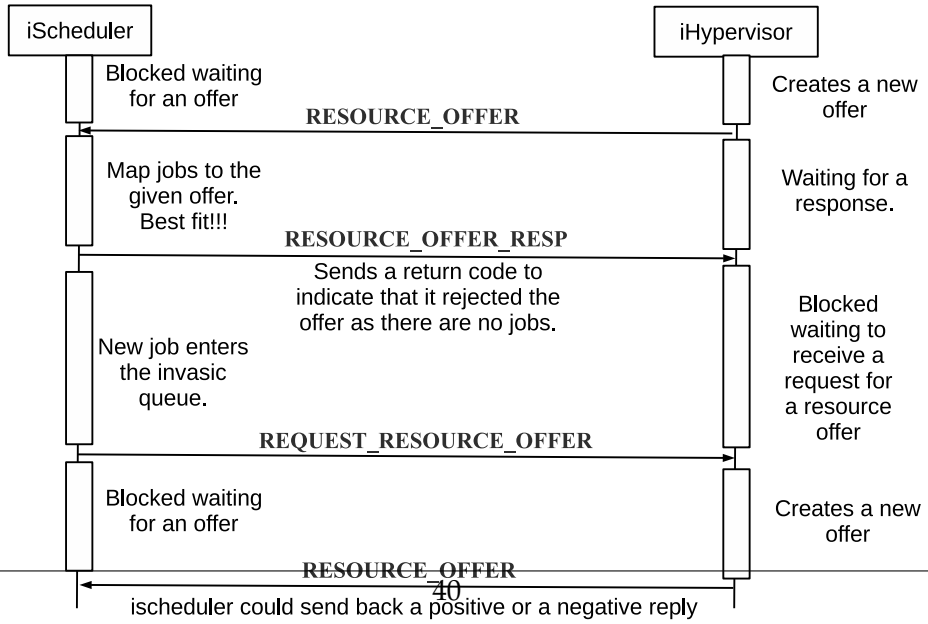


Figure 4.5: Scenario 2

- Above diagram and the ones in the following pages illustrate state machine diagrams for few of the communication phases described earlier starting first with a general diagram of how the mulithreaded component iRM agent inside iScheduler starts up and shuts down.

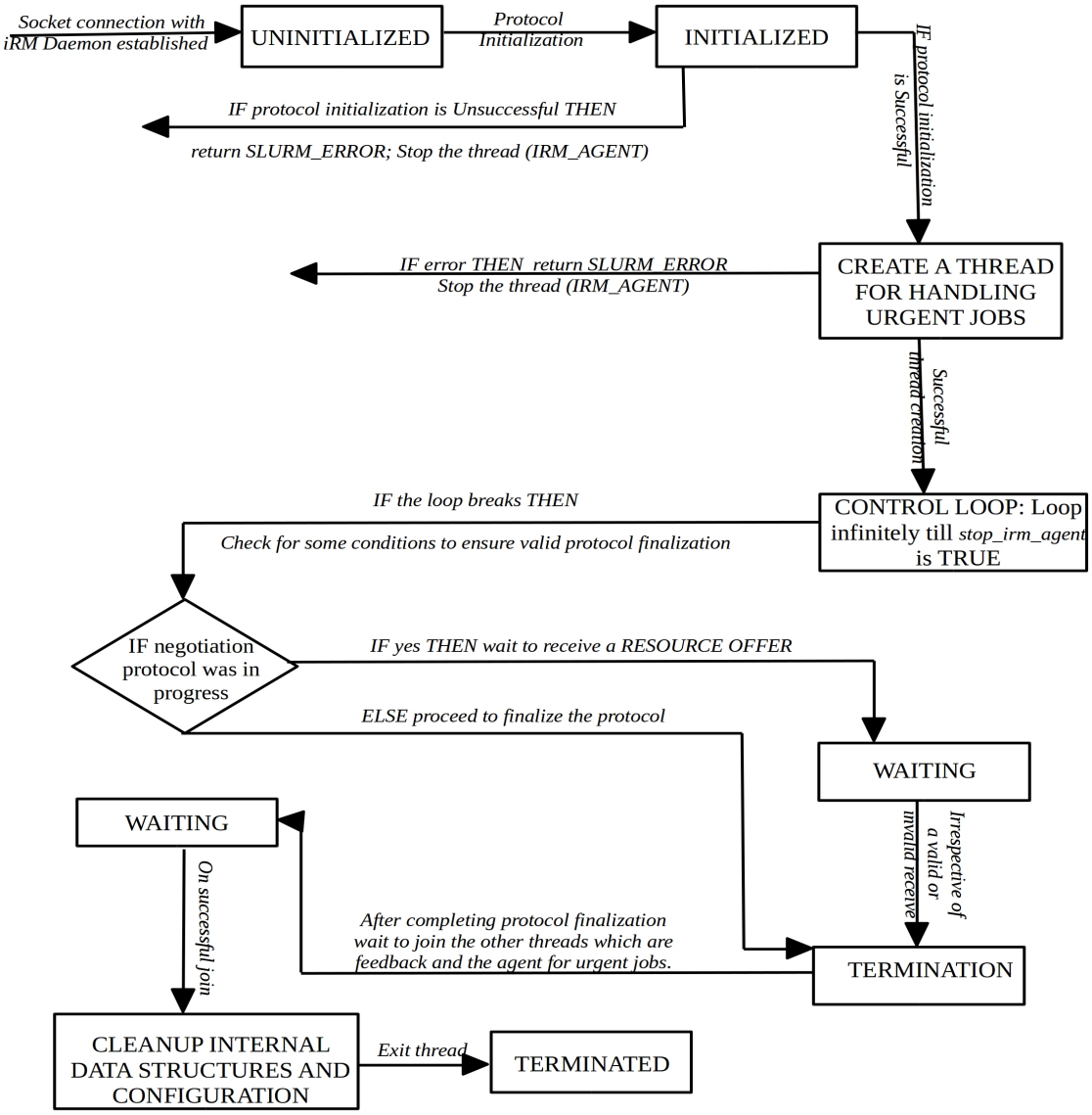


Figure 4.6: iRM Agent

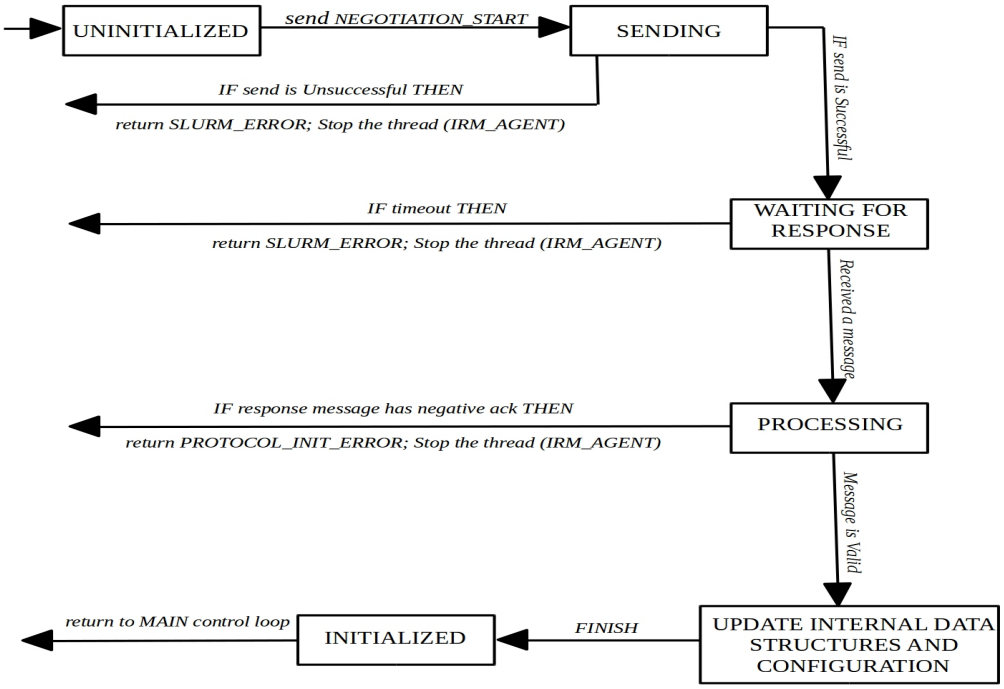


Figure 4.7: Protocol Initialization

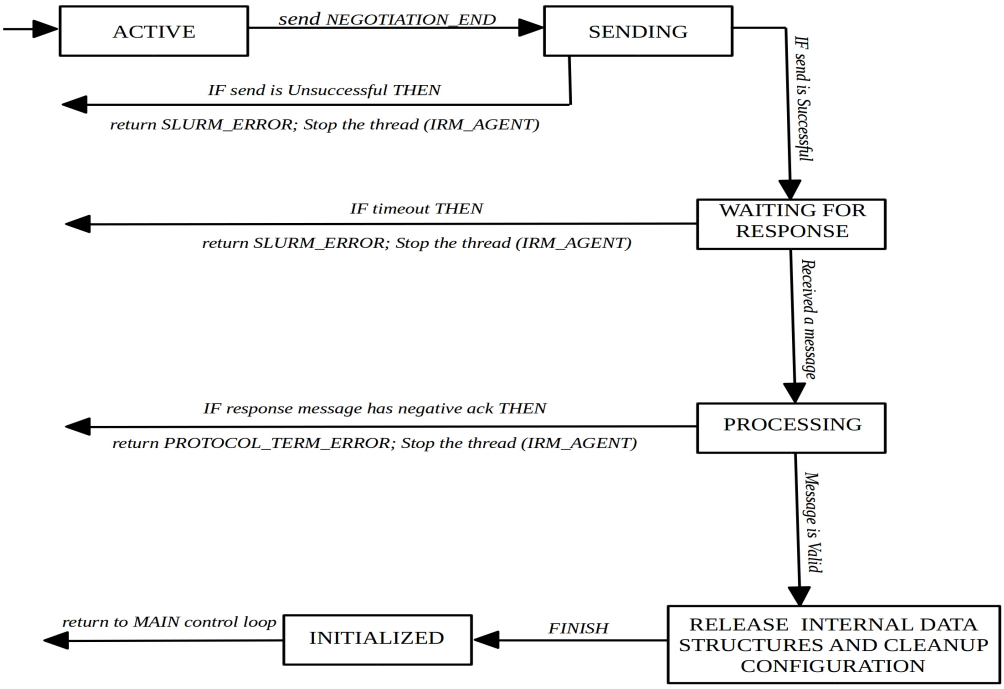


Figure 4.8: Protocol Termination

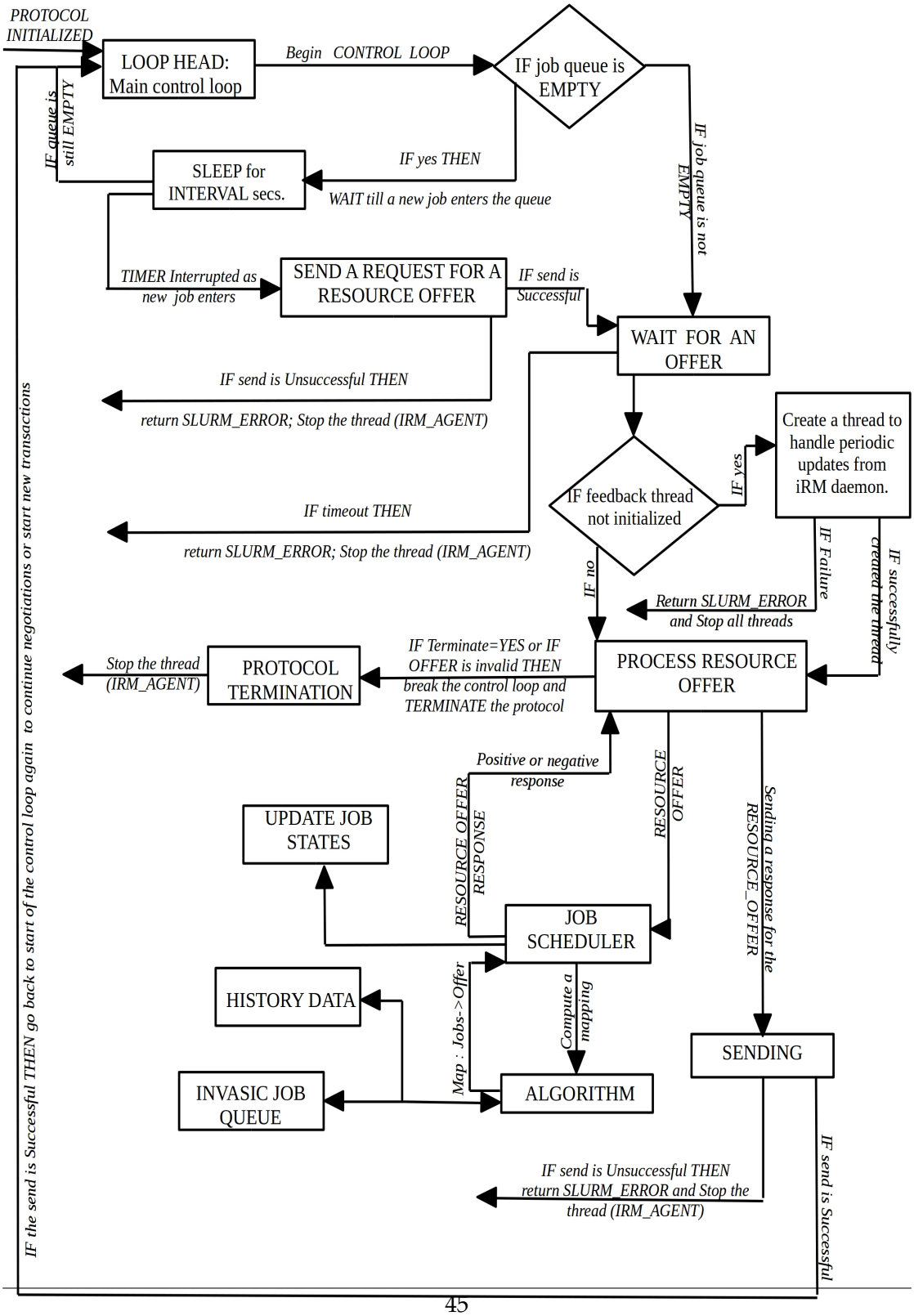


Figure 4.9: Negotiation

4.3 Invasive Jobs

5 Design["How we are building". Designing the individual modules / Components, flow charts, UML diagrams, What can it do and what it cannot

5.1 Job Mappings - Description and its generic structure

5.2 Resource Offers

5.3 Negotiation Protocol - Use state machine diagrams here

5.4 Feedback Reports

5.5 Job Scheduling Algorithms

5.5.1 Problem Formulation

5.5.2 Pseudo Code

6 Implementation

6.1 Plugin

6.2 Data Structures

6.3 Important APIs

6.4 State Machine Diagrams

6.4.1 iBSched

6.4.2 iRTSched

7 Evaluation

7.1 Method of Evaluation

7.1.1 Emulation of Workload

7.1.2 Real Invasic Applications

7.2 Setup

7.3 Experiments and Results

7.4 Performance and Graphs

8 Conclusion and Future Work

8.1 Future Work

[Pra+14] [Pra+15] [Ioa+11] [DL96] [Ure+12] [Ger+12] [Cer+10] [Mag+08] [UCA04] [KP01] [Bal+10] [YJG03] [Gup+14] [Lif95] [Sko+96] [Hun04] [Cao+10] [Zho+13] [Luc11] [TLD13] [Zho+15] [Tan+10] [Tan+05] [FW98] [FW12] [Sch]

List of Figures

List of Tables

Bibliography

[Bal+10] P. Balaji, D. Buntinas, D. Goodwell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. "PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems." In: *Recent Advances in the Message Passing Interface* (Sept. 2010).

[Cao+10] Y. Cao, H. Sun, W.-J. Hsu, and D. Qian. "Malleable-Lab: A Tool for Evaluating Adaptive Online Schedulers on Malleable Jobs." In: *Euromicro Conference on Parallel, Distributed and Network-Based Processing(PDP)* (Feb. 2010).

[Cer+10] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. Navaux. "Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic MPI." In: *International Conference on Distributed Computing and Networking(ICDCN)* (Jan. 2010).

[DL96] D.G.Feitelson and L.Rudolph. "Towards convergence in job schedulers for parallel supercomputers." In: *Workshop on Job Scheduling Strategies for Parallel Processing* (Apr. 1996).

[FW12] D. G. Feitelson and A. M. Weil. "Scheduling Batch and Heterogeneous Jobs with Runtime Elasticity in a Parallel Processing Environment." In: *International Parallel and Distributed Processing Symposium Workshops and Phd Forum* (May 2012).

[FW98] D. G. Feitelson and A. M. Weil. "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling." In: *Parallel Processing Symposium and Symposium on Parallel and Distributed Processing(IPPS/SPDP)* (Apr. 1998).

[Ger+12] M. Gerndt, A. Hollmann, M. Meyer, M. Schrieber, and J. Weidendorfer. "Invasive Computing with iOMP." In: *Forum on Specification and Design Languages(FDL)* (Feb. 2012).

[Gup+14] A. Gupta, B. Acun, O. Sarood, and L. V. Kale. "Towards Realizing the Potential of Malleable Jobs." In: *High Performance Computing(HiPC)* (Dec. 2014).

[Hun04] J. Hungershofer. "On the Combined Scheduling of Malleable and Rigid Jobs." In: *International Symposium on Computer Architecture and High Performance Computing(SBAC-PAD)* (Oct. 2004).

[Ioa+11] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra. "Phase-Based Application-Driven Hierarchical Power Management on the Single-chip cloud Compupter." In: *Parallel Architectures and Compilation Techniques(PACT)* (Oct. 2011).

[KP01] C. Klein and C. Perez. "An RMS for Non-predictably Evolving Applications." In: *Cluster Computing(CLUSTER)* (Sept. 2001).

[Lif95] D. A. Lifka. "The ANL/IBM SP scheduling system." In: *Job Scheduling Strategies for Parallel Processing* (Apr. 1995).

[Luc11] A. Lucero. "Simulation of Batch Scheduling using Real Production Ready Software Tools." In: *Iberian Grid Infrastructure Conference* (June 2011).

[Mag+08] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. "Dynamic Malleability in Iterative MPI Applications." In: *Concurrency and Computation: Practice and Experience* (Jan. 2008).

[Pra+14] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf. "A Batch System with Fair Scheduling for Evolving Applications." In: *International Conference on Parallel Processing(ICPP)* (Sept. 2014).

[Pra+15] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale. "A Batch Sytem with Efficient Adaptive Scheduling for Malleable and Evolving Applications." In: *International Parallel and Distributed Processing Symposium(IPDPS)* (May 2015).

[Sch] SchedMD. *SLURM Workload Manager*. www.slurm.schedmd.com.

[Sko+96] J. Skovira, W. Chan, H. Zhou, and D. Lifka. "The EASY - LoadLeveler API Project." In: *Job Scheduling Strategies for Parallel Processing* (Apr. 1996).

[Tan+05] W. Tang, N. Desai, D. Buettner, and Z. Wan. "Backfilling with Lookahead to Optimize the Packing of Parallel Jobs." In: *Journal of Parallel and Distributed Computing* (May 2005).

[Tan+10] W. Tang, N. Desai, D. Buettner, and Z. Wan. "Analyzing and Adjusting User Runtime Estimates to Improve Job Scheduling on the Blue Gene/P." In: *International Symposium on Parallel and Distributed Processing Symposium(IPDPS)* (Apr. 2010).

[TLD13] W. Tang, Z. Lan, and N. Desai. "Job Scheduling with Adjusted Runtime Estimates on Production Supercomputers." In: *Journal of Parallel and Distributed Computing* (Mar. 2013).

[UCA04] G. Utrera, J. Corbalan, and J. Albarta. "Implementing Malleability on MPI Jobs." In: *International Conference on Parallel Architecture and Compilation Techniques(PACT)* (Oct. 2004).

[Ure+12] I. A. C. Urena, M. Riepen, M. Konow, and M. Gerndt. "Invasive MPI on Intel's single-chip cloud computer." In: *Architecture of Computing Systems(ARCS)* (Feb. 2012).

[YJG03] A. B. Yoo, M. A. Jette, and M. Gondona. "SLURM: Simple Linux Utility for Resource Management." In: *Job Scheduling Strategies for Parallel Processing* (June 2003).

[Zho+13] Z. Zhou, Z. Lan, W. Tang, and N. Desai. "Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling." In: *Job Scheduling Strategies for Parallel Processing* (May 2013).

[Zho+15] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Wan. "I/O Aware Batch Scheduling for Petascale Computing Systems." In: *International Conference on Cluster Computing* (Sept. 2015).