Dynamic Balancing of Scientific HPC Applications on Institutional Clusters

---

Mac Lyle

May 21, 2020
Version: Final Draft
Advisors: John Dougherty and Dave Wonnacott

Abstract

In an age of ever expanding data sets, there is an increasing demand for high performance computing (HPC) in the scientific community. In order to maximize the performance of existing hardware, researchers have been looking for innovative and cost effective ways to expand and/or optimize their clusters. This thesis first provides a quick summary on the traditional methods that are currently in use in institutional clusters followed by some of the new services being offered to help researchers expand their computational resources. Most importantly, three recent papers propose new ways to optimize current hardware to increase total job throughput and utilization of a cluster. The methods proposed take advantage of dynamic allocation, both on the application level and the system level.

The first paper (Nathaniel Kremer Herman and Thain 2018) finds that it is possible to dynamically size master-worker applications, freeing up significant resources for other applications to run on the system. The work done in (Feng Liu 2018) offers a method to integrate on-demand HPC requests into a traditional batch system. This created a massive boost in system throughput and reduction in batch wait time. The final paper (Suraj Prabhakaran 2015) looks into the potential of a new type of malleable job that can be run on HPC systems. By integrating these jobs into the batch scheduler, total system throughput was increased and lays a foundation for potential future workflows.

This review is followed by the methods and results of a comparison between parallel jobs in the Chapel language and the MPI implementation. During my experience, Chapel was much easier to learn and to implement however, when scaled, it couldn't compare to the speed-up of MPI.

Acknowledgements

I would like to thank my advisors Professor John Dougherty and Professor Dave Wonnacott for being my close guides in writing this thesis. Without their dedication to my success this would not have been possible. I would also like to thank Professor Sorelle Friedler for her guidance and work to provide the beautiful template for this thesis.

Contents

1 Statement of The Problem

Computing clusters have become an essential part and in many cases a necessity of various institutions. These clusters are able to handle complex problems analyzing a massive number of parameters and also run in a timely manner. However, we are currently in an age of rapidly expanding data sets. Improvements in measuring devices and techniques paired with increasing demand for high performance computing (HPC) has institutional computing clusters struggling to keep up.

Typically to allocate resources an application will requests a specific number of computers to run on and then holds onto those resources for its lifetime. However, it is often the case that the researcher submitting the application is the one that determines how many computers to request. If too many are requested, computers sit idle when they could be running a different application. If too few are requested, the application could run for an unnecessarily long time when it could have been completed in a fraction of the time. This highlights two closely related problems. HPC machines are sitting idle even when demand is increasing and the total number of applications completed in a given timeframe is too slow. The question is how can ensure fewer machines are sitting idle as well as increasing the speed in which applications are completed? Not only this, but we must do it in a fair and economical manner. This thesis aims to give an overview and investigation into the available economical methods to expand and increase the overall efficiency of these clusters while highlighting some of the most recent advancements in the field.

2 Literature Review

2.1 Introduction

Computing clusters have become an essential part and in many cases a necessity of various institutions. These clusters are able to handle complex problems analyzing a massive number of parameters and also running in a timely manner. However, we are currently in an age of rapidly expanding data sets. Improvements in measuring devices and techniques paired with increasing demand for high performance computing (HPC) has seen many institutional computing clusters become overwhelmed. This literature review aims to give an overview of available economical methods to expand and increase the overall efficiency of these clusters while highlighting some of the most recent advancements.

2.2 Background

Many institutions use a distributed system of computers to make up their computing cluster. A distributed system in the simplest sense is a group of multiple computers that are connected to appear as a single endpoint for a user. This gives the overall system the potential to run more jobs and increase the total amount of jobs that can be finished in a given amount of time, also known as *throughput*. While these computing clusters can be very fast, it is often the case that many researchers have access to them and will submit large computational jobs to be completed. This can quickly become an issue as the number of jobs grows and resources become more limited. If jobs are simply allowed to run immediately on whatever resources are currently available on the system, this can be very inefficient. For instance, if a resource intensive job is using 6 of 10 nodes and another similarly sized job is queued up next, no jobs can use the 4 idle nodes until the running job is done, even if they use 4 or less nodes. Situations like this can cause large portions of a system to remain idle for long periods of time causing a decrease in throughput. In order to prevent this, most computing clusters utilize a *batch system*.

A batch system is a vital part of an HPC cluster as it determines how and when to run jobs. The batch system is made up of two main parts which work in tandem to run

jobs on the cluster, a scheduler and a resource manager. The system not only runs jobs but has to factor in efficient job management, resource utilization, throughput, and *job fairness*. *Job fairness* means that the all jobs are treated with an equal overall priority and will be completed in a timely manner. This prevents big low priority jobs from getting infinitely held up and high priority jobs being blocked by many low priority jobs. The batch system can also be affected by the shape and type of incoming jobs (Suraj Prabhakaran 2015)(Feng Liu 2018). This is an important factor that will be analyzed in depth later in this literature review. The overall goal of a batch system is to increase the total throughput of the system while still maintaining a balance between the aforementioned factors (Suraj Prabhakaran 2015).

Until now a job has simply been a sort of black box that sits on a given number of processors until it is finished, upon which it releases those processors. Looking closer, these jobs are large computations that are being queued into the batch system. In order to be completed in a reasonable time fashion, these jobs have to run in parallel. A parallel job is one that is able to be broken apart into multiple smaller jobs that can then be individually run on separate processors (Barry Wilkinson 2005). By breaking up the total work that needs to be done, the job can be completed significantly faster than if simply left to run in serial on a single processor. This speedup is best described by John Gustafson.

In 1988, John Gustafson published his findings in parallel computing that questioned the current ideas of how much faster a parallel application could run versus in serial (Gustafson 1988). At the time, estimated speedup that could be seen in parallel computing was determined by Amdahl's law. The law stated a problem can be broken up into two parts, the time spent on serial parts of the application s and the time spent on parallel parts of the application p (on serial processors). Therefore when applied to N number of processors, Amdahl's speedup is defined as:

$$\begin{aligned} \textit{Speedup} &= (s+p)/(s+p/N) \\ &= 1/(s+p/N) \end{aligned}$$

However, in Gustafson's research he had noticed speedups significantly greater than what would be expected from Amdahl's law. Gustafson realized that Amdahl had assumed that p and N were independent, when in reality the problem size scales accordingly to the number of available processors. He proposed a new perspective where run time was constant, rather than size. Gustafson also noted that s did not grow with problem size. Thus, he proposed the following new scaled speedup equation (Gustafson 1988).

$$\textit{Scaled speedup} = (s + p * N)/(s+p)$$

$$= s + p * N$$
$$= N + (1 - N) * s$$

It should be noted that Gustafson's speedup is still theoretical and does not consider some overhead such as inter-process communication. Still, this way of looking at speedup in parallel computing shows that efficient and scalable parallelism is more achievable than first theorized by Amdahl.

Even as more advancements are made in parallel computing and batch systems continue to optimize cluster throughput. The pace of data growth and demand on computational systems continues to be too great than current systems can handle and likely always will (Barry Wilkinson 2005).

While the promise of running HPC applications in the cloud is tempting, it is not yet totally viable. There have been many studies and reports done on the subject, many of which are touched on in the survey paper (Marco A.S. Netto 2018). Also noteworthy is The Magellan Report (Katherine Yelick 2011), which remains one of the largest and most comprehensive reports on the subject. The papers outline the potential for HPC in the cloud along with the major restraints that have prevented the migration away from HPC clusters.

The Magellan Report was carried out by the department of energy for the US government and was published in 2011. Despite its publishing date, many of the core limitations that were discovered still hold true now as is evident in (Marco A.S. Netto 2018). Firstly the report found that the initial cost for an institution to migrate all of its scientific precesses to a cloud computing system would be very high. Much of the cost of this move would not come from the cost of the actual cloud computing services but from retraining researchers on how to use new systems, setting up virtual environments that could run scientific applications, data storage and migration, as well as the creation of any custom tools needed by the institution to correctly port their applications to the new system. Another limitation of moving HPC to the cloud is simply that the cloud on the whole is not designed for HPC (Qiming He 2010). Cloud resources are mainly optimized to run business applications which can also be demanding but in a different manner than HPC, mainly geared toward volume and query heavy workflows. Scientific HPC is often very demanding on the memory, raw performance, and communication between machines (I/O). These demands act as major physical limitations for the cloud. These constraints however, are not entirely restrictive to running scientific HPC on the cloud. Simply, they dictate the type and quantity of applications that can be run effectively on the cloud.

Both the Magellan Report (Katherine Yelick 2011) and the survey paper (Marco A.S. Netto 2018) found that small to medium sized applications with minimal levels of I/O (loosely-coupled applications) can be run efficiently and sometimes even faster than supercomputing clusters on cloud resources. This leads to the conclusion that it is viable to migrate some applications to the cloud which is an appealing option for many researchers. Cloud computing offers accessible and cheap computing resources and can be scaled to handle different workflows that may present themselves over time. The flexibility of the cloud is one of its main attractions as it is cost effective to only pay for the computing resources that are needed. Yet, the inability of the cloud to run all HPC applications in a timely manner justifies the continued utilization of supercomputing clusters. Moving into the future, it will likely be the case that the fastest computing system architectures will consist of a hybrid of cloud and local resources (Marco A.S. Netto 2018) (Daniel A. Reed 2015). For smaller applications, cloud services are already appearing in order to help bridge the gap between insufficient local resources and the potentially expensive and excessive power of finding time on a supercomputing cluster.

2.2.1 Services

Often, it is not always feasible for individual researchers to maintain or even access enough local resources. This is where *software as a service (SaaS)* becomes necessary. The main goal of SaaS is to increase accessibility to any given software by removing the burden of hardware and maintenance from the user. To achieve this, the software is operated and configured centrally while the user simply accesses it, often through a web browser. This model simplifies the user experience and reduces costs for providers since there is only one centralized piece of software to maintain. As a business model users must pay to use this service, usually by subscription or pay per use (Bryce Allen 2017).

Globus is a case study on how the SaaS model can be useful for the scientific community in particular (Bryce Allen 2017). The use of online software is not new to science. Globus stands out however due to its subscription model. Most scientific gateways to online resources often rely on grants or allocation of time on computing resources. This raises concerns for the long term sustainability of these services. Globus aims to bring highly reliable, accessible, and secure data management to researchers over the long term as a subscription, allowing for its continuous maintenance. Globus is comprised of multiple services that provide capabilities to each other, yet do not rely on each other. These are also all built upon Amazon Web Services creating a reliable and elastic service. In total Globus looks to address the three main challenges of software in science: usability, scalability, and sustainability (Bryce Allen 2017).

2.2.2 Future of HPC

Geoffry C. Fox (2017) lays out a vision for what type of platform will be in place for scientific computing in the future. Currently, the future of supercomputing lies in exascale computing, the next step up in speed in high performance computing according to Moore's Law (Daniel A. Reed 2015) (Moore 1965). In order to achieve this (Geoffry C. Fox 2017) looks at the existing architectures of cloud based computing and traditional HPC. Both architectures have pros and cons depending on the type of application. In general clouds excel at data-intensive algorithms while HPC fits more with simulation data and high precision calculations (Geoffry C. Fox 2017). Geoffry C. Fox's vision for the future sees a convergence of both of these technologies into an HPC cloud platform. This new system may ideally be the future of scientific HPC but many institutions have a limited budget and can't afford to rebuild their computing resources. This shows the need for an easy access alternative to help expand computing resources in an economic manner.

2.3 Recent Advances

2.3.1 Right-Sizing Master-Worker Applications

Nathaniel Kremer-Herman and Thain (2018) propose a model for properly provisioning resources within parallel computing applications, especially on distributed systems. Often in scientific computing, researchers are tasked with the job of requesting the number of cores they need for their application. Even with intimate knowledge of the application, it can often be difficult to know the optimal number of machines needed to run the application to reduce run time due to the current network environment. This leads to an over or under-provisioning of resources. Over-provisioning of resources represents an issue for scientific computation since many of these types of jobs are run on distributed systems such as a computing networks on a university campus where other researchers are also trying to run their applications. An over-provisioning of resources means many machines are sitting idle and may never be used by the application, thus blocking other researchers from using them. Likewise, an under-provisioning of resources can slow research since the job is not being completed as fast as it could be.

The model proposed by (Nathaniel Kremer Herman and Thain 2018) attempts to dynamically adjust the number of machines allocated to an application in order to find the ideal sizing, or right-sizing, for the job at the time of execution in the network environment. Using a *master-worker framework* they are able to adjust resources to better approach an application's right-size.

A master-worker framework is a manner of parallel computing consisting of a master process which coordinates and assigns work to be done by worker processes. The workers are processes which exist only as long as they are receiving work to do from the master. This framework allows for scalability as a master takes on more workers. An increase in workers increases *parallelism* (jobs running at the same time) of the application but only up to a limit. This limit is defined by both logical and practical means. Firstly since some processes may rely on others this can limit when some processes can be executed. In terms of practical limitations, the architecture of a distributed system creates two inherent bottlenecks, execution time on the hardware available and I/O time. If the execution time is high relative to I/O, the master can be stuck waiting for data to come back and if the I/O time is high relative to exectution, again the master is stuck waiting for data to send instead of distributing new workloads. The model proposed by Kremer-Herman et. al. considers these limitations for the unique environment the application is running in and attempts to determine the ideal number of processes a master can handle. The paper calls this the application's *capacity*. Knowing an application's capacity allows for dynamic allocation of resources with the goal of reducing the idle time of the master. The model proposed is unique since it functions on the application layer. This means it is only aware of the user's input and the system availability, rather than operating at a systems level. This creates a more user-centric model, where the user's perspective is prioritized.

The capacity model looks to quickly identify the smallest number of resources which when allocated to an application, minimize execution time. In parallel computing it is important not to under or over allocate as this could cause undo slowdowns. Over allocation may not seem to be a big issue. However when more cores being run, more overhead is created for the master. This means the master can be spending its time managing unutilized cores instead of sending out new tasks, causing a slowdown. Therefore at best, the over allocation of resources wastes time on those resources, preventing anyone else accessing them.

Examining capacity further, (Nathaniel Kremer Herman and Thain 2018) makes the following observations about capacity. Under the assumption that the master process can only do I/O for one task at a time and that all the workers have identical execution time t_e and data transfer time t_{io}, the master has a maximum throughput T_m of $T_m \leq 1/t_{io}$. Meanwhile, since the worker is the one executing the task, it is max throughput T_w is bounded by $T_w \leq 1/(t_e + t_{io})$. It follows then that if the master has C workers available then $CT_w \leq T_m$. Now let C be the number of workers such that the master is never idle thus $CT_w = T_m$. Now C is defined as $C = 1 + t_e/t_{io}$ and is the capacity of the system. Note the one here is to ensure that at least one worker is available so work gets done.

The above model is a good for understanding capacity. However, the assumptions made are hardly realistic and do not address making a dynamic model. In order to dynamically define capacity, both the average capacity of all previous jobs completed and the most recent i^{th} job completed are weighed separately. A new parameter α, an exponential moving average, is used to weight the previous jobs against the latest job i. This is to ensure that the i^{th} job is weighed more since (Nathaniel Kremer Herman and Thain 2018) assumes that the most recent job is more likely to indicate the current capacity of the application. This average also works as a smoothing variable to steady erratic jumps in capacity. In practice (Nathaniel Kremer Herman and Thain 2018) determined that $\alpha = 0.05$ worked the best and for C_0, $i > 0$ and $0 < \alpha < 1$, recursively defined the capacity of the i^{th} task to be:

$$C_i = \alpha(1 + t_e/t_{io}) + (1 - \alpha)C_{i-1}$$

Until now the assumption has been that all tasks have independent t_{io}. This is often not the case as tasks can have shared inputs stored in a cache. The time to transfer these shares inputs is called t_c. Thus capacity is now defined as $C = 1 + t_e(t_{io} - t_c)$ (Nathaniel Kremer Herman and Thain 2018). In practice to determine when more workers would be advantageous, suppose that m workers are currently available with n jobs to complete. If $m \geq C$ or $m \geq n$ another worker would be wasteful since the application would be at capacity or not have enough work respectively. But for the remaining case of $m < C$ and $m < n$, it will take nt_{io} for the master to process the remaining tasks and the workers $nt_e/m + t_{io}$. Since $m < C$, the master is under capacity so $nt_{io} < nt_e/m + t_e$. This means its advantageous for r new workers when:

$$nt_{io} + rt_c < nt_e/(m + r) + t_{io}$$

This capacity model does have limitations on its effectiveness. Since the model relies on past and current task completion, if an application does not group similarly sized tasks, the model will have a hard time predicting necessary resources for each step of the application. This is because the model will always be chasing and trying to keep up with where the application's workflow is going. In other words, the model struggles when an application's I/O and execution time for tasks are widely variant from step to step.

2.3.2 Merging On-Demand and Batch Clusters

In recent years due to increased sensitivity in many experimental devices the amount of data available to researches has increased at an incredible rate. In tandem to this

rise in big data is a growing demand on many institution's computing clusters. Unfortunately, it is often the case that these clusters simply do not have the computational capacity to handle such demand. Therefore, many of these institutions are looking for cost effective ways to integrate HPC resources into their workflows. While, HPC resources are incredibly fast, traditionally they are scheduled using a batch system. This is unacceptable for applications that need rely on time-sensitive execution. Generally the solution is simply to have dedicated resources just for applications that require on-demand access. Having dedicated machines also allows for control of the environment that the application is run in, something typical HPC also lacks. Considering all of the conditions above, how can HPC be implemented in such a way to be able to satisfy all on-demand jobs while still able to run traditional batch jobs? Feng Liu et. al. in their paper "Dynamically Negotiating Capacity Between On-Demand and Batch Clusters" propose a solution that both handles all on-demand requests as well as improves batch wait time.

The proposed system is comprised of a service called the *Balancer* (Feng Liu 2018). The Balancer works on top of the existing systems and is designed to be as non-invasive as possible, meaning that both existing on-demand clusters and batch clusters will be remain as untouched as possible. The job of the Balancer is simply to dynamically shift computational resource nodes (machines/cores) between the on-demand cluster and the batch cluster. Essentially, a single large cluster with soft partitions for on-demand and batch jobs is created, compared to two distinct clusters. The overall goal being to reduce the number rejections of on-demand resource requests, while at the same time keeping the amount of on-demand resources low to decrease batch job wait time.

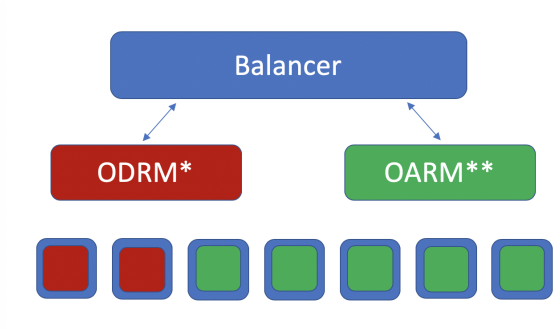


Fig. 2.1: Balancer high level architecture. Figure adapted from (Feng Liu 2018).

The architecture of the Balancer is quite simple and is illustrated in Figure 2.1. It works by moving available nodes between the two pools of resources for the scheduling managers, the on-demand pool and batch pool. These pools represent the nodes that are currently available to the corresponding system's scheduler. In this way, the Balancer does not affect the resource managers of the pools, only the total available resources the managers can access. The Balancer tracks every node

in the whole system and knows its status at any point in time. Nodes can be in one of four states: 1) OD_Reserve, 2) OD_Alloc, 3) OA_Idle, 4) OA_Busy. The nodes marked as OD_Reserve are designated to be just for on-demand jobs and cannot be allocated to the on-availability (batch) pool. These nodes act as a buffer to incoming requests and can start a job while the Balancer goes and retrieves more nodes if necessary. As the number of nodes required by the on-demand cluster grows, the Balancer negotiates moving nodes from the on-availability pool. Interestingly, the available nodes for allocation to the on-demand cluster include nodes marked as OA_Busy, which are nodes that are currently being used by the batch cluster. As one would expect the OD_reserve, OA_Idle (idle machines in the batch pool) nodes are readily available for allocation. The OA_Busy nodes are available however, only if they are predicted to finish their job before a given time limit W. This limit W is another parameter of the Balancer signifying maximum accepted wait time an on-demand job can have. If the Balancer cannot allocate enough nodes for a job before time W, the job request is rejected (Feng Liu 2018).

The Balancer uses the basic algorithm shown in Figure 2.2 to balance the node pools and two refinements that improve upon the basic algorithm are also mentioned (Feng Liu 2018). The basic algorithm waits for on-demand requests to come in and while using the reserve nodes to pad availability, goes and retrieves more nodes as additional requests come in. Once the job is completed, the nodes are released back to their original designated pool. The first refinement is called the *hint algorithm*. This algorithm is based off of the real world situation where it is understood that the size of on-demand jobs by their nature cannot be determined far in advance. However, if even only a small advanced notice is given (15-30 minutes), the Balancer can start to allocate the appropriate amount of nodes before the request arrives and therefore have an even lower rejection rate. This type of advanced notice also means that the on-demand reserve of nodes can be dynamically sized, allowing it to go all the way to zero. In turn, this algorithm also decreases the total wait time or turnaround time of batch jobs. The final refinement is the predictive algorithm. Much like the hint algorithm it has a dynamically sized reserve. The difference lies in the fact that it uses a prediction algorithm to determine when to grow and shrink the on-demand reserve rather than relying on a user advanced notice (Feng Liu 2018).

To test the effectiveness of the Balancer, the researchers modeled a real-life scenario using data from the Argonne National Laboratory (ANL). The model consisted of data from two separate clusters, the on-demand Sun Grid Engine cluster in the Advanced Photon Source (APS) and the batch cluster in the Laboratory Computing Resource Center (LCRC). The combination of these two systems created an overall system of 372, 16 core nodes, which would handle both real-time requests and batch requests. The actual data tested was one week of data pulled from two years worth

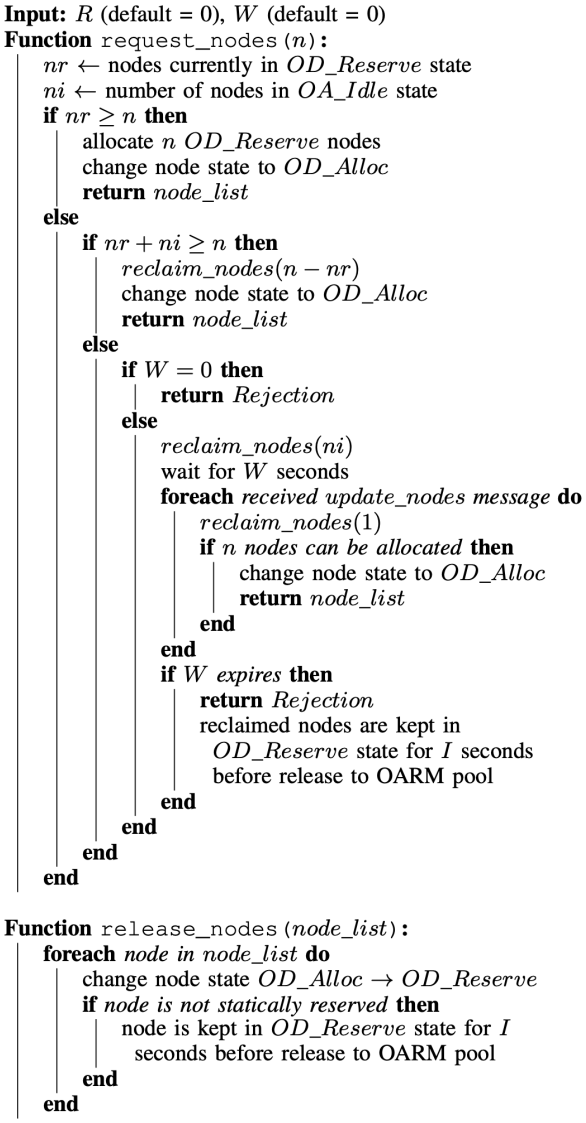


Fig. 2.2: Basic Algorithm from (Feng Liu 2018)

of job requests in order to shorten the run time of the experiment. The researchers picked the week they deemed to be the most challenging, defining challenging as the lowest average availability of nodes and highest amount of usage. The week chosen saw 24,177 batch and 141 on-demand job requests.

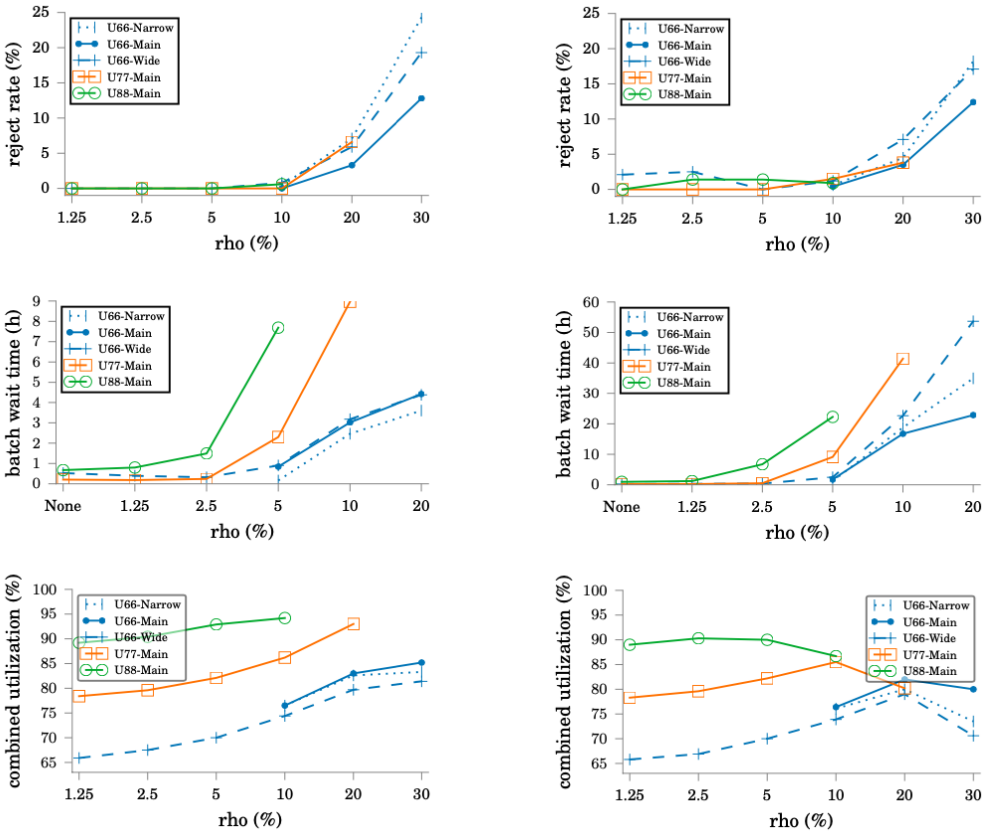
The experimental results of running the basic algorithm on the input created from the most challenging week are shown in Figure 2.3. It was found that the mean batch wait time before the Balancer was implemented was 1002.8 minutes with a lower bound of 122.5 minutes if all nodes were designated as batch only. When the Balancer was implemented, there was a total cluster utilization increase ranging from of 4.8% to 5.6% and mean batch wait times decreased ranging from 85% to 88%. The drastic decrease in wait time is due to the newfound availability of

Parameter settings	Static (Baseline)			Dynamic						
	Dedicated batch nodes			W						
	372	304	0	0	5	10	0	5	10	0
	Dedicated on-demand nodes			R						
	0	68	372	0			6			12
Combined utilization	84.4%	80.1%	1.25%	84.9%	85.7%	85.7%	85.3%	85.3%	85.3%	85.3%
Batch utilization	84.4%	78.8%	NA	84.5%	84.4%	84.4%	84.0%	84.1%	84.0%	84.0%
On-demand utilization	NA	1.25%	1.25%	0.38%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%
Batch wait time (min)	122.5	1002.8	NA	122.0	147.0	147.0	150.0	140.6	150.4	130.0
Rejections	141	0	0	30	3	3	1	1	0	0

Fig. 2.3: Experimental results from the most challenging week. Wait time W measured in minutes, reserve size R given in nodes. Table taken from (Feng Liu 2018).

previously idle nodes that were designated strictly for on-demand jobs. It is also worthy to note how the number of nodes statically reserved R and time window W effected the rejection rate. While increasing the window to 10 minutes significantly reduced the number of rejections, this proved to be too long a delay in results and was not considered further. When both R and W were at 0, there were 30 rejections. However, increasing the static reserve to just 12 nodes saw the number of rejections go down to 0 with a window of 0. This commitment of 12 nodes represents only 18% of the total cluster and met all on-demand needs on the most challenging week in the real life scenario data (Feng Liu 2018). Consequently, the on-demand cluster could be reduced by 82% and still have enough capacity to satisfy all jobs.

To further evaluate the Balancer system, synthetic workloads were generated in order to replicate varying types of workflows outside those found in the Argonne National Laboratory. The batch workloads generated had 3 different shapes, mainstream or main, wide and narrow. The mainstream workflow consisted of the same type of workflow found in the Argonne National Laboratory. Wide workloads simply created fewer but larger parallel jobs and narrow workloads created more small parallel jobs. These batch workloads were created by doubling or halving the number of nodes required per job request while keeping the same run time and ensuring total cluster utilization remained consistent. The synthetic workloads were also varied by total cluster utilizations of 66%, 77% and 88% (notated U66, U77, U88). The closest synthetic batch workflow to the average usage in the real life scenario is U66-Main, the mainstream workflow with 66% utilization. U77-Main best matches the batch workflow in the most challenging week of the real life scenario data. As one might expect, as the jobs get wider the number of rejections goes up and as jobs get narrower rejections decrease. Smaller jobs mean shorter run times and thus more space will be available when on-demand requests arrive (Feng Liu 2018). Synthetic on-demand workloads of various sizes were also generated by simply increasing the number of nodes each on-demand job requested. The max on-demand workflow tested was 30% of the combined cluster's capacity. From this data, the researchers determined that with no alterations to the basic algorithm, the Balancer would be able to support on-demand workloads below 10% of the cluster's capacity (Feng Liu 2018).



(a) Performance of hint algorithm.

(b) Performance of predictive algorithm.

Fig. 2.4: Figures taken from (Feng Liu 2018). Here rho(%) signifies the percentage of total cluster capacity requested by on-demand jobs.

In order to handle workflows where on-demand jobs are over 10% of a system's capacity, the hint and predictive algorithm come into play. The results of which are displayed in Figure 2.4. Simply designating a large static reserve on top of the basic algorithm causes for immense slowdown in batch job turnaround time. Therefore, the dynamically allocated nodes of the hint algorithm allow for the node pool to scale accordingly to incoming jobs. While the hint algorithm is very effective at reducing rejection rates of on-demand workflows over 10% and keeping batch turnaround fast, it is not as realistic since it requires all on-demand jobs give prior notice of their arrival. The predictive algorithm essentially does the same thing as the hint algorithm just not as precisely as it is guessing when jobs will come in rather than being told.

This algorithm shows that the problem of including on-demand capacity onto a traditional HPC batch system is possible with the Balancer proposed in (Feng Liu 2018). The Balancer can also be further optimized based on the type of workflow expected by the specific cluster. This solution is also economical since it increases

utilization of the system while maintaining minimal on-demand rejections and greatly improving batch turnaround.

2.3.3 Scheduling Malleable Applications

The overall performance of a cluster can be broken down to one main measurement, throughput. Higher throughput means more jobs get run and results are returned faster. Throughput can be tied to many variables including but not limited to, type of jobs, shape of jobs, resource management, scheduling, and hardware restraints. The work of (Suraj Prabhakaran 2015) focuses on how the type of jobs a system receives can effect overall throughput.

There are four categories of jobs, rigid, moldable, evolving, and malleable (Suraj Prabhakaran 2015). Rigid jobs are the most common type of job and simply require a fixed number of processors for the duration of their execution. Moldable jobs are similar to rigid jobs however, the number of processors allocated to a moldable job can be modified by the scheduler. Once resources are set for the job, the amount stays the same for its duration and can't be changed again. Since the total resources utilized by rigid and moldable jobs doesn't change during their lifetimes, they are classified as static allocation. The next type is evolving jobs. These jobs request resource expansions or shrinkages during execution. Finally malleable jobs, which like evolving jobs can grow and shrink during execution. However, they differ from evolving jobs because the expansion and shrinkage of the job is not initiated by the job itself but the batch system. Both evolving and malleable jobs can change the size of their allocated resources during runtime and are thus fit into the category of dynamic allocation (Suraj Prabhakaran 2015).

Traditionally batch systems only supported static allocation jobs. This is primarily due to the fact that writing malleable jobs with the current programming models like MPI is an incredibly difficult and time consuming task. The programmer would have to create their own functions that would manually monitor and adapt the resources to the current state of the application (Suraj Prabhakaran 2015). However, (Suraj Prabhakaran 2015) predicts that with future versions of MPI, jobs will become automatically malleable and thus being able to utilize the capabilities of these jobs will be crucial in improving the throughput of future batch systems.

Malleable jobs are of particular interest because they have the potential to obtain very high system performance. With the ability to dynamically change the resources available while jobs are being executed, the batch system can considerably improve throughput, utilization, and response times. (Suraj Prabhakaran 2015) proposed a modification on the existing Torque/Maui batch system capable of managing rigid,

```
1: while TRUE do
2:     Obtain resource information from Torque
3:     Obtain workload information from Torque
4:     Update statistics
5:     Refresh reservations
6:     Prioritize eligible static requests
7:     Prioritize eligible evolving requests
8:     Schedule static requests in priority order and create reservations (without job start)
9:     for each evolving request do
10:        Allocate idle resources
11:        if Enough idle nodes not available then
12:            Shrink expanded malleable jobs to find resources
13:        end if
14:        if Enough idle nodes found then
15:            Apply fairness policies and determine if job expansion is allowed
16:            if Expansion is allowed then
17:                Allocate resources for evolving job
18:            else
19:                Reject the dynamic request
20:            end if
21:        else
22:            Reject the dynamic request
23:        end if
24:    end for
25:    Reschedule static requests and create reservations (with job start)
26:    Update job dependencies according to the new system state
27:    for each reserved job do
28:        Prioritize malleable jobs in the order: (i) malleable job expanded for this reserved job, (ii) malleable job expanded for no specific reserved job, (iii) malleable job expanded for other reserved jobs
29:        Analyze if expanded malleable jobs can be shrunk in the above order to make enough nodes available to start the reserved job
30:        if enough nodes were found then
31:            Shrink the selected malleable jobs
32:            Start the reserved job
33:        end if
34:    end for
35:    Reschedule static requests and create reservations
36:    Update job dependencies according to the new system state
37:    for each reserved job do
38:        if job depends on one malleable job then
39:            Expand the malleable job with the available nodes
40:        else if job depends on more than one malleable job then
41:            Equipartition available resources among these malleable jobs
42:        end if
43:    end for
44:    Update job dependencies
45:    Backfill non-reserved static requests from the job queue
46:    Equipartition available idle nodes among other running malleable jobs
47: end while
```

Fig. 2.5: Batch scheduler from (Suraj Prabhakaran 2015).

evolving, and malleable jobs. This extended batch system out performed other batch systems on the all of the tests published.

The basic approach taken for this batch system revolved around multiple aspects, resource utilization, throughput, fairness, and communication with the runtime system. Resource utilization is usually a good indicator of throughput but this is not always the case. When running malleable jobs, if the system poorly selects jobs for expansion and shrinkage, it is possible for the utilization to go up but total throughput to decrease. Thus the batch system must consider job and resource dependencies to prevent such a poor selection. The system must also deliver fairness

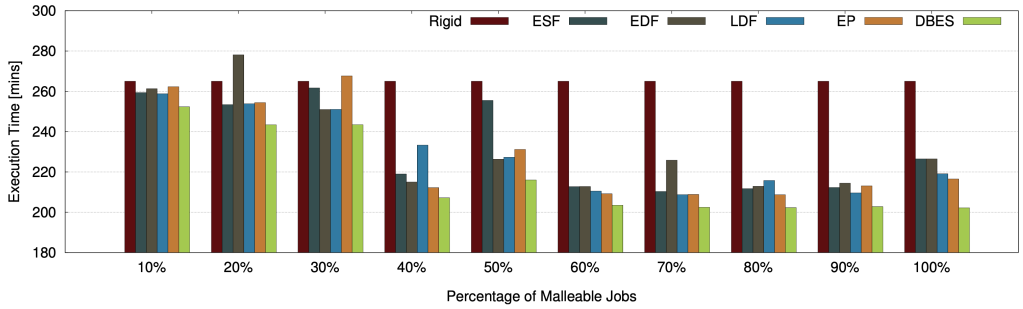


Fig. 2.6: Time for completion of the modified workloads with varying amounts of rigid and malleable jobs. Figure from (Suraj Prabhakaran 2015).

among types of jobs. Finally, malleable jobs require a mechanism to expand and shrink applications, this requires communicating with the part of the system that is actually running jobs, the runtime system. With all of this taken into consideration (Suraj Prabhakaran 2015) created Algorithm 2 shown in Figure 2.5.

The algorithm begins by prioritizing all static and evolving jobs separately. This prioritization is when the batch system creates two groups, StartNow and StartLater. StartNow jobs are literally jobs that are going to be started immediately and StartLater are jobs that are up next but are going to be put aside due to lack of available resources. Next, static requests are scheduled followed by evolving requests which have the opportunity to steal resources from the current StartNow jobs based on a given fairness policy. Once no idle nodes are available, a check is run to verify if it is possible to make room for evolving jobs by shrinking any malleable jobs. If possible, the malleable jobs are shrunk. If not, malleable jobs are marked for potential expansion later. Another round of fairness checks is run to oversee any dynamic changes made. Once all of the evolving requests are either satisfied or rejected, a new schedule of static requests is performed. It is again checked if shrinking malleable jobs or clearing invalid job dependencies will make room for StartNow jobs. Job dependencies are simply the running processes that can determine when a job starts or expands. A fairness policy is run here to determine which malleable jobs to shrink. The order is as follows: 1) expanded malleable jobs that are blocking a StartLater job, 2) malleable jobs expanded for no specific StartLater job, and 3) malleable jobs expanded for lower priority StartLater jobs. After this, another schedule of StartLater jobs is created containing a preset number of jobs. Now malleable jobs that were marked for expansion earlier are expanded based on computed job dependencies. Next a backfill scheduling is run to fill up any remaining resources with jobs that won't delay any already queued jobs. Lastly, malleable jobs are expanded to fill the final available resources.

When analyzing the new dependency based expand/shrink algorithm (DBES), the researchers ran it against several other scheduling strategies. These included: rigid

scheduling, earliest started first (ESF), earliest deadline first (EDF), latest deadline first (LDF), and naive equipartitioning (EP). The first test was done on a modified workload of varying amounts of rigid and malleable jobs, the results of which can be seen in Figure 2.6. Figure 2.6 shows that the proposed strategy consistently has the lowest execution times when compared to the others. It is worth noting the consistency of the DBES strategy as well. Not only does it perform well with a small percentage of malleable jobs but also with high levels of malleable jobs. This is unlike the EP algorithm which performs very well with a high percentage of malleable jobs (60% +) but does poorly with a lower amounts like 10% and 30%. The consistency of the DBES algorithm comes from the dependency analysis determining if it is beneficial to expand malleable jobs.

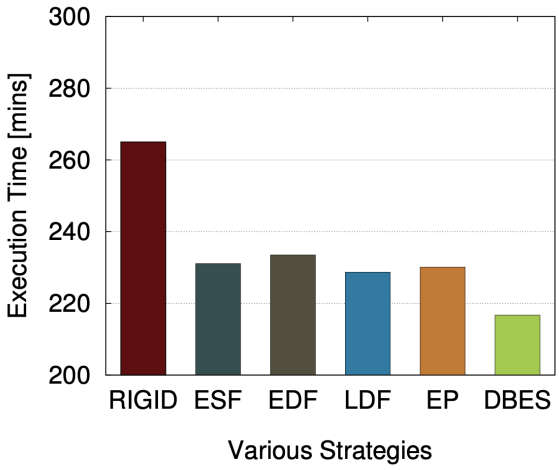


Fig. 2.7: Time completion for various strategies with a workload of 10% evolving jobs, 40% malleable jobs and 50% rigid jobs. Figure from (Suraj Prabhakaran 2015).

The researchers also analyzed a workflow which combined all types of jobs, rigid, evolving and malleable. The results are illustrated in Figure 2.7. The workflow consisted of 10% evolving, 40% malleable, and 50% rigid jobs. DBES again saw the fastest execution time with a 6% increase in throughput over the next fastest strategy LDF. Another interesting result found when testing the DBES algorithm was that during some tests, other strategies achieved higher system utilization yet still produced a lower throughput. Often higher utilization is synonymous with higher throughput, but this is clearly not the case. As the number of malleable and evolving jobs increases, the need for effective scheduling strategies grows. This novel scheduling strategy is a good step towards a new paradigm of adaptive resource management and scheduling.

2.4 Related Work

2.4.1 Dynamic Load Balancer for Iterative Applications

Harshitha Menon (2013), proposes a fully distributed dynamic load balancer for iterative applications. The goal of this is to reduce load imbalance which can lead to significant performance slowdowns in parallel computing. Having an application properly balanced in parallel computing means that across all of the processors being utilized, they all have approximately the same execution time. By using load rather than number of jobs, we can better determine how long a processor will take before it can return an answer.

Many scientific applications dynamically change over time and the given load per processor at any time is not reasonably predictable. Therefore, when assigning jobs it is possible that some processors can become overloaded while some become underloaded. Due to this, load balancers are put in place. Multiple strategies exist when it comes to designing a load balancer. A common structure is to use a centralized processor to do the balancing. However, this quickly leads to bottlenecks in scalability as one processor has to manage and decide for ever other processor being utilized for every step of the application. Another strategy is a hierarchical structure which creates subgroups of processors which each aggregate the information of all the processors and pass it up to another level of groupings. There, more aggregation occurs until it reaches the smallest and top layer. However, excess data collection and work being done while aggregating information limits these structures. The final strategy utilized by (Harshitha Menon 2013) called Grapevine is a distributed method. This method uses the data from each processor to construct a partial representation of the global state. From this representation, loads are distributed appropriately by the run-time system.

The Grapevine load balancer consists of two main stages. The first being the establishment of the global representation of the application at each processor and the second being the actual transferring of loads. In order to propagate information about every processor to all the other processors (Harshitha Menon 2013) used a gossip-protocol. This protocol spreads information analogous to how gossip spreads in the real world. Each individual processor tells a subset of all the processors its current load status. This communication is called the processor's fanout. Those processors then pass on that information along with their status to another subset of processors. While this cannot guarantee that any one processor knows the status of any other processor at a given time, the representation created produces a good enough image to effectively balance loads. This process of propagation takes $O(log_f * n)$ time (f = size of fanout, n = total nodes) for a high likelihood that the information got to any one processor (Harshitha Menon 2013). The fanout is chosen in an informed random manner. Known underloaded processors are unfavored so

the likelihood that the overloaded processors get information about underloaded processors is greater. Once a transfer is ready to be completed, one final check is made. If the proposed transfer causes a previously underloaded processor to become overloaded, the processor will reject the transfer.

Analyzing the algorithm, (Harshitha Menon 2013) found that total application run time was decreased when compared to other top end load balancing strategies. This is due to the fact that very little overhead is incurred due to its distributed nature. This means that even as it completes each step in similar time to other strategies, the overall time to completion is lowered.

2.5 Conculson and Future Work

To address the problem of growing demand on institutional HPC clusters, multiple solutions have been proposed to increase overall utilization and throughput. Firstly, for small to medium sized workflows, or for research groups with limited resources, the cloud may be an appealing option to run scientific applications. This is due to its wide availability, elasticity, and cost effectiveness. However, the cloud is not a full replacement of large scale HPC clusters. To optimize these computing resources we look to recently proposed methods (Nathaniel Kremer Herman and Thain 2018) (Feng Liu 2018) (Suraj Prabhakaran 2015).

The first of these methods looks at right-sizing master-worker applications during runtime (Nathaniel Kremer Herman and Thain 2018). This right-sizing, identifies if an application is over or under-provisioned and adjusts the number of machines allocated accordingly. Secondly, in order to increase the total systems computing power, Feng Liu et. al. proposed to combine on-demand computing resources and batch resources. This asked the question of how on-demand requests could be managed o a majority batch system. By creating a lightweight Balancer algorithm, nodes could be moved between the on-demand resource manager and the batch resource manager. This allowed a massive reduction in batch wait times. Finally, the benefits of malleable jobs were analyzed and a novel method for scheduling them alongside rigid and evolving jobs was proposed (Suraj Prabhakaran 2015). These findings work to increase system utilization and increase throughput, optimizing the current hardware to be able to handle the increasing demand.

Replicating this at scale and pushing the fault tolerance of the master-worker system proposed is necessary future work. Further work can also be done in replication on different data and workflows for the work done by (Feng Liu 2018). This is because all of the data analyzed was from one lab and in fact only done on a small portion of

that lab's workflow. Dependability studies of these dynamic balancing tools should also be pursued.

3 MPI vs Chapel Locales

3.1 Overview

In this section is will compare the performances of MPI and Chapel locales when running similar load jobs across multiple machines in the Haverford College computing labs. This required the direct translation of existing MPI applications into Chapel with the addition of timing capabilities into both implementations.

MPI stands for message passing interface and is one of the most widely used interfaces to create and run jobs in parallel. MPI is aptly named as its main objective is to transfer information between different tasks and/or processes. These tasks can be running on separate processors across a single machine as well as across a computing cluster or grid made up of many machines. MPI is necessary since the memory of the system may be distributed across multiple machines therefore requiring the transfer of data to complete a job as the results from one node could be necessary for another to complete its calculation. MPI is has also been around since the 1990's and thus has gained the reputation of being "tried and true" (Barker 2015). MPI is a specification of message passing libraries which means that it can be run on practically all machines and using it to parallelize code requires little to no modification of the existing code. These factors along with MPI being widely available and relatively quick have made it the go to method to create parallel programs.

The implementation of MPI generally works in a master-worker architecture. In this architecture one computing node is designated as the master of the application and is the one that distributes the work of the application to the workers. The master is also in charge of the eventual reduction of the workers results as they come in. Each worker simply receives a portion of the work to be done and executes it, returning the results to the master. The way that the job is divided up and partitioned and eventually reduced back together must be implemented by the developer (Barney 2020).

Chapel is a language developed by Cray Inc. with the particular goal to create a language for developing parallel computing applications at scale. Chapel uses locales as its method of running code across multiple machines. With Chapel locales a

developer can actually control what each processor in each machine is doing. Chapel is also unique in how it abstracts the scope of variables especially across multiple machines. Chapel allows for declaration and computation similar to how one would program on an entirely local application. This is what is referred to as *global-view data structures* (Chaimberlain 2015). Chapel also allows for each locale to operate more individually than the typical worker in MPI. Locales can declare and store their own unique variables and each locale doesn't have to operate like it's neighbors. For example a group of locales could be working on a section of a job where they need to be grabbing stored variables from each other but this is independent of another group of locales operating on a separate section of the job which does not need those variables and thus doesn't store or access them. In this way Chapel has easy to access layers of abstraction available to the programmer that MPI simply does not deliver.

The goal of my experiment is to determine if there is a noticeable performance differences between MPI and Chapel locales. I will also be analyzing the benefits and limitations of both implementations.

3.2 Related Work

Work similar to this experiment has been carried out before (Brown 2015). This experiment was done 5 years ago and it appears that the researchers did not have access to Chapel locales. However, the finding in this paper back up what was found before, that even with Chapel locales, the slowdown from MPI to Chapel is significant and is hard to justify with improved accessibility. The research here done though, does suggest that the researchers were testing based on industry needs. This means that accessibility and to coding was likely not considered in the evaluation of Chapel.

3.3 Motivation

The goal of this experiment is to highlight not only the speed comparisons between MPI and Chapel but also to analyze the usability and learning curves that come with them. As researchers are not necessarily the most proficient coders, the accessibility of a language or implementation means saved time developing running and correct code which in turn could get results faster. This is a key part of research as the end goal is for the researcher to get correct results in as efficient a manner as possible. With this goal in mind coding should be a valuable and efficient tool to complete calculations not a burden to learn.

The Chapel language in particular was developed with this mindset. Chapel aims to make development of parallel computing easier and more accessible by utilizing higher levels of abstraction. This is analogous to Python. Python may not be the fastest language, especially when compared to something like C. However, it is extremely popular and used extensively in all type of applications because of how easy it is to use and read. By analyzing the speed and usability of MPI and Chapel, it can be determined whether Chapel is a viable option for researchers in the future looking to run their experiments in parallel.

3.4 Experiment

For this experiment, the code used was an embarrassingly parallel implementation of a Monte Carlo pi estimation. This code functioned by randomly generating "darts" to "throw" at a square board with a unit circle in it. The ratio of the area of the circle to the square is $\pi/4$. To estimate pi simply divide how many landed within the unit circle over the total number thrown then multiply by four. Thus, two real numbers x and y between 0 and 1 are randomly generated and their coordinates determine where the dart landed. This specifically meaning if $x^2 + y^2 \leq 1$ then the dart landed in the circle if $x^2 + y^2 > 1$ then the dart landed outside the circle. This is called embarrassingly parallel as no one dart relies on any other dart thus the work can easily be divided and run across many machines with minimal communication between workers.

This results of this experiment will come from multiple performance runs on varying number of dart tosses. The final timings will be the average of 10 runs per number of tosses. This will be run on multiple variations on the Chapel code where all tasks are run on a single machine, single tasks are run on multiple machines, and multiple tasks run on multiple machines. In an ideal circumstance this would also be replicated in with the MPI code by integrating OpenMP to be able to fun multiple tasks on multiple machines. However, due to the restrictions on the Haverford College CS lab and situation with the current global pandemic, the only version of MPI that was able to be tested was multiple tasks running on a single local machine.

3.4.1 Method

A basic outline of the experimental process follows:

1. The largest step in this experiment is the implementation of existing MPI code into comparable Chapel code. This should be executed as faithfully to a one-to-one translation as possible.

2. Matching timing code should be added to both MPI and Chapel implementations. This will be a wall-clock timing implementation as wall-clock is the only timing method available in Chapel.

3. Execute performance runs on varying sized jobs.

    a) Each final result time will be an average of 10 runs.

4. Analyze and compare the results of the performance runs.

3.4.2 Translation and Timing

The MPI implementation of this code was sourced online from the Macalester College CS website (Sonsalla 2020). This code was almost ready to go for my uses immediately however, the timing used did not incorporate the reduce step and thus some timing modifications were made. The first step of the translation was to write the function that would generate the random number and count the total number of darts that landed in the circle. In both implementations, the code samples below are similar (especially in the for loops) as it is simply called and is not really part of parallelizing the code.

```
// Toss function in Chapel
var numInCircle: int = 0; // locale level var
var randStream = new owned RandomStream(real);
for i in 1..numTossesPerLocale {
  var x = randStream.getNext(); // This gives values 0-1.
  var y = randStream.getNext();
  if ((x*x+y*y) <= 1.0 ) {
    numInCircle += 1; // Increments if dart lands in circle.
  }
}
```

```
// Toss function in MPI C
long Toss (long processTosses, int myRank){
long toss, numberInCircle = 0;
double x,y;
```

```
unsigned int seed = (unsigned) time(NULL);
srand(seed + myRank); // Initializing pseudo-random numbers.
for (toss = 0; toss < processTosses; toss++) {
   x = rand_r(&seed)/(double)RAND_MAX; // Gives value from 0-1.
   y = rand_r(&seed)/(double)RAND_MAX;
   if((x*x+y*y) <= 1.0 ) numberInCircle++; // Counts if in circle.
    }
    return numberInCircle;
}
```

The most important part of the translation was the way in which the code was to be parallelized. In each case it simply initializes the given number of node that are requested.

```
// In Chapel
coforall loc in Locales with (ref finalInCircle) {
      on loc {
      // Do stuff
      }
}
```

Here Chapel uses a simple loop to spin up multiple tasks which are spread across multiple locales also known as different machines. The ref finalInCircle is for the reduce portion of the code as it gives a reference to a shared array to write results.

```
// In MPI C
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
// Do stuff
MPI_Finalize();
```

The MPI code is a bit less friendly to look at as is not in a stated basic format like a for loop as Chapel did. Rather the loop is implied as the program is being fed multiple data streams. The MPI_Init is setting up the number of tasks and data streams that are requested via the command line and Comm_size and rank are simply noting the size each of the tasks will be and the rank is simply the number given to each initialized node. The finalize step simply closes everything and frees resources allocated.

The final portion of the translation was the reduce step in which all of the results that the workers got must be combined into one final answer.

```
// Reduce in Chapel
var total = + reduce finalInCircle;
```

In Chapel this process is slightly clunky as each node needs to write its result into a shared array indexed by its id number. This is what finalInCircle is and as seen in the parallelization code in Chapel it has to be given as a reference to each node. Once all values are in the array this reduce gives the sum.

```
// Reduce in MPI
MPI_Reduce(&processNumberInCircle, &totalNumberInCircle,
1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD)
```

The MPI reduce is built for exactly what its name states. The parameters tell the reduce what is it receiving from the workers, where to put the result, how many things it is receiving from the workers, the data type, the type of reduce to use, the rank of the root or master node, and the communicator used. Much of these parameters are also used in the Chapel implementation but MPI puts them all in one place. For instance in Chapel, each locale must be told where to put its answer in a shared memory array in which the root locale will then use to reduce later. This is in part due to the simplicity of the reduce function that is built into Chapel as it can only function on a given array while MPI can access the data directly from the workers.

To time the speeds of the code it must be noted that Chapel only has wall-clock timing as of the writing of this. Therefore, wall-clock timing was the standard used across both implementations. Timing began at the start of the first toss and ended after the final reduce and calculations were completed. Chapel utilized the "Time" package with the built in timer while MPI C used C's "time.h" with the gettimeofday() function to time in microseconds.

3.4.3 Executing Performance Runs

The performance runs were executed by bash script running on the Haverford College CS lab. Noting that the lab is a shared computing space of the college, all runs were attempted at the lowest traffic times possible on the Ritchie machine. Figure 3.1 gives a summary of the results.

	Time to Completion (sec.)					
Number of Tosses	4 cores on 1 machine (Chapel) Ritchie	4 cores on 1 machine (MPI) Ritchie	6 cores on 1 machine (Chapel) Ritchie	6 cores on 1 machine (MPI) Ritchie	Single core on 6 machines (Chapel)	Mixed 4 cores on 6 machines (Chapel)
60	0.000636778	4.45556E-05	0.0008647	0.0000399	0.0141281	0.0476915
120	0.0008384	0.000057	0.0008283	0.0000402	0.0088346	0.0457614
1200	0.0008442	0.0000505	0.0007968	0.0000409	0.0078923	0.0391887
3600	0.0008035	0.0000737	0.0009083	0.0000498	0.0083161	0.0448344
6000	0.0008771	0.0000892	0.0008354	0.0000619	0.0094539	0.0403889
60000	0.0032616	0.0002903	0.0024599	0.0001843	0.0255192	0.0328529
120000	0.0060194	0.0007477	0.0043739	0.0003635	0.0302922	0.0368683
600000	0.0276458	0.0034313	0.0189964	0.0019305	0.113185	0.0695584
1200000	0.0545351	0.0061287	0.0372457	0.0033245	0.1990251	0.086907
6000000	0.2860245	0.0214597	0.2001094	0.0158516	1.0231444	0.3410732
12000000	0.5843762	0.0390662	0.4164861	0.0304091	2.031794	0.8680037

Fig. 3.1: Summary of all performance runs.

3.4.4 Analysis

During the performance runs the number of tosses was selected in order to highlight if there was any significant overhead and how that might improve as the number of operations increased. The runs also included some larger scale tests to see how the implementations would scale. Figures 3.2 3.3 do not include the 3 largest tests as they obscure the patterns seen in the first few runs.

Looking at Figure 3.2 it is clear that as the jobs scale, Chapel cannot compete with MPI in terms of time to completion. Even given the familiarity Chapel brings to the programmer does not justify the slow down seen especially as jobs grow much larger than what was tested. This trend can been seen in the final rows of Figure 3.1 where time to completion is upwards of 10x slower than that of MPI.

Figure 3.3 is great example of overhead causing a slow down in speed. As more tasks are spun up it take more overhead to keep track of them and do more reduces. Therefore, when the number of operations is low, more workers actually run significantly slower. However, as the jobs increase in size, the efficiency of more workers shows and there is a speed up. This is a perfect demonstration of why the right-sizing method proposed in the literature review (Nathaniel Kremer Herman and Thain 2018) is so important. Being able to correctly size a parallel job can see significant jumps in throughput.

The second part of this experiments revolves around the experience of learning and using both MPI and Chapel. Since MPI can be used in multiple languages, I chose to use the version in C as I was most comfortable already in C. MPI from the offset is not very approachable since it looks like nothing that I have used before and is simply a suite of functions with a large array of parameters that need to be learned. Chapel on the other hand looks like other languages and followed familiar syntax. This made it easy to read and understand code examples. The use of loops and variables with many built in functions allowed for me to quickly pick up Chapel and

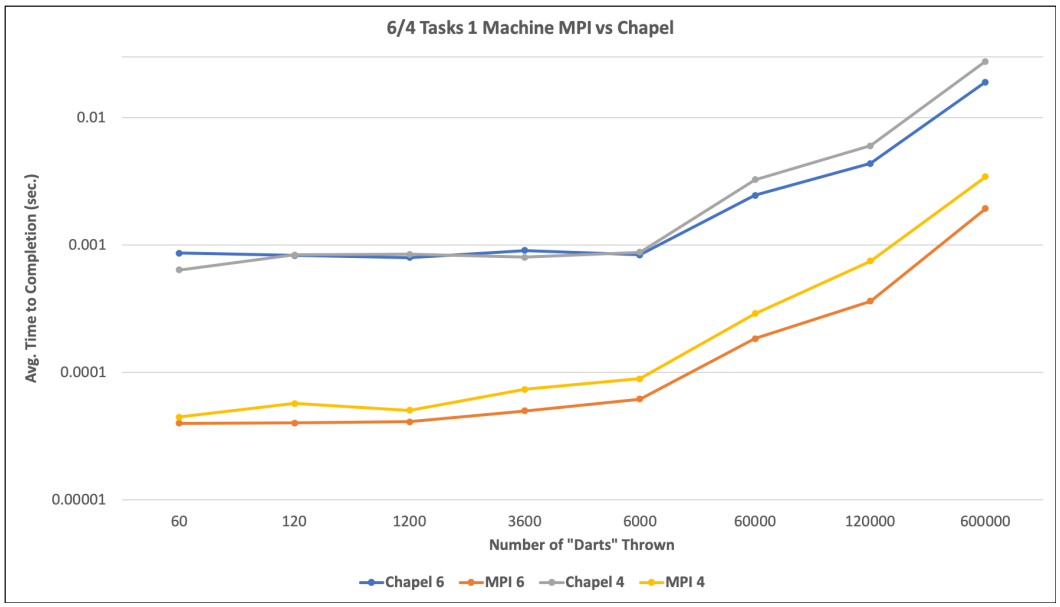


Fig. 3.2: Performance of Chapel vs. MPI running 4 and 6 tasks on a single machine.

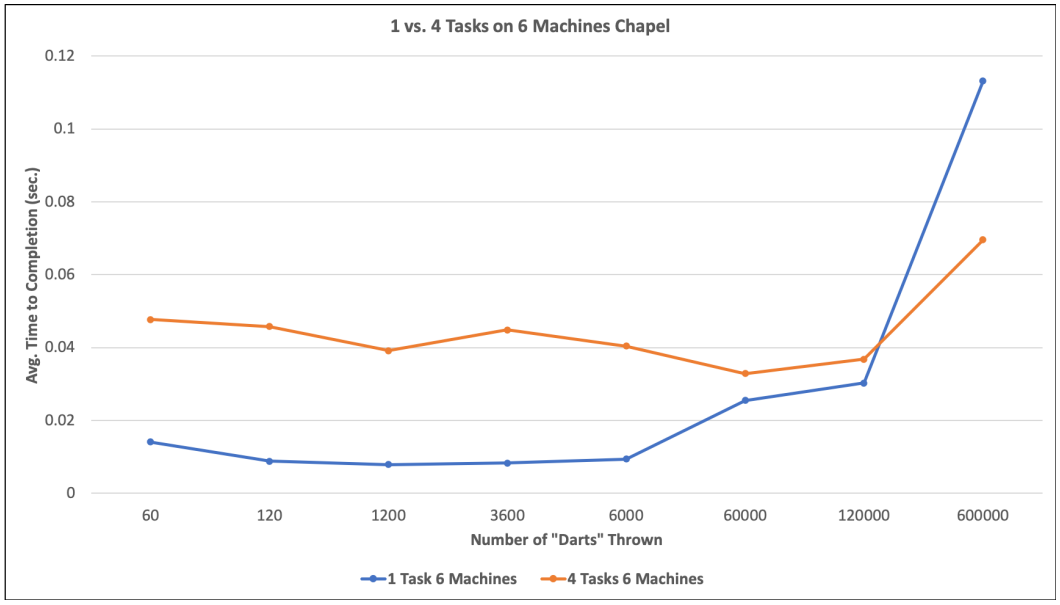


Fig. 3.3: Chapel Performance running 4 tasks on 6 different machines.

be able to write some simple parallel programs very quickly. MPI however, has been the industry standard for a long time and thus the amount of support and knowledge online is vast and much more available than that of Chapel.

On the whole Chapel is a much more accessible and easy to learn language. The high level of abstraction mean that a programmer coming from say Python would be able to pick it up quickly. The structure of the language is familiar and easy to navigate. This structure also allows the programmer to easily scale their applications into a parallel. This is one of the biggest benefits in Chapel. Converting code from serial to parallel is incredibly easy and the control over scale is easy to understand and can be tune in a matter of minutes. I was able to convert the Monte Carlo code from single task on many machines to multiple tasks on one and many machines with only a few lines of code and minimal experience in Chapel. Thus as a learning tool or as an introduction into parallel computing for smaller applications this language could see great use. The power to be able to quickly scale a project with minimal coding experience has value.

MPI on the other hand requires a significant background in the languages it can be implemented in, neither which are as friendly as Chapel (C and Fortran). These languages are themselves very fast however have much lower abstraction than Chapel. If speed is the goal of the application Chapel cannot stand in for MPI especially at scale. Not only does it run faster it also compiles faster. The learning curve for MPI while greater than Chapel is more than justified by the speed up in the end.

3.5 Challenges

1. The version of Chapel that handles multi-locale compiling is not the standard version available and required a special installation and the building of a separate Chapel compiler.

    a) This also required gasnet as the underlying communication architecture.

    b) There were a lot of issues of then being able to use the compiler as it seemed the environment had to be rebuilt too much and sometimes still would fail.

        i. With considerable help from Dave Wonnacott I was able to get a script running to act as the “compiler” for multi-locale Chapel programs.

2. Currently the version of MPI on the Haverford machines does not seem to be able to run MPI across multiple machines. However, it can be run on multiple processors on the local machine.

    a) The provided hostfile does not seem to be accessed.

4 Conclusion

Parallel computing is a vital resource for many researchers and applications today. With an ever growing demand on computing resources, the need to find more efficient methods to squeeze every last ounce of performance out of our hardware also grows. In the literature review three new proposed methods looked at how to improve throughput on institutional sized batch clusters. These methods look into how the system itself could be optimized to increase the output of researchers. The following experiment however, looked into a different manner in which parallel computing could be optimized for researchers, how the applications themselves parallelized.

MPI and Chapel are two very different implementations that allow a programmer to create applications that run in parallel. They can be split into two different categories of optimization. The first is optimizing the usability and thus the development speed of the applications themselves. This is where Chapel excels over MPI. The Chapel language uses familiar data structures and syntax that a beginner programmer would be able to read with a small learning curve. This allows for less time to be spend developing code that runs how the researcher wants it and also improves human readability. However, the trade off is in execution time.

This brings us to the second category, performance optimization. This is where MPI trumps Chapel. MPI's speed has been a big reason why it has been so widely used in parallel computing for so long. MPI runs really fast for the size and parallelization that most researchers need. This is certainly true when it comes the speed difference between MPI and Chapel from the experiment in this paper. However, the lack of ability to greatly scale and test both implementations suggests an avenue for future research. In total a potential combination of faster hardware scheduling combined with the increased speed of development from languages such as Chapel could lead to a middle ground. Thus, both the researchers can spend less time coding and tuning their applications and the hardware that they run it on can make up for the potential slow down in execution time.

Bibliography

Barker, Brandon (Jan. 2015). *Message Passing Interface (MPI)*. cornell.edu.

Barney, Blaise (Apr. 2020). *Message Passing Interface (MPI)*. Livermore national lab. URL: https://computing.llnl.gov/tutorials/mpi/.

Barry Wilkinson, Michael Allen (2005). *Parallel Programming Technques and Applications Using Networked Workstations and Parllel Computers*. Ed. by Kate Hargett. 2nd Edition. Upper Saddle River, NJ: Pearson Education, Inc.

Brown, Laura (June 2015). *A Preliminary Performance Comparison of Chapel to MPI and MPI/OpenMP*. Powerpoint in chapel-lang.org.

Bryce Allen Rachana Ananthakrishnan, et. al. (May 2017). "Globus: A Case Study in Software as a Service for Scientists". In: *ScienceCloud'17*, pp. 25–32.

Chaimberlain, Bradford L. (Nov. 2015). *Programming Models for Parallel Computation*. Ed. by Pavan Balaji. Chapter 6. MIT Press. URL: https://chapel-lang.org/publications/PMfPC-Chapel.pdf.

Daniel A. Reed, Jack Dongarra (2015). "Exascale Computing and Big Data: The Next Frontier". In: *Communications of the ACM* 58.7, pp. 56–68.

Feng Liu Kate Keahey, et. al. (Nov. 2018). "Dynamically Negotiating Capacity Between On-Demand and Batch Clusters". In: *SC18*.

Geoffry C. Fox, Shantenu Jha (June 2017). "Conceptualizing A Computing Platform for Scince Beyond 2020: To Cloudify HPC, or HPCify Clouds?" In: *IEEE*.

Gustafson, John L. (1988). "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5.

Harshitha Menon, Laxmikant Kalé (Nov. 2013). "A Distributed Dynamic Load Balancer for Iterative Applications". In: *SC13*.

Katherine Yelick Susan Coghlan, et. al. (Dec. 2011). "The Magellan Report on Cloud Computing for Science". In: *Office of Advanced Scientific Computing Research*.

Marco A.S. Netto Rodrigo N. Calheiros, et. al. (Jan. 2018). "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges". In: *ACM Computing Surveys* 51.1.

Moore, Gordon E. (Apr. 1965). "Cramming more components onto integrated circuts". In: *Electronics* 38.8.

Nathaniel Kremer Herman, Benjamin Tovar and Douglas Thain (Nov. 2018). “A Lightweight Model for Right-Sizing Master-Worker Applications”. In: *SC18*.

Qiming He Shujia Zhou, et. al. (June 2010). “Case Study for Running HPC Applications in Public Clouds”. In: *HPDC ’10*, pp. 395–401.

Sonsalla, Hannah (Apr. 2020). *Monte Carlo Estimate Pi*. online at selkie-macalester.org. URL: `http://selkie-macalester.org/csinparallel/modules/MPIProgramming/build/html/calculatePi/Pi.html`.

Suraj Prabhakaran Marcel Neumann, et. al. (May 2015). “A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications”. In: *IEEE*.