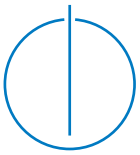
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Job Scheduling for Adaptive Applications in Future HPC Systems

Nishanth Nagendra



DEPARTMENT OF INFORMATICS

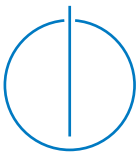
TECHNISCHE UNIVERSITÄT MÜNCHEN

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, 2016



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, 2016 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

List of Figures

List of Tables

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

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 applications 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 changing the resource set of the application during runtime or adapt to the available resources during execution.

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: ***iBSched*** (Batch Scheduler for Invasic Jobs) built as an extension into an existing batch system and ***iRTSched*** (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 5.1 shows a high level illustration of the architecture for such an Invasive Resource Management.

The above figure illustrates the proposed invasive resource management architec-

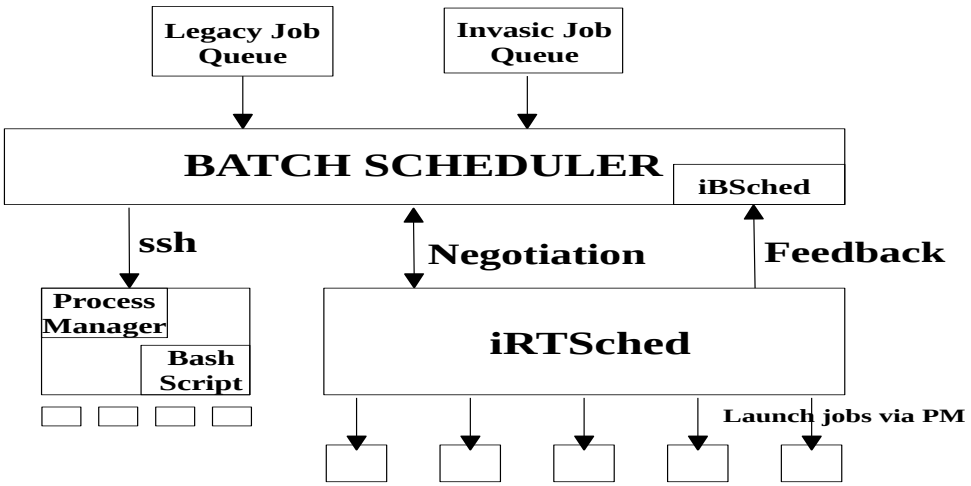


Figure 1.1: Invasive Resource Management Architecture

ture. 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 these new type of Invasic Jobs and a new component called ***Invasic Scheduler (iBSched)*** 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 (iRTSched)*** 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 iBSched and iRTSched. iRTSched talks to the iBSched via a protocol called the Negotiation protocol to receive Invasic Jobs dispatched from iBSched 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. iBSched is responsible for scheduling Invasic Jobs. The scheduling decisions are communicated via negotiation protocol to iRTSched 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 iRTSched which creates these resource offers based on the state of the partition. Upon receiving a resource offer, iBSched 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 iRTSched 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 iBSched. Similar to iRTSched, the iBSched 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 iRTSched and iBSched make their decisions on and hence both will be involved in some kind of a negotiation via the protocol to reach a common agreement. iRTSched 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.

1.3 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:

- ***Related Work:*** This section will briefly mention some of the earlier research efforts that have been made in the direction of batch job scheduling, runtime scheduling specifically to support adaptive applications and resource-aware programming paradigms to implement adaptive applications.
- ***Invasive Computing:*** This section will first introduce the concept of invasive computing in brief and explain the motivation behind this concept. This is followed by an elaborate description of the traditional resource management

approach in order to contrast it with the following section on invasive resource management that is necessary to support invasive computing.

- ***Architecture:*** This section will present an 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. This "how" is tackled in the following section of design.
- ***Design:*** This section will present the details on how we are building the system whose architecture was illustrated in the previous section. It deals with the internal details of the individual modules / components, flow charts and illustrations. It describes what it can do and what it cannot.
- ***Evaluation:*** This section will cover the evaluation of the work presented in this thesis. It will describe the approach used for evaluating the system in order to test its functionality, correctness and performance.
- ***Conclusion and Future Work:*** This last section concludes the report on this thesis 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

Batch Scheduling and Runtime Scheduling

3 Invasive Computing

Invasive Computing is a novel paradigm for designing and programming future parallel systems. Decreasing feature sizes are motivating a redesign of multi million transistor system-on-chip architectures. This can lead to a dramatic increase in the rates of temporary and permanent faults as well as feature variations. SoCs with 1000 or more processors on a single chip in the year 2020 are foreseen, hence static and central management concepts to control the execution of all resources are no longer appropriate. Invasive Computing allows a program to be resource-aware by which it can explore and dynamically spread its computations to neighbour processors in a phase called as ***invade***, then to execute portions of code of high parallelism degree in parallel based on the available invasible region in a given multi-processor architecture in a phase called as ***infect***. Later, once the degree of parallelism should be lower again or if it terminates, the program may enter a ***retreat*** phase where it can deallocate resources and resume execution again, for example, on a single processor. With the help of such resource awareness, the program has the ability to self-organise itself and be immune to faults, feature variations, be highly scalable, show performance gain and record a higher resource utilization metric.

This concept would require not just new programming concepts, languages, compilers and operating systems but a radical change in the architectural design of MPSoCs(*Multi-Processor Systems-On-a-Chip*) so as to efficiently support invasion, infection and retreat operations. Some of the main motivations behind the idea of invasive computing are enumerated below:

- ***Programmability*** How to map algorithms and programs to 1000 processors or more and how to benefit from the massive parallelism available by tolerating manufacturing defects, feature variations etc.
- ***Adaptivity*** Modern applications have unpredictable resource requirements most of which may not be known at compile-time. In addition to this, when different applications are running on a single chip, resource distribution will have to happen dynamically keeping up a high resource utilization and performance. These factors show the need for some sort of hardware / software reconfigurability of the MPSoC.

- ***Scalability*** How to efficiently run algorithms and programs on different number of resources?
- ***Physical Constraints*** Heat dissipation will be a major bottleneck. Intelligent methods and architectural support to run algorithms at different speeds to exploit parallelism under power reduction is needed.
- ***Reliability and Fault-Tolerance*** Applications must be immune to temporal or permanent faults that may be caused due to manufacturing defects, feature variations, degradation etc. This especially has a higher likelihood to happen in the case of future MPSoCs.

The paradigm of invasive computing offers a new perspective for programming large scale HPC systems. Current resource management systems manage resources via static partitioning among parallel jobs. This is a very rigid approach considering that an application will then be limited to a fixed amount of parallelism it can utilize. This, however, will not be beneficial especially in the case of future exascale systems where if one needs to derive maximum performance then the maximum number of resources will have to be allocated. The application can benefit from invasive programming during certain phases of its runtime by running at maximum parallelism and in the remaining time it can run at a lower parallelism.

Another motivation is for a specific classes of applications like multi-grid and adaptive grid. Multi-grid applications work on multiple grid levels ranging from fine to coarse grids. On fine grids, more resources could yield better performance and efficiency, whereas on coarse grids fewer resources would be sufficient. In the case of adaptive grid applications, the grid is dynamically refined according to the current solution and the application may go through different levels of parallelism in different phases.

Scaling the systems to exaflop level would consume significantly more power that would very likely cross a gigawatt, roughly the output of Hoover Dam. Reducing the power requirement by a factor of atleast 100 is a challenge for future hardware and software technologies. Invasive computing concept with invasive programming models combined with intelligent resource management and flexible scheduling mechanisms can possibly help in addressing this challenge.

Coping with run-time errors would be another major challenge. Due to design and power constraints, the clock frequency is unlikely to change and feature sizes would continue to decrease as per moore's law for the next few years. By 2020, it is envisaged that exascale systems can possibly have approximately one billion processing elements.

An immediate consequence is that the frequency of errors will increase while timely identification and correction of errors would be much more difficult. Fault tolerance would be one of the most important challenges in this regard.

Exploiting massive parallelism for current and emerging scientific applications would also be another major challenge.

3.1 Traditional Resource Management

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 almost a personal computer. MPI would typically be used to manage communications within the parallel program. A resource manager allocates resources within a cluster, launches and otherwise manages the 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.

3.1.1 Classification

The process of computing a schedule may be done by a queueing or planning based scheduler. A *Schedule* is computed for the job requests that are present in the job queue. Every request contains information such as the number of requested resources and a duration for how long the resources are requested for. A job consists of a request and some additional information about the associated application. These additional details could be the following: 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 etc. There can also be some reservation requests present. These request for resources at a specified time for a

given duration. Once the scheduler accepts such a request, it is a reservation and those exact resources are then blocked for that specified time and are unavailable for any scheduling purposes.

The classification of resource management systems is based on the planned time frame [42]. Queuing systems try to utilize the currently available resources in order to satisfy the job requests. Future resource planning for all the pending requests is not done. Hence, the pending requests have no proposed start times. Planning systems in contrast, plan for the present and the future. Planned start times are assigned to all requests and a complete schedule about the future resource usage is computed.

Queuing Systems These systems have several queues with different limits on the number of requested resources and the runtime limit for the job. 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.

In queuing systems, no information about future job starts are available. Consequently guarantees can not be given and resources can not be reserved in advance. Resource reservation will have to be done manually by the administrative staff. Usually high priority queues combined with dummy jobs for delaying other jobs are used. Job requests also come with run time limits. 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. Also, if a job runs for more than the run time limit it specified, then the system will usually kill such a job.

Planning Systems Planning systems schedule for the present and future. They assign start times to all requests and a full schedule is generated. Runtime estimates for jobs are mandatory for this planning. With this knowledge advanced reservations are easily made possible. 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 pending requests are sorted according to a scheduling policy in addition to clearing their previous planned start times. All the pending requests are then re-inserted at the earliest possible start time in the schedule. After this step each request is assigned a planned start and end time.

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 standard job requests which can float around in the schedule. It does not have to be sorted as all reservations are accepted and therefore generate no conflicts in the schedule.

One drawback in a planning system is the cost of scheduling. Furthermore, The usage of reservations should be monitored as users may misuse this facility without really needing it. This can be avoided by automatically releasing unused reservations after a specific idle time.

3.1.2 Job Scheduling

Typical resource management systems store job requests in list-like structures. 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 and FCFS is probably the most known and used scheduling policy as it simply appends new requests at the end of the queue. With this example the term fairness is described [79].
- by duration: Both increasing and decreasing orders are used. Sorting by increasing order leads to SJF (shortest job first) respectively FFIH (first fit increasing height). 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. FCFS, SJF, and LJF use this approach.
- first fit: 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. 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.

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 has become a 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 job schedulers for parallel supercomputers employ a straightforward version of variable partitioning. (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.

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

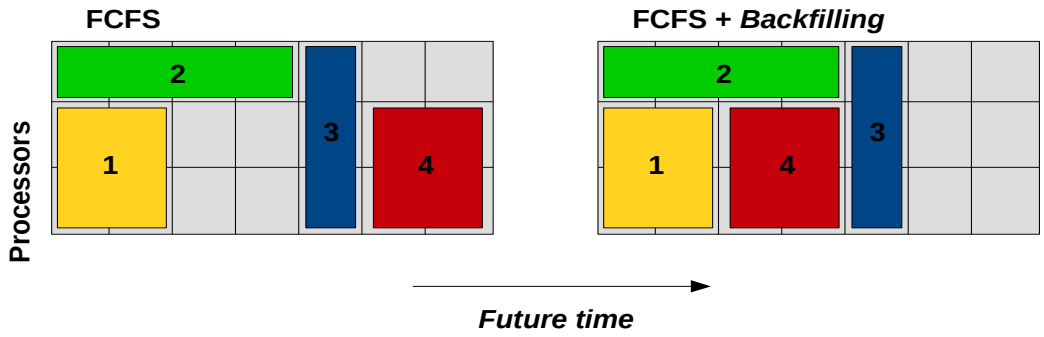


Figure 3.1: FCFS with and without Backfilling

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). 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.

Variations on Backfilling This section briefly mentions some of the various tunable knobs of backfilling algorithms.

Parameter	Description
Number of reservations	Default is ***1***. This is called "***Aggressive Backfilling***" where only the first queued job receives a reservation. This can cause delays in execution of the other waiting jobs. The alternative is "***Conservative Backfilling***" where other waiting jobs are also allocated reservations. The number of reservations to allow is configurable by the administrator.
Looseness of reservations	This refers to a "***Selective Reservation***" strategy depending on the extent different jobs have been delayed by previous backfilling decisions. This is similar to "***Flexibe Backfilling***" strategy where the backfilling is allowed to violate the reservation upto a certain slack.
Order of Queued Jobs	Usually FCFS order is used. An alternative is where the jobs are prioritized in some way and the scheduler picks the jobs according to this order including for backfilling. The factors on which priority could be calculated are: job characteristics, user, group, user priority, fairness, weight, duration etc.
Partitioning of Reservations	Machine is divided into several disjoint partitions with the freedom to move around idle processors dynamically based on current needs. Each partition is associated with its own job class, runtime limit for jobs, wait-queue and reservation. Backfilling candidate is chosen in a round-robin fashion, each time from a different partition respecting the reservations.
Adaptiveness of Backfilling	Simulates the execution of recently submitted jobs under various scheduling disciplines and switches the algorithm to the ones which gives the highest performance.
Lookahead into the queue	Default behavior is to consider the queued jobs one at a time that may lead to loss of resources. Alternative is to look at a window jobs at a time and pick a mapping that gives the maximum utilization but respects all the reservations.
Speculative Backfilling	The Scheduler is allowed to backfill jobs even if it interferes with an existing reservation as it speculates that the job will finish earlier than estimated time.
Minimize the electric power demand	The current direction of research in job scheduling is targeted towards power efficient exascale HPC systems. Scheduling decisions will have to be made considering power constraints of the machine, power stability, energy efficiency.

Figure 3.2: Backfilling Variations

3.1.3 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 written in C whose development started in the year 2002 at Lawrence Livermore National Laboratory as a simple resource manager for Linux Clusters. A few years ago it 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 flavors like AIX, Linux, Solaris and is also fault tolerant, highly scalable, and portable.

SLURM has a centralized manager called ***slurmctld***(controller daemon) that is the main nerve center of SLURM. SLURM operates in a style simlar to the Master-Slave paradigm where the Master is the ***slurmctld***. It takes centralized decisions to monitor resources and work. In the event of a failure, there may also be a backup controller. Each of the nodes in the cluster has a daemon running on it called as ***slurmd*** and these are the slaves. These daemons are started on every node and they are responsible for monitoring them. This can resemble a remote shell: it waits for work from the controller, executes that work, returns status and waits for more work. The *slurmd* daemons provide fault-tolerant hierarchical communications and also are responsible for spawning an additional daemon called ***slurmstepd***. The step daemon as it is called is responsible for the node local part of the job step that are the subset of processes running on the local node. A job step in SLURM refers to an application started with the help of srun and its allocated resources. srun could be used independently to launch jobs or one can specify the same within a batch script while using sbatch. ***srun*** is one of the tools SLURM provides that allows the user to launch interactive jobs on the cluster, ***sbatch*** to launch batch jobs and several others relating to accounting, job status, cancellation operation etc.

The figure x.x shows the high level architecture of SLURM with the interaction between the several of its key components. It also shows the interaction between an MPI application through the PMI(Process Management Interface), *slurmd* daemon of a node and the *slurmstepd*. **Plugins** are dynamically linked objects loaded at run time based upon configuration file and/or user options. 3.3 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.

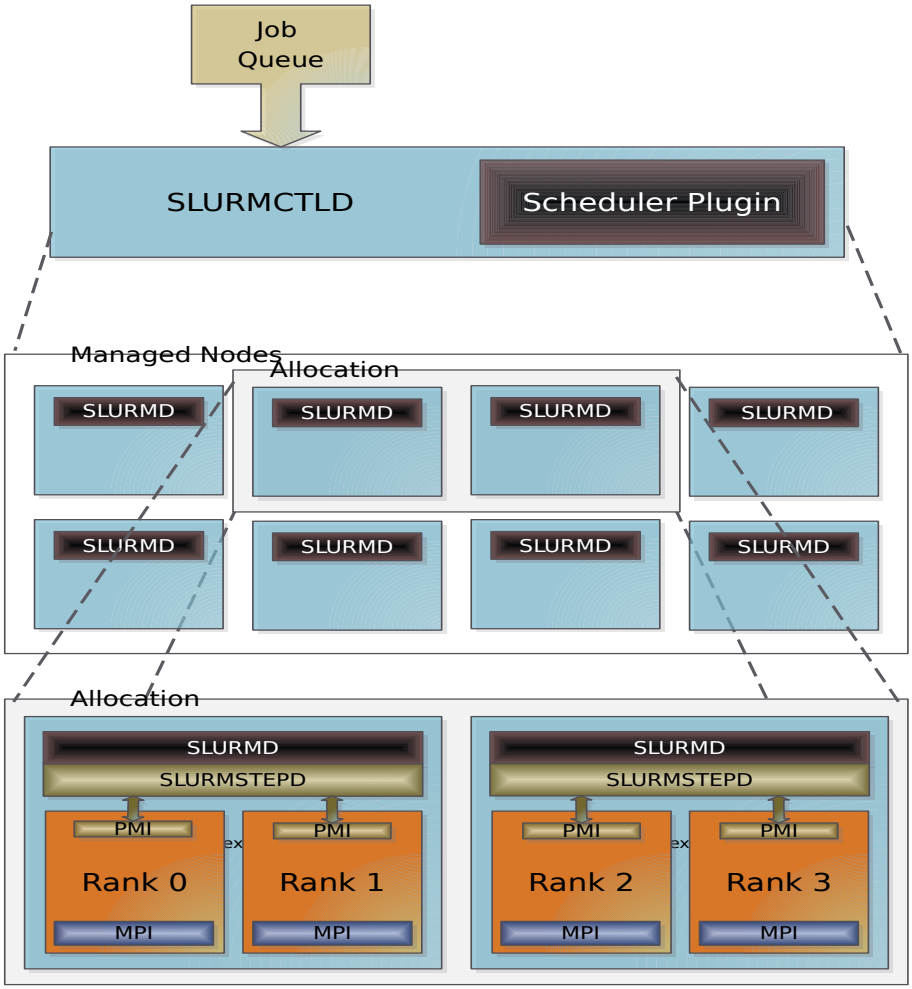


Figure 3.3: SLURM Architecture

- ***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.

3.2 Resource Aware Programming

3.2.1 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.

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.

3.2.2 Invasive Programming Models

In this section, we will briefly look into the details of the earlier invasive extensions done 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. These invasic extensions provide us with a new parallel programming model that

allows us to implement resource aware programs. Depending on the semantics of these new extensions, the resulting application can either be evolving(application dictates the changes to its resource set) or a malleable job(resource manager dictates the change in resources to which the application must adapt). Resource awareness could mean that either 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.

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 library to allow for such adaptive operations of an invasive parallel application.

iOMP
The OpenMP parallel programming model for shared memory systems was extended to support the programming of resource aware applications and is named as Invasive OpenMP or iOMP. 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 and the most common approach is through the environment variable, or through OpenMP library call or as an additional clause in its directives. It has language bindings for C, C++ and Fortran. iOMP has been implemented as a library in C++ using an object oriented approach and provides two important methods / operations 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*(not the C++ Class) in iOMP refers 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 important part of the iOMP implementation, 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.

iMPI

Similar to **iOMP**, previous research effort in this project was also directed towards extending parallel programming models for distributed memory systems. iMPI which stands for Invasive MPI is an extension to the MPI library that can support resource aware programming. The Single-chip Cloud Computer(SCC) from Intel Labs was an

experimental CPU that integrates 48 cores and is basically a distributed memory system on the chip. This hardware platform along with its interesting memory features was used in order to evaluate this invasive programming model.

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.

The following are the invasive extensions to MPI:

- ***MPI_Comm_invade***: The main purpose of this operation is to reserve resources and for this it looks into what resources are currently available and invades them.
- ***MPI_Comm_infect***: This operation is used by the application to specify the total number of cores to infect and which ones are preferred. This number can be less than or equal to the total number of cores that were reserved by the invade operation.
- ***MPI_Comm_retreat***: This operation does the reverse of the invade+infect sequence. Instead of reserving and claiming resources, it returns them so that other invasive applications can claim them.

The above extensions were based on the MPICH2 library. A new process manager called **Invasive Process Manager(IPM)** was also developed as a part of the iMPI implementation. It was responsible for launching of the MPI jobs, as well as spawn operations(invade+infect) with low latency and functionality for resource awareness.

Other Resource-Aware Programming Models

3.3 Invasive Resource Management

In this section, we will look at the latest extensions done to the MPI library for programming on HPC systems like clusters, supercomputers etc and also the extensions done to a resource managers managing these HPC systems. This is in contrast to the earlier version of the iMPI which was targeted towards the Intel SCC platform. The MPI and Resource Manager extensions are not accomplished as a part of this thesis but are essential to be described here in order to get the right context for adaptive applications and their scheduling.

3.3.1 Invasive MPI

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 shortcoming 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();
```

3.3.2 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 SLURM components have been extended:

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 specific to SLURM:

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 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

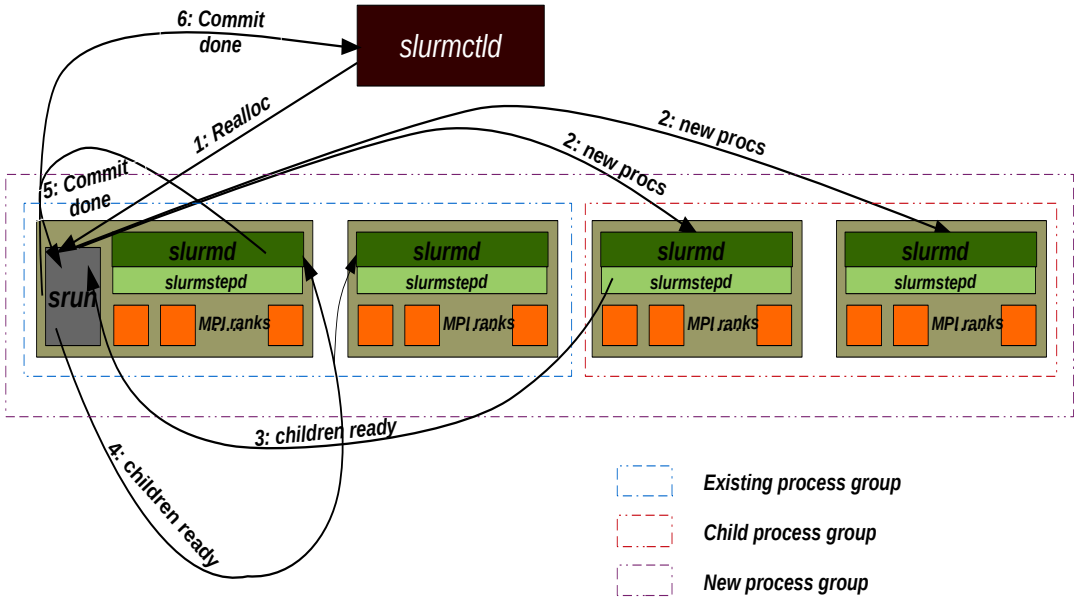


Figure 3.4: SLURM Extension

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

This section illustrates and describes the high level architecture of the software implemented along with a few protocol sequence diagrams. It will help to understand at a high level about the components which form a part of this system to support the new approach of invasive computing and how will its components interact with each other through new protocols or extensions of existing protocols in order to integrate such an invasive resource management into existing 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 into 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). 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 "iRTSched" will be implemented which needs to communicate with the batch scheduler and influence the scheduling decisions taken by it. The iRTSched sits between the top layer and the process manager.

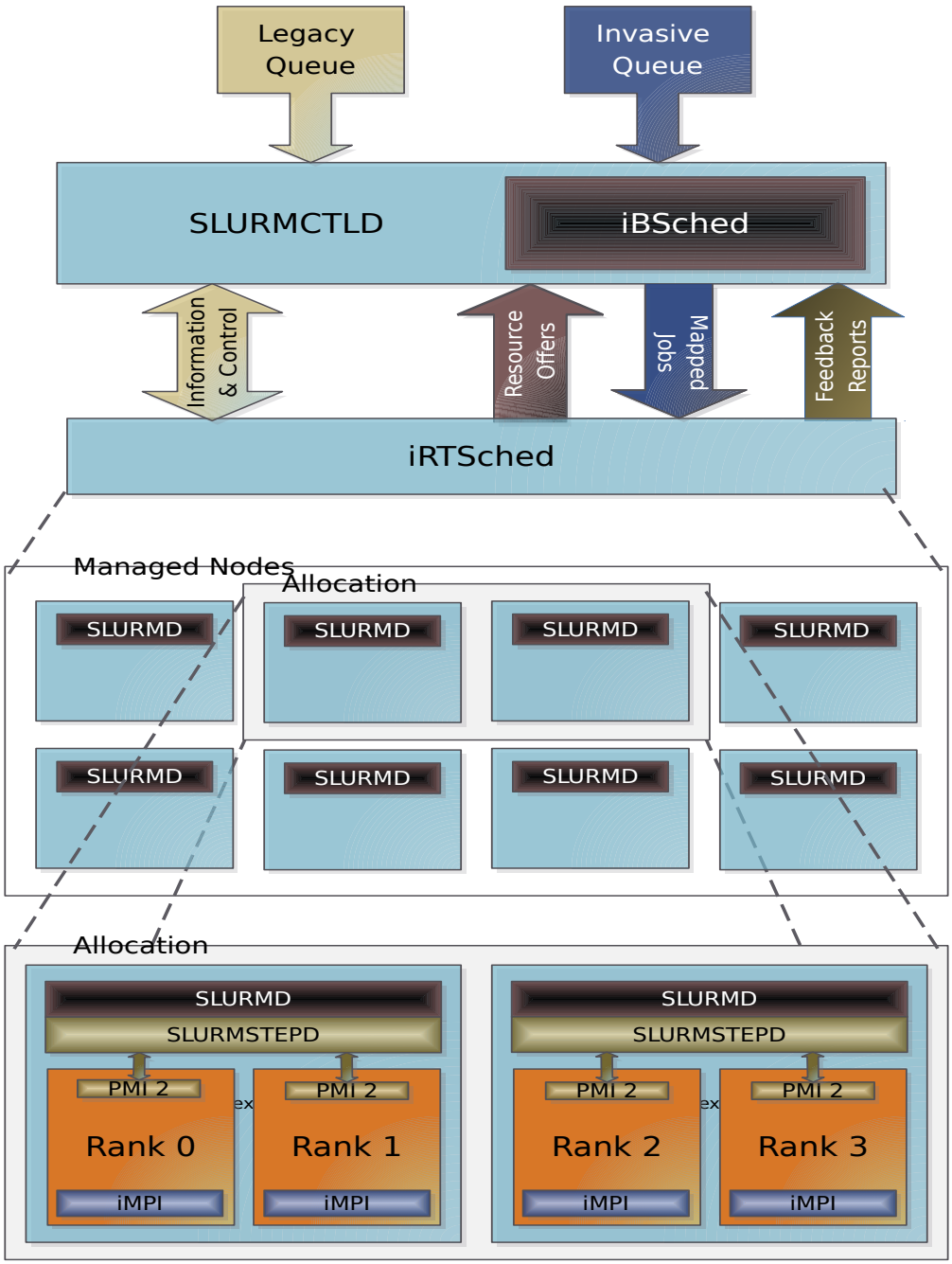


Figure 4.1: Invasive Resource Management Architecture

- 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 "iBSched" will be implemented for SLURM to handle job scheduling specifically for invasic jobs.

- Communication between iRTSched and iBSched will involve the negotiation protocol as explained in the previous chapter but will also include periodic / event-driven feedbacks being sent by iRTSched to iBSched. These will contain some useful information about the current state of the running jobs, their energy consumption, other job characteristics etc. 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 (iBSched and iRTSched) 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 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 the communication between iRTSched and iBSched 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 iRTSched and iBSched 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 will begin at some later point of time.

<EVENT> <=> <PACKET>	DIRECTION OF COMMUNICATION
<REQUEST_RESOURCE_OFFER>	$iBSched \rightarrow iRTSched$
<RESOURCE_OFFER>	$iRTSched \rightarrow iBSched$
<RESPONSE_RESOURCE_OFFER>	$iBSched \rightarrow iRTSched$
<NEGOTIATION_START>	$iBSched \rightarrow iRTSched$
<RESPONSE_NEGOTIATION_START>	$iRTSched \rightarrow iBSched$
<NEGOTIATION_END>	$iBSched \rightarrow iRTSched$ $iRTSched \rightarrow iBSched$
<RESPONSE_NEGOTIATION_END>	$iBSched \rightarrow iRTSched$ $iRTSched \rightarrow iBSched$
<STATUS_REPORT>	$iRTSched \rightarrow iBSched$
<URGENT_JOB>	$iBSched \rightarrow iRTSched$
<RESPONSE_URGENT_JOB>	$iRTSched \rightarrow iBSched$

Figure 4.2: Message Types

- ***Feedback:*** This concerns the periodic / event-driven feedback sent by the iRTSched to the iBSched containing useful information as mentioned earlier. iRTSched will also send a performance model of every completed job in the feedback that can be stored in some database as a part of the history of executions for this job. This will help the iBSched in the future if the same job is submitted when there will be additional performance specific information available about this job that can be used by the batch scheduling algorithm to make better decisions for this job. 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.

4.1 Dynamic Resource Management

Separation of Concerns: In this thesis, We explore the idea of separating the concerns of batch and runtime scheduling into two different software layers / components in contrast to the existing systems where both are merged together. A negotiation protocol will be implemented as a means of communication between the two. The motivation behind the negotiation protocol is the conflicting set of objectives between batch scheduler(user perspective: faster response time, fairness for jobs etc.) and run time scheduler(system perspective: maximize utilization, throughput, energy efficiency etc.). The role of the batch scheduler is to forward jobs to a runtime scheduler which is managing the resources in the partition(s) and supports runtime scheduling of adaptive applications. Runtime scheduler will make intelligent expand / shrink decisions by observing scalability behavior of the running applications and use it to predict future resource requirements. The proposed protocol will also allow us to integrate such adaptive resource management systems into existing batch systems and thereby allowing for an easy migration from legacy systems to invasive resource management systems. Negotiation also helps us to realize a dynamic and flexible scheduling strategy by balancing the conflicting objectives at the two layers.

We will briefly look at the important components and their respective roles in this architecture in order to support adaptive applications on HPC systems by dynamically managing the resouce allocation of running jobs.

4.1.1 Invasive Batch Scheduler

This component will be an extension to the existing batch systems for batch scheduling of invasive jobs. The scheduling decisions are communicated via the negotiation protocol to iRTSched. 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. Batch scheduler is making its decisions to optimize for certain local metrics such as reduced job waiting times, fairness, deadlines, priorities etc. The purpose of a batch scheduler is to select jobs from its queue according to some algorithm and dispatch this to the runtime scheduler. This is similar to a long term scheduler which by its definition: admits new processes to the system and controls the degree of multiprogramming(number of processes in memory). The batch scheduler for HPC workloads also does something similar but it is also merged with the runtime scheduler. In this work, we have separated the two because of which it now looks analagous to a long term scheduler(batch scheduler) that admits jobs into the HPC system and a short term scheduler(runtime scheduler) which will manage the running jobs. Batch

scheduler runs less frequently and there may be quite a lot of time gap until the next job may be admitted into the system. This depends upon available resources, any events like job termination, completion where we can expect to receive a resource offer from the runtime scheduler.

This batch scheduler is termed as invasive batch scheduler because it supports the negotiation interface to talk to an invasive run time scheduler. It will now consider a mix of job types which are: *malleable, moldable, rigid, and evolving* and perform the following tasks as per the existing functionalities provided by SLURM:

- It is based on an event-driven batch scheduling where scheduling decisions would be made only when an offer is received from iRTSched. Scheduling is performed here at the granularity level of nodes.
- Dispatch jobs from the batch queue that have been specifically submitted to the invasic partition according to a scheduling algorithm. This algorithm will consider the resource constraints of the job that can include the number of nodes(exclusive / shared), memory size, duration, quality of service parameters etc. Scheduling here is done at the granularity of nodes.
- With every negotiation attempt, the batch scheduler will try to relax the constraints of those jobs which could not be mapped to the available resource offer from the runtime scheduler. As the negotiation attempts increase the degree to which the constraints are relaxed also increases. By doing this the batch scheduler is trying to bargain as best as it can in order to get an offer from iRTSched to best fit as many jobs as possible.
- It will process the feedbacks received from iRTSched and update the job details in the queue such that it is then visible to the administrator in the graphical user interface provided by SLURM to view the current status of the jobs.
- The batch scheduler can also process jobs which have an urgent priority. These jobs will receive special service and are going to be dispatched immediately to iRTSched. It will also consider jobs that may have some quality of service requirements such as deadlines and start time.
- Based on the negotiation and the offers received from the iRTSched, the batch scheduler may try to bias its scheduling decisions towards certain jobs.
- If a job that is in the queue has already been submitted in the past and there is some history stored in the system about it, Then the batch scheduler will use the

history information from the database to update the job details. This can lead to better scheduling decisions for this job.

4.1.2 Invasive Run Time Scheduler

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 runtime scheduler is like a short term scheduler that manages the running jobs via space sharing, time sharing or both. The runtime scheduler here is invasive because of the different job types it handles and the ability to support elastic execution of adaptive jobs. It can now redistribute resouces to running jobs by either expanding or shrinking them based on their performance and scalability. Following points describe the role of iRTSched:

- It manages all the resources in the partition and is responsbile for runtime scheduling at the granularity of cores / sockets, resource management and process management(infrastructure to launch, adapt and monitor parallel jobs).
- It will use an intelligent expand / shrink algorithm to dynamically adjust the resources of running jobs which are adaptive in nature.
- Runtime scheduler will bias its decisions towards maximizing throughput, energy efficiency, optimize for topology, resource utilization etc.
- Similar to iBSched, iRTSched will increase the degree of transformation of the runtime state of the system to better serve the batch scheduler. As the negotiation attempts increase, the scheduler will try its best to fit every batch job forwarded by performing an aggressive transformation of the running jobs. Aggressive here means that the scheduler will try to either expand / shrink jobs to their maximum / minimum.
- In addition to expand / shrink strategy, The runtime scheduler can support scheduling algorithms like backfilling, gang scheduling, preemption etc.
- iRTSched will build a performance model of the running application or will refine an existing model if it was forwarded by the batch scheduler in the job details. It will also make its scheduling decisions using this performance model to estimate the scalability, performance and efficiency of the application at different job sizes.
- It will send the generated models and other relevant information about a completed job like runtime, energy consumption, IO consumption, performance model in a feedback report to the iBSched.

4.1.3 iMPI Process Manager

The iMPI process manager in this architecture refers to multiple components as they all are necessary to successfully provide an elastic execution framework for adaptive applications. These components are ***srun***, ***slurmd*** and ***slurmstepd***. The previous chapter already described as to how the extended MPI library called as iMPI allows the programmer to create invasive MPI applications and how the extensions on the resource manager side in coordination with iMPI provide a complete infrastructure necessary for the elastic execution of adaptive applications.

4.2 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.

4.2.1 Protocol Sequence Diagrams

- Above diagram illustrates a scenario where both iBSched 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 iBSched for a resource offer that will happen when new jobs arrive.
- iBSched makes scheduling decisions at a coarser level of granularity which is nodes whereas iRTSched does at the granularity of cores and sockets. Both will negotiate with each other till they reach an agreement.

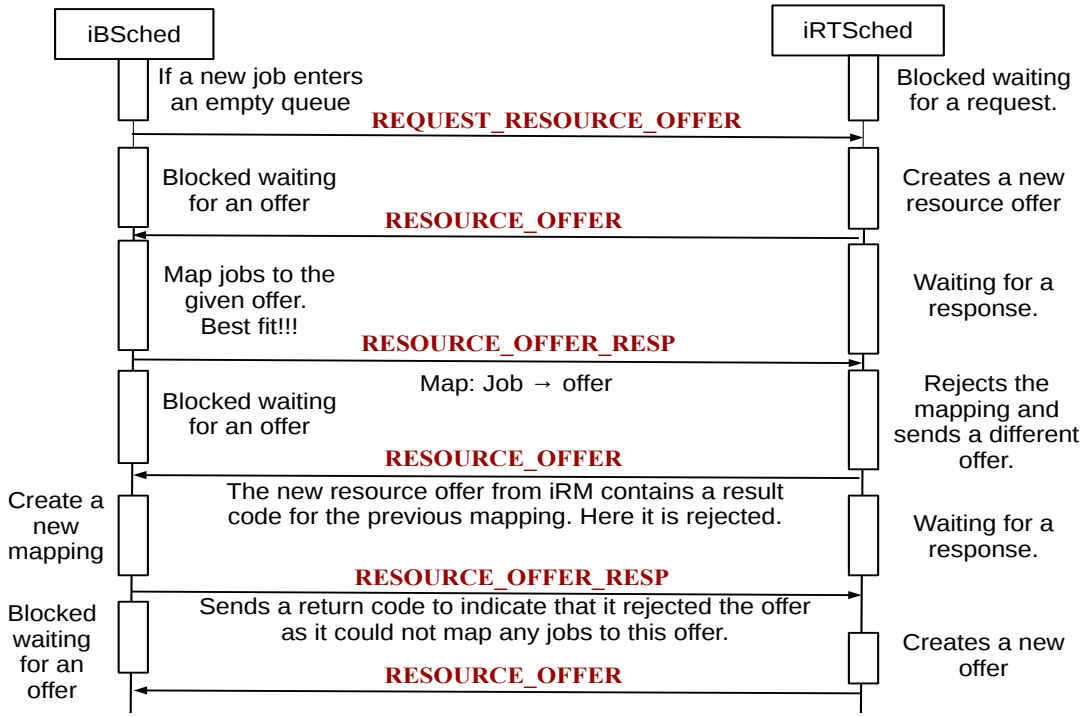


Figure 4.3: Scenario 1

- It is an event based scheduling which means iBSched makes a scheduling decision only when it is triggered by receiving a resource offer from iRTSched. 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 iBSched will have to explicitly send a request message to iRTSched for a resource offer otherwise at all other times scheduling is event based.

4.3 Invasive Jobs

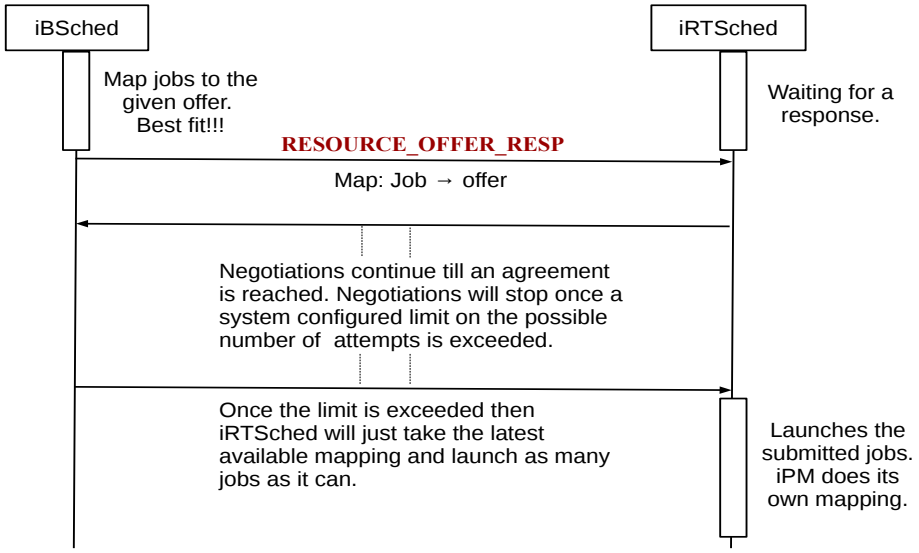


Figure 4.4: Scenario 1 contd.

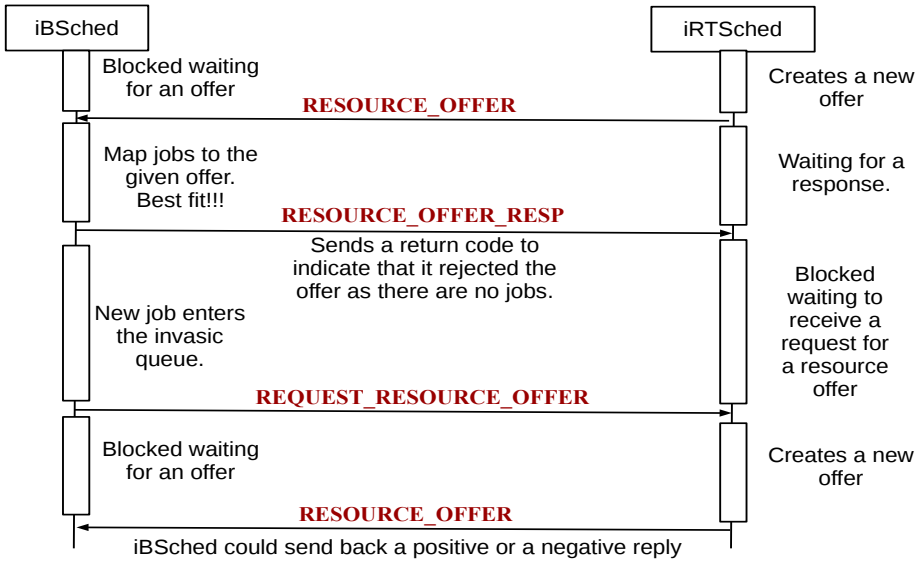


Figure 4.5: Scenario 2

5 Design

In this section, we will describe the internal details of the architecture specifically on how the scheduling algorithms work in order to realize the negotiation, the meaning of job mappings, resource offers and feedback reports.

5.1 Entities

A design entity is an element(component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced. They result from the decomposition of the software system requirements. The objective is to divide the system into separate components that can be considered, implemented, changed, and tested with minimal effect on other entities. Examples of design entities are: ***protocol, application layer, state machine, data model, object, task, sub-systems, modules*** etc.

Job Mappings
A job mapping is basically a job and its allocated set of resources. A forward job record is a job mapping and other related job details important for its launch. A list of such forward job records are sent by the batch scheduler in response to receiving a resource offer from the runtime scheduler.

$$
\begin{aligned}
M_i &:= \textit{Job Mapping} := \{jobid, nodecount, bitmap\} \\
Map_i &:= \textit{Forward Job Record} := \{M_i, details_i\} \\
Map &:= < Map_1 >< Map_2 > ... < Map_n >
\end{aligned}
$$

In the above description $details_i$ refers to the details of any i^{th} job such as *constraints, priority, runtime estimate, max nodes, min nodes, job name, start time, deadline*. In these details, the constraints are specific to the purpose of negotiation and this can be related to node count that the job is requesting for, memory size, exclusive / shared resource access etc. Currently, we support only the constraints for node count. The node count constraint will again have a min and max both of which will fall within the window of max and min nodes of the job. It is using these constraints that the batch scheduler will negotiate with the runtime scheduler and with every negotiation attempt it will

relax the constraints to bargain better. By realxing the constraints we mean that the min constraint of node count will be lowered.

Resource Offers
A reservation is basically a job being guaranteed a set of resources for a certain period of time. A resource offer is a list of such reservations and the set of nodes which are free in the partition after the runtime scheduler has guaranteed these reservations. The runtime scheduler will send such a resource offer to the batch scheduler if it rejected its mapping or it is sending a new offer that it generated.

$$Resv_i := \textit{Job Reservation} := \{jobid, nodecount, bitmap, starttime, endtime\}$$
$$IdleNodes := \textit{Residual Nodes} := \{bitmap\}$$
$$Offer := < Resv_1 >< Resv_2 > ... < Resv_n >, IdleNodes$$

The $\boldsymbol{Resv_i}$ entry in the resource offer corresponding to a particular job id must match the job id of the $\boldsymbol{Map_i}$ entry in the forwarded jobs. This means that the sequence in which jobs were forwarded to runtime scheduler will be the same sequence of jobs for which reservations will be sent to the batch scheduler.

Feedback Reports
Periodic or event-driven feedback reports on the latest status of the running jobs are sent by the runtime scheduler to the batch scheduler. It is event-driven for events such as job termination, job completion, acceptance of a mapping received from batch scheduler.

$$\boldsymbol{P_i} := \textit{Performance Data} := \{\boldsymbol{model, energy, io, memory, hint}\}$$
$$\boldsymbol{Report_i} := \textit{Status Report} := \{\boldsymbol{jobid, nodecount, bitmap, runtime, endtime, state, P_i}\}$$
$$\boldsymbol{Feedback} := < \boldsymbol{Report_1} >< \boldsymbol{Report_2} > ... < \boldsymbol{Report_n} >$$

The feedback reports can also contain performance specific data such as energy consumption, performance model of completed jobs that can be used by batch scheduler for making better decisions on same jobs submitted again in the future. For running jobs, the reports can contain similar information but most important ones would be the current assigned node count, expected end time, memory usage etc.

5.2 Scheduling Algorithms

<Include the decision logic algorithm also here in this section> Following pages present the pseudo code for both the batch and runtime scheduling algorithms.

5.2.1 Batch Scheduling

Algorithm 1 presents a high level pseudo code of the batch scheduling algorithm. Following points describe the algorithm:

- It takes as input a resource offer from iRTSched and maps jobs from its queue to the offered resources.
- ***Line* 1**: A separate job queue is constructed by scanning the main job queue for jobs which have been submitted specifically for the invasic partition. These jobs are then sorted according to their priorities.
- ***Line* 8-34**: For every job in the invasive queue: If it is found in the reservation list from the resource offer and if it can start immediately(has a bitmap allocated) then we will just append this to the Map after creating a new forward job record.
- If it starts in future(has a NULL bitmap) then we will relax its node count constraints by a step size calculated as below and try to map it to the residual nodes.

$$step\ size = \frac{(Job.MaxNodes - Job.MinNodes)}{MAX\_NEGOTIATION\_ATTEMPTS}$$

- If ***step size*** in the above calculation comes up as **0** then we will set it as **1**.
- ***job.node_count.min = job.node_count.min - step size***. If the computed value is negative or less than the job's minimum number of nodes then we just set ***job.node_count.min = job.min_nodes***.
- If the job is not found in the reservation list then this must be a new job that was submitted after the previous dispatch of jobs was completed by batch scheduler in the previous negotiation attempt. We will directly try to map it to the residual nodes.
- If a job could not be mapped, then we will just append it to the Map by creating a new forward job record. If it was mapped, then we will set the current node count of that job to the count of the allocated nodes. The node count constraints remain the same.
- ***Line* 35**: We will try to map reserved(could not be mapped due to lack of sufficient resources in the offer) batch jobs by shrinking jobs which have been currently mapped. The shrink operation considers mapped jobs in the reverse order of their priority so that the operation can start from the lowest priority mapped job till the highest one.

Algorithm 1: Batch Scheduling Algorithm

```
   Data: Resource Offer
   Result: Map : Jobs → Offer
 1 build invasive job queue
 2 if empty queue then
 3 |   return empty Map
 4 end if
 5 sort queue by job priorities
 6 reservations ← Offer.Reservations
 7 freenodes ← Offer.ResidualNodes
 8 while not at end of the invasive job queue do
 9 |   read queue entry
10 |   if entry.jobId in reservations then
11 |   |   resv ← reservations(entry.jobId)
12 |   |   if resv.bitmap is NULL then
13 |   |   |   entry.start_time ← resv.start_time
14 |   |   |   entry.node_count ← 0
15 |   |   |   adjustNodeCount(entry.jobId)
16 |   |   else
17 |   |   |   entry.start_time ← 0
18 |   |   |   entry.node_count ← resv.node_count
19 |   |   |   entry.bitmap ← resv.bitmap
20 |   |   |   create forward job record using entry
21 |   |   |   append record to Map
22 |   |   |   mapped ← true
23 |   |   end if
24 |   end if
25 |   if mapped is true then
26 |   |   continue
27 |   end if
28 |   try mapping to the residual nodes
29 |   if successfully mapped then
30 |   |   update the residual nodes
31 |   end if
32 |   create forward job record using entry
33 |   append record to Map
34 end while
35 try_to_best_fit(Map, invasive job queue, Offer)
36 return Map
```

- Result would be a best fit mapping of the invasic jobs ready to be forwarded to iRTSched.
- ***Algorithm 2:*** This algorithm shows how the best fit mapping is computed. It considers all those jobs which could not be mapped. For every such job, it will try to shrink the mapped jobs in an increasing order of priorities until sufficient nodes have been found to map the job.
- If we were successful in mapping any new jobs by shrink operations, then we must rescan the invasive job queue to fill the Map by jobs which had been previously skipped due to a limit on the reservation depth.
- ***Algorithm 3:*** This algorithm shows how the analysis of running jobs are done in order to shrink them to satisfy a job request. Analysis will consider shrinking the jobs by a step size computed as shown below. If analysis is successful, then jobs are shrunk.

$$step\ size = \frac{(Job.node\_cnt - Job.node\_count.min)}{MAX\_NEGOTIATION\_ATTEMPTS}$$

Algorithm 2: Best Fit Algorithm

```
   Data: Map, Invasive Job Queue, Offer
   Result: Updated Map : Jobs → Offer
 1 avail_bitmap ← Offer.ResidualNodes
 2 while not at end of the Map do
 3 |   read Map entry
 4 |   if (entry.start_time EQ 0) AND (entry.bitmap NEQ NULL) then
 5 |   |   continue
 6 |   end if
 7 |   /* Try to map this job by shrinking other mapped jobs */
 8 |   try_sched(entry, Map, avail_bitmap)
 9 |   if successfully scheduled then
10 |   |   update avail_bitmap
11 |   end if
12 end while
13 if avail_bitmap not empty then
14 |   Rescan the invasive job queue to fill the residual nodes with some new jobs
15 end if
```

Algorithm 3: Try Schedule Algorithm

Data: Entry, Map, avail_bitmap
Result: Updated entry: Mapped to some nodes

```
/* The mapped jobs in the Map would be analyzed in the reverse order
   which is increasing in the priority */
```

Analyze in increasing order of priority if the mapped jobs in the Map can be shrunk to find enough nodes for entry
if *sufficient nodes available* **then**
    shrink the mapped jobs as per the analysis
end if
entry.bitmap ← bitmap(available nodes)

5.2.2 Runtime Scheduling

Algorithm 4 presents a high level pseudo code of the algorithm to generate a resource offer. Following points describe the algorithm:

- If batch scheduler accepted the previously sent offer, then we will repeat the system transformation of the previous negotiation attempt to check if runtime scheduler will accept this mapping. If the mapping was accepted, then the runtime scheduler will commit the mapped jobs to the running list.
- If this is the first attempt and there are no forwarded jobs then it means that the runtime scheduler is generating a new offer. In this case it will perform a partial transformation and will return back from the function to send the offer.
- If the mapping was rejected after repeating the previous transformation, then the runtime scheduler will reset the runtime state of the system and perform a new transformation for generating an offer to satisfy the forwarded job requests.
- If the response from batch scheduler was reject, then the runtime scheduler will go through the same step as described in the previous point.

Algorithm 5 presents a high level pseudo code of the runtime scheduling algorithm. Following points describe the algorithm:

- Run the backfill algorithm to schedule the forwarded job requests. Some jobs can immediately start whereas for others it will give reservations. Both of these are written into the resource offer.

Algorithm 4: Algorithm for generating a resource offer

```
   Data: Attempts, Jobs2Map, Error Code, Offer<empty>
   Result: Offer<Reservations, Residual nodes>
 1 if Error Code is SUCCESS then
 2 |   /* Batch Scheduler accepted the previously sent offer */
 3 |   if (Jobs2Map NEQ NULL) AND (Attempts GT 1) then
 4 |   |   initialize runtime state
 5 |   end if
 6 |   /* Repeat the transformation of the system which was done for the
   |      previous attempt */
 7 |   schedule((Attempts GT 1) ? (Attempts - 1) : Attempts, Jobs2Map, Error
   |   Code, Offer)
 8 |   /* Offer being generated for the first time if attempts is equal
   |      to 1 */
 9 |   if Attempts EQ 1 then
10 |   |   return SUCCESS
11 |   end if
12 |   if schedule was successful AND Jobs2Map NEQ NULL AND Attempts GT 1 then
13 |   |   commit the mapped jobs to the running list
14 |   |   return SUCCESS
15 |   end if
16 end if
17 /* Batch Scheduler rejected the previously sent offer */
18 if Error Code NEQ SUCCESS then
19 |   initialize runtime state
20 |   schedule(Attempts, Jobs2Map, Error Code, Offer)
21 end if
22 return error code
```

- If there were no forwarded job requests received then this was a new offer being generated and therefore it will return back from the function with the constructed resource offer.
- Update job dependencies according to the new system state. This means that some running malleable jobs may be maintaining invalid job dependencies. If a forwarded job has been mapped successfully to some nodes and a running malleable job has its dependency on this forwarded job from before because it was reserved, then this dependency should be cleared now as it is no longer valid.

- Analyze running malleabe jobs according to the approach mentioned in the pseudo code for shrink operations to generate sufficient resources for immediate start of reserved jobs. If analysis was successful, then shrink the jobs.
- Reschedule forwarded job requests using backfill algorithm and compute reservations(without job start). In this step we will not consider jobs which could get immediate start time in the previous steps.
- Update job dependencies according to the new system state similar to point **3**.
- For every reserved job, if it has a dependency on a running malleable job then we will expand that running job to the maximum possible node count and update the dependency of that running malleable job to this forwarded job. Expansion of the running job will also consider shrinking other running jobs for its needs. For the purpose of shrinking, it will follow the same approach as shown in ***Line* 7** except that it will follow only the second and third steps.
- Once all the previous steps of the algorithm have been completed, runtime scheduler will decide whether to accept or reject the mapping based on some decision making logic. If the number of negotiation attempts have reached the limit then it will just accept. If it rejects then it will send back an offer to iBSched.

Algorithm 5: Runtime Scheduling Algorithm

```
   Data: Attempts, Jobs2Map, Error Code, Offer
   Result: Updated Offer
 1 schedule requests and create reservations(without job start)
 2 if Jobs2Map EQ NULL then
 3 |   return /* new resouce offer generated */
 4 end if
 5 update job dependencies according to the new system state
 6 for each reserved job do
 7 |   Prioritize malleable jobs in the order: (1) malleable job expanded for this
   |   reserved job, (2) malleable job expanded for no specific reserved job, (3)
   |   malleable job expanded for other reserved jobs
 8 |   Analyze if expanded malleable jobs can be shrunk in the above order to
   |   make enough nodes available to start the reserved job
 9 |   if enough nodes found then then
10 |   |   Shrink the selected malleable jobs
11 |   |   Insert the job as an entry in Offer.Reservations
12 |   end if
13 end for
14 reschedule requests and create reservations(without job start)
15 update job dependencies according to the new system state
16 for each reserved job do
17 |   if job depends on a malleable job then
18 |   |   Expand the malleable job with the available nodes
19 |   |   Update the dependency information for this job
20 |   end if
21 end for
22 if Error Code is SUCCESS then
23 |   /* Runtime scheduler will decide now whether to accept / reject
   |      the mapping */
24 |   if Attempts LT MAX_LIMIT then
25 |   |   decision ← decision_logic(Offer, Jobs2Map, Attempts, count(idle nodes))
26 |   end if
27 end if
28 if Attempts EQ MAX_LIMIT then
29 |   decision ← accept
30 end if
```

```
31 if Error Code is SUCCESS AND decision is reject then
32 |  return
33 end if
34 equipartition the available idle nodes among other remaining running malleable
   jobs
```

5.3 Negotiation

The above figure illustrates one possible scenario of the negotiation between batch and runtime scheduler. Following points describe it in detail:

- The alphabets **A,B,C,D,E** represent runtime states(running jobs and idle nodes in the invasic partition) of the system. **TRF** means transformation of the system state through expand and shrink operations. This happens when a list of forwarded job requests are received from batch scheduler and the runtime scheduler runs its algorithm to fit as many requests as possible by expanding / shrinking running jobs.
- **INIT** means initialize state. It will reset the transformed state back to the original state that the system was at the beginning of the negotiation. This is done in order to perform a new system transformation during every negotiation and this must start from the original state otherwise the state of the system will be saturated very soon.
- Saturation means that we can no more perform any expand / shrink operations on the running jobs as they will be either at there min or max node counts due to a result of the previous transformations. This would result in making no progress on the following negotiation attempts.
- The green boxes labelled **"Algo 1"** represent the batch scheduling algorithm which will run every time the batch scheduler receives a resource offer. On every such attempt, it will relax the constraints of the job much more than its previous attempt for all those jobs which could not be mapped.
- The box labelled **"Update"** will update its job details once it receives a feedback. This is important as a subsequent negotiation must not result in the batch scheduling algorithm dispatching a job that is already running.
- Once the negotiation completed then the runtime scheduler will accept the mapping and commit the jobs. The box labelled **"COMM"** represents the step

where the commit of new forwarded jobs to the running list happens. These jobs would be started very soon. The runtime state of the system will take up the transformed state going forward as shown by the box **E**.

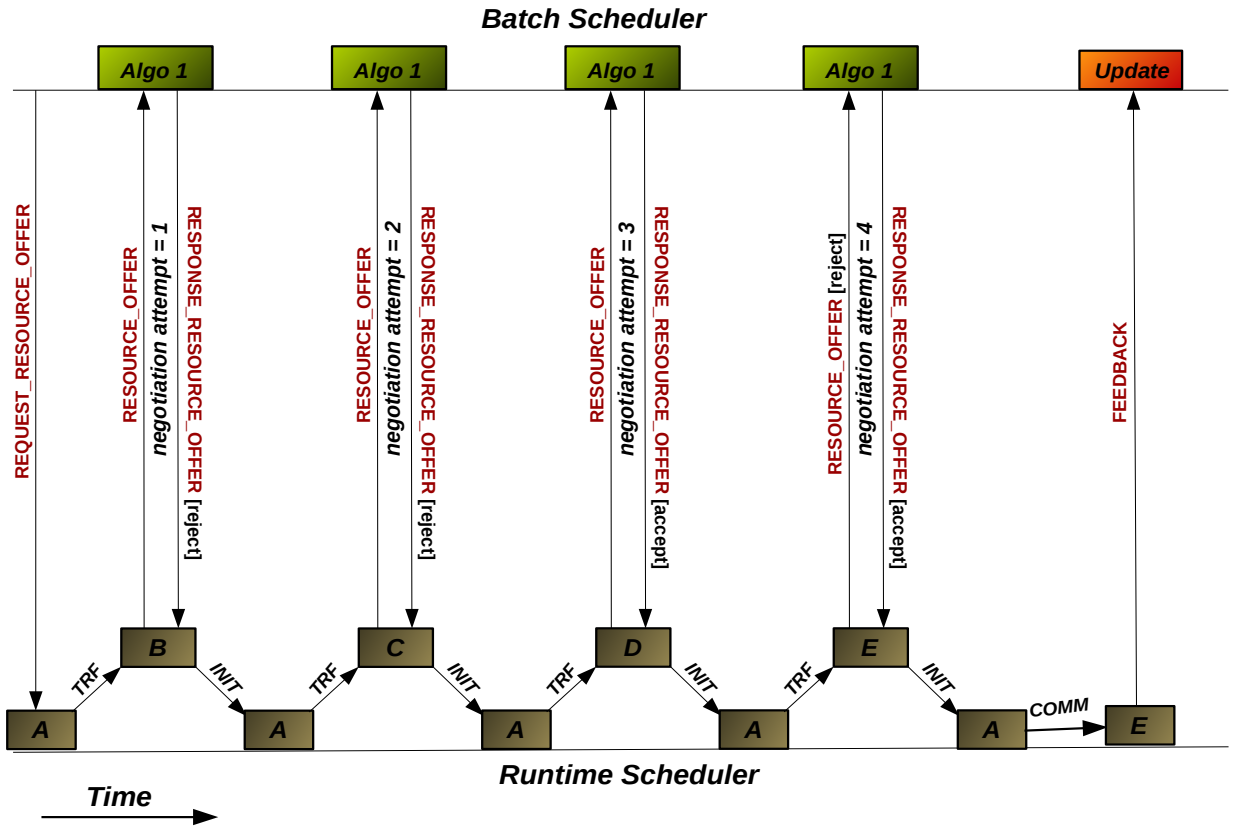


Figure 5.1: Negotiation protocol

6 Implementation

6.1 Plugin

iBSched is implemented as an extension to SLURM in the form of a scheduler plugin. A Slurm plugin is a dynamically linked code object which is loaded explicitly at run time by the Slurm libraries. A plugin provides a customized implementation of a well-defined API connected to tasks such as authentication, interconnect fabric, and task scheduling. There are several other existing scheduler plugins of SLURM such as: the default(FCFS) plugin, Backfill, wiki and wiki2(Interfaces to external schedulers) etc. A related plugin that is of importance to scheduling is the priority plugin because the job queue is first constructed and then sorted based on priorities before the scheduling algorithm runs through it. Slurm provides a basic priroity plugin and a multifactor priority plugin. The basic version provides a standard FIFO based job priority whereas the multifactor version provides priorities to job based on several factors such as age, fair-share, job size(also considers the job duration to calculate a weight factor for job size relative to time), partition, quality of service etc. The multifactor priority plugin has been refactored for this work to be better suitable for our implementation. Specifically, the multifactor priority plugin now assigns priority to jobs based on age and size(also considers the job duration to calculate a weight factor for job size relative to time). As its default behavior it also re-calculates periodically the priority of the jobs.

For the current scope of the work, iRTSched has been implemented as an independent binary that talks to iBSched via the negotiation protocol. The purpose of this independent binary is to fake an actual runtime scheduler by simulating the runtime states of the system in time by employing the runtime scheduling algorithm and also negotiating with iBSched. In reality, a full-fledged runtime scheduler will be managing the resources in the invasic partition, will launch the jobs and perform dynamic resource management. In order to evaluate the negotiation-based separation of batch and runtime scheduling system, a simulation of the same involving a fake runtime scheduler would be very useful and sufficient for our objectives.

6.2 Data Structures

This section will enumerate some of the important data structures that are a part of the implementation.

Job Mappings
Below is enum that represents the possible characterstics of a job. This hint can be supplied at job submission time or can be inferred during runtime and profiling the application performance. By default it will be considered as NONE.

```
enum job_hint {
        COMP_BOUND,
        MEMORY_BOUND,
        IO_BOUND,
        NONE
};
```

Below is an enum that represents the type of a job. This needs to be specified at submission time by the user. By default it will be considered as rigid.

```
enum job_type {
        RIGID,
        MOLDABLE,
        EVOLVING,
        MALLEABLE
};
```

Below structure represents the node count constraint that is used by the batch scheduler during the negotiations with runtime scheduler. At the submission time max will be set to the job's max nodes and min will be set to a value of max nodes - step_size where step_size is calculated from the formula given in x.x. This step will be repeated when a new transaction starts.

```
typedef struct qos_rsrc {
        uint32_t min;
        uint32_t max;
} qos_rsrc_t;
```

Below structure represents the details of a job necessary for its launch. This is packed within the following structure and it gets forwarded to the runtime scheduler.

```
struct forward_job_details {
        qos_rsrc_t node_count; /* Job resource requirements */
```

```
        uint16_t cpus_per_task; /* number of processors required for
                                   * each task */
        uint8_t hint; /* Job characteristic: IO / Compute / Memory bound
            */
        uint8_t job_type; /* Type of job: Static or Dynamic */
        uint32_t max_cpus; /* maximum number of cpus */
        uint32_t min_cpus; /* minimum number of cpus */

        /* job constraints: */
        uint32_t pn_min_cpus; /* minimum processors per node */
        uint32_t pn_min_memory; /* minimum memory per node (MB) OR
                                   * memory per allocated
                                   * CPU | MEM\_PER\_CPU */
        uint32_t pn_min_tmp_disk; /* minimum tempdisk per node, MB */
        uint8_t share_res; /* set if job can share resources with
                                   * other jobs */
        uint8_t whole_node; /* 1: --exclusive * 2: --exclusive=user */
};
```

The below structure makes up a single entry in the Map that is constructed by batch scheduler. A list of such entries make up the mapping of jobs to offer which is sent to the runtime scheduler. In addition to the details of the job as mentioned for the previous structure, this structure include details such as the bitmap of nodes allocated, priority, job id, min / max nodes etc.

```
struct forward_job_record {
        struct forward_job_details *details; /* job details */
        uint32_t job_id; /* job ID */
        uint32_t magic; /* magic cookie for data integrity */
        char *name; /* name of the job */
        bitstr_t *node_bitmap; /* bitmap of nodes allocated to job */
        uint32_t node_cnt; /* Current node count assigned */
        uint32_t min_nodes;
        uint32_t max_nodes;
        uint32_t priority; /* relative priority of the job,
                                   * zero == held (don't initiate) */
        time_t start_time; /* Expected or Actual start time */
        uint32_t time_limit; /* time_limit minutes or INFINITE,
                                   * NO_VAL implies partition max_time
                                       */
```

```
        uint32_t time_min; /* minimum time_limit minutes or
                                    * INFINITE,
                                    * zero implies same as time_limit */
};
```

Below structure represents the message used by batch scheduler to request for a resource offer. It contains a value field used for the purpose of protocol communication followed by the most important field which is the Map.

```
typedef struct request_resource_offer_msg {
        uint16_t value; /* info */
        List jobs2map; /* This is the list of jobs waiting to be
            dispatched. And communicates the current requirements to rt
            scheduler which
                        can then try to suitably construct a resource
                            offer to satisfy the requirements as best as
                            possible */
} request_resource_offer_msg_t;
```

Below structure represents the response to a resource offer sent by the batch scheduler. It resembles very closely to the previous message but has in addition the error code and the error msg fields. These fields which identify the error are important for batch scheduler to convey its response to the runtime scheduler for the offer it sent.

```
typedef struct resource_offer_resp_msg {
        uint16_t value; /* info */
        List map_jobs2offer; /* Jobs mapped to the given offer. This
            depends on whether the batch scheduler accepted / rejected /
            countered
                              the resource offer it received */
        uint32_t error_code; /* error code on failure */
        char * error_msg; /* error message on failure */
} resource_offer_resp_msg_t;
```

Resource Offers

Below structure represents the reservation for a job which can begin immediately or in the future. This depends on the response from the runtime scheduler.

```
typedef struct job_resv {
        time_t end_time; /* end time of reservation */
        uint8_t full_nodes; /* when reservation uses full nodes or not */
        uint32_t job_id; /* job ID */
        bitstr_t *node_bitmap; /* bitmap of reserved nodes */
```

```
        uint32_t node_cnt; /* count of nodes required */
        time_t start_time; /* start time of reservation */
} job_resv_t;
```

Below structure represents the message format for resource offer message sent by runtime scheduler. It contains a list of reservations for each of the jobs that were sent by batch scheduler in its mapping. It contains the set of residual or free nodes available.

```
typedef struct resource_offer_msg {
        uint16_t value; /* info */
        uint8_t negotiation; /* if negotiation is ongoing then this value
            is 1 else it becomes 0 to indicate ischeduler that previous
            negotiat
                              ion is over */
        List resource_offer; /* List of node space records */
        List resv_jobs; /* List of jobs with actual reservations. Can also
             include jobs with virtual reservations that are those with
                                future start / service times */
        uint32_t error_code; /* error code on failure */
        char * error_msg; /* error message on failure */
} resource_offer_msg_t;
```

Feedback Reports

Below structures represent the format of a feedback report and a list of reports respectively. For the scope of this current work, the feedback does not send back any performance model of the running / completed application or any kind of job characteristic and energy consumption. It sends across basic details about the status of running / completed jobs.

```
typedef struct job_status {
        uint32_t job_id;
        time_t run_time;
        time_t end_time;
        uint32_t node_cnt;
        bitstr_t *node_bitmap;
        uint32_t job_state;
} job_status_t;

typedef struct status_report_msg {
        List status_reports;
} status_report_msg_t;
```

Running Job Record
The below structure is very important as the runtime scheduler uses this as the job record for each of the running jobs. Since the scope of this work is simulation, We do not include many other details of a job that would be necessary for its launch, logging etc. Dependency of a forwarded job onto a running malleable job is saved in the variables depend_job_id and depend_job_prio of a running job record. This dependency information is used by the PDBES algorithm.

```
struct run_job_record {
        uint32_t job_id;
        bitstr_t *node_bitmap;
        time_t start_time;
        uint32_t priority;
        uint32_t time_limit;
        uint8_t hint;
        uint8_t job_type;
        time_t end_time;
        time_t orig_end_time;
        uint32_t job_state;
        uint32_t orig_node_cnt;
        uint32_t node_cnt;
        uint32_t last_node_cnts[2];
        bitstr_t *next_node_bitmap;
        uint32_t next_node_cnt;
        uint32_t depend_job_id;
        uint32_t depend_job_prio;
        uint32_t save_depend_job_id;
        uint32_t save_depend_job_prio;
        uint32_t min_nodes;
        uint32_t max_nodes;
        uint8_t adapt; /* 0 - No change, 1 - Expand, 2 - Shrink */
        uint8_t transformed;
        time_t exp_end_time;
        time_t last_resize_time;
        uint32_t adapt_interval;
};
```

Node Space Map
Below is an existing structure used by SLURM for its backfilling algorithm. This is used to form a chain of records chronologically ordered in time that represent the set of

available nodes in those timeslots. The backfill algorithm uses this structure to prepare the view of resources in time and resembles a two dimensional space of nodes and time. This is used in order to compute when and where a job can start by looking at the reservations that higher priority jobs already have and these must be respected by lower priority jobs in case they can start now.

```
typedef struct node_space_map {
        time_t begin_time;
        time_t end_time;
        bitstr_t *avail_bitmap;
        int next; /* next record, by time, zero termination */
} node_space_map_t;
```

6.3 Important APIs

```
extern List schedule_invasic_jobs(resource_offer_msg_t *, List, uint16_t
    *, uint32_t *);
```

This routine is responsible for creating a map of jobs to the available offer. It will fill the offer that has been passed as a an argument to it with the list of invasive jobs that have been selected according to its algorithm. Jobs in this list would be mapped to some number of nodes that satisfy the job constraints. Those jobs which could not be mapped will have their constraints relaxed and just forwarded along with other jobs. Also, The batch scheduler has a limit on the number of jobs that can be forwarded but have not been mapped to any resources as it did not satisfy its constraints. This is the reservation depth similar to what is used in backfill algorithms. Once the limit has been reached the scheduling algorithm will then just scan the rest of the invasive job queue to look for jobs that can fit the remaining available nodes in the offer.

```
extern uint32_t adjustQoS_node_count(struct job_record *);
```

This routine relaxes the node count constraint of a job. The original constraints are min and max nodes supplied at the job submission time. Negotiation begins by setting a node count constraint(node_count.min, node_count.max) within this window of min and max. With more negotiation attempts in a single transaction, the constraints would keep getting relaxed and node_count.min would approach the original min value.

```
extern void resetQoS_node_counts();
```

This routine is called before the start of a new set of negotiations. It will reset the node count constraint for every job in the queue to their original values.

```
extern int _decision_logic(List, int, uint32_t);
```

This routine is where the batch scheduler makes a final decision on whether the received resource offer form runtime scheduler is accepted / rejected. It does so by checking through the mapping that has been constructed after the scheduling algorithm processed the offer. The mapping is supplied as the first argument to this function.

```
extern void *irm_agent(void *);
```

This is the routine that is associated with the main runtime scheduler thread. It gets called once the thread has been created. It basically runs an infinite control loop for the scheduler to initiate various operations such as processing the response for an offer from batch scheduler, performing a transformation of the runtime system state, waiting for a request for resource offer in a separate spawned thread, sending back an offer to batch scheduler, terminating the scheduler etc.

```
extern int process_resource_offer(resource_offer_msg_t *, uint16_t *, int
    *, List *, List);
```

This routine initiates the processing of a resource offer, calls the appropriate scheduling algorithm.

```
extern int _request_resource_offer(slurm_fd_t); /* Wrapper for
   request_resource_offer to construct the list of job requirements to be
    sent */
```

This routine sends a request for resource offer to the runtime scheduler. It does so by constructing a list of job requests to be forwarded.

```
extern int process_rsrc_offer_resp(resource_offer_resp_msg_t *, int,
   resource_offer_msg_t **);
```

This routine is responsible for processing the mapping received from the batch scheduler and invoking the appropriate runtime scheduling algorithm. If the response from batch scheduler was accept for its previous offer, then it will recreate the transformation

of the previous attempt else it will create a new transformation. It will run the algorithm to satisfy as many jobs forwarded as possible by using the PDBES algorithm and then determine whether it will accept / reject this mapping. If it does not accept then it will send a new offer created as a result of this new transformation.

```
extern int create_resource_offer(int, List, uint16_t,
   resource_offer_msg_t **);
```

This routine is called from within the "process_rsrc_offer_resp" routine to initiate the scheduling algorithm to run through the mapping. It is also called in the "irm_agent" routine in situations when a request for a resource offer is received. Or, a new offer needs to be generated by the runtime scheduler since some running applications have shrunk to create a gap in the resources that can be utilized to serve some new batch jobs.

```
static int
_schedule(int attempts, List job_queue,
              uint16_t *recv_err_code, resource_offer_msg_t **
                 rsrc_offer_msg);
```

This is the runtime scheduling algorithm whose steps have been described in the form of a pseudo code x.x.x. It will use the PDBES algorithm to scan through the mapping and provide start times for each of them.

```
extern bitstr_t * _try_transf(bool);
```

This routine performs a runtime transformation of the system state by performing expand / shrink of the running malleable jobs considering their performance and scalability. In case the result of this transformation are some free resources, then those will be sent up to batch scheduler as a new resoruce offer.

```
extern int _commit_state(bool);
```

This routine will commit the forwarded jobs(with immediate start time) to the running jobs list. It will also change the node bitmaps of running jobs to their new node bitmaps if they have been subjected to some sort of transformation during the phase of runtime scheduling algorithm. In alignment with the new node counts for many of the running malleable jobs, there expected end times will also be updated based on whatever performance model is know until this point.

```
extern bool _get_delta(uint32_t *, time_t, time_t);
```

This routine will compute the next time slice for which the runtime scheduler will sleep. This could be a fixed time interval for periodic wake ups of the runtime scheduler or it can wake up earlier if there was some job that is expected to complete within this time interval.

```
extern void _initialize_state(void);
```

This routine resets the state of the system back to the point before the start of the negotiation. During the negotiation process, a lot of the details in the running job records would be temporarily updated to account for negotiations and the associated expand / shrink operations. With every negotiation attempt within a transaction, the state will have to be reset before the runtime scheduler can run its algorithm again.

```
extern int _decision_logic(resource_offer_msg_t *, List, int, bool);
```

The runtime scheduler decides whether to accept / reject the mapping received from batch scheduler by calling this routine. This routine checks through the mapping to see how many jobs can be started immediately and based on how well it is suitable for its metrics a decision will be taken.

6.4 State Machine Diagrams

6.4.1 iBSched

6.4.2 iRTSched

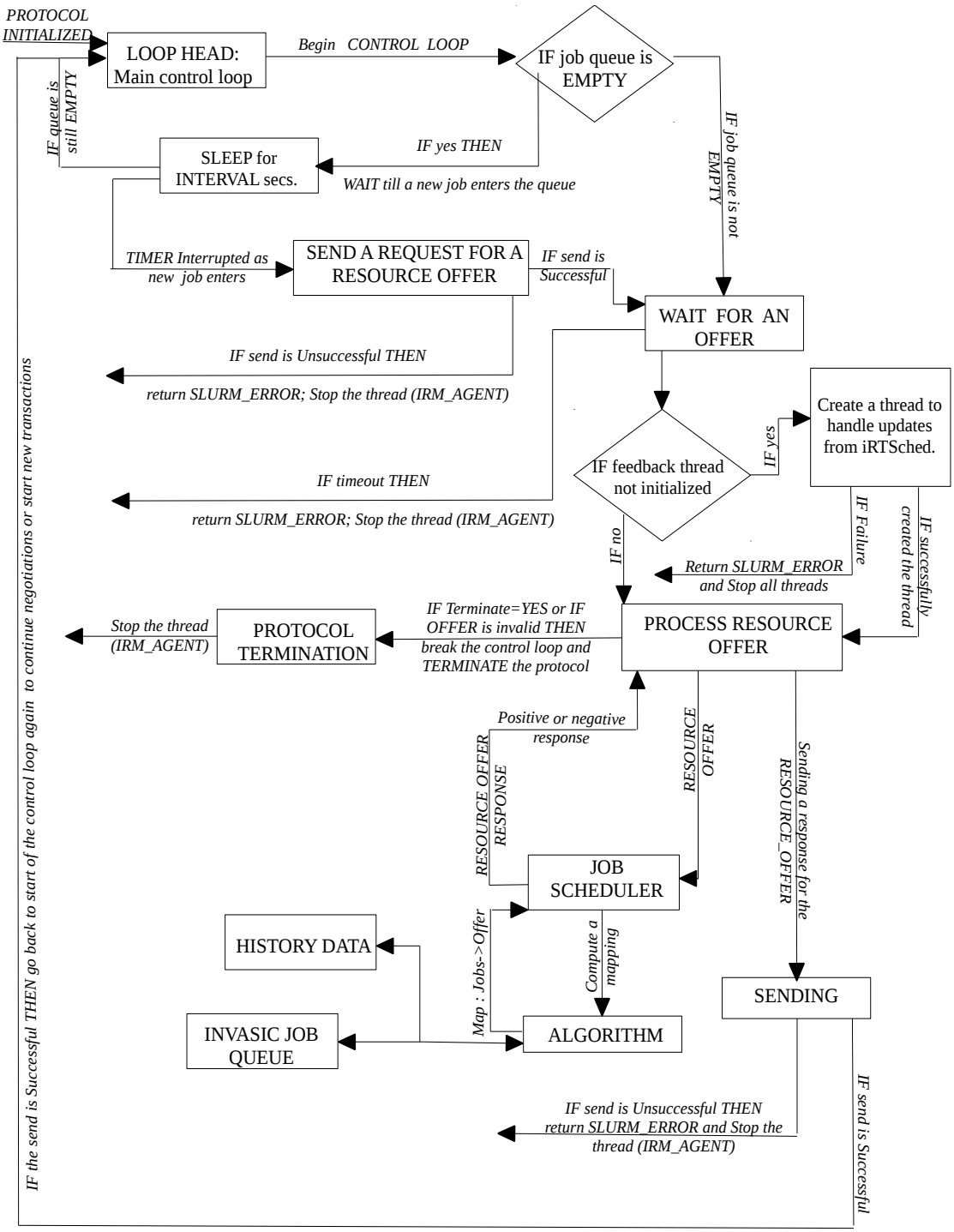


Figure 6.1: iBSched

Figure 6.2: 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]

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).