A FAULT TOLERANT GRID GENERATION TECHNIQUE

By

Matthew D. O'Connell

Steve L. Karman, Jr.
Advance Research, Pointwise Inc.
(Chair)

James C. Newman III
Professor of Computational Engineering
(Committee Member)

Michael A. Park
Research Scientist, NASA Langley
Research Center
(Committee Member)

Craig R. Tanis
Assistant Professor of Computer Science
(Committee Member)

A FAULT TOLERANT GRID GENERATION TECHNIQUE

By

Matthew D. O'Connell

A Dissertation Submitted to the Faculty of the University
of Tennessee at Chattanooga in Partial Fulfillment of
the Requirements of the Degree of Doctor of
Philosophy in Computational Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2016

ABSTRACT

Automatic and parallel mesh generation has been highlighted as a bottleneck for large scale automated Computational Fluid Dynamics analysis. The desire for large scale automated CFD is driven by the growing computational capabilities in large scale supercomputers. Unfortunately, as compute clusters grow in size, they also suffer more failures. Left unchecked, the increased frequency of failures may stymie any efforts to fully utilize these machines.

This work aims to tackle one component required for automated large scale engineering analysis by developing a fault tolerant mesh generator. The mesh generator uses a novel communication layer written using the transport layer ZeroMQ and is made fault tolerant through an integrated in-memory checkpoint and recovery strategy. Benefits of using in-memory checkpoints vs traditional in-disk checkpoints are discussed. By relying on in-memory checkpointing, it is demonstrated that the mesh generator to be capable of generating Cartesian meshes in parallel. The generator continues to operate even while the compute cluster it is running suffers failures. The generator is shown to be high performing, including being capable of generating an 8.6 billion element mesh in just over 1 minute while creating multiple in-memory checkpoints.

DEDICATION

This work is dedicated to my family.

ACKNOWLEDGEMENTS

This work represents not only long hours at the keyboard, but a rich array of conversations, encouragement, triumphs, and failures. No man is an island. First, I would like to thank my advisor, Dr. Steve Karman, for his inexhaustible patience, and guidance. Additionally, I would like to thank the committee members: Dr. Park, Dr. Newman, and Dr. Tanis. Each helped me keep an eye on target.

I would like to thank the Computational Aerosciences Branch at NASA Langley Research Center for the support I received, including access to the compute hardware that was critical to conduct this research. But beyond funding and computer resources, this work would not have been possible without the invigorating conversations, mentoring, and culture of excellence of this fantastic group.

I also want to thank the UTC SimCenter for the support throughout my graduate work. And to each of the professors with a contagious excitement for high fidelity simulations. A special thanks goes out to Kim Sapp for all the help navigating between two organizations.

A big thank you to Judith Hill of the Scientific Computing Group at Oak Ridge National Laboratory for the conversations and insights into current and future hardware on leadership class computer systems.

Research is the extension of curiosity, observation, critical thinking, and hard work. I would like to thank the following educators and mentors who not only taught me science, but these values

as well: Mr. Brown, Mr. Digby, and Dr. Taylor. I also want to thank an elementary school teacher's assistant, whose name I've long forgotten, but whose words have stuck with me for more than two decades.

Finally, I want to thank my family; and it's a big one! My aunts and uncles, my grandma, my cousins (MBAAMTJR+M), my in-laws Cindy and Kempf, my sisters Amber and Julia, and especially my parents Dawn and Kevin, and my wife Emily. Your support, encouragement, and love were vital not only to the completion of this work, but the years of preparation which came before it. I am eternally grateful for the sacrifices you made so that I could one day write these words.

Thank you, everyone!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Today's supercomputers are built on a dizzying complexity of hardware and layers of software that all must work in concert to provide a platform for high performance computational analysis and design. Any one hardware or software component is a link in the proverbial chain where one weak link defines the chain's strength. The larger we make our supercomputers, the longer we make our computational chain, and the more likely it is that our chain will have a weak link. If any one link fails, the simulation stops.

There is a push currently within supercomputing to reach "exascale" by building a supercomputer which is capable of 10^{18} FLOPS (floating point operations per second). As of June 2016 the largest supercomputer in the world is the Sunway TaihuLight computer at the National Supercomputing Center in Wuxi, China. [1]. The Taihulight consists of 10 million compute cores across 41,000 compute nodes. The Taihulight has a theoretical peak computing capability of 125 petaflops, and Linpack performance of 93 petaflops. However, to reach a machine capable of 1 exaflop, a computer will need to be built with approximately 100 times more compute power.

Exascale supercomputers are expected to exceed the current machines- future machines. Exascale machines were expected to have millions of cores and tens to hundreds of thousands of

nodes. [2,3] More recently these estimates have been revised upwards and now exascale machines are expected to contain between 100 thousand and one million nodes. [4] Unfortunately, by increasing their size, these machines will have more components that could fail. Increasing the number of hardware components will put extreme scale applications at risk of suffering system failures unless future hardware is more resilient.

Unfortunately, it does not appear that newer technology suffers less failures. [5–7] A study published in 2007 evaluated 10 years of failure data from Los Alamos National Laboratory covering over 24 thousand processors, it summarized "...there is little indication that systems and their hardware get more reliable over time as technology change". [8] The Intel Corporation in a patent granted in 2011 warned that "due to various technology related issues, cores in a many-core processor will exhibit higher failure rates than single core or dual-core processors." [9] Evidence supports this as failure rates have increased at about an 8% rate between hardware generations. [10, 11] It has even been shown that compute clusters at higher elevation suffer more failures due to hardware interactions with cosmic-ray-induced neutrons. [12, 13]

Researchers have raised concern that future exascale computers will be less resilient than current generation high performance computing (HPC) systems, and resilience has been often highlighted as a critical component of exascale systems. [5, 7, 14–17] A DARPA funded "ExaScale Computing Study" finds that an exascale machine is expected to suffer a failure every 35-39 minutes. [3] In 2011, the International Exascale Software Project Roadmap warned that "insufficient resilience of the software infrastructure would likely render extreme-scale systems effectively unusable". [18]

We don't have to look to future extreme scale computers to find the impact of fault tolerance. Between August 2008 and February 2010, Jaguar (the 3rd largest supercomputer on the top 500 at the time) suffered from 2.33 average hardware failures a day. [19] Zheng has given a surprising analysis at these smaller scales. Assume a hardware node has an average lifetime of 20 years (a *very* generous assumption). A job running on 5000 nodes for 400 hours has a failure probability of 99.9989% ! [20]

HPC applications are not only subject to hardware failures, but software bugs as well. Schroeder et al. reported that while 50% of failures occur because of hardware failures, a substantial 20% come from software bugs. [8]. In order to make effective use of large compute clusters, we must be able to tolerate some system failures regardless of the source. We must keep the simulation running.

This work aims to support the future of CFD running on large distributed compute clusters. The current common CFD workflow consists of geometry clean up, mesh generation and computing numerical solutions to conservation equations describing fluid flow. As Figure 1.1 illustrates, early stages in the CFD process require much more user effort to prepare geometry and generate the mesh and much less user effort to run the flow solution. On the contrary, early stages in the CFD process require relatively small computer resources while numerically computing the solution can be very computationally expensive. Furthermore, the CFD process is dependent on previous steps. A practitioner may need to revisit each step in the process after analyzing the flow solution.

The significant user effort required for geometry preparation and mesh generation can limit the complexity of problems that can be solved quickly. If it is to become common that complex

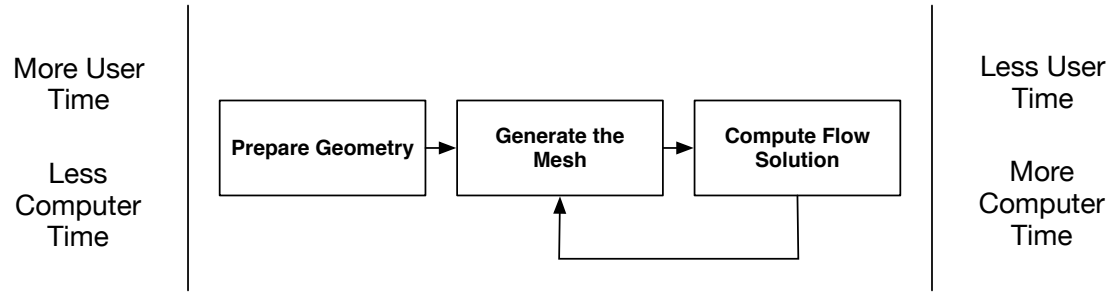


Figure 1.1 A common CFD workflow

engineering solutions can be found quickly, the entire computational analysis process must be automated.

In 2014, a survey study of the state of the art of Computational Fluid Dynamics was published titled "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences." The study highlighted mesh generation and adaptivity as a key issue which must be addressed to achieve robust and reliable CFD simulations. The study states: "Mesh generation and adaptivity continue to be significant bottlenecks in the CFD workflow, and very little government investment has been targeted in these areas." [21]

Meshing has been identified as a workflow bottleneck outside of aerospace applications. In the summer of 2015 the Department of Energy held a workshop on turbulent flow simulation at the exascale. Among the applied research topics for large scale turbulent flows was nuclear energy. A review of the workshop cited "meshing is a bottleneck in the workflow at large scales" and "is expected to be amplified on future architectures." [7]

Automated CFD analysis will require heavy investment in parallel mesh generation, geometry handling. It will also require coupling the entire CFD process into software frameworks which can be deployed onto large scale, state of the art machines. These future CFD applications may need to be resilient at all stages of analysis from geometry, to mesh generation, to simulation, to post processing. Furthermore, it is expected that disk based read and write speeds will continue to lag behind main memory read and write speeds. A fully automated CFD package may want to avoid using permanent storage. The entire CFD analysis workflow, geometry, mesh, solution, error analysis, mesh adaptation, and post-processing, will all need to be integrated into a software ecosystem that runs diskless.

1.2 Objective

The primary objective of this work is:

> Develop and demonstrate a technique for mesh generation which is capable of running on compute clusters and recovering from process failures.

There has been no demonstration of a parallel mesh generator with an integrated in-memory fault tolerance scheme to the author's knowledge. To accomplish this goal, this work will also describe a simple messaging system that is capable of detecting failed processes and recovering from those failures. Additionally, it will describe a fault tolerant checkpointing framework which is capable of redundantly storing data within a compute cluster to increase resiliency of parallel applications.

Many events can cause a computational task to fail including hardware failures such as Central Processing Units, Graphics Processing Units, Random Access Memory, board, network, and software. This work focuses on detecting and recovering from single point, and small point failures. Single point failures are the most common failures in compute clusters accounting for approximately 70% of all hardware related failures. [13, 19, 22–24]

The mesh generator used in this work generates isotropic Cartesian meshes using an octree based technique. The mesh generator is implemented for use in parallel with distributed memory compute clusters. The mesh generator is capable of generating isotropic body fitted grids for simple 3 dimensional geometries using projection based techniques. Projection mesh generation techniques are not a mature technology, and this work makes no advancements in this area.

This work assumes that it will be desirable to continue to increase the size and complexity of problems analyzed by CFD techniques and increasing the capability of CFD analysis. This work, and the software systems it describes, was developed based on recent predictions about the future of computer hardware. Critically, this work assumes that future computer hardware will be no more resilient than today's hardware and that future communication systems will continue to be faster than permanent disk based storage schemes.

If automation for large scale capability CFD is a goal of the industry then the entire automated framework must embrace failure as a normal part of operation. Each stage in the CFD workflow: geometry queries, mesh generation, and computation of the flow solution, must be prepared to react when failures occur. It should be expected that the majority of failures will occur while the flow solution is being computed because flow computation will consume the majority of the run time. Still, a failure during any stage must be anticipated. This work chooses to focus on incorporating

fault tolerant techniques into the mesh generation stage. Mesh generation can be used as a test bed to explore for how fault tolerant techniques might be incorporated into other stages in the CFD workflow.

1.3 Grid Generation

Automated grid generation techniques for Computational Fluid Dynamics simulation have seen a rise in popularity in recent years. Generating large scale meshes with these methods has, up to this point, largely been relegated to generating the mesh on a "grid generation" machine. To handle large meshes, the "grid generation" machines can be equipped with more than 10 times the memory of average machines. Large meshes often take hours or even several days to generate.

The power of next generation compute clusters will enable simulations using exceptionally large grids. There has been a growing interest in parallelizing the mesh generation process to harness the power of the next generation of computing hardware.

1.3.1 Grid Generation with Spatial Trees

The grid generation landscape is filled with a wide array of different techniques spanning both the structured and unstructured domains. Because of their speed and simplicity, a growing number of grid generation programs are using tree based methods. Techniques which use spatial trees can generate elements very quickly by focus on creating Cartesian aligned elements. The most common spatial tree is the octree. Octree based techniques generate isotropic Cartesian elements.

In recent years a number of researchers have developed dynamically load balanced parallel octree methods. Octor [25] and p4est [26] are both examples of parallel octree codes, which use space-filling curves to repartition and dynamically load balance during the mesh generation process. Very large scale octree meshes have been generated quickly using parallelization. In 2014 Lintermann et al. demonstrated generating 17 billion elements on a 65 thousand core compute cluster in 17 seconds. [27]

Cartesian aligned meshes can be generated very quickly; however, high Reynolds number CFD techniques generally require body fitted grids with anisotropic viscous spacing within the boundary layer. Some mesh generation efforts have explored hybrid techniques that use Cartesian aligned elements away from the boundary layer but transition to body fitted anisotropic elements near the geometry surface.

Betro introduced a hybrid method that relied on an initial surface mesh. The highly anisotropic viscous layers were generated using an extrusion based technique. The nearbody and offbody regions of the mesh were connected using a tetrahedral based technique.. [28] Betro did not use an isotropic octree and instead used a specialized binary tree called a split tree which allowed for anisotropic elements in the offbody region.

Some grid generation efforts aim to eliminate the need to generate an initial surface mesh. Hexagrid is one such effort which generates Cartesian elements throughout the domain and then projects and optimizes elements near the body surface. [29] Hexagrid has seen increased performance with the addition of shared and distributed memory parallelism through use of the building cube method. Boxer Mesh also utilizes a hybrid technique generating Cartesian elements in the

offbody region and body fitted viscous elements near the surface. [30] Distributed memory parallelization has also been implemented in Boxer Mesh and demonstrated out to 48 processors. [31]

1.3.2 Projection Based Near Body Grid Generation

Many common automatic mesh generation techniques are based on hierarchical trees for offbody grid generation combined with a near body gridding technique. Some techniques include cut-cell methods such as in Cart3D, [32] SPLITFLOW, [33] and PHUGG. Other techniques, such as Hexagrid, [34] project nodes from the offbody mesh onto the geometry. FASTAR [28] was developed using a type of binary tree called a split-tree. It uses tetrahedral meshing to combine a hierarchical technique for offbody grids with a body conforming viscous mesh generated using extrusion.

The proposed work focuses on Cartesian off body grids blended to unstructured body conforming near body grids. These techniques can be referred to as projection based techniques since during the near body generation process nodes are projected onto the surface of the geometry.

A common technique during projection based grid generation is to capture sharp geometry features using explicitly defined geometry. [34–36] With these techniques, explicit points and curves are captured during the creation of the nearbody grid. The explicit geometry can be a series of NURBS patches, or more simply, the geometry may be represented by a triangulated surface mesh. [34, 35]

An alternative approach to explicitly defined geometry is to use implicit geometry. Implicit geometry can be represented using a distance field where the location of the geometry surface

is implicitly defined as an isocontour at a specified distance. Distance fields have been common geometry representations in computer graphics as a morphable geometry representation for animation. [37, 38] Dawes et al. have developed Boxer Mesh to generate grids on geometries represented implicitly by level sets. [31] They have further used the level sets of the distance function to create viscous layers in the near body region. [30]

To date, no significant fully automated nearbody projection technique has been demonstrated as robust to any input geometry. Hexotic cites challenges with sharp geometric angles below 30°. [35] In some instances, Hexagrid requires manual labor to clean input surface triangulations. Boxer Mesh will sacrifice geometry integrity to obtain what they call a "solveable" mesh. Despite these continuing challenges, nearbody projection techniques have been deployed for use in computational analysis.

A future fully automated CFD workflow will need to automatically generate very large meshes in parallel for geometries with arbitrary complexity. The combined grid generation and flow solver must be capable of simulating compressible and high Reynolds number flows. For this reason a hybrid approach will be used for this research following a hybrid approach. The offbody elements will be generated using a parallel octree based technique. The nearbody grid will be optimized and projected onto the geometry surface to create body conforming grids. Projection based techniques such as those used with Hexagrid and Boxer Mesh are still areas of active research. This work makes no contributions in this area. Furthermore, this work focuses only on isotropic grids. Despite this limitation, the projection based nearbody technique was carefully chosen because it has been demonstrated to be capable of supporting anisotropic viscous layers.

1.4 Fault Tolerance

The term “fault’ will be used as a catch all term to mean any event which causes the computational environment to be unresponsive. The cause of the failure could be related to the underlying hardware or software of the compute cluster. Not all faults are recoverable. This work focuses on faults of a single component, or small number of components, within the compute system. Small, isolated, failures are the most common type of failure. Full system failures are less common. Full system failures are not addressed by this work.

The main technique to be studied by this work to achieve a fault tolerant implementation is fault recovery. In fault recovery techniques, first a fault is detected by the implementation, and then some method is applied to recover from the fault. Not all faults will be recoverable, for example a compute cluster which unexpectedly loses power without a backup supply may simultaneously shut down every compute node killing any application currently running. However, some faults are recoverable given some additional prior preparation. For example, a common fault recovery technique used by many parallel simulation codes today is disk based checkpointing, where a copy of the working state of the application is saved to disk periodically. If a fault brings down the simulation, a portion of the work can be recovered by restarting the simulation from the last saved checkpoint.

The following sections cover this two step process (detect fault, recover from fault) in more detail. This work is focusing on fault recovery techniques for applications running on commodity hardware using the Message Passing Interface (MPI) as the parallel programming paradigm. [39] This combination of commodity hardware and MPI has been the work horse for HPC simulations since the 1990s. [40–42]

1.4.1 Fault Detection

The first challenge for a fault tolerant application is that the application or the parallel runtime has to detect that a problem has occurred. Timeouts are a common concept in networking used to detect a failure within a network. In the context of this work, a timeout is a specified length of time during which a receiving node in a network must send an acknowledgement that the message was received. If that acknowledgement is not returned to the sending node within the time allotment, then the sending node assumes that a failure has occurred with the receiving node. Many researchers have proposed detecting timeouts during communication in MPI programs. [19, 20, 43–45]

Timeouts could be detected different ways. One method was proposed by researchers from the University of Illinois at Urbana-Champaign using *buddy processors*. [19, 20] Buddy processors are two or more parallel tasks which are in charge of keeping track of each other. They periodically message each other to ensure the buddy is responding to infer functionality. If a parallel task misses a scheduled status update, its buddy reports the task as being lost.

Other researchers have proposed a ticketing system where each message sent requires a "ticket" to be sent back confirming receipt of the message. [46–48] If the ticket is not received in a prescribed amount of time, then the sending process assumes the receiving process has gone bad.

The fault tolerant "heartbeat" has also been used in different fault tolerant implementations. [44, 45] With a heartbeat, a controlling thread periodically sends a request for a status update both internally to other threads on the same process and externally to other nodes. If a thread or external process misses a status update, then that process or thread is determined to have failed. Heartbeat detection could be centralized, with each node periodically sending update messages to a managing node. Heartbeat detection can also be decentralized, as is used with *buddy processors*.

Some researchers have proposed using fault prediction rather than timeout based fault detection. Chakravorty et al. have proposed predicting hardware failures by using temperature sensors on compute hardware. [49] Researchers at IBM have used Bayesian network models based on logs collected from a 350 node cluster over the period of one year. They were able to predict hardware failure to within a 70% accuracy. [50]

1.4.2 Checkpointing and Recovery

Checkpointing is a very common fault tolerance algorithm dating back to the 1960's. [51] In fact, most large scale HPC applications already implement fault recovery based on checkpointing. Fun3D for example can write out the flow field and recent history as a restart file periodically. [52] If a hardware failure kills a job, it can be restarted from the last restart file. This type of fault recovery is referred to as a disk-based checkpoint since the checkpoint is written to a reliable storage disk. Checkpointing currently utilizes 75-80% of the I/O traffic on current HPC systems. [53]

However, checkpointing on disk can be prohibitively expensive. A disk based system checkpoint can take up to 40 minutes for top performing machines depending on the size of the checkpoint. [8] Large checkpoints can incur an overhead as large as 20-25% of the total running time [54, 55], and may even be less performing than triple redundant calculations at exascale. [56]

A more efficient checkpointing strategy involves in-memory checkpointing. In-memory checkpointing creates a checkpoint which is stored in-memory redundantly across compute nodes. Since the communication network is often many times faster than file I/O, in-memory checkpointing has less of an impact on the performance of the application. In-memory checkpointing has been

shown to be up to 50x faster than than checkpointing based on a reliable shared Network File System (NFS) disk. [19]

Checkpointing techniques vary in the number of in-memory and disk based checkpoints that are created. The simplest in-memory checkpoint technique is similar to a RAID1 disk configuration where two copies of the data are stored in the cluster. This allows for any one node to fail and the run is still recoverable. Checkpoints can also be both in-memory and in-disk for further resiliency.

Checkpoints can be application specific or be application agnostic. Many application agnostic solutions require checkpointing the entire address space of an application. Some of these techniques virtualize the entire application environment and therefore require little or no application changes. Co-Check was the first MPI implementation to use virtualization for MPI processes. [57]. Researchers at Illinois have written an implementation of MPI on top of the Charm++ runtime which virtualizes the entire MPI process and virtual address space associated with the process. [20] That state is checkpointed redundantly in-memory and in-disk. All of these works can use recover from failures by reverting the state of the application

Bronevetsky et al. also implemented full application level checkpointing. In their work, function calls could be placed into a C application's source code. The function calls identify places in the code where the application may checkpoint. The entire application is then passed through a custom precompiler which instrumented the code to save the entire address space of the application.

Berkeley Lab Checkpoint/Restart (BLCR) is a widely used checkpoint / restart library. BLCR is a high performing checkpoint and restart system which can checkpoint an entire application's working environment including registers, file descriptors, and signal handlers. [58] BLCR

uses specific assembly code to save state of CPU registers. While this level of integration enables full environment checkpointing, it does so at the cost of portability as hardware paradigms change.

Techniques which checkpoint the entire application address space can be simple to implement with legacy applications, but they come at a high memory cost. For example, consider an implicit PDE solver. A virtualized checkpoint would store the current solution state and the mesh used for discretization. However, the checkpoint would also contain the jacobian matrix of the implicit system. The jacobian matrix will likely require much more memory than the solution state or the mesh. The checkpoint could be made smaller by not storing the jacobian matrix, and instead recalculating the matrix during restart. However, virtualization techniques will not make this optimization.

Recovering the state of the entire system from a coordinated checkpoint is a simple recovery technique, but it in the event of a failure coordinated recovery requires a lot of redundant calculations. An alternative technique is message logging. With message logging, only the failed process needs to be recalculated from the previous checkpoint. During execution any messages sent are logged. In the event of a failure, the failed process is restarted from the previous checkpoint, and any messages sent since that checkpoint are reapplied from the log. The comparative performance of these two techniques depends on the rate of failure and the amount of data which needs to be checkpointed. [59]

1.4.3 Reliability and Availability

Reliability and availability are two common measures of stability for High Performance Computing systems. This work will adopt the definitions of availability and reliability from the work of Torell et al. [60] Availability "...is the degree to which a system or component is operational and accessible when required for use." While reliability is "is the ability of a system or component to perform its required functions under stated conditions for a specified period of time."

Availability is simply the ratio of uptime to total time for some interval. Many IT organizations have goals of reaching well over 99% availability on their services. However, availability is not the most important measure of stability for HPC systems. Even short interruptions requiring no repair, such as a memory error which causes a node to reboot, can cause a researchers application to crash. These short interruptions may not dramatically affect a system's availability, but instead they decrease its reliability. It is this reliability that is critical for long running and large scale simulations.

The serviceable lifetime of a component or system are often reported using Mean Time To Failure (MTTF) and Mean Time Between Failures (MTBF). These terms are often incorrectly used interchangeably. While these two metrics are related, they are distinct. Figure 1.2 illustrates these two concepts.

Common understanding of serviceable lifetime is complicated further by misconceptions that vendor reported MTBF is a measurement of the expected number of hours a component will operate before failing. The mean expected lifetime of a hardware component reported by hardware vendors can be many years long [61–63]. However, these reported measurements are inferred from necessarily shorter duration tests. An empirically measured mean lifetime of a component would

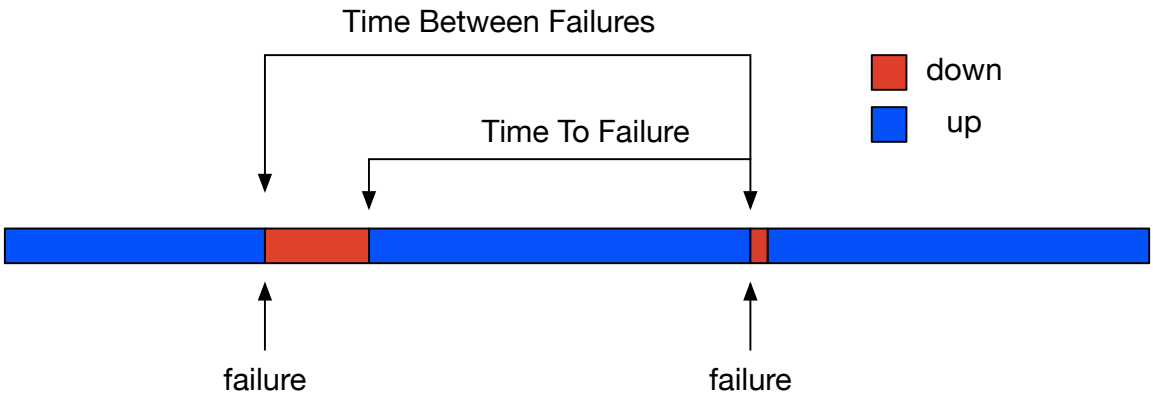


Figure 1.2 Time between failures and Time to failure

be useless information by the time it could be verified. Instead a sample batch of the hardware is tested and the reported MTBF is the testing length divided by the failure rate. A similar approach could be used to determine the lifetime of humans by determining the cumulative failure rate of humans at 25 years. The resulting expected MTBF for a human would be nearly 800 years! [60] Vendor reported MTBF can still be useful in comparing stability of different hardware components, however it should not be used in predicting the fault rates in HPC machines.

1.5 Intra-Process Communication

Problems with failure detection and failure recovery have been shown to be solvable in one form or another, but MPI still remains a major hurdle. The MPI standard defines that default behavior when MPI has detected a failure is that MPI aborts. [64] That default behavior can be

changed with the MPI settings MPI_ERRORS_RETURN where an MPI error could be returned from an MPI call, returning control to the user application. Some researchers have pointed to this as an indication that MPI in its current standard can be fault tolerant. That conclusion is not universally agreed upon. Since version 2.2, the MPI standard states: *"MPI does not provide mechanisms for dealing with failures in the communication system ... Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures."* [39, 64, 65] Even if the default behavior is set to MPI_ERRORS_RETURN the standard does not define how MPI should behave after the error has been detected; furthermore, not all MPI implementations enforce MPI_ERRORS_RETURN. [66] The MPI fault tolerance working group made a proposal for MPI 3 to change this behavior, however as of MPI 3.1 which was finalized June 4, 2015 this change has not been accepted.

There are different strategies used to circumvent this oversight by the MPI standard. A portable approach has been proposed by researchers at the University of Tennessee, Knoxville called Checkpoint-on-Failure. [67] This technique uses double in-memory checkpointing during the execution of the application and sets the default behavior of MPI to MPI_ERRORS_RETURN. If an mpi error is detected by a process, that process writes its checkpoints to reliable disk storage and then exits without making any additional MPI calls - including MPI_Finalize(). Exiting without calling MPI_Finalize() is detected as an error and MPI eventually returns with a failure notification on every process. The application can then be restarted by resubmitting the job into the queue system. This technique of checkpoint-on-failure only requires an MPI implementation which is fully MPI-2.2 compliant and allows for fault tolerance in an application without the overhead of on-disk checkpointing unless a failure is detected.

Despite the MPI standard lacking direction for with fault tolerance, there are a myriad of proposals for extending MPI standard to handle fault tolerance and many implementations of MPI have already implemented some extensions. These include AMPI [20], FT-MPI [68], LA-MPI [43], MPICH-V [69], and MPICH-V2 [70]. Additionally there is an MPI working group devoted to fault tolerance in MPI; [71] though the MPI standard has not adopted any proposals regarding fault tolerance.

It is not very productive to wait for a fault when studying fault tolerant implementations. Furthermore, if you want any product to have a particular behavior you must test for that behavior. The video streaming service Netflix has embraced failures as part of normal operation. Netflix runs a series of tools collectively called "Simian Army." These tools intentionally attack running instances of their video streaming service by deliberately causing different faults to occur. [72] Zheng, et al. have also decided to proceed in researching fault tolerant algorithms and application implementations by simulating hardware failures. These researchers inject a DieNow command into their application at random to test their applications fault recover. The DieNow command forces a process to stop responding to any communication, but it does not cause an MPI error. This has allowed researchers to test their application's fault recovery mechanisms and continue studying different algorithms and implementations while bypassing MPI's current undefined behavior.

1.6 Summary

In-memory checkpointing has emerged as a simple and high performing solution to combat the growing instability in compute clusters. Currently, the author knows no implementation of fault tolerant grid generation.

In practice, grid generation is still widely a serial process. The serial nature of mesh generation has been identified as a bottleneck for large scale computational analysis. Parallel implementations of grid generation techniques have emerged with varying degrees of success and scalability. Body fitted projection techniques, like the used for this work, have not been shown to scale for machines larger than a few tens of cores. Furthermore, scalable parallel generation of body fitting grids is still an open problem using any technique. The best scalable mesh generation techniques use tetrahedral dominated mesh generation. Some tetrahedral dominated mesh generators have been demonstrated to scale using a few hundred cores. [73–75]

Many of the performance hurdles in parallel grid generation stem from improper load balancing. To mitigate load balancing issues researchers have proposed techniques which dynamically load balance during the grid generation process. Much of the machinery required to do dynamic load balancing can be reused in fault tolerant implementations which use double in-memory checkpointing. In both dynamic load balancing and in-memory checkpointing, the implementation must be able to communicate pieces of the grid domain through the network.

CHAPTER 2

THE CURRENT AND FUTURE COST OF FAILURES

No compute machine is made of perfect hardware or software components. Imperfect systems incur additional cost for High Performance Computing (HPC) organizations in the form of wasted compute hours. Wasted compute hours waste electricity, and slow the rate of innovation by extending the cycle for discovery on new science and technology.

Imperfect compute systems also incur risk. The Return to Flight effort following the Columbia accident in 2003 tasked NASA engineers to establish a process to perform during flight assessments. Part of this process included personnel, Computational Fluid Dynamics (CFD) software analysis tools, and HPC hardware positioned and ready to perform high fidelity CFD analysis inside an 18 hour window. [76] Projects which depend on timely simulation results for critical decision making may not have the luxury of simply running the simulation again.

On today's production machines, failures occur often enough that mitigation strategies are used. Disk based checkpointing and recovery is a common mitigation strategy. Mitigating failures through disk based checkpointing incurs relatively low overhead, especially on machines with fast disk arrays. However, as machine sizes grow, so do failure rates. It has been widely projected that tomorrow's machines must consider failures as part of normal operation and not an exception. Sufficient attention must be pid to the failure mitigation strategy or it may consume compute time.

This work proposes a parallel grid generation technique which has increased resiliency from an integrated mitigation strategy. This chapter aims to answer the question of what costs can be expected by protecting from failures and how much risk can these protections mitigate. Unfortunately, to answer questions of specific job failures we need to know something about the current or recent state of the machine before launching the job. There is some work being done to predict specific hardware failures. Teams have used hardware sensors and recent failure characteristics to try to predict when specific hardware will fail (often to replace the hardware before it causes a failure in production). [8, 77, 78]. However, these techniques are still experimental and not all compute clusters have deployed the required hardware sensors and software monitoring tools.

This chapter looks to answer questions of the cost of failures from a statistical, rather than a deterministic approach. A simple model is built to emulate failures on compute clusters. The model is tuned to to reproduce Mean Time to Failure (MTTF) for known compute clusters. Once tuned, the model is used to examine the statistical likelihood of failures and the additional costs incurred on notional computational jobs on small and large clusters.

2.1 A Model of Failure

A simple model is created to explore the likelihood of failures, and the associated costs of those failures. The model consists of a collection of compute nodes. Each node has a percentage likelihood of failure per hour, λ. Initially, it is assumed that λ is invariant across time and across nodes within a system. Then the model is expanded to allow for compute nodes with different

failure rates which closer matches published data. For both assumptions the value of λ is tuned until aggregate runs of the model properly predicts the published MTTF of large HPC machines. The model is then used to simulate a run of a notional compute job at three scales, a small job, a medium job, and a large job. A series of Monte Carlo simulations are performed using the model to determine the statistical nature of the number of failures which occur during each job, and how the total run time of the job responds to these failures.

2.2 Methodologies

The model takes as an input the computational cost in hours that the job must take to complete, t^*. This is the time the job would take if there were no failures. The simulation iterates hour by hour ticking down the hours in t^*. At each iteration the simulation computes a different random percentage, r_i, for each compute node. If for any compute node i

$$r_i < \lambda_i$$

the node is considered to have failed and the job needs to be restarted from the last checkpoint. If a job must be restarted, the additional cost of a restart (C_r) is added to the run cost.

The elapsed simulation time in hours is tracked. When an iteration completes successfully the elapsed time increases by one hour. If a failure occurs then the elapsed time increases by a random fraction of an hour as if the job had progressed for some amount of time during the next hour before failing.

The cost of creating a checkpoint (C_c) must also be considered and added to the elapsed running time each time a checkpoint is created. For simplicity all the results shown below are for applications which checkpoint once an hour.

Currently in-disk checkpointing is an often used method of creating checkpoints. Recovering an application from a failure usually requires restarting the entire application and re-running the application's start up. This may include reinitializing any runtime systems used by the application, such as MPI. With many organizations, restarting a failed job also requires resubmitting a job through a queueing system. Queue lengths can vary greatly between different organizations and time spent in a job queue does not waste electricity. However, when lengthy queues are combined with frequent failures can greatly slow the cycle of discovery for computational researchers.

It has been well documented that in compute clusters that the majority of single point failures occur on a relatively small number of compute nodes. Additionally, compute clusters are more likely to suffer a failure, if the machine has suffered a failure recently. [13, 22, 23, 79, 80] This is why two sets of cases are run, one with λ assumed constant and one with λ assumed to follow a Weibull distribution curve. Weibull distributions are commonly used to describe failure rates. A Weibull curve was identified as a suitable match for hardware failure rates in the Titan supercomputer. [79]

2.3 Model tuning

For the model to be useful it must have a reasonable value for λ, the percentage likelihood of failure of a node. There are a few choices to determining failure rate. The first is we could

choose a failure rate based on the manufacture's reported MTTF for some hardware (perhaps the CPU or GPU) on a compute node. However, these failures rates are not empirically determined. Instead MTTF from a manufacturer is typical simply the inverse of the Failures In Time (FIT) for that hardware during testing. These rates typically do not match reported aggregate failure rates on compute clusters. Instead, appropriate values of λ can be determined using the simulation itself while trying to match data taken from real world compute clusters.

Oak Ridge National Laboratory has a long record of operating world class compute clusters. They have reported failure rates on some of their compute clusters. Recent published data in [79] describes failures on the Titan supercomputer caused by GPU hardware. During the time period of the study Titan operated on 18688 compute nodes. The average GPU MTTF across all of Titan was approximately 44 hours. The study also highlighted that there was temporal coupling on failures in GPU nodes. The failure rates were shown to match a Weibull distribution with a shape parameter of 0.64. A shape parameter less than one implies a high infant mortality rate and, indeed, the most common mode was under 2 hours between two consecutive failures.

Failures due to GPUs are not the only failures which can cause a node to become unresponsive. The MTTF for Titan's predecessor, Jaguar, is reported to have had a MTTF of 10.3 hours between 2009 and 2010. [19] Newer studies on hybrid CPU + GPU compute nodes have shown MTTF of as low as 4.1 hours. [81] Hardware running on the Oak Ridge supercomputers undergo stress testing to identify bad components before going into production, this may explain why Oak Ridge clusters exhibit higher MTTF than researchers have experienced on other systems.

2.3.1 Constant Likelihood of Node Failure

The model developed for this work was tuned to determine appropriate values of λ. First experiments were run with constant λ across the entire compute system of 18688 nodes. The value of λ was fixed and then the simulation was run ten thousand times while recording the time to first failure. After completing all ten thousand simulations the MTTF was computed along with a confidence interval. The confidence interval was computed by:

$$x_i = \frac{1}{n}\sum_{i}^{n} x_i$$

$$\sigma_i = \frac{1}{n}\sum_{i}^{n} (x_i - x_n)^2$$

$$x_n - \phi\frac{\sigma_n}{\sqrt{n}} < x < x_n + \phi\frac{\sigma_n}{\sqrt{n}}$$

A value of $\phi = 1.96$ corresponds to a confidence of 95%. A few iterations of a binary search was used, and a final value of $\lambda = 1.190e-4$ was found to reproduce a MTTF of 44 hours with a 95% confidence interval of (44.1, 44.9). Similar experiments were run and $\lambda = 4.94e-4$ produced MTTF of 10.3 hours with a 95% confidence interval of (10.1, 10.5). Figures 2.1 and 2.2 show the running average time to failure during the ten thousand runs. In both figures, the black line shows the current MTTF and the blue shaded region covers the running 95% confidence interval.

Figure 2.3 shows the relationship between λ and MTTF for two systems, one with 18688 nodes and one with 9344 nodes. This experiment was run to help build an understanding of the response of MTTF as the reliability of each hardware node changes. It is clear that MTTF grows

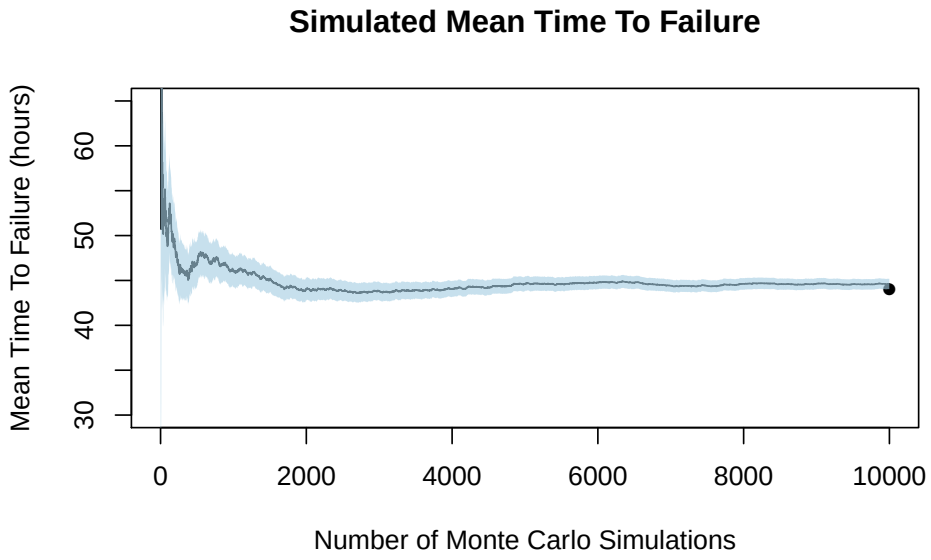


Figure 2.1 Mean time to failure convergence history

non-linearly as the reliability of each node decreases. Furthermore, the size of the machine also greatly impacts the MTTF.

The tuned values of λ can reproduce system MTTF. However, assuming constant λ does not reproduce the distribution of failures reported in the literature. Figure 2.4 shows a Quantile-Qualtile (QQ) plot of the TTF distribution compared to a Weibull curve with shape parameter 0.64. This is the distribution extracted from failure logs on the Titan supercomputer caused by GPU errors.

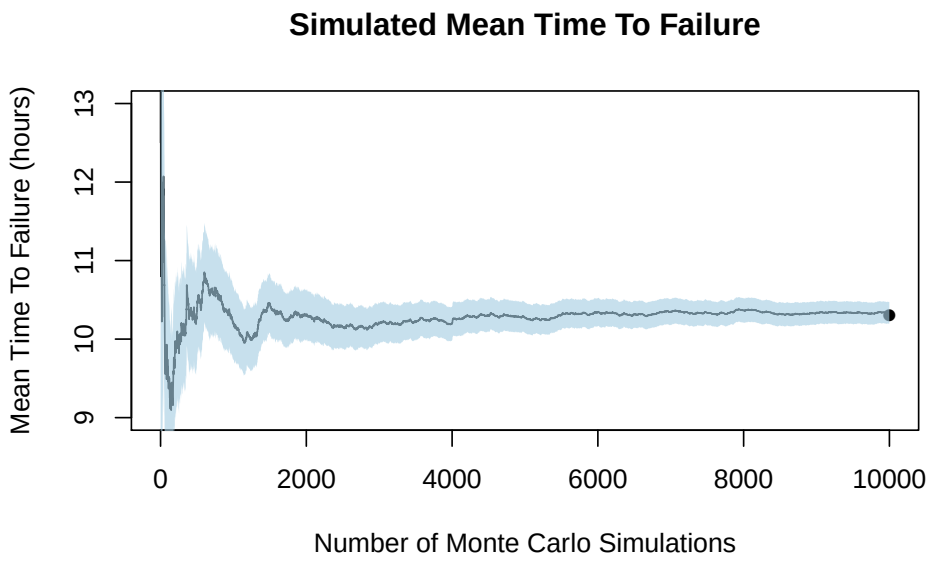


Figure 2.2 Mean time to failure convergence history

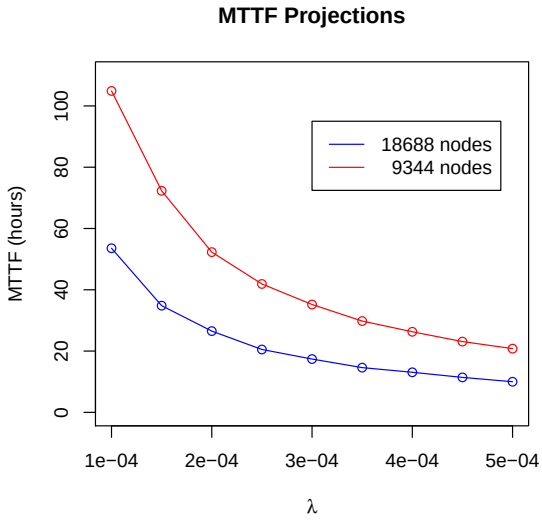


Figure 2.3 Mean time to failure to likelihood of failure

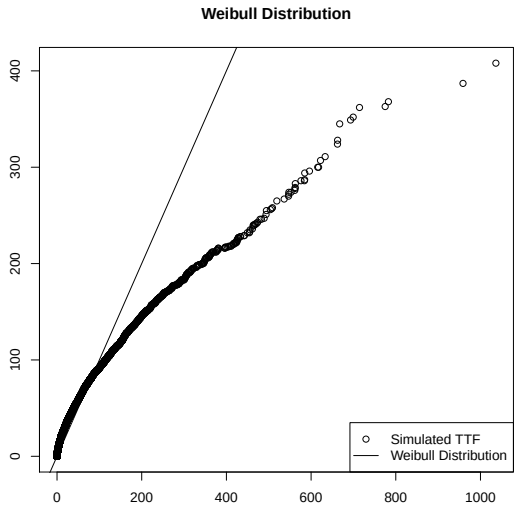


Figure 2.4 Temporal distribution of failures assuming constant likelihood of failure

2.3.2 Variable Likelihood of Failure

Not all compute nodes exhibit the same rates of failures in large scale compute systems. In most systems the majority of failures occur on a small number of nodes. Therefore, assuming that λ is constant across all compute nodes may be a poor assumption. Tiwari et al. reported that the distribution of times between failures matches well with a Weibull distribution. [79] It was then assumed that the failure rates (λ) for nodes may also match a Weibull distribution.

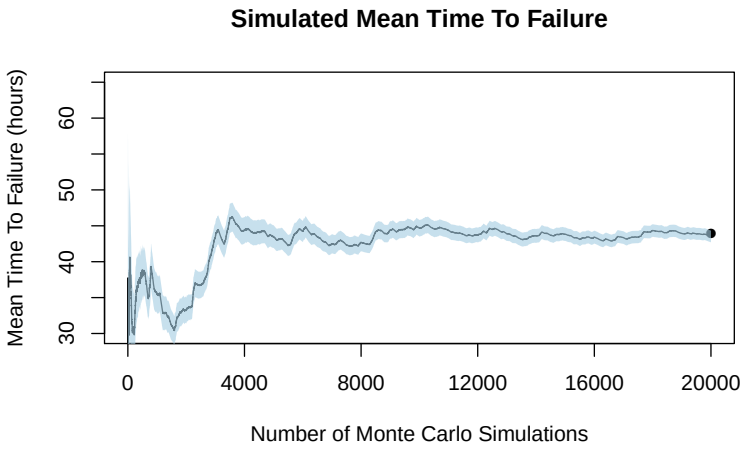


Figure 2.5 Mean time to failure convergence history

Tuning experiments were run to determine appropriate scale and shape parameters. It was found that the parameters shape = 0.0765 and scale 8.5e-14 reproduced a MTTF of 44 hours when run on 18688 nodes. A 95% confidence interval for the MTTF was (43.7, 45.8) after 20,000 runs.

Figure 2.5 shows the running TTF during the 20,000 iterations. A QQ plot is shown in Figure 2.6 comparing the simulated distribution of times to failure with the reported distribution of times to failure extracted from failure logs on the Titan supercomputer. As the plot shows, expanding the model to allow for non-constant failure rates can predict the system mean time to failure and better match the distribution of times to failure.

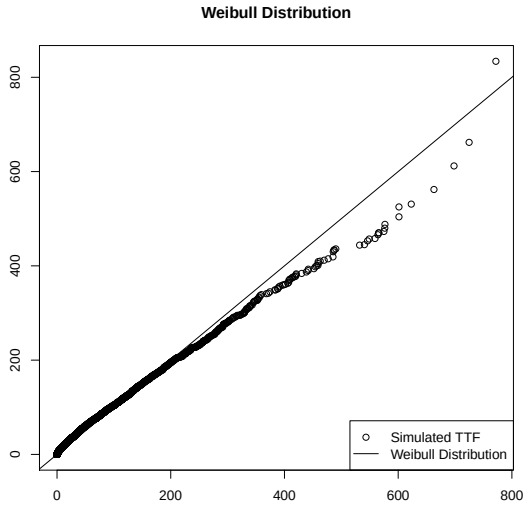


Figure 2.6 Temporal distribution of failures with varying likelihood of failure

Tuning experiments were repeated to determine appropriate scale and shape parameters for λ distribution which reproduced a MTTF of 10.3 hours. It was found that shape = 0.09600 and scale = 1.0e-10 was sufficient to emit a MTTF of 10.3 hours on a 95% confidence interval of (10.1,

10.5), running averages during the simulation are shown in Figure 2.7. Figure 2.8 shows a QQ plot of the total system failure rates from the simulation compared to the failure distribution for Titan as reported. The 10.3 hour MTTF measurement is the reported system time of the supercomputer Jaguar during 2009, distribution data is not available for comparison of that time frame. In 2012 the Jaguar supercomputer received upgrades, including interconnect and GPU hardware, after the upgrade the supercomputer was renamed Titan. Due to the similarities of the system (similar size, operated using similar policies) it is assumed that the distributions are also similar.

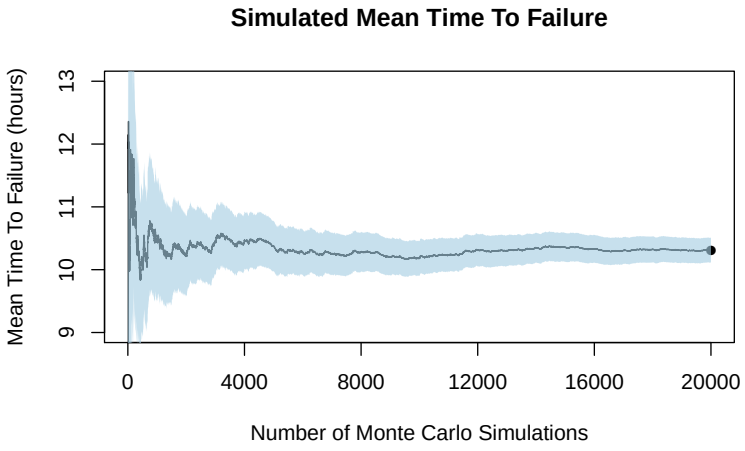


Figure 2.7 Mean time to failure convergence history

Suitable values for likelihood of failure per hour of a node has been determined for both a constant and variable likelihood of failure. Preparing for failures by creating checkpoints, and recovering from failures both cost time. The model used in this section to match MTTF rates for

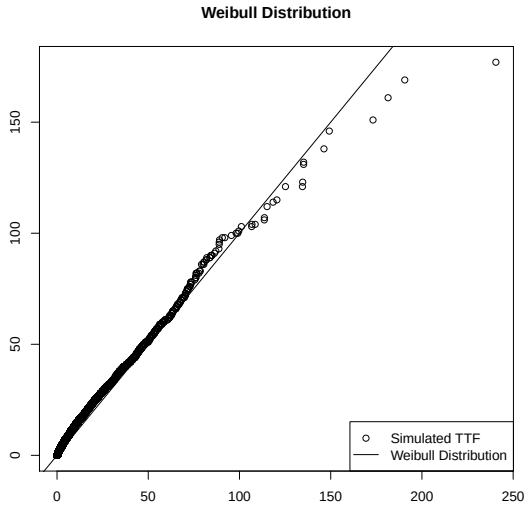


Figure 2.8 Temporal failure distribution with varying likelihood of failure

large systems can be modified to estimate the aggregate increased cost. The next section explores the impact of failures on notional CFD computations.

2.4 The Cost of Failure

Computational simulations can vary greatly in computational cost scaling from a few seconds on a laptop to many weeks or months on a large supercomputer. Aeroacoustic simulations are among the most expensive simulations run with CFD. These simulations must be run time dependent with high resolution in both the spatial and temporal domains. One such example comes from 2012 where FUN3D was used to compute noise caused by nose landing gear. [82] The grid used contained 145 million control volumes and took approximately 1280 compute hours using

180 dual socket compute nodes. It is estimated that accurate full spectrum noise prediction for a full aircraft may require at least 1-2 billion control volumes with some researchers anticipating mesh sizes near 10 billion. [83] Luckily, high performing simulation codes such as FUN3D have shown good weak scaling performance. If weak scaling holds, it can be assumed that a simulation of full aircraft noise prediction using 1.4 billion control volumes (roughly 10 times larger than nose landing gear case) will take 1280 compute hours on 1800 compute nodes. Furthermore, a complex configuration, may contain deployed slats, flaps, landing gear, and possible flow control. This fully complex configuration could take as much as 14 billion control volumes. Which with perfect scaling take approximately 1280 compute hours on a compute cluster of 18,000 compute nodes. Notional values of compute length and machine size can be used to explore failure characteristics and the total computational cost of computing early 21^{st} century CFD simulations.

Each notional case was compute twice, once by assuming that each node has the same likelihood of failure, and once by assuming that the likelihood of failure obeyed a Weibull distribution across the system. Appropriate values for the probability of failure were determined in the previous section using a similar Monte Carlo based model.

2.4.0.1 Cost Evaluation for Constant λ

Figure 2.9 shows the estimated total run time for a job with an expected computational time of 1280 hours running on 180 nodes. Four plots are shown varying both C_c and C_r. A checkpoint was created once an hour. Figure 2.10 shows the corresponding failure distribution for these four cases. The likelihood of failure was set to $\lambda = 5.0e-4$ which corresponds to a MTTF of approximately 10 hours, as shown in Figure 2.3. Even for this relatively small case the most

likely outcome is one failure occurrence during the job's execution. And the likelihood of a job encountering more than 6 failures was very unlikely. With this small number of failures the run time is dominated by the computational cost (1280 hours) when the checkpoint cost is low.

If no checkpointing had been enabled, there was only approximately a 30% chance that the job would have successfully completed. Even if checkpointing and recovery were set to a very expensive 60% of the running time every job would have completed within 2560 hours (2× of the original simulation length). Without fault recovery, there is only approximately a 70% chance that the job would finish during the same window of 2560 hours.

Next the job was increased to include 18000 nodes. The CFD Vision 2030 report identifies mesh sizes of 10-100 billion control volumes for aircraft in maneuvering flight. A machine size of 18000 nodes could perform calculations on a 15 billion control volume mesh with approximately the same number of control volumes per core as the 180 node case. The wall time was held constant at 1280 hours. Again λ was assumed constant at 5.0e-5. The checkpoint and restart costs were varied to simulate both inexpensive and expensive checkpoint and recovery costs.

The larger system of 18,000 nodes the probability distribution of failures changes. Figures 2.11 and 2.12 show the projected cost and associated failure distribution for a case running on 18,000 nodes. The same value of $\lambda = 5.0e-4$ was used for these runs. The most likely number of failures has shifted to between 120 and 200 failures during the run. The total clock time of the run increased 6% over the base computational cost, even when both creating checkpoints and recovering from checkpoints takes only seconds per hour. The recovery time C_r must also be considered since the number of failures has grown.

Since the computer used was so large, it is nearly impossible that the job with no fault tolerance considerations would have completed. The number of failures has shifted from a skewed distribution near 1 on 180 nodes, to a normal distribution with well over 100 failures occurring during the simulation.

Checkpoint cost dominates the overhead cost of fault tolerance. Creating quick checkpoints will have the biggest impact on minimizing the increase in run length caused by failures. The checkpoint and recovery time can be brought down to under a minute then expected time to complete a simulation would only be approximately 6% more than the base run length. Ignoring failures on the 180 node case increases the total run time by about 60%. It is statistically extremely unlikely that the notional 15 billion control volume simulation would complete without some form of fault tolerance.

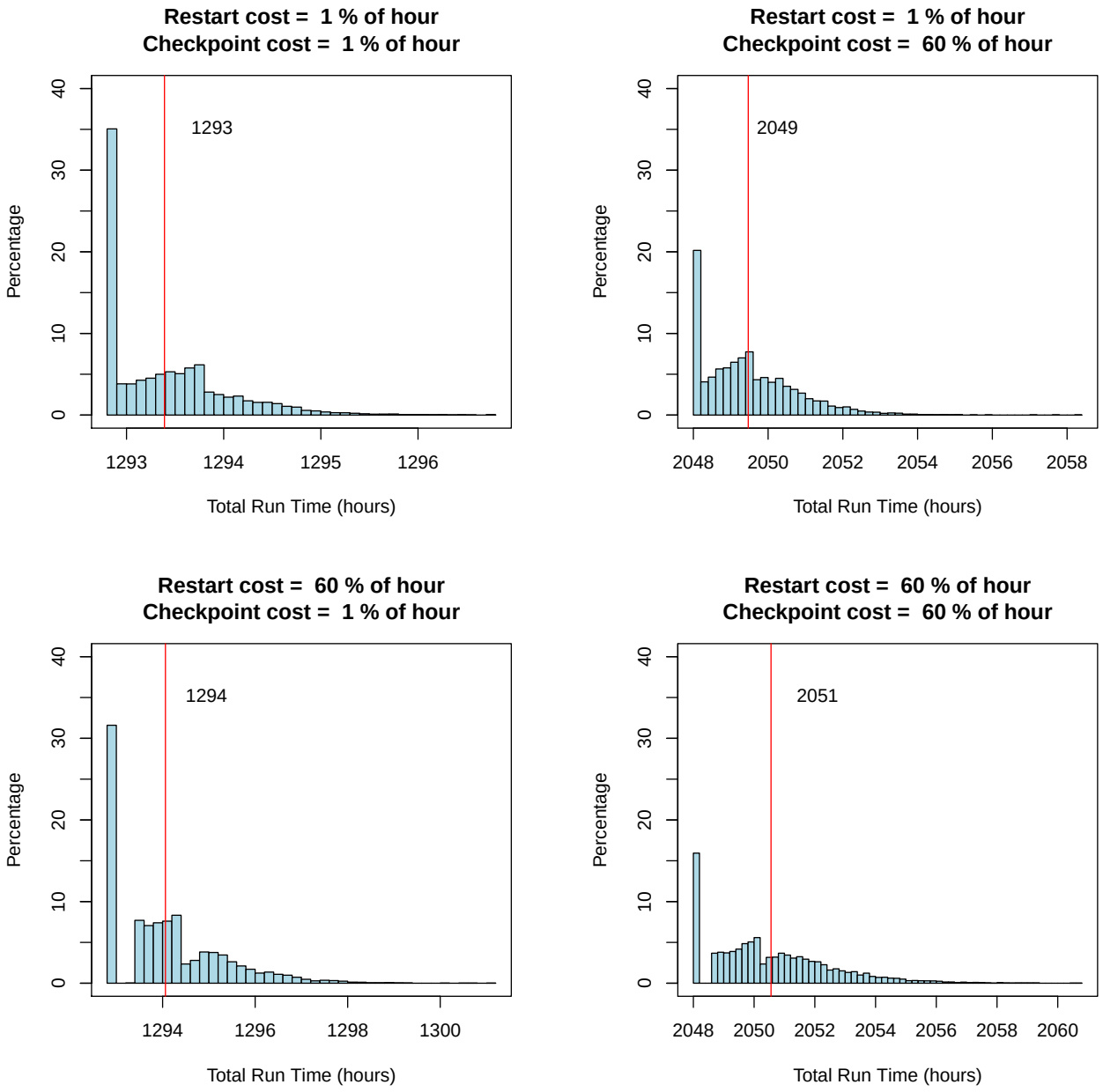


Figure 2.9 Simulated run length for 180 nodes

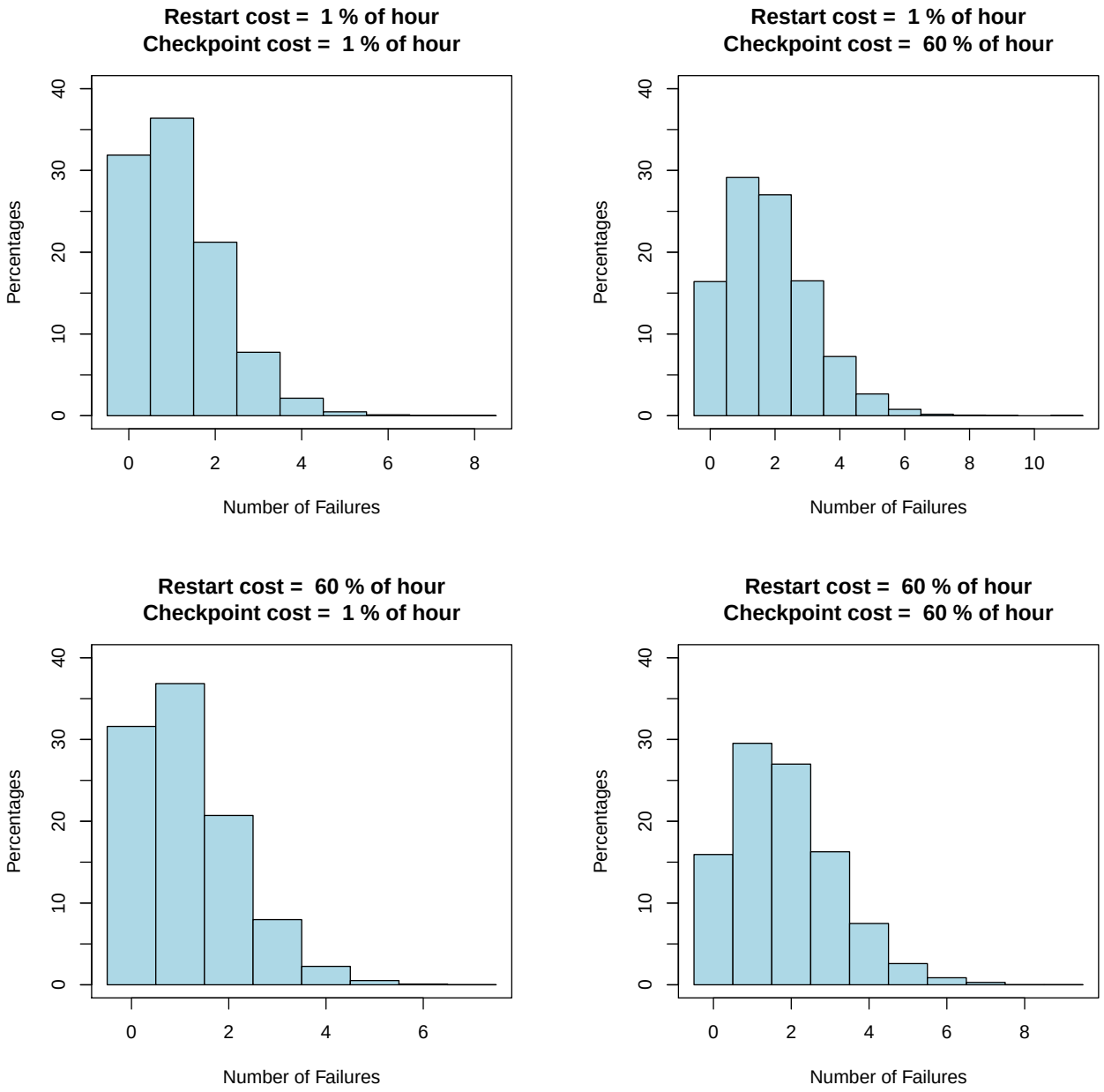


Figure 2.10 Number of failures for 180 nodes

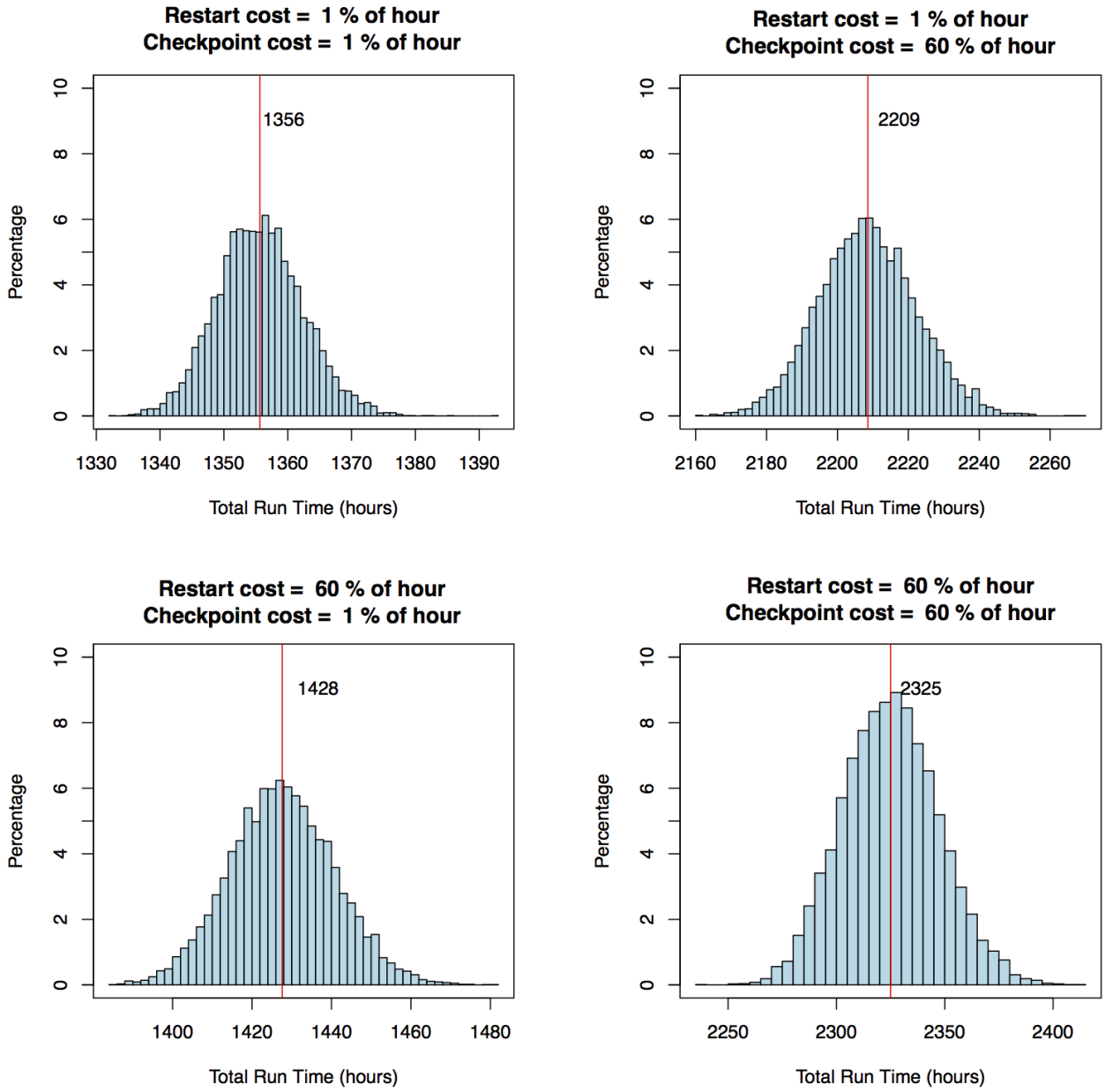


Figure 2.11 Simulated cost for 18,000 nodes

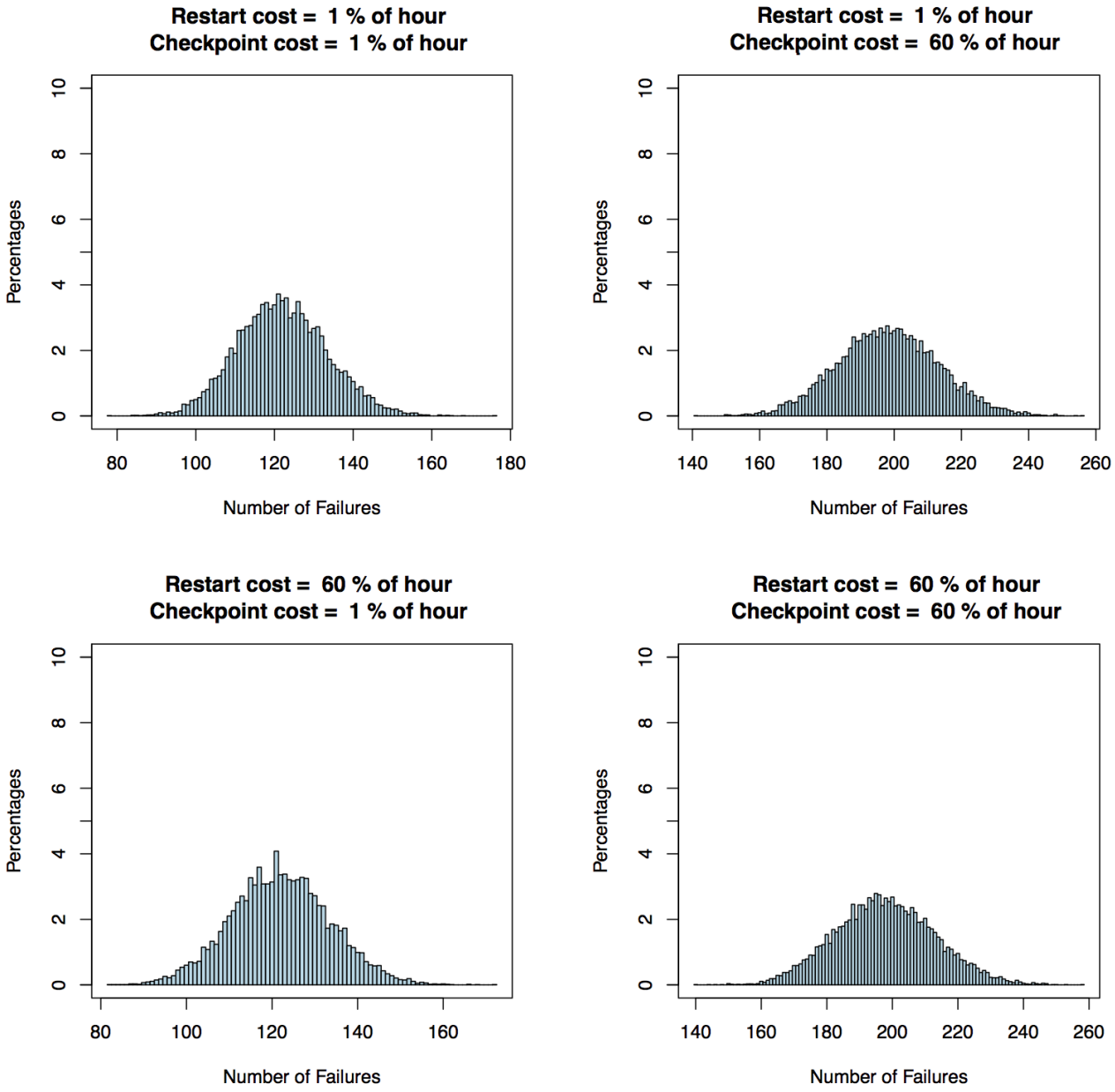


Figure 2.12 Simulated failures for 18,000 nodes

2.4.0.2 *Cost Evaluation Variable Likelihood of Failure*

Section 2.3 demonstrated that using constant values for the likelihood of failure (λ) did not reproduce the distribution for Time To Failure (TTF) reported from the literature. The cost experiments were conducted again. This time each compute node was assigned a unique value of λ. The values of λ were taken from Weibull distributions which were tuned in the previous section to reproduce a total MTTF of 10.3 hours.

Figure 2.13 shows the distribution of time to solution for a simulation which is takes 1280 hours to compute on 180 nodes. Again 4 cases were run with varying checkpoint and recovery costs. The average time to completion with these cases is nearly the same as the cases run with a constant likelihood of failure. However the distributions have changed. The most striking change is shown in Figure 2.14. In all cases the likelihood of failures decreased. Each case shows approximately a 70% chance of encountering zero failures during the run.

Figure 2.15 shows the time to solution distribution. Figure 2.16 shows the failure distributions. Again both the time to solution and the failure distributions appear to follow normal distributions. And like the 18000 node case with constant λ, the number of failures ranges from approximately 100-250 depending on the cost of creating a checkpoint.

While the failure occurrences were very different for the 180 node case, they are almost unchanged for the 18000 node case. This indicates that the constant λ assumption may be suitable for large CFD simulations involving billions of control volumes. The constant λ assumption over predicts failures for smaller simulations, those with under one billion control volumes when compared to Weibull distributions for λ.

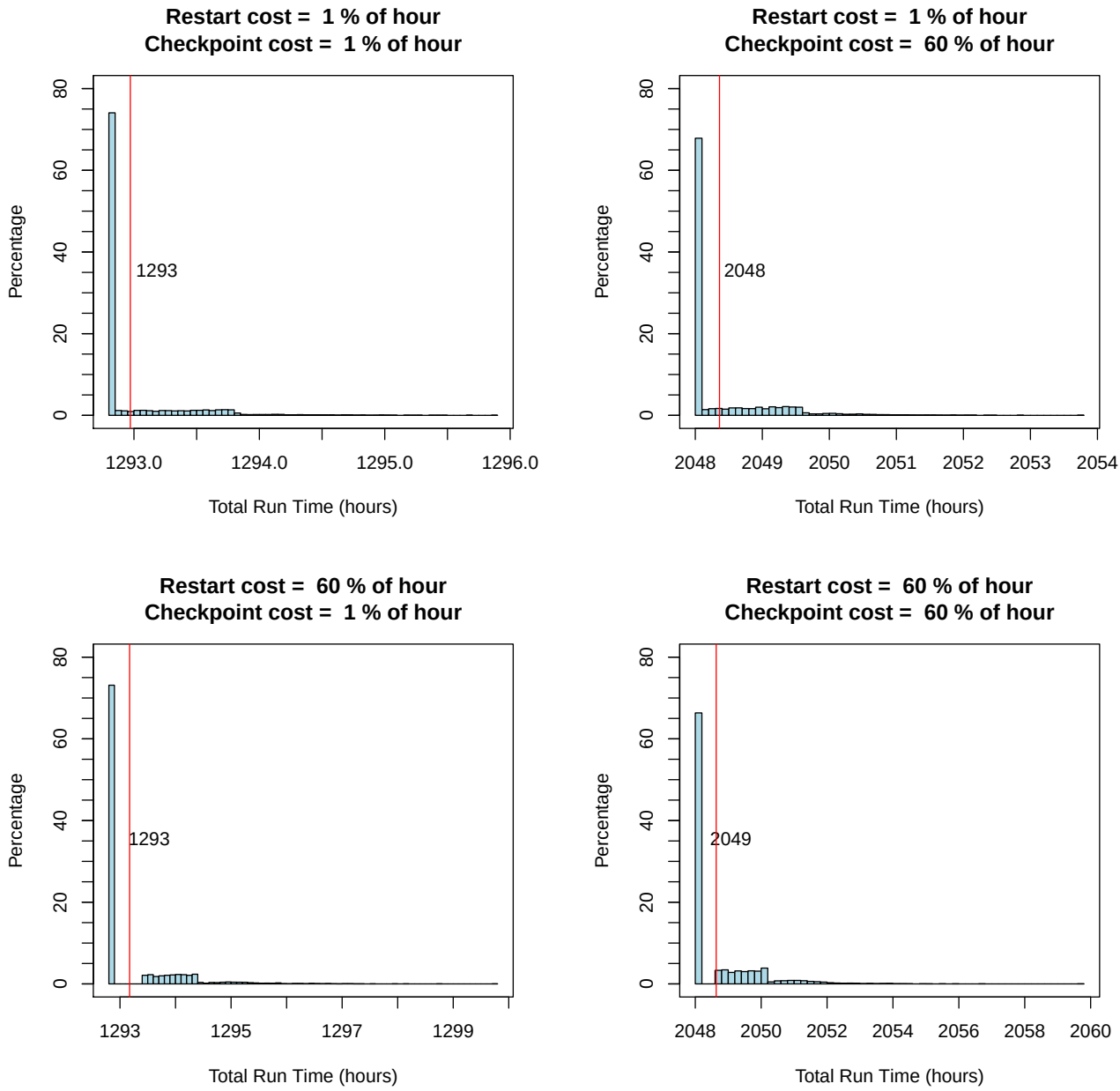


Figure 2.13 Simulated run length for 180 nodes

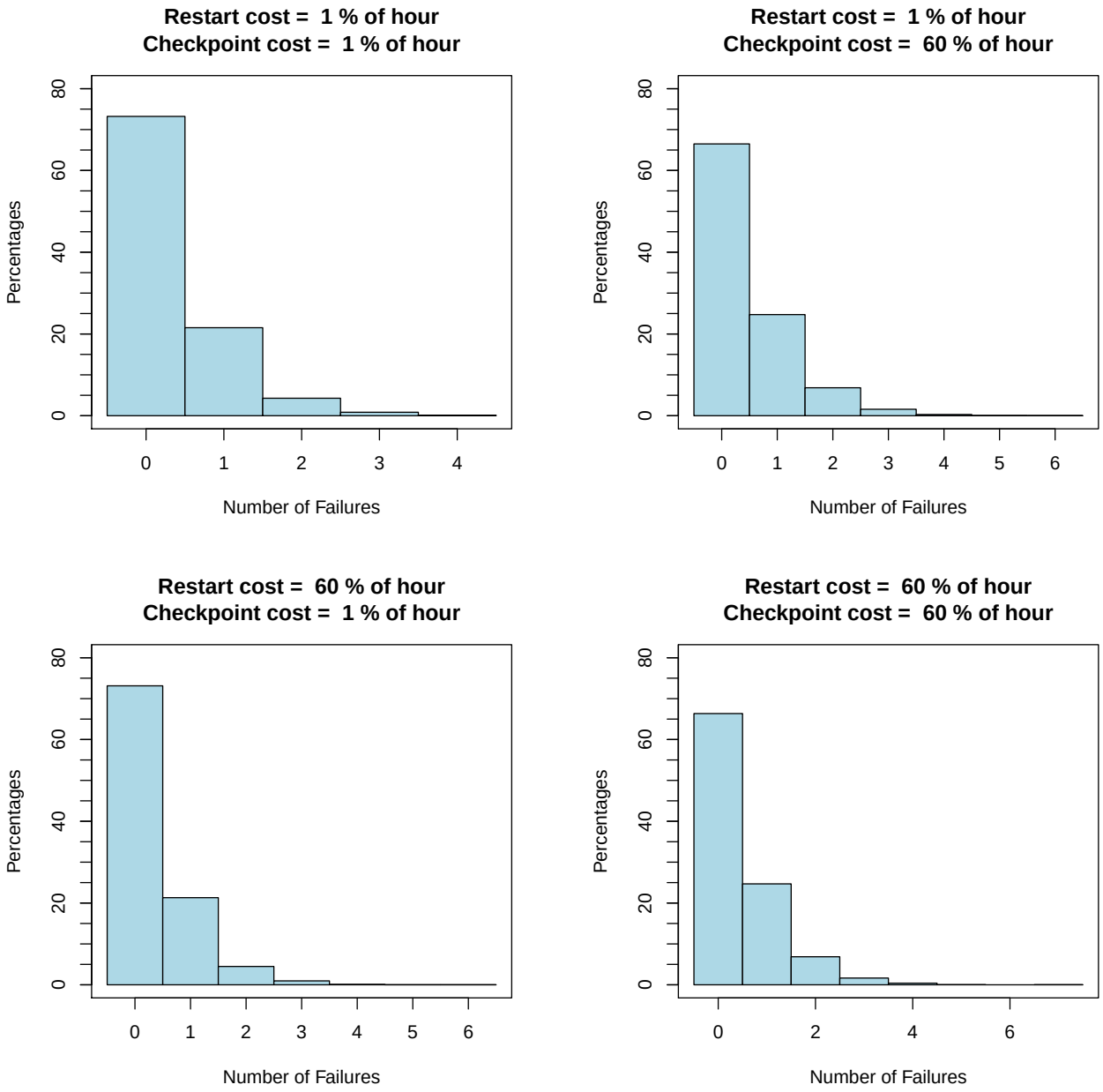


Figure 2.14 Number of failures for 180 nodes

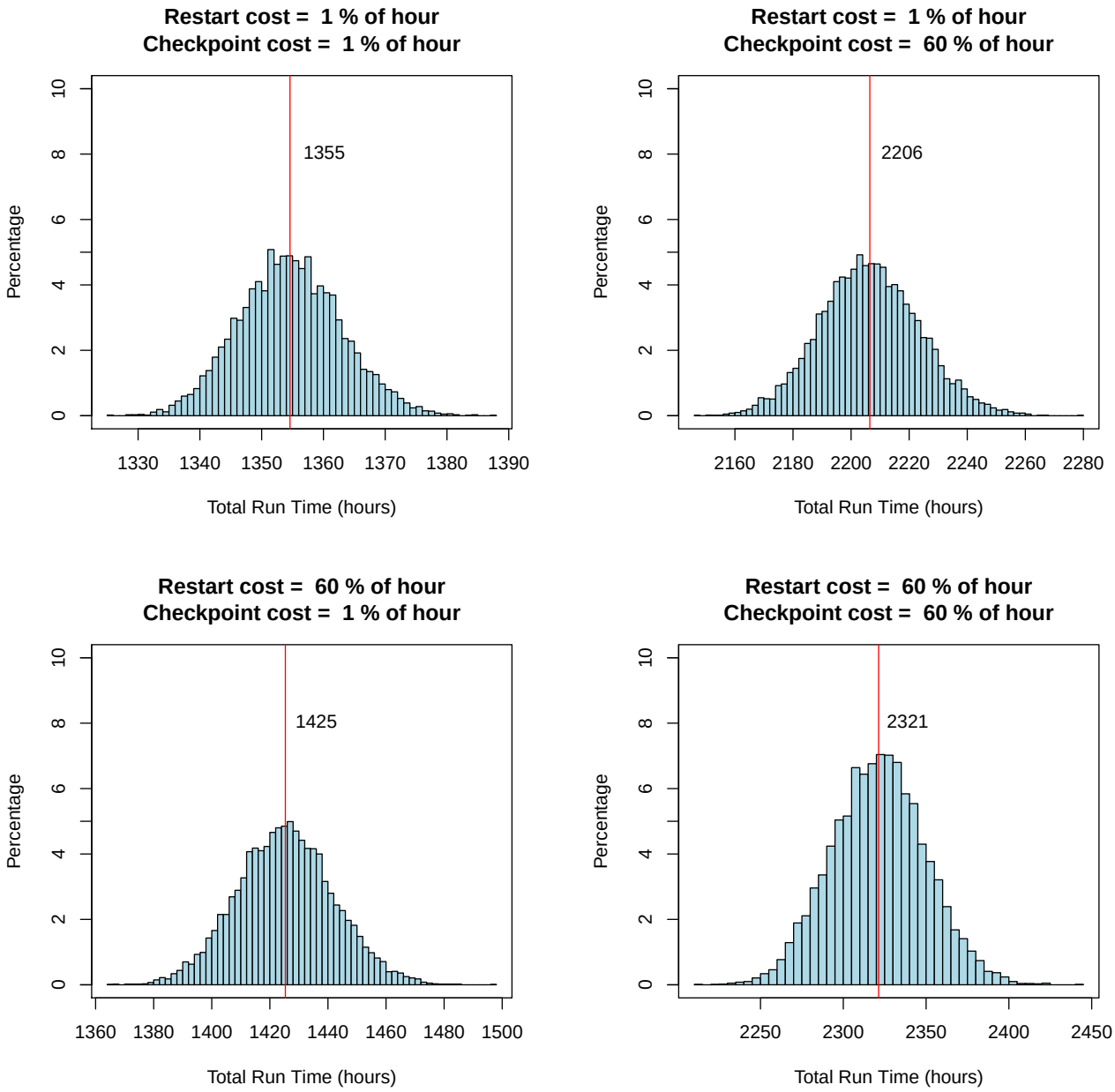


Figure 2.15 Simulated run length for 18,000 nodes

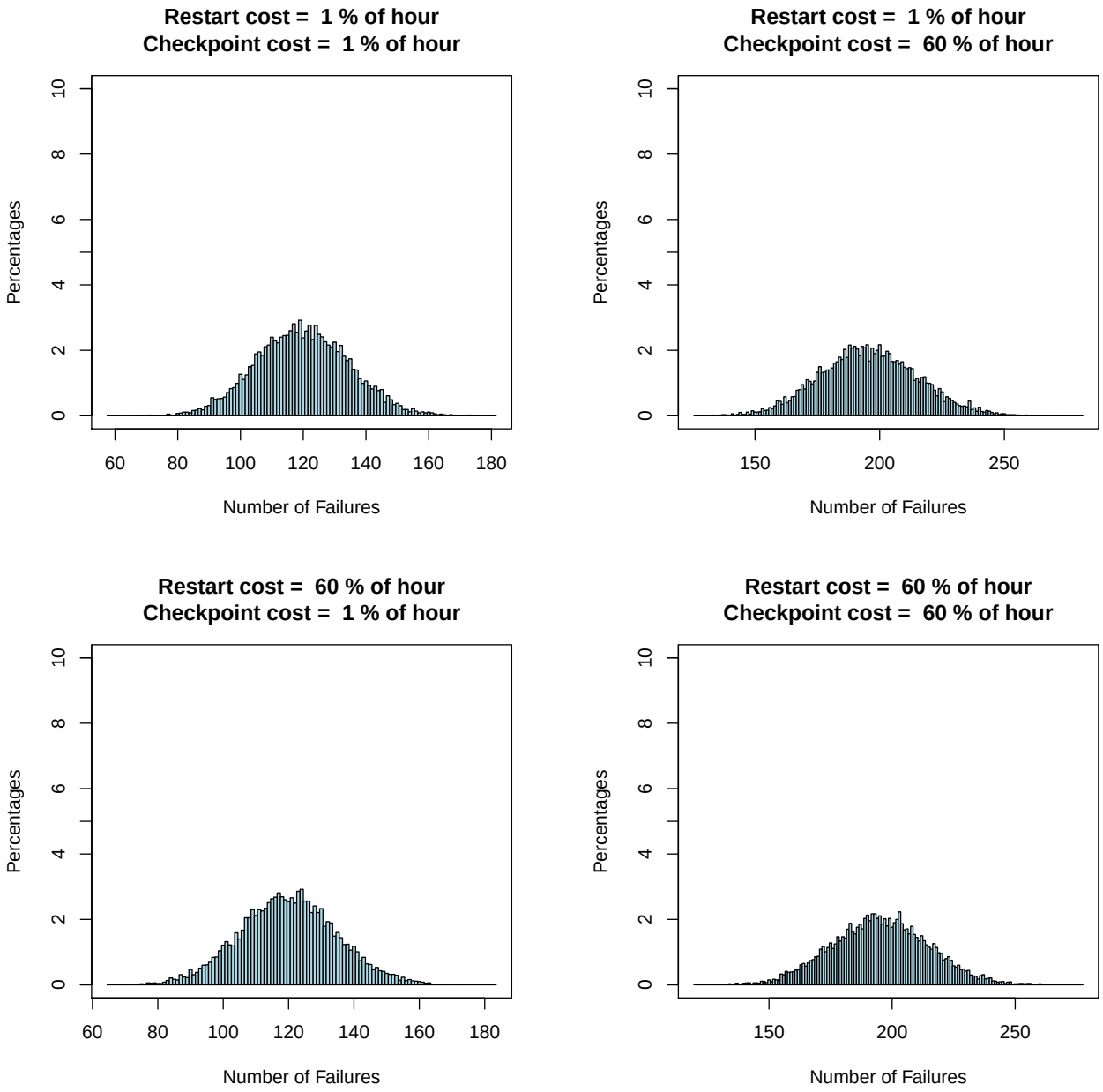


Figure 2.16 Simulated failures for 18,000 nodes

2.5 Summary

Large computational jobs consume many node hours of computational resources. The likelihood of a failure on the underlying compute system grow when a job requires more node hours of compute time. Checkpoint and restart based resiliency techniques allow a job to make progress while operating on imperfect compute systems. These techniques incur additional cost when checkpoints are created, and incur a different cost when a failure must be recovered from. Ideal checkpoint and restart implementations would take only the minimum amount of time to allow the maximum amount of the compute resources to be applied to running the simulation.

Average total time to complete was computed using a simple model for failures on a compute cluster. This model demonstrated that even small overhead costs for checkpointing and restarting can lead to significant overhead costs. For the notional problem of full aircraft in maneuvering flight, the overhead cost of protecting against failures could be as small as 6%. Alternatively, it is statistically extremely unlikely that the simulation would complete at all without some form of fault tolerance.

Cases were conducted by first assuming that each node in the compute system had the same likelihood of failure, and then by allowing some nodes to fail more often than other nodes. The distribution of the failures was tuned to match overall system TTF distributions taken from the Titan supercomputer. The effects of a variable likelihood of failure had significant impact on parallel jobs which were running on 180 compute nodes. This suggests that models which assume each compute node is equally likely to fail may over predict the number of failures which occur on a system. Assuming each compute node has an equal likelihood of failure was shown to over predict the number of failures on 180 node clusters during a simulations when compared to a variable

distribution for the likelihood of failure. The differences between equal likelihood of failure and variable did not greatly impact simulations running on 18000 node clusters.

At today's production scales of a few hundred computational nodes failures are still relatively rare occurrences. Computational jobs on 180 nodes for approximately a month have only a 30% chance of being affected by a failure during the run. However, when solving larger problems using larger computers the likelihood of suffering a failure grows until failures cannot be treated as one-off or rare events. Future simulations, such as those discussed in the CFD 2030 vision, will require between 10 and 100 billion control volumes. These simulations will require tens of millions of node hours and the likelihood of suffering at least one failure is well over 99%. At these larger scales failures occur so regularly that a high performance simulation code must treat the occurrence of a failure as expected. Simulation codes running at this scale must carefully consider the fault tolerant strategy used. Codes which use common checkpoint and restart to mitigate the cost of failures will need to work to keep the cost of creating checkpoints low. Furthermore, simulation codes must also keep the cost of recovery low.

Typical serial mesh generation techniques will have a difficulty generating meshes large enough to support simulations with tens or hundreds of billions of control volumes. Instead, parallel mesh generation techniques, such as the one described in the next chapter, will be a key component of the CFD workflow.

CHAPTER 3

PARALLEL GRID GENERATION USING SPATIAL TREES

Grid generation has been identified as a key bottleneck for Computational Fluid Dynamics (CFD) at large scales. [1] The common CFD workflow requires a human in the loop to discretize the fluid domain around a geometry. Furthermore, grid generation is widely still a task performed on a single workstation. Grids are often made by hand, on a workstation. The grids are then often used in combination with a CFD analysis code that is running in parallel on a compute cluster. A more desirable workflow would be to generate the grids automatically and in in a distributed memory parallel environment. This chapter describes a technique which does just that.

The grid generation technique described in this chapter utilizes a hybrid approach to automated and parallel grid generation. The grids are generated using Cartesian elements away from the geometry surface. Grid elements near the geometry surface are not Cartesian aligned and are instead smoothed and projected to create a body conforming nearbody region. The entire process operates in a distributed memory parallel environment which allows for very large grids to be generated without the need specialized hardware with above average memory.

[1] see Section 1.1 for a full discussion.

3.1 Off Body Generation

The off body grid generation process consists of basically three steps: creating the initial octree for each partition, subdividing the octree where voxels cross the geometry by progressing the tree from the top down, and filling out the breadth of the octree to ensure reasonable cell size gradation by progressing from the bottom of the tree to the top.

3.1.1 Initial Partition Creation

Parallel grid generation begins by creating a large number of work partitions. Each rank process will be responsible for managing some number of work partitions. Each rank operates at a minimum one partition; however, it is more typical for each rank to operate 10 or 20 partitions. Over-decomposing the total work into a large number of partitions has many benefits including hiding the latency of data movement either from cache or from the network. However, the most critical benefit is that over-decomposing lowers the load imbalance that could be caused from a node failure. In the event of a failure, the work assigned to one node can be simply distributed to a larger number of surviving nodes.

Each partition contains an octree. The root voxel in every partition's tree has the same extent box. The extent box is large enough to encompass the entire domain which is to be meshed. Each partition begins subdividing voxels which cross the geometry surface level by level down to some initial starting depth, typically 7 or 8. This work is done redundantly on each partition. Each partition is identical once the initial refinement completes. The partitions then perform some collective communication sharing what number of leaf voxels each partition has. The partitions

then take ownership of only a small number of voxels by deleting any voxel to which they are not assigned. At the end of this ownership phase each leaf voxel in the mesh is uniquely owned by only a single partition. This is the starting place for the parallel offbody generation. Ownership of a leaf voxel is determined by its location along a space filling curve.

3.1.2 Space Filling Curves

Space filling curves are widely used with cartesian aligned mesh generation. In particular, Z-order curves, or Morton curves, have been shown to have nice compact properties for octree data structures [25, 26, 31, 84]. Figure 3.1 shows two example Morton curves. Notice that the curve exhibits a Z pattern. Morton curves are sometimes called Z curves because of this pattern. A Morton curve is defined hierarchically and never self intersects.

Morton curves are created by assuming an implicitly defined cartesian background grid which spans the domain of the mesh. Each location on the background grid has a unique coordinate (x, y, z). The x, y, and z components of the coordinate can be encoded to determine that coordinate's location along a curve which passes through each point on the grid. The encoding of coordinate data to a Morton curve is a Morton Id.

Morton Ids for this work were stored in 64 bit integers. The x, y, and z coordinates on the background grid were allowed 21 bits each. The x, y, z coordinates on the background grid are implicit, and not stored. Instead the coordinates are interleaved, and the resulting Morton Id fits into a 64 bit integer (with 1 bit to spare). Therefore, the background grid is always a discretized cube where each side is 2^{63} long. The root voxel in the octree has a spacing that is always 2097152

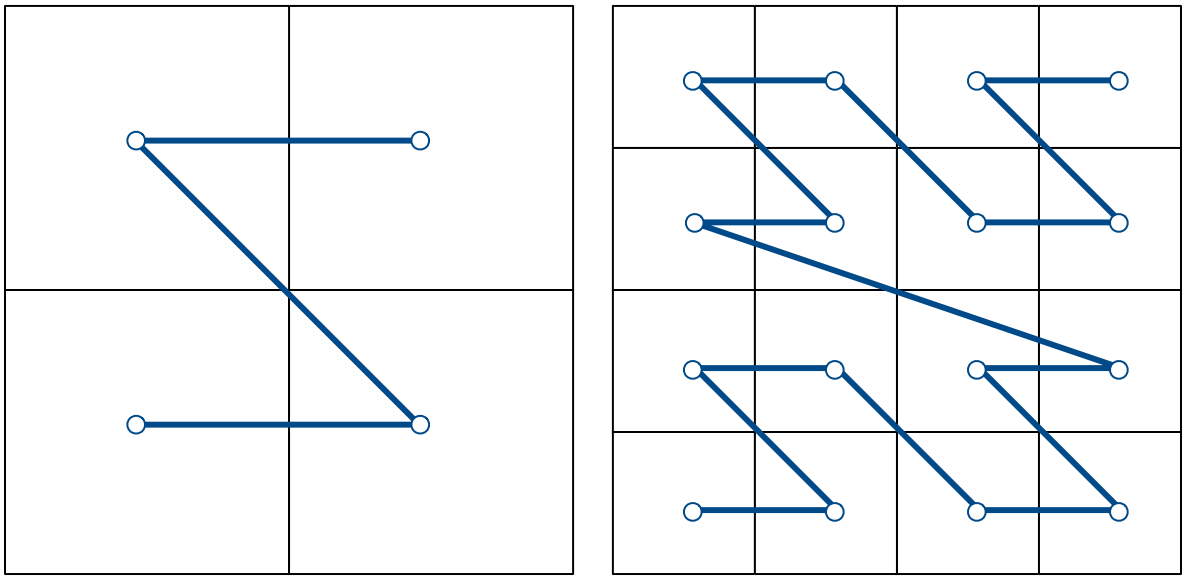


Figure 3.1 Two dimensional Morton curve

units long on the implied background grid. The physical distance of the root voxel changes to accommodate the physical domain which needs to be meshed. The spacing width is cut in half as the depth increases. At a depth of 21, the width of a voxel is 1 unit long in the background grid. This puts a maximum depth limit of 21 in the octree. This limit could be raised in future work by increasing Morton Ids to 128 bit integers. The maximum tree depth would then be 42.

The compact representation Morton Ids means that arrays of Morton Ids consume very little memory. Compact Ids can be communicated, saving bandwidth when voxels need to be shared across the network or if voxels need to be written in checkpoints. A voxel can be reconstructed using the root domain's axis aligned bounding box, the Morton Id of the voxel, and the depth in the tree of the voxel. This means that the entire mesh, or part of a mesh, can be stored very

compactly. During the mesh generation process each voxel's axis aligned bounding box is also stored using double precision float point numbers. This bounding box stores the voxel's position in physical space, not the integer based background grid. The bounding box in physical space is used to compare the voxel to the geometry. However, double precision bounding boxes consume 48 bytes of data each, while the voxel can be stored using Morton Id and depth using only 12 bytes.

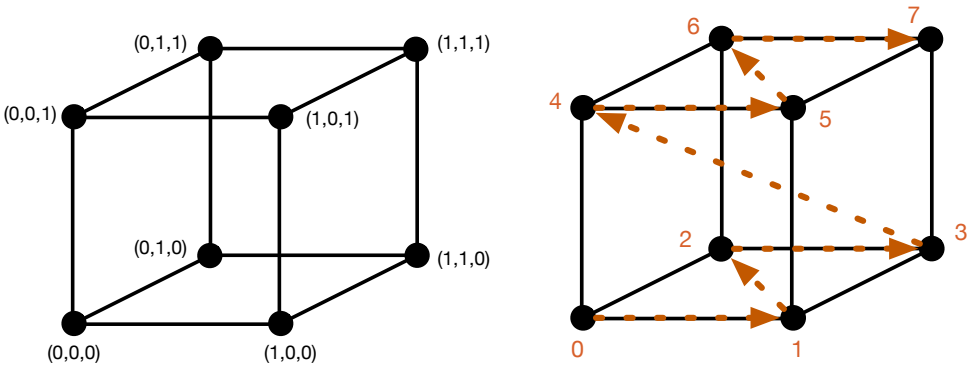


Figure 3.2 Morton curve through three dimensional voxel

Figures 3.2 shows a simple cube which is one unit wide. The x, y, z coordinates are marked for each node on the cube. Figure 3.3 then shows how the bits of each coordinate are interleaved to create the encoded Morton Id. Notice that each coordinate can be described using only a single bit (since the cube's domain is only $[0, 1]^3$). The resulting Morton Id is 3 bits long. The Morton Id has encoded the entire coordinate into a single number.

Physical Coordinates	Morton Bits	Morton Id
(0,0,0)	000	0
(1,0,0)	001	1
(0,1,0)	010	2
(1,1,0)	011	3
(0,0,1)	100	4
(1,0,1)	101	5
(0,1,1)	110	6
(1,1,1)	111	7

Figure 3.3 Conversion of physical coordinates to Morton Ids

Each voxel stores the Morton Id of the voxel's lower leftmost corner. The voxel also contains the depth of the voxel in the tree. The depth can be used to determine the width of the voxel in the background cartesian grid. Width (w) of a voxel is given by:

$$w = 2^{21-d}$$

The width can be used to determine the voxel's range along the Morton curve. The Morton Id of the lower left corner of the voxel can be decoded to determine the x, y, z coordinate of that corner. Then the coordinates can be incremented by w and re-encoded to determine the Morton Id of the upper right corner of the voxel. Any point inside the voxel will have a Morton Id that is between the Morton Id of the voxel's lower left and upper right corners. This means that all children voxels will have Morton ranges that are within their parent's range.

The Morton curve is also used to partition the offbody mesh following the work of Tu et al. [25] Each partition is responsible for a range along the Morton curve. Leaf voxels are assigned to partitions based on the Morton curve which passes through each leaf voxel's lower left corner. The curve is evenly divided into a series of segments, one segment for each partition.

A compressed map of the Morton Id ranges for each partition can be created and stored on each rank. The map has a length of number of partitions + 1 and each entry consumes only 8 bytes. Even if the mesh was divided across a million partitions, each rank would only allocate 8 megabytes to storing the entire map.

The range that each partition is responsible for can be rebalanced during generation. A rebalance is done by counting up the number of leaf voxels in the entire mesh and assigning an equal number of voxels to each partition. A new compressed map is generated showing the new Morton Id range of each partition. Rebalancing voxels across partitions then becomes simple. Partitions simply use an RPC to send voxels which are no longer inside the partition's assigned range to the new owner. Partitions also receive any new voxels that exist in the partition's Morton range. Figure 3.4 shows an outline of this process.

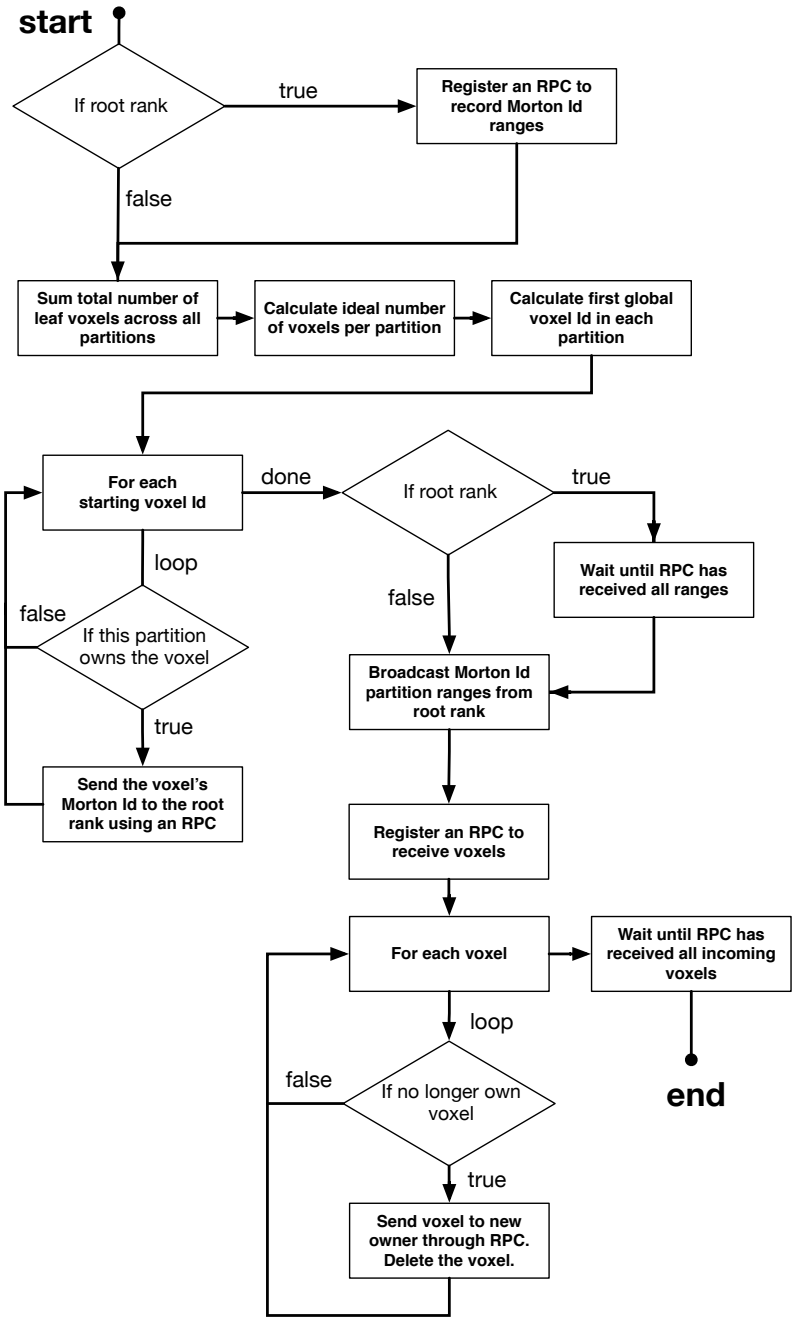


Figure 3.4 Rebalancing algorithm

3.1.3 Top Down Subdivision

The Octree begins to refine once all the partitions have been instantiated. The tree is refined level by level with all partitions completing refinement to a depth before the next depth adaptation begins. Only voxels which cross the geometry are refined during the top down subdivision step. The mesh generator simply loops over the leaf voxels at the depth which is to be refined and asks the geometry object if there is geometry within a voxel's extent box. The only way for a voxel to be at the current depth of interest is if its parent voxel was crossing the geometry.

Refining level by level allows rebalancing to occur between depths. Rebalancing between depths is not to load balance for computational time. The entire top down process consumes only a fraction of the total running time. Instead load balancing is done in order to to load balance the memory consumed and lower the peak memory used.

The initial partitioning is created to equally distribute leaf voxels for a very coarse mesh. As the mesh is refined level by level only voxels near the geometry will be refined. The partitions quickly becomes imbalanced as some partitions will almost certainly have been assigned a region of space which has more geometry than other partitions. If the process does not rebalance the imbalance will become worse and worse as the grid refines until a small number of partitions are responsible for the vast majority of the grid. In many cases this can cause a partition to run out of memory while other partitions store only a fraction of the mesh.

A load balancing ratio is monitored during top down generation. This ratio is computed as

$$r = \frac{\texttt{average number of leaf voxels per partition}}{\texttt{number of leaf voxels in largest partition}}$$

If this load balancing ratio falls below a threshold, generally between 0.8 and 0.9, then the partitions are rebalanced using the method described earlier.

Figures 3.5 and 3.6 demonstrate a fictional array of Morton Ids which has been partitioned into 3 sections. The three arrows in the figure denote what Ids each partition is responsible for. For example, partition 0, shown by the orange arrow, is responsible for all voxels in the Morton range [0,40). Initially the load balancing ratio was $r = .92$. If the mesh is refined in the region between Morton Ids 30 and 40 all, of the new cells will be created on partition 0. The work becomes unevenly distributed and the load balancing ratio drops to $r = .63$. The mesh can be rebalanced by reassigning the range each partition is responsible for. After rebalancing the load balancing ratio rises to $r = 1.0$, a perfect balance.

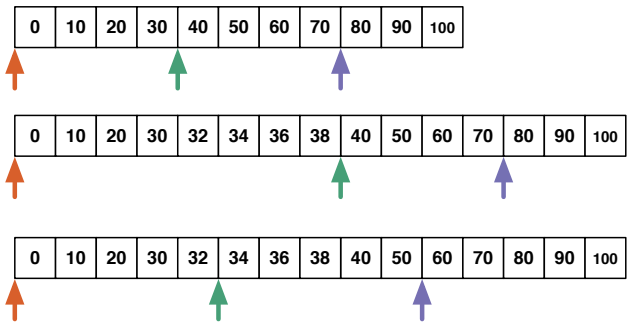


Figure 3.5 Rebalancing in Morton space

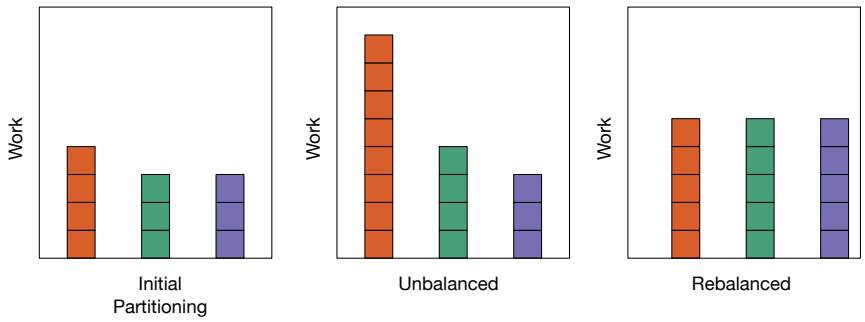


Figure 3.6 A well balanced partition

3.1.4 Geometry

Geometry can be represented in many forms, including Computer Aided Design (CAD), Non-uniform Rational Basis Splines (NURBS), Stereo Lithography (STL), and signed distance fields. The mesh generator is not designed to work with any specific geometry input. Instead, the mesh generator interacts with the geometry through an abstract interface. Any object which implements this interface must answer the questions shown in the code listing 3.1.

Listing 3.1 The interface required by a geometry object.

```
bool doesExtentContainGeometry(const Extent<double> &e) = 0;
int getStatusAtPointWithRadius(const Point<double> &p, double r) const = 0;
Point<double> getClosestPointOnSurface(const Point<double>& p) const = 0;
```

Each function is targeted to a specific mesh generation function. During top down subdivision only voxels with extent boxes containing geometry are subdivided. Deleting voxels internal

to the geometry is done with a flood fill which is seeded with the `in` or `out` status of points in the mesh domain. Finally, points are placed on the surface by using the closest point on the geometry's surface. So long as the geometry can be used to answer these simple questions, the mesh generation can generate a mesh on that geometry.

3.1.4.1 Stereo Lithography Collections

Most examples in this work were done with a geometry object based on a collection of triangular facets read in from a Stereo Lithography (STL) file. The functions listed above are all implemented to allow for STL collections to be used for mesh generation. STL collections can be output from many CAD programs or 3 dimensional sculpting applications. The STL format is a common format used in 3D printing.

An STL collection simply contains a collection of triangles. There is no requirement in the STL file format that the collection defines a surface mesh. There is no adjacency information connecting one facet to any neighboring facets. The collections are not required to be watertight, or even manifold. STL collections which represent geometry are often optimized to minimize the number of triangles in the collection. Because of this, the collections are generally unsuitable as surface meshes for computational simulation. The collections often have facets with very high aspect ratios as well as extreme changes in facet size. Figure 3.7 shows an example STL collection with both high aspect ratio triangles and regions of extreme gradation in facet size.

During top down subdivision, the geometry must quickly answer if an extent box contains geometry. The triangular facets are stored in an Alternating Digital Tree (ADT) which allows for

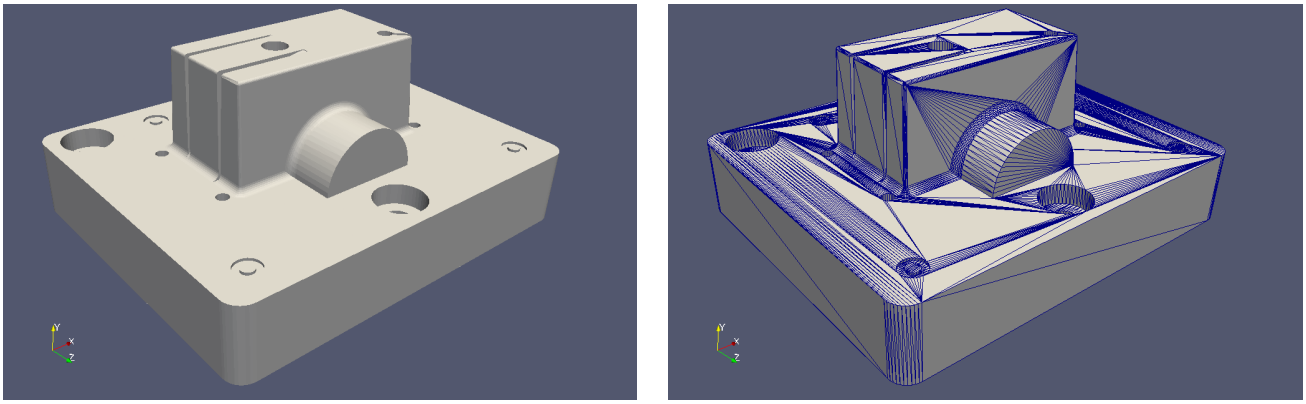


Figure 3.7 A faceted geometry surface

logarithmic query time of all the facets within an extent box to quickly determine if a facet exists within a voxel's extent box. The other two functions for determining status and projecting points onto a surface are more difficult.

One major difficulty when dealing with STL collections is floating point imperfections in the surface description. It is common for STL collections to have small imperfections in the surface they describe since it is not a strict requirement that STL surfaces be manifold.

3.1.5 In Out Testing

The geometry must be able to determine if a query point in space is inside or outside the geometry. This information is used by the grid generator to determine what voxels will remain in the computational domain and which voxels are not required. The STL based geometry class

determines in and out status by casting multiple rays from a query location, and counting the number of intersections each ray makes with the facets of the STL collection.

From a query point in space, rays are cast along each axis line. The rays extend past out of the meshing domain. The number of intersections with the ray and the geometry along each ray is added up. If the number of intersections along each line is odd, then the point is inside the geometry. If the number of intersection points is even, or zero, then the point is outside the surface. Figure 3.8 diagrams these two cases.

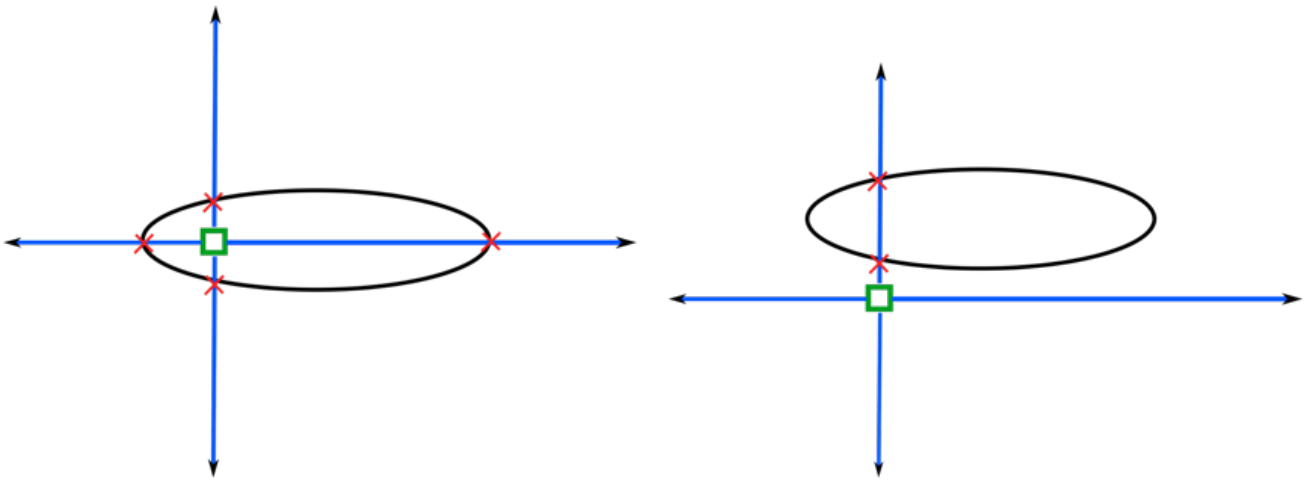


Figure 3.8 Even and odd intersections for ray casting

STL collections often have geometric imperfections such as gaps and regions of overlap. Geometry with these imperfections are sometimes referred to as "dirty" geometry. It is desirable

that the mesh generation process be tolerant to these issues. Primarily this means that in and out test of the STL collection should be robust.

Ray casting techniques can produce erroneous results if the casted ray passes through an ambiguous region of the geometry. Figure 3.10 shows a ray which passes at a tangent to the surface of the geometry. In this example, three rays will report that the query point is outside the geometry, however, one ray will report that the point is inside the geometry. Geometric features, gaps, overlaps, and numerical error can all cause ray casting algorithms to fail.

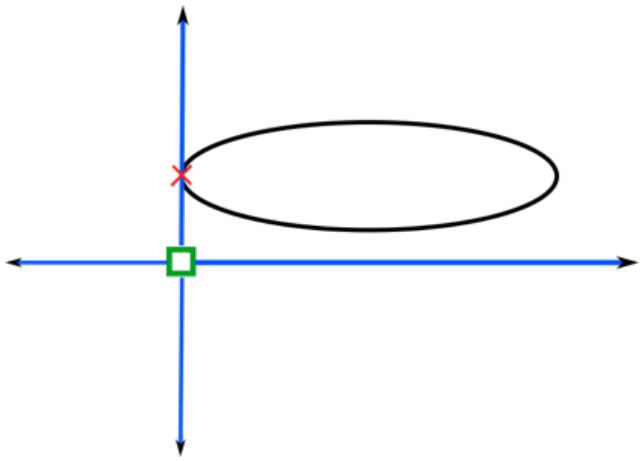


Figure 3.9 Surface tangent ray casting

A more robust algorithm for in-out testing can be built by recognizing that all causes of errors in ray casting only create local regions of ambiguity. Furthermore, the mesh generator does not need to know the in and out status of specific points. Instead, the mesh generator is only trying to determine if a voxel is inside or outside the geometry. Consider Figure 3.10. The mesh generator

needs to know if the green voxel is inside or outside the geometry. To determine the status the in-out test will cast rays from inside the green voxel. If the rays are cast from the voxel's centroid then rays will be in disagreement with 3 rays determining that the voxel is outside the geometry, but one ray determining inside. It does not appear that the algorithm has much *confidence* in its answer. This lack of confidence implies that the ray casting algorithm is locally ambiguous.

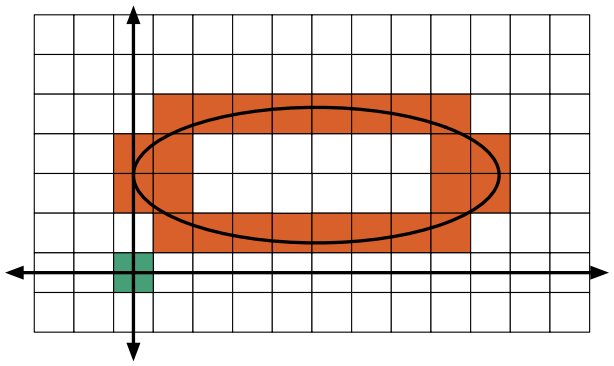


Figure 3.10 Surface tangent ray casting at voxel discretization

Notice that the selection of the voxel's centroid as the end point of the rays was arbitrary, but is also the source of the problem. Any point within the voxel is equally valid to determine if the voxel is inside or outside the domain. If the initial algorithm cannot determine a confident status it can try again using a new query point. Figure 3.11 shows that even a slight change in the location of the query point yields no ambiguity that the query point is outside the geometry.

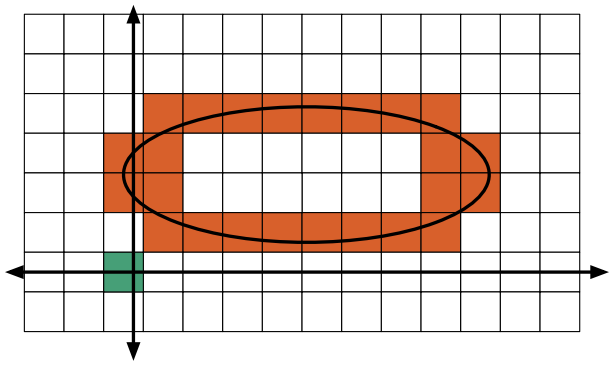


Figure 3.11 Shifted ray casting

In all cases where the results of the in-out query are not confident, the procedure is the same: try a different origin point for the rays. This assumes that any imperfections in the geometry representation are small.

3.1.6 Detecting Ambiguity with Ray Casting

Detecting ambiguity is possible since multiple rays are being cast simultaneously. The outcome of the rays is compared. If there is not a unanimous decision the result is considered ambiguous. However, it is possible with dirty input geometry that even if all the rays determine the same in-out status, the status could still be incorrect. If confidence in the test was only determined by a unanimous decision of ray casting then the answer would be reported as confident - and incorrect. This is because the individual ray casting algorithm can suffer from errors.

Ray casting techniques can suffer from numerical error. Some researchers have proposed using arbitrary precision arithmetic to mitigate numerical error. [85, 86] However, these techniques

still assume that the underlying geometry is generally free from imperfections. And these techniques can be slow on modern computer architectures. [87] Work has been done for mesh generation with dirty geometry. Wang et al. determines in-out status by user selection of cells in the fluid domain region. [88] Lahur et al. states that they use ray casting; however, they do not specify what algorithm they use or how they handle cases with overlaps and gaps. [34, 89, 90].

The ray casting algorithm used in this work was designed to try to detect when a ray is intersecting a triangle where the intersection test may be erroneous. The confidence in the outcome of the ray casting is reported along with the determined in-out status. If a result is not confident it can be thrown out and the in-out test for a voxel can be retried. The ray triangle intersection algorithm which was selected is based on the work by Möller et al. [91, 92]. A point (T) on a triangle described by three vertices (V_0, V_1, V_2) is given by:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

where (u, v) are barycentric coordinates, each on the range [0,1], and t is the distance along the ray's direction. A point on a ray can be described as

$$R(t) = O + tD$$

where O is the origin or the ray, and D defines the normalized direction the ray points. If the ray and triangle intersect, they do so when

$$R(t) = T(u, v)$$

$$O + tD = (1 - u - v)V_o + uV_1 + vV_2$$

Which can be cast as the following linear system:

$$[\,-D,\ V_1 - V_0,\ V_2 - V_0\,] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{3.1}$$

and solved for the barycentric coordinates on the triangle (u, v) and the distance from the ray's origin t.

The technique creates a coordinate system u, v using two of the triangle's edges. The location of the intersection with the ray and the triangle is determined on this coordinate system. If the intersection occurs at a location where both u and v are on the interval [0,1] then the ray and the triangle intersect.

Let the edges of the triangle be defined by $E_1 = V_1 - V_0, E_2 = V_2 - V_0$, and allow the substitution $T = O - V_0$. The solution to 3.1 can be obtained using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D\,, E_1,\ E_2|} \begin{bmatrix} |\ \ T, & E_1, & E_2\ | \\ |\ -D, & T, & E_2\ | \\ |\ -D, & E_1, & T\ | \end{bmatrix}$$

From linear algebra, substitutions can be made using: $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$ yielding the final form for computing the intersection of the ray and the triangle.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix}$$

If the barycentric coordinates (u, v) are outside the interval [0,1], or $t < 0$ (the triangle is behind the plane normal to the ray), then the ray and triangle do not intersect. However, if one of the coordinates u or v have values close to 0 or 1, then the ray intersects the triangle close to the

edge of the triangle. For a surface triangulation with small gaps and small regions of overlap, a ray passing very close to the edge of a triangle may mean that the ray passes close to one of these imperfections. Figure 3.12 shows two cases. The ray in the first case intersects the triangle near the triangle's center. There is little doubt that there is geometry at that intersection point. However, the second case shows a ray which intersects very near one edge of the triangle. This case can be marked as ambiguous. Ambiguous ray intersections can be retried using a different origin point of the ray. If the triangle represents a sharp feature in the geometry, or the triangle is near some small imperfection such as a gap or overlap, a small change in the ray's origin point may yield a different answer.

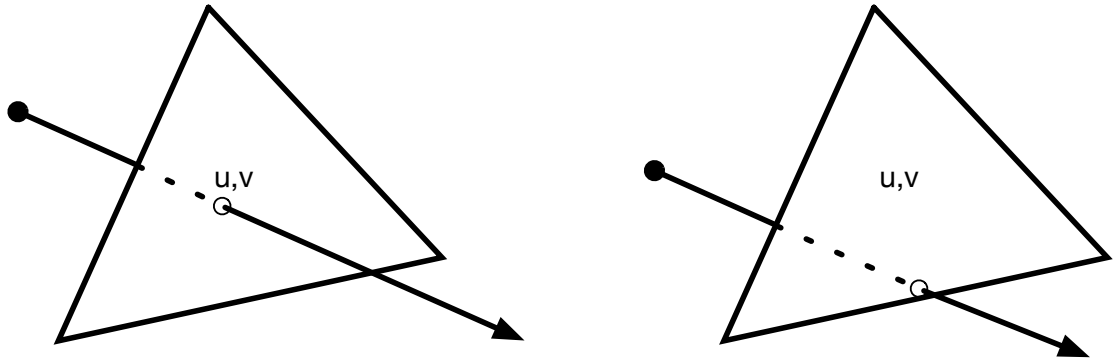


Figure 3.12 Triangle ray intersection

In most cases all the rays agree whether a point is inside or outside the domain. However, it is possible that one or more rays intersect the surface at a tangent, as shown in Figure 3.10. In-out

status could be determined for the point by taking the most often found outcome of the rays. In Figure 3.10 the most often found outcome was that the point is outside the surface.

Allowing the rays to vote to determine the in-out status is a good solution when the surface is well behaved. However, the problem is compounded by imperfections commonly found in STL files which may have small gaps or regions of overlap between surface triangles.

3.1.7 Deleting Internal Elements

The next stage in mesh generation is removing voxels which are inside the geometry. Voxels which cross the geometry are tagged as crossing during top down subdivision. All voxels which do not cross the geometry are left as untagged. Some of these untagged voxels will be inside the geometry and should be deleted. Figure 3.13 shows an example mesh divided into four partitions. The orange voxels are tagged as crossing, all other voxels are untagged.

Each partition can query the geometry to determine if a point in space is inside or outside the geometry. However, it is inefficient to query the geometry for each voxel. If the status of one voxel in a region is known, then a painting or flood fill algorithm can be used to mark the rest of the voxels in that region.

The painting algorithm iterates through the voxels until it reaches a voxel with an untagged status. The algorithm then queries the geometry to determine if the voxel is inside or outside geometry. Once the status of the geometry is known the seed voxel and the seed tag are passed to the painting algorithm to successively spread that mark to the neighboring voxels. The painting algorithm does not spread to voxels which already have a tag, so the algorithm does not spread

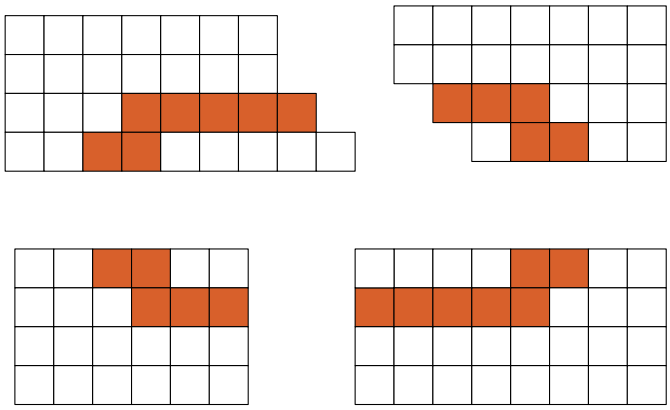


Figure 3.13 Geometry and domain partitioning

to voxels already tagged as crossing the geometry. Listing 3.2 shows pseudocode for the painting algorithm.

Listing 3.2 The painting algorithm.

```
void paintTagFromSeed(seed_voxel, seed_tag){
    Queue process;
    process.push(seed_voxel)
    while(not process.empty()){
        auto voxel = process.front();
        tree[voxel].tag = seed_tag;
        for(auto neighbor : tree[voxel].neighbors())
            if(tree[neighbor].tag == UNTAGGED)
                process.push(neighbor)
        process.pop();
    }
}
```

The painting algorithm can fail if the geometry is not closed surface, but small imperfections in the geometry are allowed. But these imperfections must be small with respect to the isotropic spacing of the mesh in the region of the imperfection. Figure 3.14 shows a curved geometry with a small gap. If the mesh spacing is coarse, this gap does not cause a problem for the painting algorithm. However, if the mesh spacing is small, the painting algorithm will fail.

The painting algorithm only operates on a local partition. It does not communicate tags across partition boundaries. Each partition has access to the geometry and determines in-out status for each untagged voxel in the partition without communication. After all voxels have been tagged, the voxels with the `in` tag are deleted.

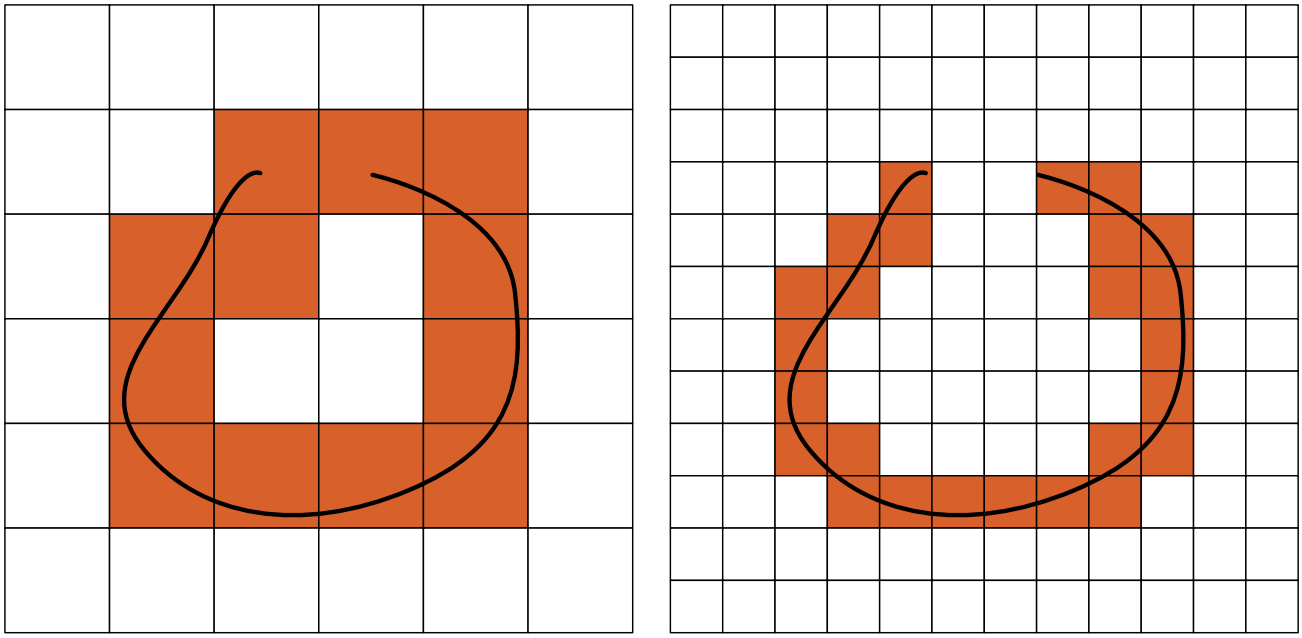


Figure 3.14 Feature imperfections and voxel spacing

3.2 Filling the Tree Bottom Up

Once top down subdivision is complete, the mesh has appropriate isotropic spacing near the geometry, but mesh spacing grows too quickly away from the geometry. Gradation is enforced by traversing the tree from the bottom layer of the tree and climbing up.

An example of a mesh which has undergone top down refinement can be seen in Figure 3.15. Notice that the mesh has large cells adjacent to much smaller cells. High gradation creates problems for many simulation techniques. Rules on cell growth rates must be enforced both locally within a partition and across partition boundaries.

Smooth cell size gradation is enforced with two characteristics. The first characteristic is that no cell has more than one hanging node per edge. This restriction is common among hierarchical

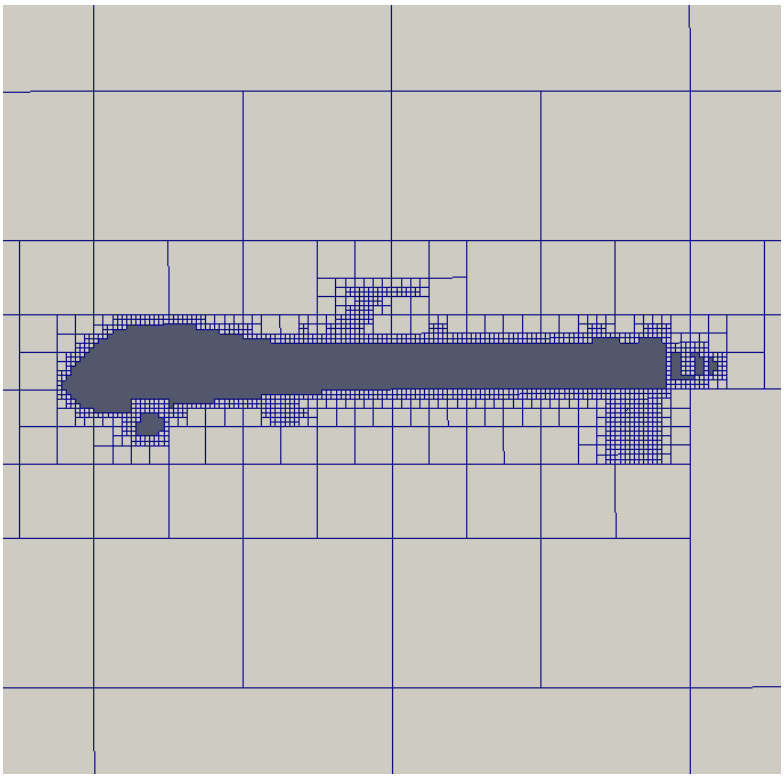
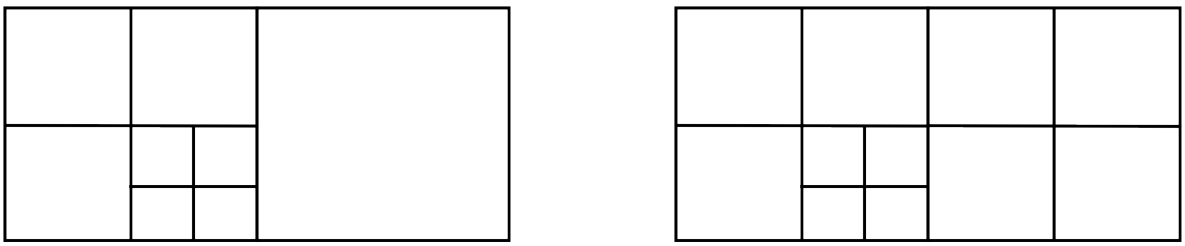


Figure 3.15 Top-down mesh with invalid hanging nodes

mesh generators and is referred to as a 4:1 neighbor gradation (2:1 in 2D), or sometimes simply the balancing rule. A violation of the balancing rule is illustrated in Figure 3.16. The large voxel on the right in Figure 3.16a has three neighboring voxels on the same side. The large voxel's left side has two hanging nodes. The balancing rule requires that each voxel side has at most only one hanging node. The balancing rule is being satisfied if all voxels are within 1 depth of all their neighboring voxels. A violation of the balancing rule can be resolved by subdividing any voxels which are too large (at too shallow a depth). Figure 3.16b shows the same set of voxels after a

subdivision refinement has been performed on the largest voxel. Now each voxel has at most one hanging node on any given side, and the balancing rule is now being satisfied.



(a) A violation of the balancing rule.

(b) The balancing rule enforced by subdivision.

Figure 3.16 Hanging node rule violation and resolution

The second quality characteristic attempts to control the growth rate of cell sizes. Typical best practices for mesh generation have been to use geometric growth rates to control cell sizes. Growth factors between 10 and 20% are commonly used. [93, 94]. Octree Cartesian techniques can only exhibit discrete cell sizes. If the mesh contains a cell of length d the mesh may also contain cells of length $\frac{1}{2}d$ or $2d$. The mesh will not contain cells with length $1.15d$. Because of discrete cell sizes, octree Cartesian meshes cannot produce cell sizes which vary as smooth as general unstructured meshing techniques. Cartesian meshes can specify layer thicknesses. These can be used to approximate continuous growth rates. Figure 3.17 shows a layer which is four cells thick.

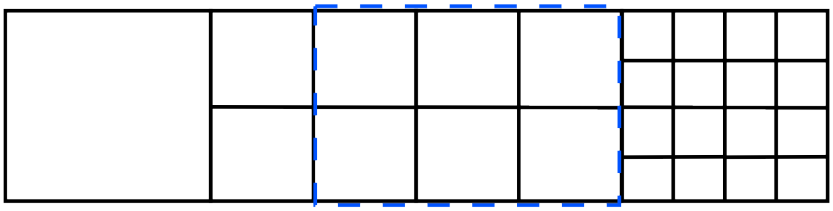


Figure 3.17 Thickness parameter enforcement

A region of a mesh with the same size cells is referred to as a layer. A layer thickness can be used to describe the minimum number of adjacent cells that must be within a layer before the cell spacing can change. Figure 3.18 shows cell spacing as a function of distance assuming the cell spacing is 1.0 at a distance of 0. Ideal cell spacing is shown for a 15% and a 20% growth rate. The voxel spacing (shown in green) grows in discrete jumps as spacing doubles from 1.0 to 2.0 to 4.0 and onward. If the layer is required to have a minimum of 5 cells, then the cell spacing is conservatively below a 20% growth rate. Increasing the minimum layer thickness to 7 cells conservatively matches a growth rate of 15%. Growth rates of 10% can be conservatively obtained by using layers with a minimum of 10 cells.

The enforcement of both the balancing rule and layer thickness occurs simultaneously and is handled in a bottom up sweep. The technique begins at the deepest level of the octree corresponding to cells of the smallest spacing. An extent box is created around each of these cells. Each extent

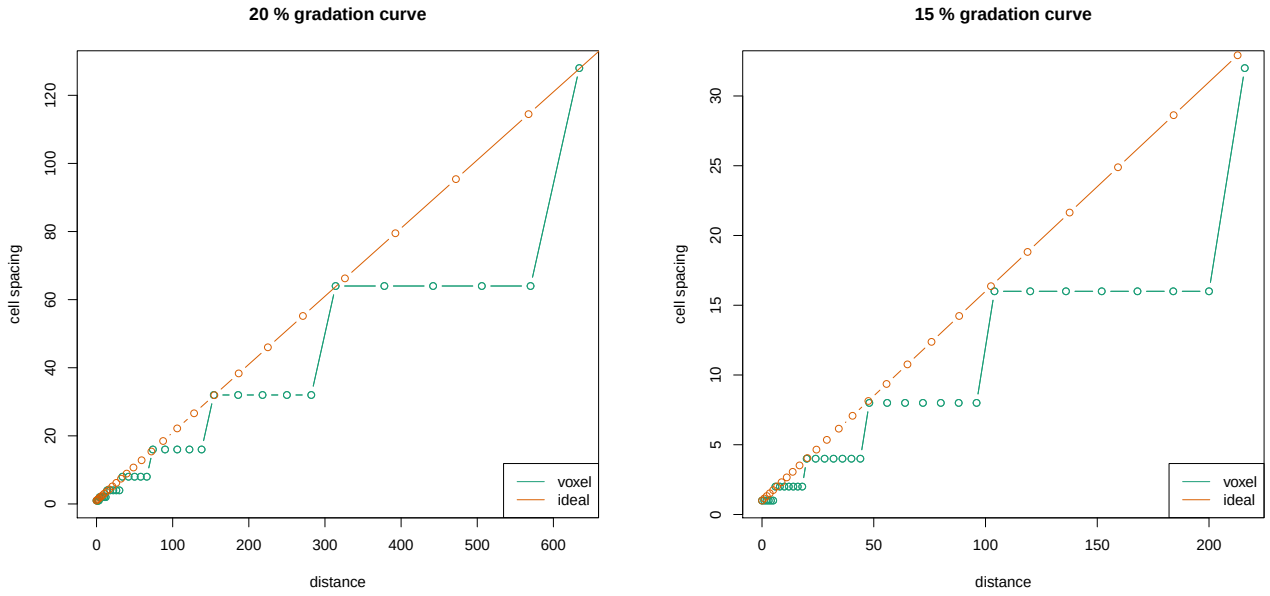


Figure 3.18 Thickness parameter compared to geometric growth rate

box is enlarged using the gradation parameter n described above. Once enlarged, all extent boxes are used for refinement until all cells inside the extent boxes match the smallest spacing. This

step creates a layer of the smallest cells. Listing 3.3 shows how layers are created within a single partition.

Listing 3.3 An example supervisor function which refines the mesh.

```
void OctTree::createLayers(int num_layers) {
    double offset = 0.25 / 1 << 21;
    for (int depth = getMaxDepth(); depth > 1; depth--) {
        double depth_spacing = 1 << depth;
        double layer = num_layers * depth_spacing + offset;
        for (auto& voxel : voxels) {
            if (not voxel.isLeaf())
                continue;
            if (voxel.depth != depth)
                continue;

            Extent refinementBox = voxel.extent;
            refinementBox.lo -= Point(layer, layer, layer);
            refinementBox.hi += Point(layer, layer, layer);

            refine(refinementBox, depth - 1);
        }
    }
}
```

After all the cells in the smallest layer have been created, the balancing rule is enforced on all voxels adjacent to the newly created layer. This is done by enlarging extent boxes for all cells at the smallest spacing. When enforcing the balancing rule, extent boxes are only enlarged enough to capture neighboring cells. Neighboring cells are subdivided until they satisfy the 4:1 requirement. Now, there are no cells at the smallest spacing that are adjacent to any cell that is greater than 8 times its size, and each layer of cells of the same size has the proper thickness. The process is

repeated at each depth of the tree beginning at the deepest level of the tree and proceeding to the highest.

3.3 Nearbody Grid Creation

Nearbody grid generation begins by removing all voxels crossing the geometry and by additionally removing voxels near crossing voxels. The resulting cavity contains space for both the geometry and for the nearbody mesh which will be created. Any voxel remaining in the mesh which is exposed to this cavity is marked as exposed and added to a list of voxels which will make up the nearbody mesh. Neighboring voxels in the vicinity of exposed voxels are also added to the growing list of voxels which will make up the nearbody mesh. Each partition then begins the process of converting voxels to nearbody cells.

The nearbody grid is an unstructured graph. The graph is stored using a collection of cells, unlike the offbody grid which is stored as simply cartesian extent boxes. For simplicity only one type of cell is allowed in the nearbody. This cell is simply a collection of the 26 possible nodes which may exist in a voxel, the node ordering is shown in Figure 3.19.

Translating from the offbody to the nearbody occurs in three stages. First nearbody cells are initialized. Then the nodes are generated and marked in their owning cells. Finally, non-local graph information is transferred to neighboring partitions.

The array of nearbody cells is initially allocated based on the number of voxels being exported. Each cell begins with all its adjacent nodes being marked as uninitialized. Up to this point, no nodes have been generated. It is unknown which of the potential 26 nodes in each cell will

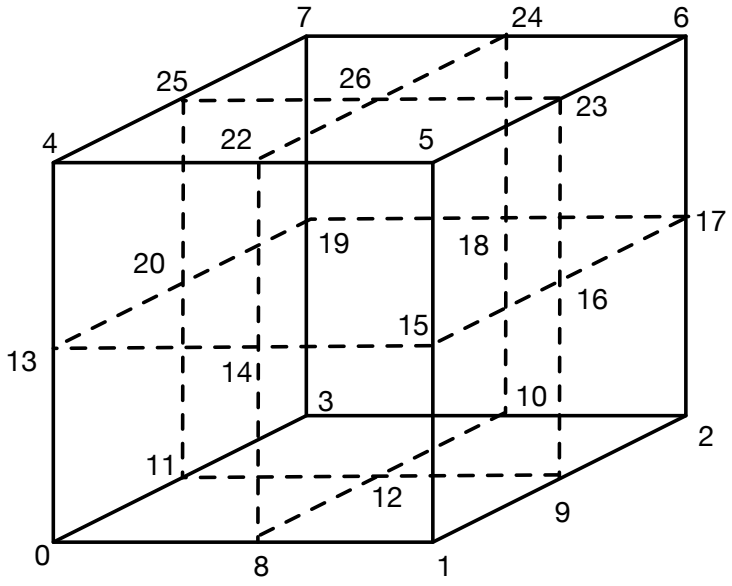


Figure 3.19 Potential voxel nodes

exist. Each cell will always have 8 corner nodes. But a cell may need additional nodes depending on the size of the neighboring cells. Some cells will need mid-face and mid-edge nodes to match with corner nodes of neighboring smaller cells.

Once the cell array is allocated and initialized, the nodes are created. The process iterates to each voxel being exported and checks the 8 corners of the cell associated with that voxel. If a corner node is still marked as uninitialized, then the node must be created. The coordinate value of that node is determined based on the extent box of the voxel. The coordinate is appended to the list of node coordinates, and the index of that coordinate is recorded. The newly created node coordinate and index are then passed recursively down the octree to each voxel containing the node's coordinate. If the node reaches a voxel where one of the 26 voxel nodes has the same coordinate value then the node's index is marked for that cell's local node. There is no risk of getting an incorrect comparison between a voxel's local node and the input coordinate. The minimum edge length of the mesh is known and can be used to a tolerance for the floating point comparison.

Once all the voxel corners have been processed many cells will have mid-edge and mid-face nodes. However, not every possible combination of mid-edge and mid-face nodes are allowed. For simplicity, a rule is then applied. Any face adjacent to an edge with a mid-edge node is required to have a mid-face node. An example of this is shown in Figure 3.20. One sweep through the nearbody cells is required to check for any missing mid-face nodes. If any are found, they are created and set using the same octree based method that was used for generating the original nodes.

Some vertices will exist in multiple partitions. These vertices are said to have remote-residency (that is they also *reside* on non-local partitions). During the nearbody projection stage only one partition will "own" vertex. That partition will be responsible for computing that vertex's new

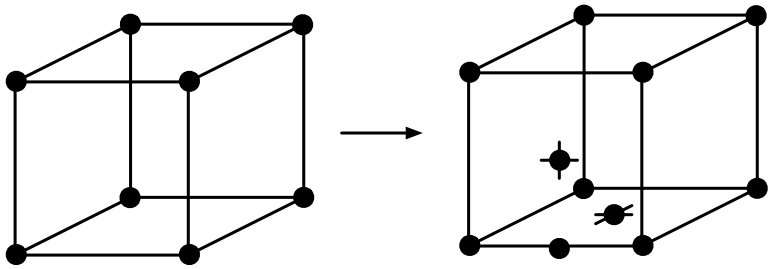


Figure 3.20 Mid-face and mid-edge nodes

coordinate location and notifying other partitions when that location changes. To facilitate updating information on non-remote partitions, every partition which contains a vertex also maintains a list of remote residencies for that vertex. The residencies are stored as partition identifier and local vertex identifier pairs. No global node identifier is generated. It is not needed by any part of the grid generation process. It is easier to map directly from a vertex local index to the remote partition - remote identifier pair than it is to map from local index, to global index, then to a remote partition's local index.

This remote residency information must be generated. Each partition exports a list of local vertices which are in voxels marked as having non-local residency or those voxels' neighbors. The list contains both each vertex's coordinate location, and the local vertex identifier. These lists are broadcast to all other partitions. The receiving partitions process each vertex, by passing the coordinate down the local octrees to determine what local vertex shares the same coordinate. When a shared coordinate is found, the mapping between local vertex and remote residency is created. When a node has multiple residencies, the partition with the lowest partition number is assigned the owner.

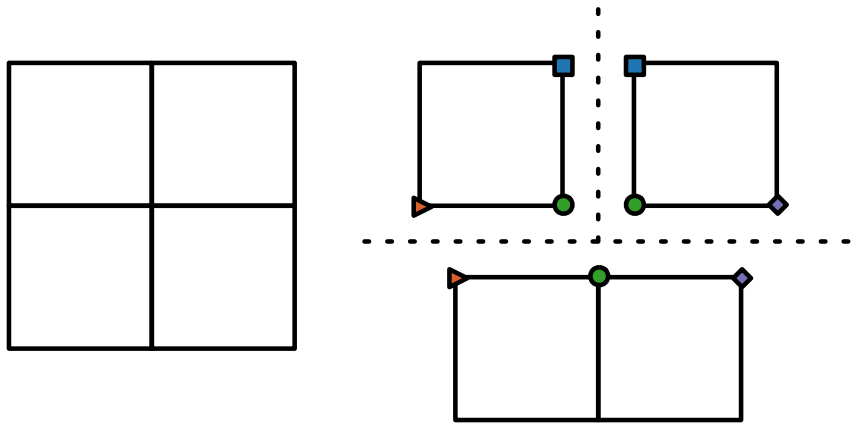


Figure 3.21 Remote node residencies

3.4 Body Conforming Grid Using Projection

Prismatic elements are created once the the nearbody voxels are translated to the nearbody mesh data structure. These prismatic elements are created beginning from the exposed faces of the nearbody voxels. These faces are duplicated creating initially degenerate prismatic elements. All the nodes which will be projected onto the surface mesh are tagged as SURFACE. All the nodes in the nearbody mesh which are also contained in cartesian locked voxels in the offbody mesh are tagged as FROZEN. All other nodes in the nearbody mesh are tagged as INTERIOR. During surface optimization FROZEN nodes will not be allowed to move. Nodes labeled INTERIOR will be allowed to move to increase the quality of adjacent elements, both voxel elements and prismatic elements. Nodes tagged as SURFACE will be allowed to move to optimize both quality of adjacent elements and to ensure that the geometry is captured.

Nodes marked SURFACE will be projected onto the geometry. The position of surface nodes is projected onto the geometry using algebraic smoothing. A new node's location is computed as the weighted average of the node's previous location and a distance weighted average of the node's adjacent surface faces. This is given by:

$$p_{n+1} = (1 - \omega)p_n + \omega \frac{\sum_i f_i d_i}{d_i}$$

where p_n, p_{n+1} is the node's old and new location, ω is a relaxation weight (usually set to 0.5), f is the centroid of an adjacent surface element, and d is the distance between the node and the face centroid. This smoothing step is repeated four times. The final step of projection is to move SURFACE nodes to the closest point on the geometry using the `getClosestPointOnSurface()` function implemented in the geometry wrapper.

The smoothing technique does not capture sharp geometric features. Projection techniques which include geometric feature capturing have been explored by other researchers, and it remains an area of active research. [30, 34, 90] This work makes no contribution in this area and mostly follows the work of Lahur et al. [34]

The location of SURFACE nodes is fixed once they are moved directly onto the geometry surface. The last step of nearbody mesh generation is to smooth nodes marked INTERIOR. Algebraic smoothing is used again for interior nodes.

$$p_{n+1} = (1 - \omega)p_n + \omega \frac{\sum_i c_i d_i}{d_i}$$

For interior nodes adjacent volume element centroids (c) is used instead of adjacent face centroids.

3.5 Adding Resiliency

The grid generation process must be broken into sections where checkpoints can be inserted in order to operate within this fault tolerant model. If a fault has been detected than the process can restart from the last checkpoint.

3.5.1 Resilient Units

Figure 3.22 shows an overview of the parallel grid generation process. The process is split into four categories: Setup, Off Body Generation, Near Body Translation, and Near Body Optimization. These high level categories can be broken down further. The highlighted steps Gradation Loop, and Optimization Loop are, the two single most expensive operations in the grid generation process consuming 65 and 11 percent respectively of the total running time.

Minimizing the longest elapsed time between checkpoints will minimize the amount of potential work lost in the event of a failure. Figure 3.23 shows where creating checkpoints have been inserted into the process.

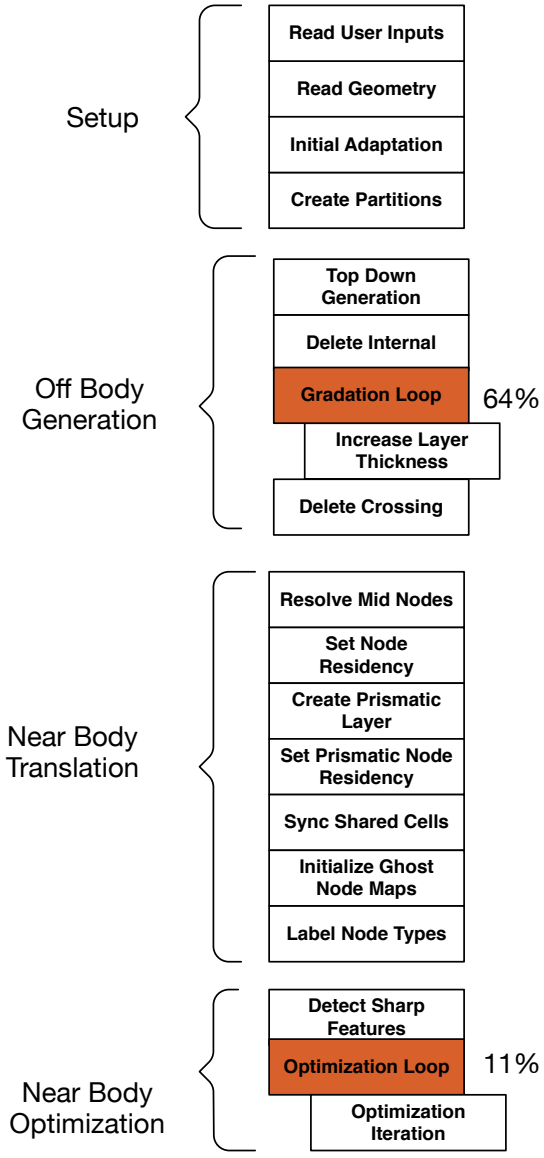


Figure 3.22 Parallel grid generation overview

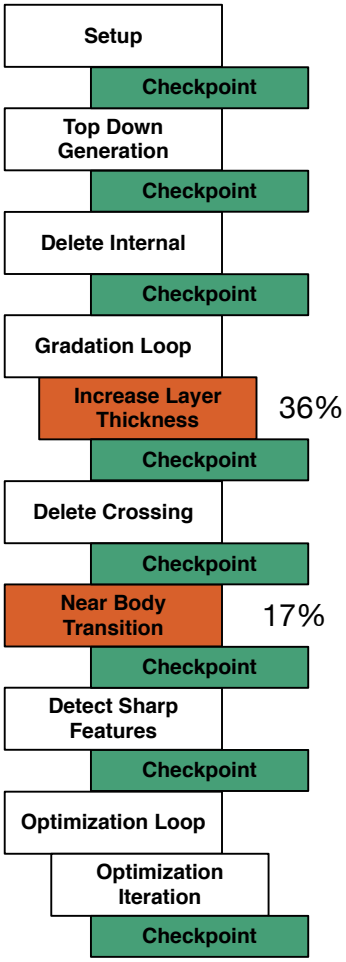


Figure 3.23 Parallel grid generation with checkpointing

3.5.2 Supervisors

Each communication operation has the potential to throw a timeout exception. Communication operations must be prepared to capture these exceptions and perform the required fix. A simple pattern can be used to manage these operations which may fail, the supervisor pattern.

The supervisor pattern has two components: a set of worker agents, and a supervisor. The role of the supervisor is to monitor the operation of the agents, and if an agent fails the supervisor is responsible for recovering from the failure. The supervisor pattern is simply implemented in the mesh generation code using timeouts. Pseudocode for a supervisor pattern is shown in Listing 3.4. The outer function `refineToDepth` is responsible for refinement, including recovering from any failures which occur during refinement. The delegated method `refineToDepth_agent` is only responsible for the logic of refinement; it is not fault tolerant.

When the supervisor function returns, it guarantees that the refinement operation was completed. It also guarantees that the progress has been saved in a checkpoint.

Listing 3.4 An example supervisor function which refines the mesh.

```
void refineToDepth(Mesh& mesh, Geometry* geometry, int depth){
  bool successful = false;
  while(not successful){
    try {
      mesh.refineToDepth(geometry, depth);
      mesh.createNewCheckpoint(vault);
      successful = true;
    }
    catch(TimeoutException& e){
      Messenger::recover();
      mesh.rollbackToLatest(vault);
    }
  }
}
```

CHAPTER 4

FAULT TOLERANCE: METHODOLOGIES AND ALGORITHMS

4.1 Introduction to Multiple In-Memory Checkpointing

Checkpoint and restart is a common strategy used by high performance applications. A checkpoint is a set of working data needed by an application. If the application was to halt unexpectedly a checkpoint must contain any data needed for the application to resume calculation from the point where the checkpoint was created. Typically the checkpoint is written to permanent storage; this is called in-disk checkpointing. If the application fails during execution, it is simply restarted beginning from the latest checkpoint file. Unfortunately, read and write bandwidth to file systems is much slower than network bandwidth. Disk based checkpointing may become prohibitively slow if failures become more commonplace. As shown in chapter 2, simulations of full aircraft in maneuvering flight man suffer hundreds of failures during computation.

Of the many strategies which can be employed to increase resiliency of large scientific systems, multiple in-memory checkpointing was selected to be used in this work rather than in-disk checkpointing. With in-memory checkpointing, the data in the checkpoint is stored in main memory on the compute nodes which are running the application. Multiple in-memory checkpointing stores multiple copies of the checkpoint data on different nodes within the network. Take a look at Figure 4.1 as an example. This is a double in-memory backup since a backup of each work unit is stored

twice. Once on the processes which owns the work unit, and once on an adjacent process. If these processes are properly mapped onto different compute nodes, then this backup strategy is able to survive a failure on any single compute node. It could even survive multiple simultaneous failures so long as the surviving processes were either A and C, or B and D.

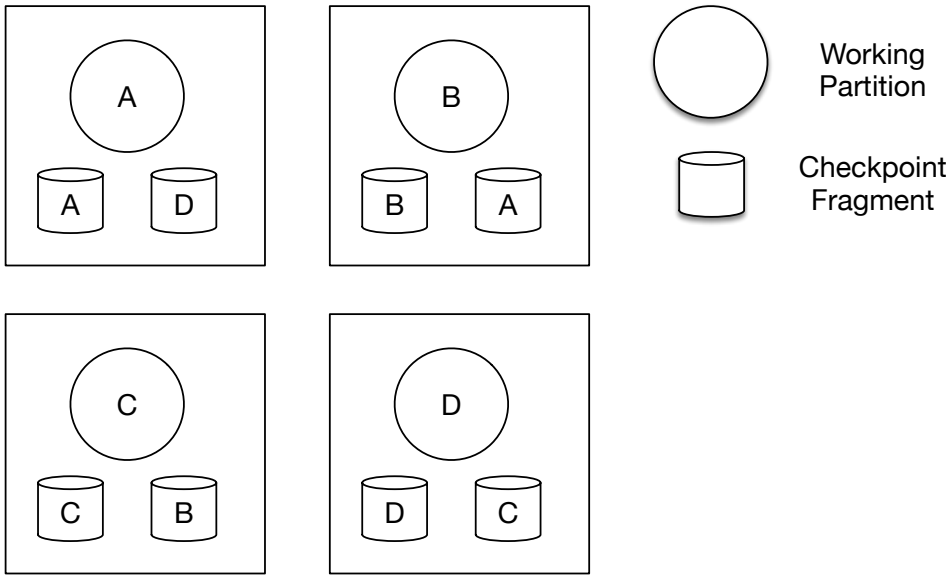


Figure 4.1 Double in-memory checkpointing

The remainder of the chapter discusses how the software developed for this work implements a multiple in-memory checkpointing strategy. The software must be able to redundantly store checkpoint data, detect failures, recover the messaging layer after a failure, and finally retrieve checkpoint data if a complete set of data survives a failure.

4.2 Software Overview

In-memory checkpointing is a critical component of a fault tolerant software package which was developed for this work. The package can be used to develop parallel applications which are more resilient to failures within the compute network. The software package is made of a fault tolerant messaging system and a multiple in-memory checkpointing library.

The package is sensitive to communication latencies within the network and will report that a process has failed if communication with the process stops. The failure is passed to the application layer which can then attempt to recover. The package aides in that recovery by determining what processes remain and what checkpoint data those processes contain. The application can then rollback to when the checkpoint was created if a complete checkpoint exists. If a complete checkpoint did not survive the failure, the program will exit.

The software framework is split widely into two user facing modules. The first is called `Messenger` which manages the message passing on behalf of the end user's application. The second module is called `Vault`, and it manages the redundant storage and retrieval of checkpoints as requested from the user application.

4.2.1 Messenger

`Messenger` is the user facing module which delivers messages between processes on behalf of the user application. Parallel scientific code is often written to the MPI standard, so it was desirable to make the API of Messenger similar to MPI. Messenger maintains the abstractions of

rank and *size* borrowed from MPI communicators. Point to point messages can be sent between any two processes within the network. Additionally, collective communications are also implemented.

`Messenger` supports more messaging types than is common in MPI. Most notably, `Messenger` supports Remote Procedure Calls (RPC). A user application can register a function using a unique identifier referred to as the *channel*. Any other process within the network can then remotely call that function. `Messenger` doesn't know anything about the function so the user application can register any function. Using an RPC protocol make it easier to use algorithms where the runtime communication pattern is highly dynamic making it difficult to precisely match send and recv calls.

`Messenger` expects that each communication sent by the user application will be completed within a reasonable timeout length. If the message is not completed by the end of the timeout, an exception is thrown to the user application notifying the user application that a process has failed and recovery must be initiated. The user can then call the recover function within Messenger which blocks until the abstractions of rank and size are again valid. A process' rank may change during a recovery. This is to ensure that the set of ranks is always the interval [0-size).

4.2.2 Vault

Storing and retrieving checkpoints is handled through a module named `Vault`. Checkpoints stored in `Vault` can be partitioned into pieces called fragments. A full checkpoint is made of an arbitrary number of fragments. Each fragment is assigned a unique identifier. If a checkpoint is partitioned into n fragments, all fragments [0-n] must survive a failure for the checkpoint to be

complete. `Vault` allows any user process to store a fragment and manages redundantly storing the fragments, in-memory, throughout the network of compute nodes.

All fragments which make up a checkpoint must be stored in memory before a failure occurs, or the application will not be able to recover. It is the responsibility of the user application to create fragments, and to give the fragments to `Vault` so they can be properly stored.

4.3 Data Structures

Checkpoint fragments are implemented in a specific C++ data structure called a `Stream`. `Stream` allows checkpoint fragments to be flexible in what data is stored, and are still simple to use. Each `Stream` stores data in a linked-list. Each element in the list is a tuple containing a pointer to a contiguous block of memory and the size, in bytes, of the block. Each block of memory can have a different size so that any data structure can be stored within a `Stream` regardless whether it is structured or unstructured data. Objects are stored and retrieved with a First In First Out (FIFO) strategy and `Stream` overloads the left and right shifting operators for convenience. A simple example use of these operators is shown in listing 4.1. Notice that after the variable `b` is pushed into the stream its value is changed. However, the value is pulled back out of the `Stream` is the unchanged value. That is because data is copied when it is pushed into a `Stream`. Stream works with any C++ Plain Old Data (POD) type without modification and also has support for std::vector types of POD as well, as shown in Listing 4.2.

Listing 4.1 A simple usage of stream to push integers into a `Stream`, then extract them out.

```
MessagePasser::Stream stream;
int b = 5;
stream << b;
b = 7; // invalidate b, testing that the data was copied
int a;
stream >> a;
REQUIRE(a == 5);
```

User defined data types can be made to work with Stream. There are two ways of making a user defined type work with `Stream`. If a type is contiguous in memory then nothing has to be done to make it work with Stream. When an instance of that type is pushed into a `Stream` it will be copied using a `memmove`. Many simple data structures, such as 3 dimensional points, fall under this category.

Listing 4.2 Stream also works with std::vector of POD

```
MessagePasser::Stream stream;
std::vector<int> input = {1,2,3};
std::vector<int> output;
stream << input;
input.assign(3,9);
stream >> output;
REQUIRE(( output == std::vector<int>{1,2,3} ));
```

More complex user types must know how be packed into a stream and how to be unpacked from a stream. The user defined type can describe how the object should be packed and unpacked by defining functions for the left and right shifting operators. An example of a user type (called

MyClass) which has added support for stream is shown in Listing 4.3. Notice that the memory layout of this class is not contiguous because the class contains a data field of the type `std::vector`, therefore, simply a call to `memmove` would only perform a shallow copy. If the stream was then communicated across the network, the remote process would not receive the complete state of the object. The MyClass type has defined both the input and extraction operators which describe how to perform a deep copy. Listing 4.4 demonstrates how stream can be used to store and retrieve an instance of the MyClass type using a stream.

Listing 4.3 An example of adding the stream input and extraction operators to a generic user class.

```
class MyClass {
public:
    int a;
    std::vector<int> vec;
    double d;

    MyClass(int a, const std::vector<int>& vec, double d)
            : a(a), vec(vec), d(d){}
    inline friend MessagePasser::Stream & operator<<(
        MessagePasser::Stream &stream, const MyClass& myClass){
            return stream << myClass.a << myClass.vec << myClass.d;
    }
    inline friend MessagePasser::Stream & operator>>(
        MessagePasser::Stream &stream, MyClass& myClass){
            return stream >> myClass.a >> myClass.vec >> myClass.d;
    }
};
```

Listing 4.4 Storing and retrieving instances of MyClass with Stream.

```
MyClass sending(9, {1,2,3}, 5.5);
MessagePasser::Stream stream;
stream << sending;
MyClass recving;
sending.a = 9001;
sending.vec[0] = 9001;
stream >> recving;
REQUIRE(recving.a == 9);
REQUIRE((recving.vec == std::vector<int>{1,2,3}));
REQUIRE(recving.d == 5.5);
```

4.4 Messenger Implementation

The first step in fault recovery is detecting that a problem has occurred. Unfortunately, standard MPI does not offer any reliable strategies for detecting failures. This was the primary motivating reason to develop a messaging platform specifically designed with fault tolerance in mind. The following section discusses the implementation details of Messenger.

An overview of the design of the Messenger package is shown in Figure 4.2. It is important to note that, as shown in this figure, a simulation code using this package only interacts with the topmost layer of the package. This topmost layer, simply called `Messenger`, is the public facing API.

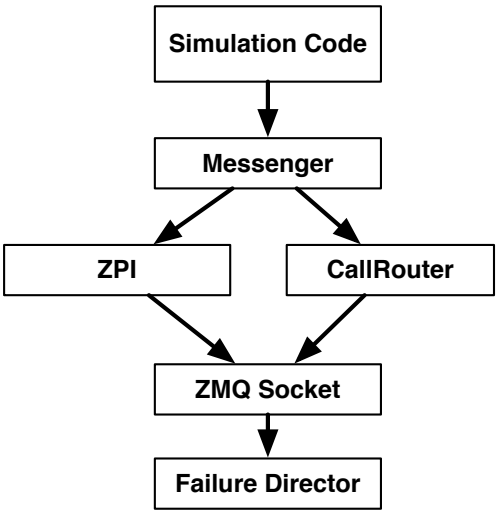


Figure 4.2 Internal layout of Messenger

4.4.1 Messenger, the API Layer

The topmost layer of the messaging package is known as `Messenger`, this is the only layer that directly interacts with the simulation code. The full public interface of the Messenger API is shown in Listing 4.5.

Listing 4.5 The interface of `Messenger` module.

```
class MessengerZMQ {
public:
    friend class ChannelZMQ;
    static void initialize();
    static void finalize();
    static int rank();
    static int size();
    static void barrier();
    static void send(MessagePasser::Stream &s, int destination);
    static void recv(MessagePasser::Stream &s, int source);
    template <typename T> static void send(const T & a, int destination);
    template <typename T> static void recv(T & a, int source);
    template <typename T> static void send(const std::vector<T> &a,
                                           int destination);
    template <typename T> static void recv(std::vector<T> &a, int source);
    template <typename T> static void broadcast(T &a, int source);
    template <typename T> static void broadcast(std::vector<T>&a, int source);
    template <typename T> static std::vector<T> gather(T& a, int destination);
    template <typename T> static std::vector<T> allGather(T&a);
    template <typename T> static T max(T t);
    template <typename T> static T min(T t);
    template <typename T> static T sum(T t);
    static ChannelZMQ registerChannel(int channel,
                                      std::function<void(MessagePasser::Stream)>);
    static std::vector<int> channelsInUse();
    static void recover();
};
```

A developer who is familiar with MPI can see some common concepts: send, recv, broadcast, gather, allgather, and barrier. In fact, the only functions which may look foreign are the final

3: `registerChannel, channelsInUse, recover`. `Messenger` uses `registerChannel` and `channelsInUse` to support Remote Procedure Calls (RPC). The last function, `recover`, tells `Messenger` to recover the messaging package after a failure. The recover function will be discussed in detail later.

The communication routines in `Messenger` are templated so that they can take many data of different types. The caveat is that the type passed must be contiguous in memory and therefore can be communicated by copying the object byte by byte. Many common C++ data structures cannot be communicated in this way. In fact, perhaps the most useful data structure in C++, `std::vector`, does not fit this description. Specialized functions for send, receive, and broadcast were all written to ensure that `std::vector` can be used easily and efficiently with `Messenger`. These routines in turn assume that the data stored in vector are simply copyable. This may sound like a dangerous assumption that would astonish a user of the `Messenger` API and cause all manner of hard to find bugs. Luckily, this is not the case. The C++ templating system ensure strong typing that is checked at compile time. The data structures passed to `Messenger` are checked at compile time using `std::is_trivially_copyable` to ensure that simple `memmove` copies are sufficient. The C++ compiler will refuse to compile if the object passed to `Messenger` does not meet these requirements. This strong typing is checked both on objects passed to `Messenger` directly, or indirectly through a`std::vector` of that object. The user has the option of packing complex data structures which cannot be trivially copied into a `stream`. The stream can then communicated using `Messenger` message passing calls.

`Messenger` supports Remote Procedure Calls (RPC) in addition to supporting simple data communication through message passing. A user can register a function to enable it for remote

calling. By registering a function the user receives a `Channel` object in return. The `Channel` object is how the user application makes remote function calls. `Channel` objects have only one function, `send`, which sends a `Stream` to a specific *rank*. The `send` command will trigger `Messenger` to communicate the data stored in the input `Stream` to the *rank* specified. The receiving *rank* will then pass along the `Stream` to the function which was registered on the `Channel`.

`Messenger` is only the publicly facing API of the messaging package. The functions of `Messenger` ensure that the type checking occurs at compile time, and then they delegate the communication to lower level objects at runtime. Internally `Messenger` only stores pointers to the lower level messaging subsystems, `ZPI`, and `CallbackRouter`. We will ignore these two subsystems for now and instead jump right to a discussion of `ZMQ Socket` which is the lowest layer in the messaging package. The discussion will then circle back around to discuss both the messaging subsystem `ZPI` and the RPC subsystem `CallRouter`.

4.4.2 ZMQ Socket Layer

The entire messaging package was written on top of ZeroMQ, an open source messaging library which supports distributed messaging. [95] ZeroMQ can be configured to run over many different communication protocols, however for this work only TCP is used. ZeroMQ has some significant benefits over simply using TCP sockets directly. ZeroMQ sockets can be started up in any order. The send call can be posted before the receive call, or vice versa. Critically, messages sent via ZeroMQ are guaranteed to arrive in full and intact or not at all. This guarantee relieves `Messenger` from having to verify the integrity of received messages.

ZeroMQ supports multiple communication paradigms; however, the only one used by the messaging package is a simple client server strategy. The basic outline of a client server exchange is shown in figure 4.3. In a simple client server interaction the client sends an initial communication called a request to a server. The server takes that request, performs some action to process the request and sends back a reply. Finally, the client sends back an acknowledgement of the reply. This is the basis of all the communication within the messaging package. Ultimately all communication goes through a ZeroMQ socket and is sent in this client server manner.

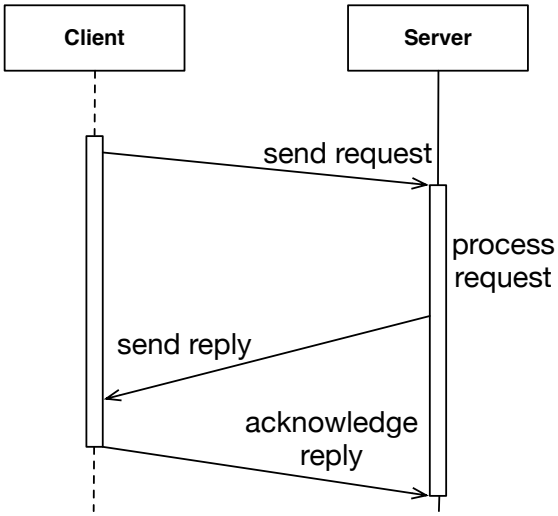


Figure 4.3 A client server interaction

ZeroMQ's C bindings have been wrapped into the`ZMQ Socket` layer of `Messenger`. The `ZMQ Socket` layer was split into two modules: `ZMQServer` and `ZMQClient`. Each process must

have a server running which is ready to accept incoming requests from clients at any time. These servers are called `ZMQServer`. Both subsystems `ZPI` and `CallbackRouter` run a `ZMQServer`. The server will receive requests from a specific port. `ZMQServer` is running for each communication subsystem on each process across the compute cluster. When a message is sent from one process to another, the sending process creates a `ZMQClient`. The client connects to the receiving server and sends the message. The receiving server processes the message and then replies with the size of the message. The sending client receives the reply, and returns a 0 in acknowledgement. If the sending client receives the reply, the original message can be freed from memory. Once the receiving server receives the final acknowledgement it can proceed to process the next incoming message.

The `ZMQ Socket` layer of the messaging package wraps up the underlying ZeroMQ library calls and additionally adds two pieces of functionality. First, both the client and the server socket in the `ZMQ Socket` layer can convert from `Stream` data types to the data types required by ZeroMQ. Secondly, `ZMQ Socket` clients and servers can be configured to simulate failures. Before any send or receive call on either the server or the client sockets, they first make a call into `FailureDirector` to see the socket should simulate a failure. The `FailureDirector` has only one public function `shouldFail` which simply returns `true` if the socket should fail, and `false` if it should not. The `FailureDirector` can be configured to fail randomly, when it receives a network communication, or when a particular file exists. If a socket is told it should fail, the socket immediately enters into a loop where it gets stuck until either the program exits, or a `wakeUp` message is sent to it.[1]

[1]This wake up message is useful to cleanly shut down processes at the program's exit which are simulating failures. Otherwise they have to be killed manually from the command line, which can be tedious in a compute cluster.

Both the client and server wrappers in the ZMQ Socket Layer have simple interfaces. A socket is constructed with the IP address and port number which they should communicate through. The client socket will bind to this IP address and port number and then begin listening. Client sockets will connect to servers through IP address and port number and send requests. The server socket supports three methods: recv_request, send_reply, and have_incoming_request. The client socket supports three similar methods: send_request, recv_reply, and have_incoming_reply. The send and receive calls are written to support communicating data through pairs of pointers and byte lengths, as well as supporting communicating Stream objects directly.

The have_incoming functions are used when polling on the socket. These functions take a length of time in milliseconds that they will listen for an incoming message. If a message is received during that time, or if a message was already pending, the function immediately returns so that the message can be processed. If there is no incoming message, then the thread calling this function sleeps for the full amount of time that was requested. Figure 4.4 shows two simple diagrams. The top shows a case where the calling thread sleeps for the full amount of time allowed. In the bottom case, the calling thread sleeps for a short amount of time, before it is waken up by an incoming message. The calling code can then immediately begin processing the incoming message.

Without have_incoming, the calling code would have to poll often in order to decrease the latency in the messaging system. Unfortunately, polling too often would starve other threads of needed compute resources slowing down the entire system. Using have_incoming allows both messaging subsystems ZPI and CallbackRouter to be responsive to incoming messages and also to be efficient with the available compute resources.

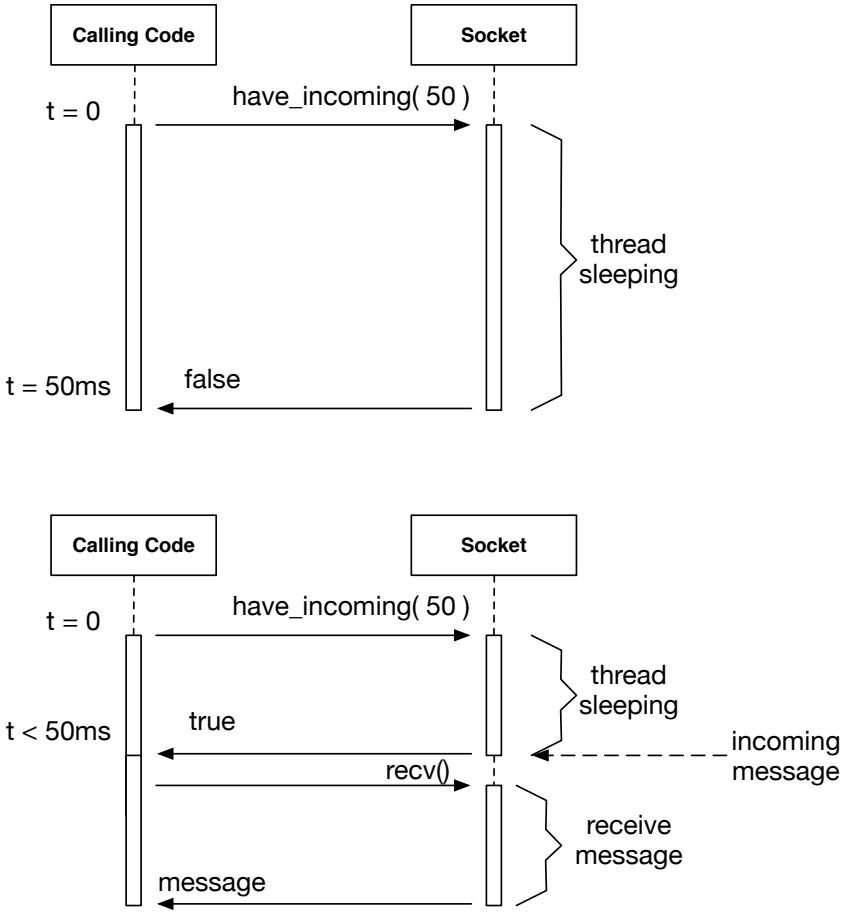


Figure 4.4 Efficient socket polling

`ZMQ Socket` is the heart of the messaging package. It detects failures, and actually makes the networking calls. The socket layer also can be configured to simulate failures. This is the point of entry for simulated failures will will be used in the next chapter to test resiliency. No other part of the simulation code or the messaging system knows if a failure is real or simulated.

4.4.3 ZPI, for Message Passing

`ZPI` is one of the two subsystems of the messaging package. ZPI, as its name suggests, supports the MPI-like messaging. No part of `ZPI` is exposed to the user and instead is only accessed by `Messenger`. `ZPI` (which loosely stands for ZeroMQ Passing Interface) only has two public functions, `send`, and `recv`. The declarations of these functions are shown below:

```
void send(const void *a, size_t byte_length, Address address, int source);
void recv(void *b, size_t byte_length, int source);
```

The send and receive functions only work on contiguous data structures. All communication sent by the user is split into a series of send and receive functions. `ZPI`'s simple public interface is supported by the messaging package's most complex implementation. The following discussion will walk through first the process `ZPI` goes through to send a message, and then go through the more complicated process of receiving one.

Sending a message through `ZPI` is relatively simple. The calling code (`Messenger`) calls `send` with the appropriate pointer, message length, ip-address and port number, and the rank of the calling process. When the calling thread posts the send call, the call will block until the send is

complete. Internally ZPI starts up a ZMQ Client Socket which then connects to the IP address and port number of the receiving process. ZPI sends first a message header which contains the sending processes rank and an integer tag, and then sends the message as a request to the receiving processes ZMQ Server Socket from the newly created client socket. The client socket waits to get a reply from the receiving server socket, then replies in turn with an acknowledgement completing the send. Execution is then returned all the way back to the code which originally called Messenger.

ZPI is configured with a timeout length. After the message has been sent, the sending thread waits using have_incoming_reply for the timeout length. As soon as the reply is received, the sending thread sends back an acknowledgement and the function returns. If the receiving process has died, then the reply will never be received by the sending thread. The sending thread will continue to wait for the timeout length. At that time, the send call from ZPI will throw and TimeoutException. This exception must be caught in the calling stack above the send call, or it will kill the entire process. Figure 4.5 shows how a send command travels down the messaging system's layers and out to the network.

From the perspective of the simulation code, the receiving side is very similar. The simulation code calls the appropriate recv call within Messenger. That sending thread blocks until the receive has finished. However, internally the recv call in ZPI requires more effort.

TCP networking requires a client socket to know the IP address and port number of the server socket where a request will be sent. The IP addresses and port numbers used for Messenger are initialized before the start of the run. The list of IP addresses can usually be found in files

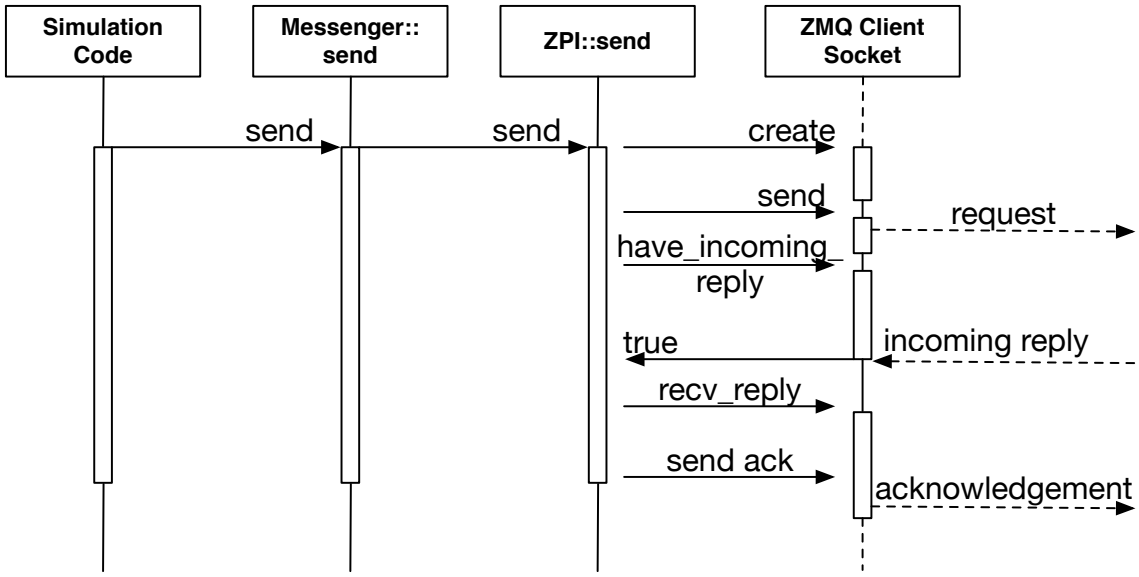


Figure 4.5 Messenger executing a send

written by the queueing system on the compute cluster.[2] Port numbers must also be known both by the sending and receiving sockets. There are only a limited number of ports which can be used and only one socket can be listen to a port at a time (that socket is said to be "bound" to the port). Furthermore, it is common in modern compute clusters to run multiple processes on a single compute node and therefore many processes may share the same network interface, and must share the same port space.

Ports are a limited resource. On large systems there are not enough ports to open a receiving port for each incoming message type. `ZPI` and `CallbackRouter` each use only one port to receive communication. This is not symmetric with the sending side of the communication where any

[2]For example, PBS (the portable batch system) writes the list of computers being used during a job to a file which is accessible on the root node of a parallel job. The file can be found using the `$PBS_NODEFILE` environment variable.

number of client sockets could be sending data simultaneously. On the receiving side, both ZPI and CallbackRouter can each process one incoming message at a time. Multiple server sockets cannot be launched on the same port to handle concurrent incoming messages. Any concurrency in receiving calls with ZPI must be managed internally.

When a ZPI recv function is called it creates a ReceiveRequest and stores it in a binary tree. The ReceiveRequest contains a pointer to the buffer where the incoming message should be copied. It also stores a timestamp of when the request was made, and copy of the expected header (source rank of the incoming message, and a tag). Finally, the ReceiveRequest contains a condition variable which is used to notify different threads as the status of the request changes. Once the request is stored, the thread which made the receive call waits on the condition variable. The receive request is now the responsibility of the thread which manages all incoming messages for ZPI.

ZPI creates a separate thread on each rank at launch which is responsible for processing all the incoming messages. This thread is called the listening thread and it has 3 jobs.

1. Receive incoming messages sent to the port number it is assigned to listen to.
2. Match those incoming messages with ZPI::recv calls.
3. Report when a ZPI::recv call has been waiting more than the configured allowable timeout length.

On startup, the listening thread creates a ZMQ Server Socket which will receive all incoming messages on the port number assigned to ZPI. All remote instantiations of ZPI in the compute cluster will communicate with a given process through that server socket. The listening thread

processes a simple loop shown in Figure 4.6. First, the thread uses `have_incoming_request` to check for incoming messages or wait a short while for one to be received from the network. If a pending message exists it is processed.

The listening thread processes an incoming message by reading the message in two parts, the header and the message body. A `PendingMessage` struct is created each time `ZPI` receives a message. The `PendingMessage` stores the header, a pointer to the message, and a timestamp of when the message was received. The `PendingMessge` is stored in a binary tree which is sorted by the header of the message so that the message can be quickly retrieved later.

Once the incoming message is processed by storing its associated `PendingMessage`, the listening thread then tries to match any `RecvRequest`s with the current list of `PendingMessages`. The listening thread traverses the tree which stores the `PendingMessage`s. It checks those headers against the tree which stores the `ReceiveRequests`. When a match is found, the data from the incoming message is copied into the buffer that was given by the user. Then the condition variable in the `ReceiveRequest` is notified that the receive has been fulfilled. The thread which created the original receive call is woken up, and the function can return, giving control back to the simulation code.

Once all matches have been made for the current collection of incoming messages and receive requests, the listening thread checks if any of the remaining pending messages or receive requests have timed out. The listening thread iterates through first the receive requests, then the pending messages, and checks their recorded timestamps against the current time. If the elapsed time is greater than the allowable time than the request or message has timed out. When a receive request times outt an exception is thrown on the thread which posted the original receive call.

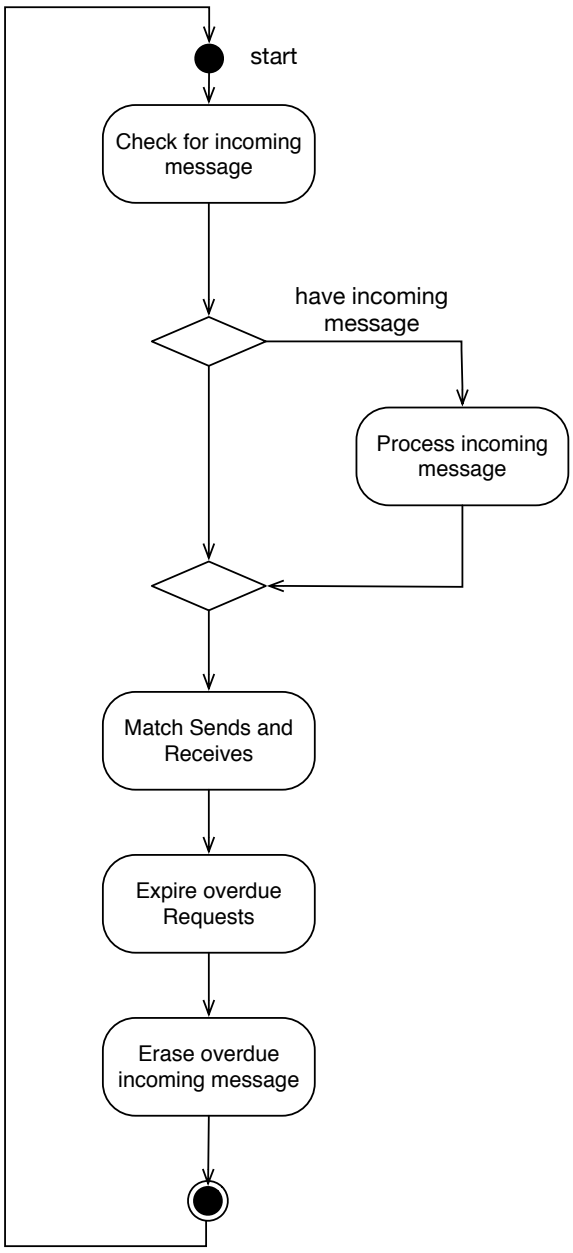


Figure 4.6 The ZPI listen thread

Timeouts on pending messages are not reported. Instead, the message is simply erased. Erasing an incoming message will cause a timeout later for the receive call which expects that incoming message.

4.4.4 Callback Router, Remote Procedure Calls

Remote Procedure Calls within `Messenger` are done through the `CallbackRouter` subsystem. Remote procedure calls are not as common in HPC applications as message passing. They can be a powerful tool for building distributed applications. Simply put, remote procedure calls are when one process calls a function on a different process. This is done by sending a message to the remote process and having some mechanics in place so that the message triggers the appropriate function call.

A callback is simply a function which stored like data and the function is then "called" at some later time. The most common form of callbacks are done through function pointers. C++ has a larger set of approaches for storing callbacks. Most notably C++ supports `std::function` which can generically store any callback, whether it is a lambda, functor, bound member function, or the traditional function pointer.

The `CallbackRouter` subsystem is simpler than its sister package `ZPI`. However, like `ZPI`, at the heart of `CallbackRouter` is a listening thread efficiently polling a `ZMQ Server Socket`. `CallbackRouter` is how the RPC `Channel` objects are supported. Each `Channel` has a unique number, called a channel number. By assigning a unique channel number, an arbitrary number functions can be registered simultaneously. The registered functions are saved in a binary tree which

is sorted by the associated channel number. Each remote procedure call send from a `Channel`'s `send` command is prepended with the channel number. This allows for `Messenger` to quickly find the associated function for a particular message. Every `Channel` object cleans up when it goes out of scope. The registered function is automatically removed from `Messenger` and the channel number can be used again for a different purpose.

`Messenger` allows for channels to be registered either with a specific known channel number, or it can dynamically find an open channel. Dynamically finding an open channel requires that every rank will register the channel. If a dynamic channel number is requested the root node finds an open channel, and then broadcasts the channel number to all other ranks. Dynamic channels offer a quality of life simplification for users of `Messenger`. However, if a large number of channels are being created and destroyed often, the additional broadcasts may become costly.

The `CallbackRouter` subsystem uses one port number per process, just like the `ZPI` subsystem. All incoming remote procedure requests come in through that port. These messages contain a message header that contains channel number that the message is being sent to. When a callback is registered it is bound to a particular channel. The callbacks are then stored in a binary tree, sorted on channel number. An overview of the steps take by `CallbackRouter`'s listen function are shown in Figure 4.7.

The listen thread in `CallbackRouter` isn't the only thread managed by the subsystem. `CallbackRouter` also launches multiple threads forming a thread pool. All the threads in the thread pool share a single queue of callbacks which need to be run. When the listen thread receives an incoming message, it finds the callback for the channel the message was sent on and queues that callback along with the received message. The thread pool works on calling the callbacks stored in

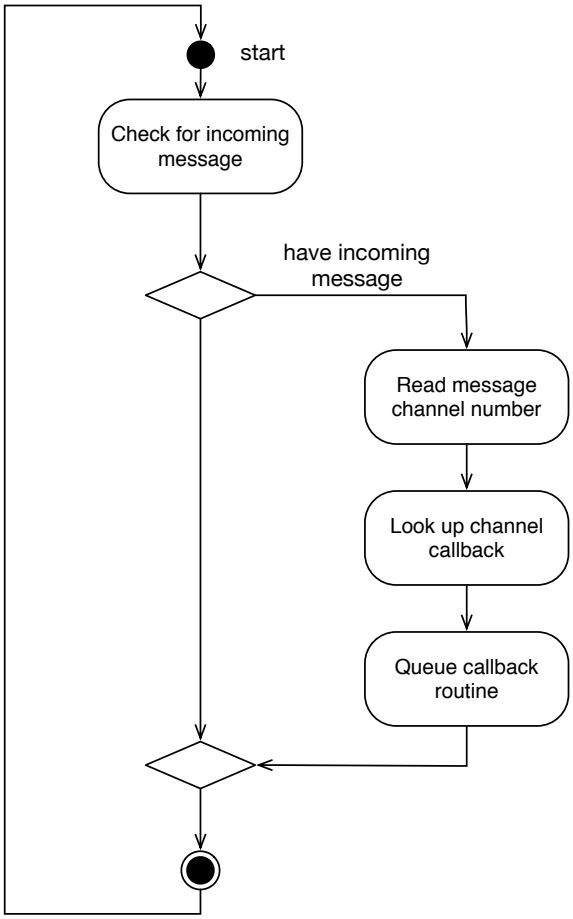


Figure 4.7 The CallbackRouter listen thread

the queue. Threads in the thread pool sleep when the queue is empty, they are woken again when work enters the queue. The thread pool automatically load balances the work of executing the callbacks because of the shared single queue. The shared queue with multiple worker threads also enables dynamic load balancing while executing the callback functions. The messaging system will not be bogged down by execution of a single long RPC since the listening thread used by `CallbackRouter` does not actually execute the callback.

4.4.5 Summary

Two communication paradigms are supported through this messaging system. The first is a fairly standard message passing system which matches explicit sends and receives. The second is a less common strategy based on remote function calls. Both systems are implemented using ZeroMQ configured over TCP. The entire messaging system is sensitive to failures through the use of communication timeouts. Furthermore, the ZeroMQ socket wrappers at the lowest level of the messaging package can simulate failures by simply refusing to return from either a send or recv function. The rest of the messaging system can then respond to these failures by reporting exceptions to the above software layers. Eventually these exceptions make it to the simulation code where a decision can be made on how to proceed.

4.5 Vault Implementation

`Vault` is the multiple in memory checkpoint package which was written to support this work. An overview of the internal structure of the package is shown in Figure 4.8. Notice that only

two components of the vault package are exposed to the simulation code. For the simulation code to use the checkpointing package, it must only need to create a `Vault` object where checkpoints can be stored. The simulation code must create checkpoint fragments which it then stores in the `Vault`. This section discusses the internal operation of how the checkpointing package works.

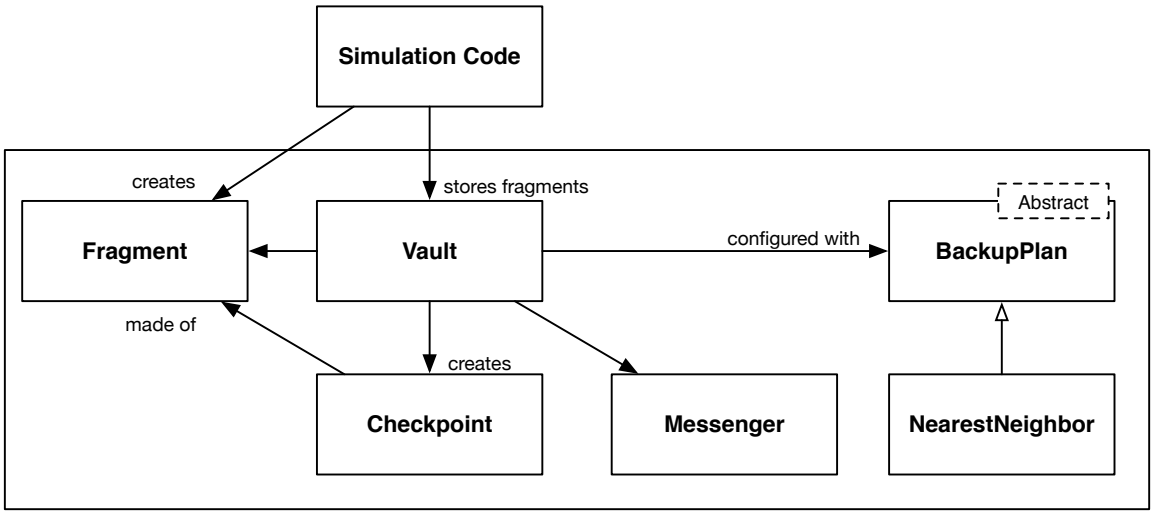


Figure 4.8 Internal layout of Vault

4.5.1 Vault, the API layer

`Vault` itself is a C++ object which must be made by the simulation code. Each rank must make a `Vault` object. `Vault` communicates with other remote instances using both message passing and RPC strategies.

The `Vault` object exposes only a small number of functions used to store and recover checkpoints. The listing of these functions is shown below. There are three steps to storing a checkpoint using `Vault`; start the checkpoint, add fragments, end storing the checkpoint. The start and end functions are synchronous across the entire compute system. When a fragment is added locally, it could be sent to any number of remote processes. `Vault` must ensure that remote instances are prepared to receive fragments when a local process tries to store them. The start function initializes the channels used to store fragments from remote processes. Once the channels are initialized, a barrier is called across the entire system. When the barrier is completed the start function returns. The application code can they begin to store checkpoint fragments. The application code stores all the fragments for the current checkpoint and then calls the end function. The end function does not return until the local instance of `Vault` receives an acknowledgement for each fragment which was redundantly stored remotely. `Vault`'s public interface is shown in Listing 4.6.

Listing 4.6 The interface of `Vault`

```
void startCheckpoint(BackupPlan *backupPlan, int checkpoint_id, int channel);
void endCheckpoint(int checkpoint_id);
void addFragment(int checkpoint_id, int fragment_id, Stream s);
Checkpoint* getCheckpoint(int id);
std::vector<int> getListOfCurrentCheckpoints() const;
void eraseCheckpoint(int checkpoint_id);
```

In between the start and end calls, the simulation code can store checkpoint fragments. The fragment identifiers across the system must be unique, and all together the fragments identifiers must be complete between 0 and the maximum fragment identifier. However, it does not matter

in which order the fragments are stored, or on what rank a fragment is stored. Furthermore, the simulation code does not need to know how many fragments will be created when it starts storing fragments. It is only when recovering from a failure that the simulation code has to report how many fragments there are in a given checkpoint.

Checkpoint fragments are stored in a `Stream`. The checkpointing package does not need to know how to parse the streams. It only communicates them, stores them, and retrieves them on behalf of the simulation code. It is up to the simulation code to know how to create and parse them.

`Vault` registers an RPC channel for each checkpoint. It will use this channel to send fragments that are to be stored redundantly within the system. `Vault` creates a `Checkpoint` object for each checkpoint. It stores references to these checkpoints in a binary tree which is sorted on the checkpoint identifier. In turn each checkpoint stores a binary tree of fragments sorted by fragment identifier. When vault receives a fragment through the add fragment call it, stores the fragment in the appropriate location, overwriting an old fragment if one existed with the same checkpoint and fragment identifier.

The user application may choose to have one fragment created for each rank, or if the domain is over-decomposed, there may be more. It is common in MPI applications to have each rank work on one partition of the overall problem, so one fragment per rank would seem natural. However, this approach alone may not be ideal. Consider a problem running on n computational units. Likewise the problem is partitioned equally onto n work units. Now a failure occurs. The number of computational units will drop to m. Typically there will be a load imbalance trying to map n equally sized work units onto only m compute units. A few strategies exist to combat this

problem, which will be discussed later. Generally it is better if the user application can dynamically repartition to load balance.

4.5.2 Backup Plans

Each checkpoint must be configured with a `BackupPlan` which tells `Vault` how to redundantly store checkpoints. `BackupPlan` is an abstract interface with only one function `getKeeperOfFragment`. This function takes in a fragment identifier and returns a list of which ranks should store the fragment. `Vault` then sends any incoming fragments to the ranks it is told by the plan. Figure 4.8 shows one example plan: the `NearestNeighbor` plan. This plan stores fragments on the current rank, as well as the rank to the left (rank - 1) and the rank to the right (rank + 1). More complex backup plans could also be implemented without changing any of the underlying code in the `Vault`.

4.6 Recovering from failures

The work of detecting unresponsive ranks and redundantly storing checkpoints is all done in preparation so that the system can recover when a failure occurs. This section walks through the process of recovering beginning with the messaging system throwing an exception signaling a failure.

Once an exception has been raised, it will travel up the call stack until caught by an appropriate level in the simulation code which is prepared to handle the recovery. The rank which

detected the failure will stop all incoming or outgoing communication. This will cause a chain reaction of failures until eventually the entire remaining system will report failures.

When a failure occurs, there are two steps to recovering. The first is recovering the messaging system to a useable state. This means returning the messaging system to a state where the rank and size abstractions are again valid. This probably means that the messaging system needs to relabel certain processes to makes sure that the ranks are complete on the interval [0,size). Many rank assignments will change after a recovery; so, if the simulation code maintains state based on rank, that state will no longer be valid. In addition to *rank* and *size*, `Channels` which are still registered also need to be remapped onto the new rank and size to that they can continue to function. The second phase of recovery is to actually recover the simulation code to a state where it can continue executing.

When the simulation code catches the exception, it can then perform any action it needs to perform in order to clean up the state of its application and prepare for rolling back to whatever state the application last stored a checkpoint. Then the simulation code initiates a recovery by calling `Messenger::recover`.

The recovery phase for messenger begins by opening a new server socket on each rank. This server socket uses a new port number which is distinct and is only used for recovery. The root process starts listening for incoming messages from any still remaining non-root processes. The non-root processes connect to the root process and send a message notifying root that the process is still very much alive, and will be available to continue working. The root process keeps this

window open for a configurable amount of time.[3] Any rank which fails to register during this window is considered dead. Dead ranks are not assigned a rank number for the next phase of work. Once the registration window's time has elapsed, the root process then assigns new rank numbers to the remaining processes. Finally, the root rank broadcasts the list of old rank numbers, new rank numbers, IP addresses and port numbers to all the remaining processes. Once a process has this information it can begin sending messages on behalf of the simulation code.

Astute readers will have noticed that there is no redundancy built into the system for a failure on the root rank. In that case, the entire system cannot recover and the application will crash. Luckily, as shown in Chapter 2 on sufficiently large systems like the ones where fault tolerance is an issue, it is simultaneously statistically likely that at least one node will fail while being very unlikely that a specific node (say the root node) will fail.[4] A fully redundant system could be implemented with a succession plan that accounts for the root rank failing. If a non-root node attempts to communicate with the root node during recovery and the communication times out then the node may assume that the root has failed. The node may then attempt to resend the message to rank 1. If rank 1 detects a failure with the root node (rank 0) it could assume the responsibility of root node and open the communication window. If rank 1 also proves unresponsive, the system could fall back to rank 2, then 3, until a surviving rank is found.

Once the messaging system has recovered, the simulation code can begin attempting to recover. Each process can query `Vault` to determine if a copy of each checkpoint fragment has survived the failure. `Vault` checks if a recovery is possible by gathering all the fragment

[3] [With current testing on approximately 10 nodes, 30 seconds seems to be sufficient.]
[4] See 2 for a full discussion.

identification numbers onto the root process and checking if they span the space expected by the checkpoint. The checkpoint is considered complete if at least one copy of each fragment exists somewhere on the recovered system.

`Vault` reports that a complete copy of a checkpoint exists if there is enough data to recover. Each rank of the simulation code can then query `Vault` to see which fragments that process is responsible for retrieving from `Vault` and restarting. `Vault` only assigns any given fragment to a single rank and tries to equally assign fragments across the remaining ranks. At this point the simulation code can request fragments by their identifier. It is up the the simulation code to know how to use the fragments to recover the system. The simulation code may decide to rebalance since the compute hardware would have changed. If the problem was sufficiently over-decomposed the simulation code may be satisfied with the basic load balancing performed by `Vault`.

4.7 Summary

This chapter described a multiple in memory checkpointing framework. The framework can store user defined fragments which are pieces of a complete system checkpoint. The checkpointing framework redundantly stores these fragments in main memory on different processes throughout the compute system. The system is configurable, allowing the user to decide what level of redundancy to use.

Storing and retrieving checkpoints are only two components to increasing resiliency through multiple in memory checkpointing. Another critical component is detecting when compute hardware has failed. Two messaging paradigms were developed to accomplish the task of detecting

failures. One is based on standard message passing. The second is based on the less common strategy of remote procedure calls. Both strategies were implemented using ZeroMQ configured over TCP. ZeroMQ allows for detecting failures using message timeouts. Though MPI is more common in high performance compute clusters, ZeroMQ was chosen as the underlying messaging technology because ZeroMQ easily supports timeouts. MPI has a lower latency as discussed in the next chapter.

CHAPTER 5

RESULTS

5.1 Load Balancing for Size

Dynamic load balancing of the offbody mesh was implemented using space filling curves. There are generally two reasons to dynamically load balancing for distributed computer applications. The primary reason to load balance is to evenly distribute access to some limited resource. Often in HPC applications load balancing is used to lower total wall clock time by minimizing the total amount of time where compute resources are idle. However, for this work load balancing is done not to minimize wasted cpu time. Instead, load balancing is a means to generating very large grids with poor initial partitions.

Static load balancing in mesh generation often gives poor initial partitions for storing the mesh which is to be generated. For any but trivial cases, it is very challenging to know precisely how to partition the mesh domain so that the each partition will end up with an equal portion of the mesh.[1] The challenge is especially acute for mesh generation techniques which make use of refinement. An initial partitioning may be suitable for an initial mesh. But if the mesh is refined in

[1] In fact, a priori domain partitioning presents a chicken and egg problem. If you knew precisely how much mesh was needed in each region of the domain, part of the grid generation problem has already been solved.

a non-uniform manner, the partitioning will become imbalanced. If the imbalance is large enough it could cause a partition to run out of memory while there is still memory available on other nodes.

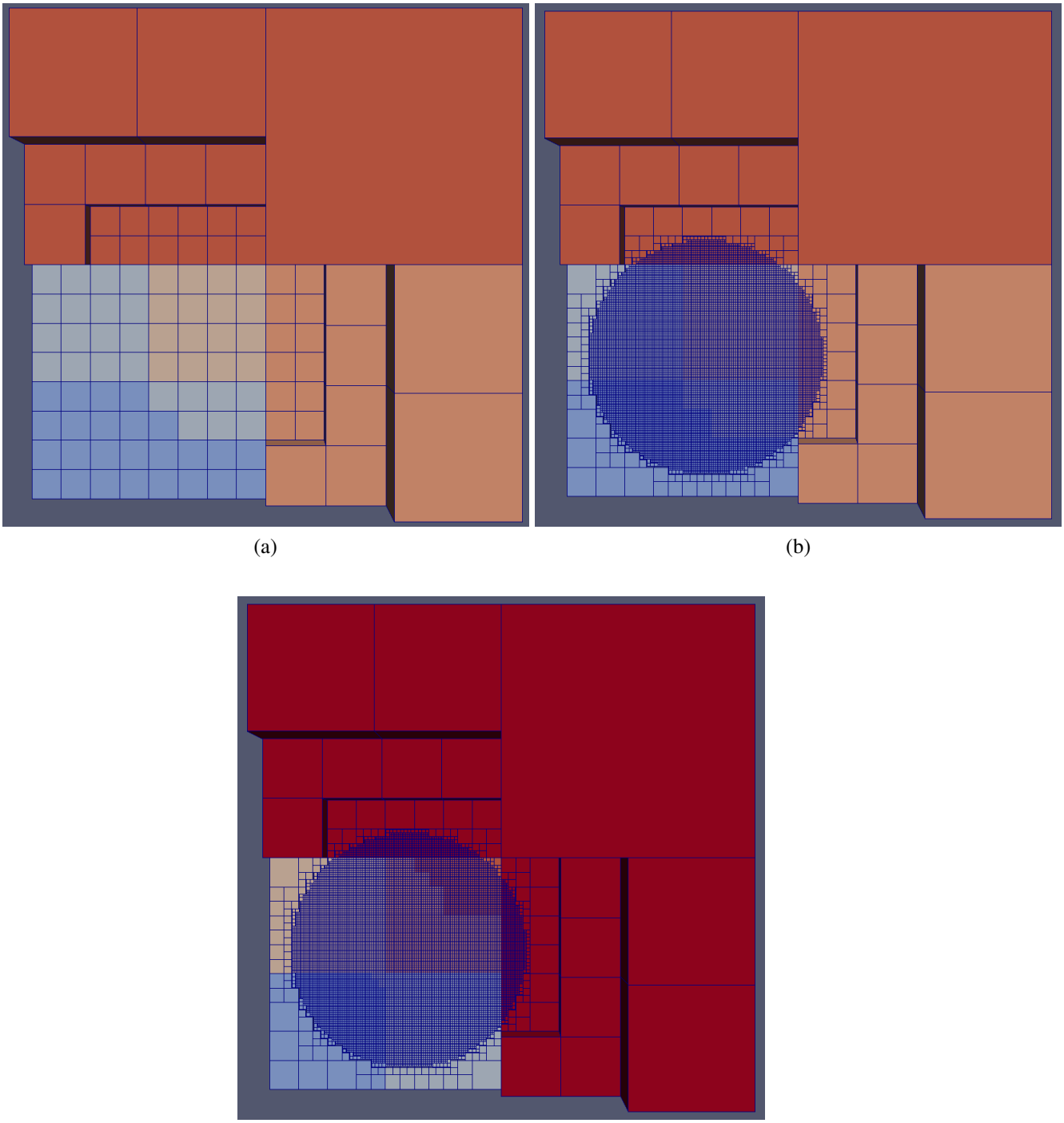


Figure 5.1 Mesh rebalancing to minimize peak memory (a) Initial mesh generated and partition (b) Additional mesh refinement (c) A rebalanced mesh

5.2 Communication Performance

Performance experiments were conducted to compare bandwidth performance of the communication layer `Messenger`. A message was sent from one node (named the client node) to another node in the network (the server node). Upon receipt of the message the server node simply returned the message back. The client node recorded the time it took to send the message and to receive the reply. The experiment (also known as the ping pong test) was conducted over a range of message sizes from 8 bytes to 1 GB. For each message size 100 cycles of send - receive were conducted. The median elapsed times were recorded.

The experiments were conducted on the NASA Langley K3 cluster which uses an Infiniband interconnect. The experiment was repeated for three different communication software libraries. The first is MPICH 3.0, an open source implementation of the MPI standard. MPICH was configured to run over the K3 cluster's infiniband interconnect using RDMA. The MPI tests were conducted using pairs of matching MPI_Send and MPI_Recv. The second software library was ZeroMQ version 4.0.4. ZeroMQ was configured to run over the Infiniband interconnect using TCP since ZeroMQ does not support the RDMA protocol. The ZeroMQ tests were conducted using a client socket on one node, and a server socket on the other. The final software package tested was the `Messenger` software developed for this research. `Messenger` also uses ZeroMQ client and server sockets configured over TCP. `Messenger` also maintains both MPI-like abstractions, and manages recovering those abstractions in the event of failure. Maintaining these abstractions incurs overhead. Quantifying this overhead was the primary goal of the communication performance experiments.

Figure 5.2 shows the timing results for all three performance experiments. MPI out performed both native ZeroMQ and `Messenger` across all message sizes. Figure 5.3 shows the slowdown factor for both native ZeroMQ and `Messenger` with respect to MPI. Native ZeroMQ starts out approximately 8 times slower than MPI but the gap falls as the message size increases. The performance difference between `Messenger` and MPI is much larger. The gap with `Messenger` being about 100 times slower than MPI, the gap falls as message size increases. At a message size of 1 GB `Messenger` is approximately 10 times slower than MPI. This indicates that latency may be impacting bandwidth at smaller message sizes.

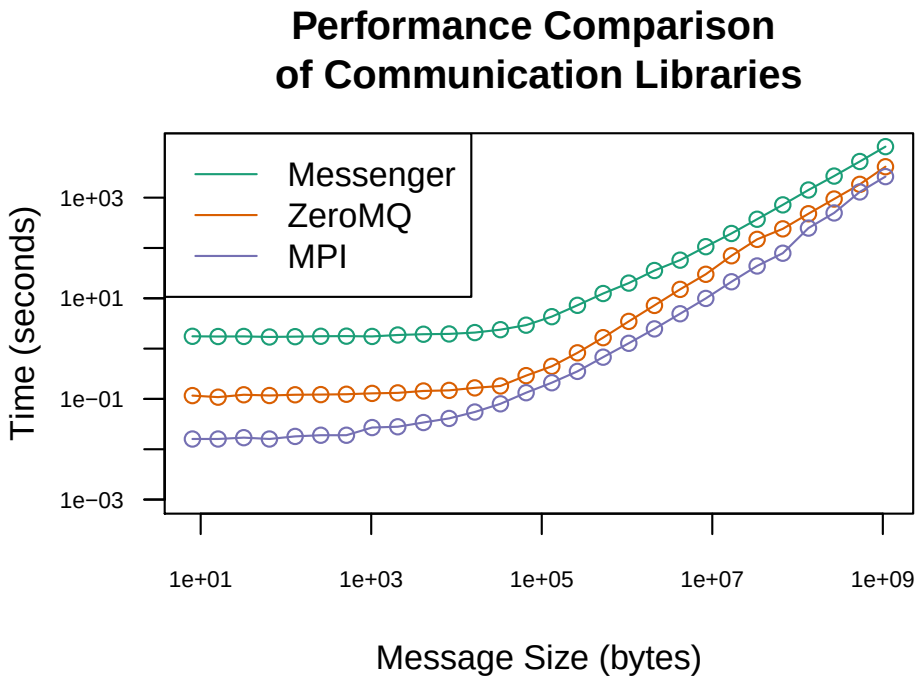


Figure 5.2 Performance timing of communication libraries

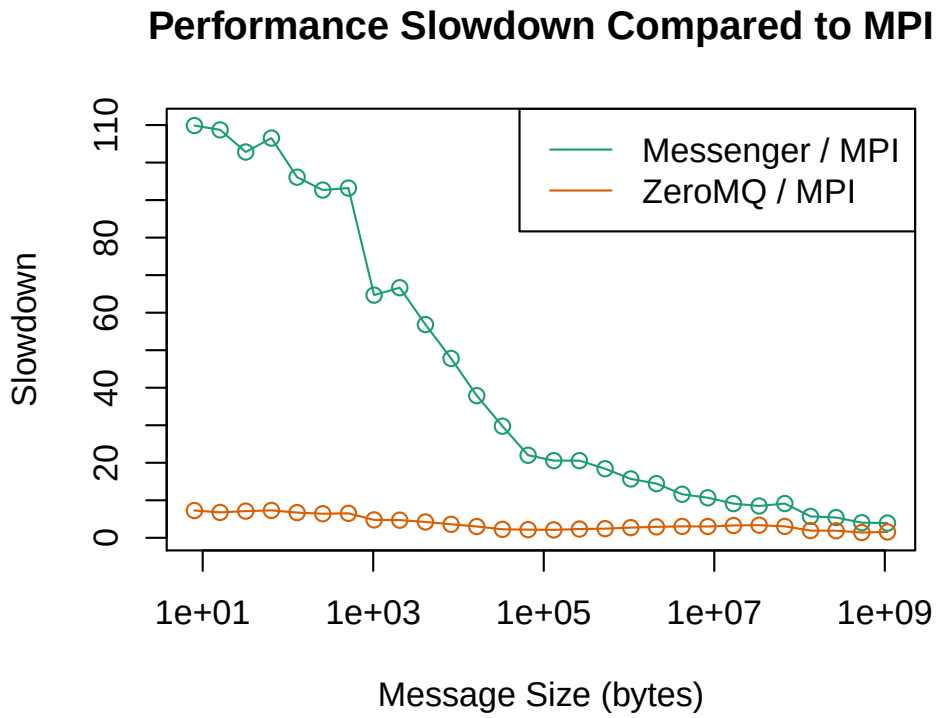


Figure 5.3 Performance slowdown of Messenger compared to MPI

5.3 In-Memory vs. In-Disk Checkpoint Performance

Performance of in-disk and in-memory checkpointing were explored. A series of experiments were conducted using 8 compute nodes. A checkpoint fragment was created on each node and then checkpointed. The size of the fragment varied from 1MB to 1GB. Timing data to store the checkpoint was recorded.

`Messenger` and `Vault`, described in Chapter 4 were used to perform the in-memory checkpointing. The double in-memory backup strategy `RoundRobinBackup` was used; therefore,

one copy of each fragment was stored on the host node and one was communicated to a neighboring node.

The in-disk checkpointing experiments were using `Messenger`'s RPC functionality. Each node sent their fragment through a registered RPC on the root node. The root node then wrote each fragment to a binary checkpoint file. The binary checkpoint file was written to permanent storage through `fwrite()`. File access of the cluster nodes is done through the Luster file system configured over Infiniband. [96]

Timing results for both in-memory and in-disk checkpointing is shown in Figures 5.4 and 5.5. Across all fragment sizes the in-memory checkpointing out performed in-disk checkpointing. The performance differences between in-memory and in-disk checkpointing are small for fragment sizes below 16 MB. However, at 40 MB and above in-memory checkpointing performs much better than in-disk; in-memory exhibits more than a 5× speedup over in-disk at 50 MB before settling down to approximately 4.5× for checkpoint sizes between 50 MB and 1 GB.

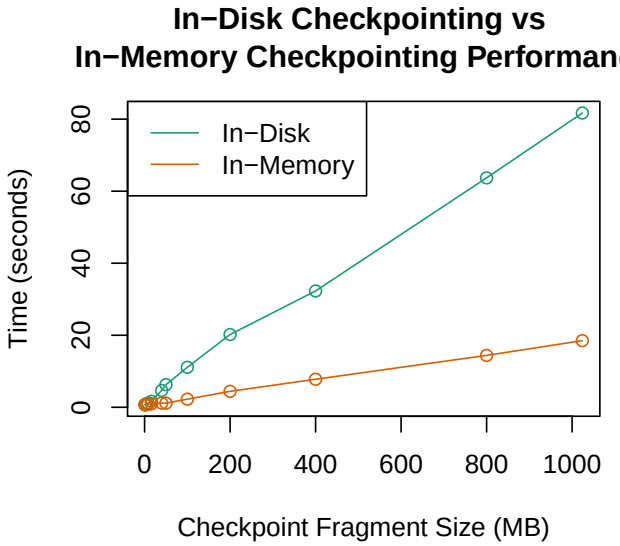


Figure 5.4 Performance of in-disk and in-memory checkpointing

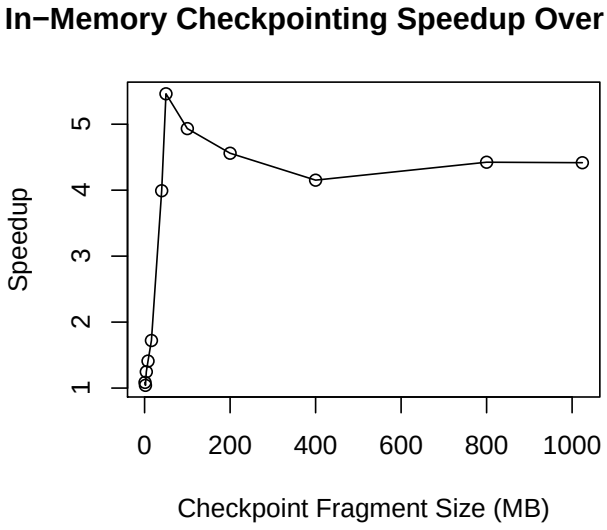


Figure 5.5 In-memory checkpointing speedup

5.4 Global Refinement Top Down

Weak scaling experiments were conducted to determine performance of the Top Down portion of the algorithm. The mesh was globally refined within a unit cube. Experiments were performed using node counts of 1, 8, and 64 which corresponds to 16, 128, and 1024 cores respectively. On one node, using 16 cores, the unit cube was refined to a tree depth of 9 yielding a mesh spacing of $1.2x10^{-3}$ and 134 million total cells. Each successive experiment then increased the depth by 1, increasing the total cell count by a factor of 8 each time. The final mesh contained 8.6 billion elements.

The experiments were run on NASA Langley Research Center's K3 cluster. Each node in the cluster consists of two Intel™ E5-2670 Sandybridge 8 core 2.6 GHz processors. Each node also maintains 32 GB of main memory. The goal of this work was not a parallel implementation for flops performance. Instead the mesh generation process was made parallel so that very large meshes could be generated. Because of this the 32 GB per node on the K3 cluster is the primary resource which is desirable. The K3 cluster's interconnect uses the 4X FDR Infiniband which supports up to 56 Gbit/sec throughput. However, since ZeroMQ is configured to run over TCP and will not benefit from the Infiniband's support of Remote Direct Memory Access (RDMA).

Table 5.1 shows the running time, total cell count and parallel efficiency for the three experiments. The single node case generated cells at a rate of approximately 400 thousand cells per second, per core. That rate drops to approximately 340 thousand cells per second per core across 1024 cores. The rate of generation is consistent with other efforts from the literature. The parallel mesh generator p4est has been shown to generate at a rate of 110 thousand cells per second per

core across 220 thousand cores. [26]. Lintermann et al. demonstrated a rate of 12 thousand cells per second per core across 65 thousand cores. [27]

Table 5.1 Parallel efficiency with checkpointing disabled

core count	total cell count	running time	efficiency	total memory used
16	0.13×10^9	20.5 s	1.0	19 GB
128	1.07×10^9	23.2 s	0.88	154 GB
1024	8.60×10^9	25.1 s	0.80	1.25 TB

Weak scaling experiments were repeated with checkpointing enabled to measure the effects of checkpointing. Each case began with the mesh refined down to a depth of 4 during the setup phase. Then one checkpoint was stored after each depth in parallel. A total of 5 checkpoints were created on the single node case, 6 checkpoints on the 8 node case, and 7 checkpoints on the 64 node case. The backup strategy used was a double in memory strategy where each checkpoint fragment was stored locally on the owning host node, and one copy was stored on a single adjacent node.

Figure 5.6 compares weak scaling with and without checkpointing enabled. Table 5.4 shows the running time efficiency and total memory used with checkpoints enabled. The scaling efficiency drop from 0.80 to 0.48 across 1024 cores indicating that creating and storing checkpoints is less scalable than the general top down algorithm. The total memory used with checkpoints enabled increased by approximately 6% over the baseline case with no checkpointing.

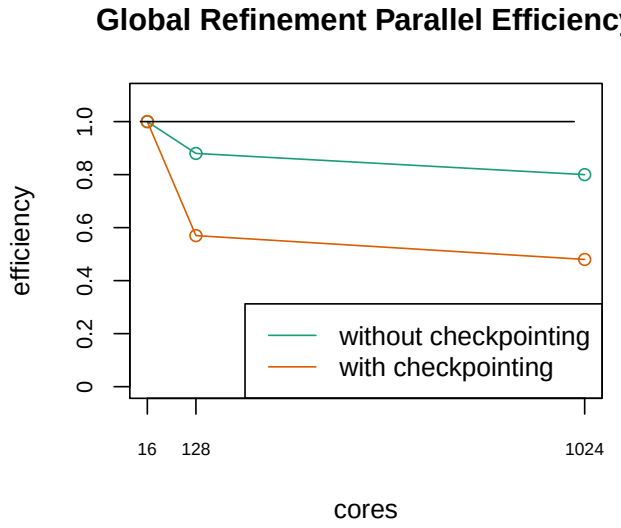


Figure 5.6 Parallel efficiency of top-down grid generation

The small memory overhead is due to the compact form which the offbody mesh is stored. An offbody checkpoint fragment. The size of an offbody checkpoint fragment is given by:

$$\text{checkpoint size} = 48 \text{ bytes} + 16 \text{ bytes} * \text{number of cells}$$

Each offbody checkpoint fragment stores the root domain of the mesh in double precision (48 bytes), then for each leaf node in the tree the fragment contains the morton Id (8 bytes) hte depth of the leaf in the tree (8 bytes) and a flag marking if the leaf cell is in, out, or crossing the geometry (4 bytes).

Table 5.2 Parallel efficiency with checkpointing enabled

core count	total cell count	running time	efficiency	total memory used
16	0.13×10^9	29.5 s	1.0	19.6 GB
128	1.07×10^9	51.7 s	0.57	162 GB
1024	8.60×10^9	61.1 s	0.48	1.32 TB

The working memory of a partition stores an extent box in double precision for each node in the full tree as well as the in, out, or crossing flag. A very compact form of the offbody mesh can be stored redundantly for resilience while the working memory is uncompressed and high performing.

5.5 Full Case Scalability

Scaling experiments were conducted to determine the performance of the entire mesh generation process including top-down, bottom-up, as well as, generation and projection of the nearbody mesh. A body fitting mesh was generated around a unit sphere with the total mesh domain being $[-10, 10]^3$. Scaling experiments were run on the K3 cluster with 16, 128, and 1024 cores. The mesh was initially refined to a spacing of 0.04 during setup phase. Then the mesh was refined to spacings of 2.4×10^{-3}, 1.2×10^{-3}, and 6.1×10^{-4}. The initial scaling experiments were conducted with checkpointing disabled to establish a baseline understanding of the performance of the implementation.

Cells were computed at a rate of 1,250 cells per second per core for the 16 core case. 870 cells per second per core on 128 cores, and 140 cells per second per core across 1024 cores.

Table 5.3 Parallel efficiency with checkpointing disabled

core count	total cell count	running time	efficiency	total memory used
16	19.4×10^6	16m37s	1.0	8.09 GB
128	76.8×10^6	11m29s	.70	32.8 GB
1024	305×10^6	35m50 s	.11	140 GB

Weak scaling experiments were repeated with checkpointing enabled. All the parameters were the same from the previous case. The only difference was that this time checkpoints were created. The `RoundRobinBackup` was used which is a double in-memory checkpointing system. One copy of each fragment was saved on the host node, and one copy was sent to node with the next highest rank.

The base case of 16 cores was run only on a single node. `RoundRobinBackup` directs checkpoints of rank (size-1) to be sent to rank 0. Since there is only one rank, the send copy of each checkpoint fragment is being sent back to the host node. There is no resiliency being added in this case. If any component of node 0 dies the job is forfeit. However, it is still a valuable case to explore performance. The results of the study is shown in 5.4.

Overall there was only a small impact on run time by enabling checkpointing. The single node, 16 core, baseline case showed no change in the run time and still generated 1,250 cells per second per core. At 128 cores the time increased slightly and the overall cell generation rate fell from 870 cells per second per core to 840. The run time was most impacted across 1024 cores,

Table 5.4 Parallel efficiency with checkpointing enabled

core count	total cell count	running time	efficiency	total memory used
16	19.4×10^6	16m37s	1.0	20.1 GB
128	76.8×10^6	11m51s	.70	144 GB
1024	305×10^6	37m24 s	.11	544 GB

which increased by 94 seconds. The cell generation rate fell from 140 cells per second per core to 133.

While the run time was not greatly impacted by creating checkpoints, the memory consumption increased dramatically and shown by Figure **??**. In section 5.4 only the offbody Cartesian mesh representation was used. The mesh in this form has a compact representation using Morton Ids and this compact representation is used to create and store checkpoint fragments.

However, the current case includes a nearbody mesh elements which are not Cartesian aligned and are not compactly stored for checkpoints. Furthermore, the nearbody mesh stores much more information than is used in the offbody, Cartesian, mesh. A checkpoints fragment for the nearbody mesh also stores node status flags, and topological adjacencies such as node to neighboring cell, and node to neighboring surface cell.

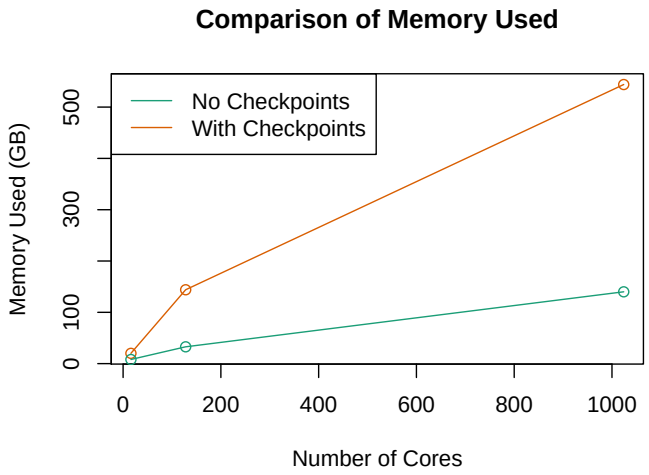


Figure 5.7 Memory requirement with and without checkpointing

5.5.1 Hurricane Strong Scaling Study

Strong scaling experiments were conducted on a body conforming case. A 14.6 million element mesh was generated around an STL surface of the Katrina hurricane. The mesh was generated on the K2 cluster at NASA Langley Research Center. Each node in the cluster consists of two Intel™X5675 Westermere 6 core 3.07 GHz processors. Each node contains 24GB of ram. The mesh was generated on a single node of 12 cores as the baseline case. The number of nodes was then doubled successively up to 32 nodes, for a total of 384 cores. The Figure 5.8 shows the partitioned nearbody mesh while running on 32 nodes and Figure 5.9 shows a close up view of the body conforming mesh near the eye of the hurricane.

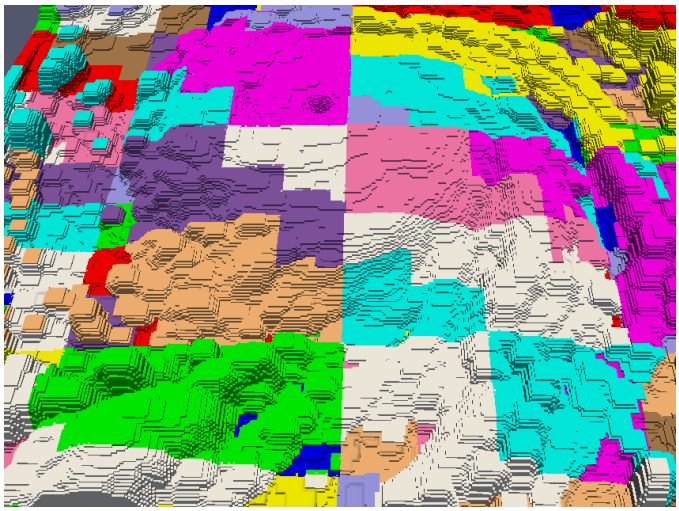


Figure 5.8 The partitioned nearbody of a hurricane

The baseline case on a single K2 node generated the 14.6 million element grid in 38 minutes 32 seconds. On 32 nodes (384 cores) the mesh was generated in 3 minutes 12 seconds. The mesh generator created 13 checkpoints during each case. Figure 5.10 shows speedup from 12 to 384 cores. At 384 cores mesh was generated 12 times faster than using only a single node of 12 cores.

Additional experiments were performed using 32 nodes to measure the time and memory cost of creating checkpoints and recovering from failures. First, the mesh was generated while all checkpointing was disabled total memory used was recorded along with the total run length. Then, checkpointing was reactivated and again the mesh was generated while measuring memory used and run length. Finally, the mesh was generated with checkpointing enabled and a single point failure on node 1 was triggered during a random communication. The mesh generator detected

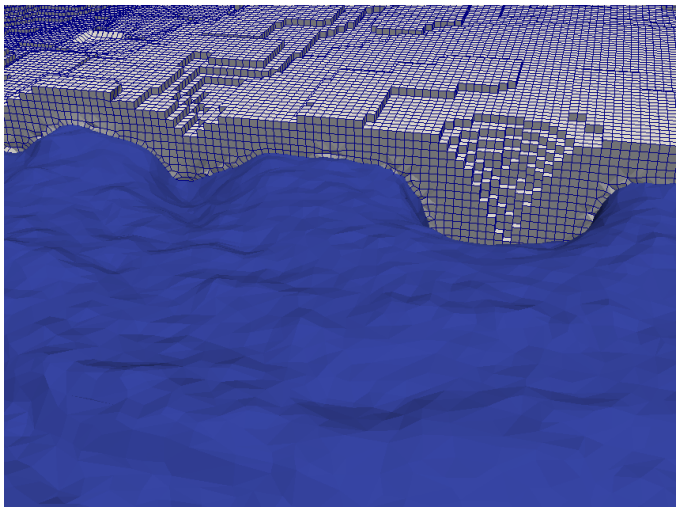


Figure 5.9 Close up of the mesh in the eye of the hurricane

the failure, recovered the messaging system, then recovered the mesh, and continued generation. Again, run length and total memory was recorded. The run lengths and total memory used were recorded in table 5.5 and a scaling plot is shown in Figure 5.10.

Table 5.5 Strong scaling experiments

	Running Time	Total Memory Used
Checkpointing Disabled	184s	4.3 GB
Checkpointing Enabled	192s	11.8 GB
Single Node Failure	225s	11.4GB

Similar to the test case described in 5.5, the memory requirement increases by approximately a factor of 3. Just like in the last case, the memory requirement is dominated by the unstructured mesh in the nearbody region of the domain. The nearbody region cannot be compressed using the Morton curve technique which is used in the offbody. The total memory required to complete

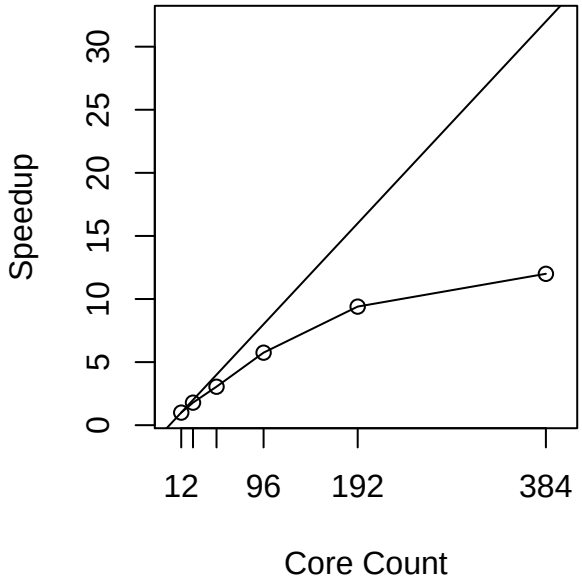


Figure 5.10 Strong scaling experiments for a hurricane mesh

the run with a node failure is 400 MB lower than the case without memory. There is data stored once per node redundantly (such as the geometry), this data does not migrate during a failure. The majority of the memory footprint is consumed by the mesh and checkpoint data. This data is migrated during recovery of a failure. Detecting and recovering from the failure only increased the run length by 33 seconds.

5.6 Recovery

The fault tolerant framework redundantly stores backups as described by the configured `BackupPlan`. The `BackupPlan` decides how many copies of a fragment are created, and where those fragments are stored. The simplest `BackupPlan` is a double in-memory backup strategy which is implemented by `RoundRobinBackup`. `RoundRobinBackup` stores each fragment twice, once on the rank which created the fragment, and once on an adjacent rank. Double in-memory strategies store enough copies of the data to recover from any single point failure.

However, rank 0 is a critical centralized location from within the compute network where recovery is managed. When any rank detects a failure it attempts to recover by coordinating with rank 0. If rank 0 is the rank which suffers a failure there is no way for the rest of the compute nodes to coordinate a recovery. This is a limitation of the implementation of the fault tolerant system. It would be possible to extend the recovery procedure to include a succession plan. That way if rank 0 failed another node could coordinate the recovery.

Resilience experiments were run to demonstrate the effectiveness of the fault tolerant framework. Eight nodes were used to generate a mesh using the technique described in chapter 3. Verification files were written for each offbody and nearbody partition. The offbody regression files contained the sorted Morton Ids of each leaf voxel in the partition. The nearbody files contained the sorted list of points of each partition.

Initially the mesh was generated with failures disabled and the verification files which were written were saved. The code was then altered so that it could be failed on command for specific ranks. If a rank was chosen to fail it would do so by simply calling `exit(0)`. A failing rank would not communicate in any way with the rest of the compute nodes. The rest of the system was

unaware that the process would die. The remaining nodes were allowed to progress and complete generating the mesh. If the process completed, verification files were again written and these new files were compared with the verification files from the run where no failures occurred. A run was considered to have successfully completed if there was no difference in the two sets of verification files.

Table 5.6 shows a matrix of runs which were completed and the status of the run. A ✓at entry (i, j) demonstrates that nodes i and j were simultaneously failed and the run recovered and completed successfully. A × denotes that the run was not successful.

Table 5.6 Double in-memory recovery matrix

	0	1	2	3	4	5	6	7
0	×	×	×	×	×	×	×	×
1		✓	×	✓	✓	✓	✓	✓
2			✓	×	✓	✓	✓	✓
3				✓	×	✓	✓	✓
4					✓	×	✓	✓
5						✓	×	✓
6							✓	×
7								✓

There are two types of failures from which the framework was unable to recover. First, note that all cases where node 0 failed were not recovered. This is an expected outcome since there is no redundancy in the framework for the task of coordinating recovery. The second type of failure which was not recovered is when two adjacent nodes failed simultaneously. In these cases

the fragments which were generated on node n and stored on nodes n and $n + 1$ were lost when both nodes went down. There was not enough information to recover from the failure, and the case exited. In all other cases where only a single node failed were successfully recovered. The double backup allows for recovering from these types of failures. As discussed in Chapter 1, single node failures are critically important since they are the most common errors.

5.7 Supersonic Sphere

For demonstration an inviscid mesh was generated. The mesh contained 1.3 million nodes. The sphere has unit diameter. The sphere is traveling at Mach 1.53. Fun3D was used to perform this inviscid calculation. [52] A density plot is shown in Figure **??** with a corresponding mesh in Figure 5.13. The initially mixed element mesh contained quadrilateral faces which were non-flat. The smoothing technique described in chapter 3 does not attempt to ensure flat faces. The mesh was converted to only have tetrehedra using Fun3D and following the work of Dompierre et al. [97]

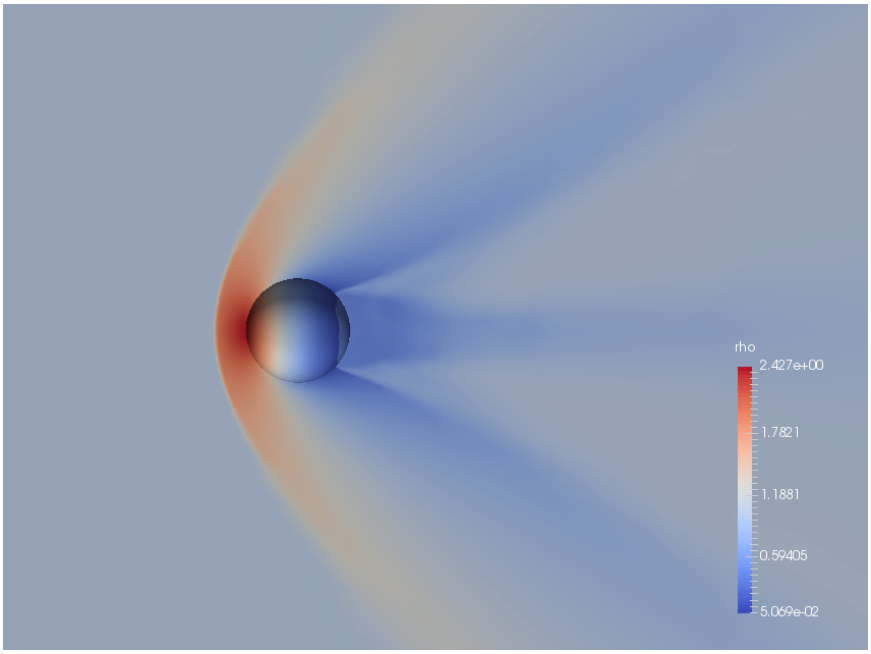


Figure 5.11 Supersonic sphere solution obtained from Fun3D

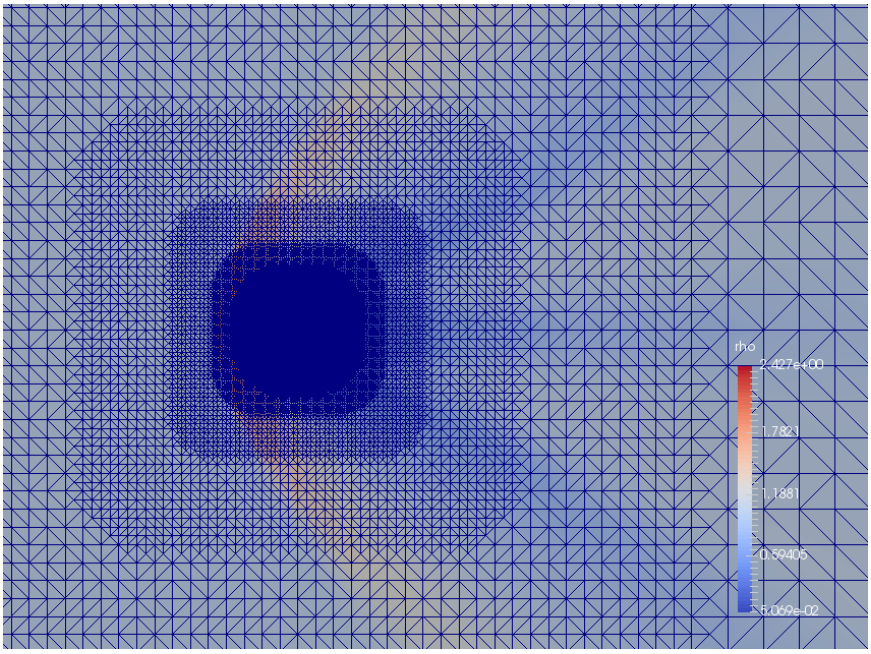


Figure 5.12 Tetrahedral elements used with Fun3D

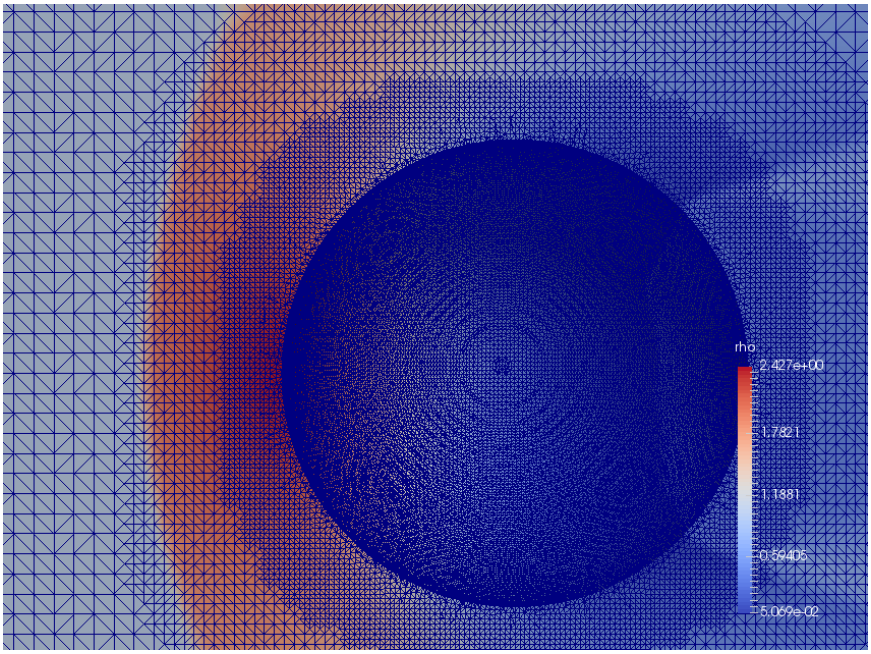


Figure 5.13 Sphere grid near the geometry

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The fields of computational science progress on two fronts: numerical algorithms, and computational hardware. However, progress on these fronts is not independent. Algorithms must be developed through an understanding of the strengths and weaknesses of the computational hardware on which the algorithm will be run. High fidelity Computational Fluid Dynamics (CFD) runs on large scale compute clusters. As the compute clusters grow in scale, and in complexity, the failure rates will also grow. Precautions must be taken in our CFD algorithms to ensure efficient utilization of large scale HPC clusters.

Rapid turn around for high fidelity CFD analysis may require a parallel software ecosystem which can start with geometry and end with engineering output. This means that mesh generation, flow solution, mesh adaptation, and post processing may all have to exist in parallel and the entire ecosystem will need to be fault tolerant. This work demonstrated one way the piece of this ecosystem may operate.

6.1 Contributions

This work made novel contributions to the young field of parallel mesh generation. This work developed and demonstrated a novel communication system which was developed using

ZeroMQ but that maintained MPI abstractions of rank and size. Additionally, this work extended the communication system to include Remote Procedure Calls. The communication system was used to implement a multiple in-memory checkpoint/recovery system. Finally, and critically, the communication system, and the checkpoint/recovery system were integrated into a novel parallel mesh generation application which is capable of running on a compute cluster suffering failures.

6.2 Summary

This work was motivated in Chapter 2 using a relatively simple model of failures for large scale HPC clusters. The model was tuned using published data for real world supercomputers of large scales. The model demonstrated that computational jobs running on a small number of well maintained nodes will not suffer many failures. However, as the complexity of the engineering problem grows, and more compute nodes are utilized, the failure rates can grow dramatically. The CFD Vision 2030 report has identified some grand challenge problems that may require 10-100 billion control volumes. These problems include: full flight aircraft in maneuvering flight, aircraft with full engine simulation, or a space vehicle launch sequence. [21] Computer systems running these simulations may require tens of thousands of nodes for weeks at a time. For interesting, large computational problems, system failures cannot be ignored and must be treated as normal operation or the run time of these computational jobs will be dwarfed by checkpoint and recovery times.

Chapter 3 described a parallel grid generation technique which used octrees to create Cartesian meshes. Regions of these meshes can be projected down onto a geometry surface creating a body fitting mesh. The technique was designed to run on standard compute cluster

hardware. The technique is can be dynamically load balanced so that very large meshes can be generated without the need for specialized compute nodes with large amounts of main memory.

The communication and in-memory checkpoint restart system was described in Chapter 4. The messaging system was designed to support two communication strategies: message passing, and remote procedure calls. The messaging system detects failures in the system by reporting failures when a message times out. These failures can trigger the messaging system to recover. Chapter 4 also described the in-memory checkpoint and restart technique. The exact checkpointing strategy is extendable and is capable of supporting an arbitrary number of in-memory checkpoints including both double and triple in-memory copies.

Finally, a series of experiments were run on the fault tolerant mesh generation application and including the messaging and checkpoint / recovery modules. Experiments on the in-memory checkpointing module demonstrated faster times to store a checkpoint fragments using double in-memory checkpointing than to write those checkpoints into permanent storage. Performance experiments on the communication module demonstrated much slower bandwidth than either native ZeroMQ point to point messages, or messages sent with MPI.

While generating body conforming meshes the parallel efficiency of the application fell to 11% over 1024 cores. The addition of in-memory checkpointing did not affect the application's performance while generating body conforming meshes. However, the in-memory checkpointing system did negatively affect the parallel efficiency of generating purely Cartesian meshes.

The next section discusses future work including some key proposed changes to the checkpointing system which may increase performance. Even so, ideal raw performance should not be the only metric of success when running on unreliable systems. Section 5.5.1 demonstrated

generating a mesh while suffering a node failure. The mesh generator was able to successfully generate the mesh with only a 22% overhead over the non-fault tolerant case. Furthermore, adding 13 checkpoints over the entire process only increased the run length by 8%. If the failure rate of the entire system is 22% or greater, the fault tolerant implementation has a lower expected total run time. Chapter 2 showed that failure rates that high are already commonplace for long running jobs on 180 nodes. If failure rates continue to increase it will become critical to look at performance of applications from a statistical nature taking into account failure rates.

6.3 Future Work

This work was simply a first step in achieving a fully automated and fault tolerant CFD workflow and still much work remains. The development and evaluation of the integrated grid generation technique presented here identified key elements of what techniques should be explored in the future.

6.3.1 Mesh Generation

Parallel mesh generation techniques with integrated checkpointing systems may benefit from exploring compression techniques for storing arbitrary unstructured meshes. The compact Cartesian mesh representation using Morton curves was shown to be very beneficial in limiting the amount of memory required to store redundant checkpoints in memory. The body conforming region of the mesh has so far required much more memory to store redundantly.

Furthermore, this work only demonstrated body conforming mesh generation for simple smooth geometries. The Cartesian meshes generated with this technique may prove suitable for immersed boundary methods, or with Lattice-Boltzmann techniques to handles geometry configurations with arbitrary complexity.

6.3.2 Communication

The bandwidth results comparing `Messenger` to native ZeroMQ and MPI point to `Messenger` having a high constant or nearly constant overhead cost for each message. As the message size grows this overhead cost is being amortized over the additional time to communicate the large message through the network. Future work with `Messenger` could work to decrease the number of times data is copied during the communication of a message.

Timeout violations were detected on messages sent by the application layer and this technique has been proven useful. However, relying only on monitoring application communications does have drawbacks. False failures can be reported if there are work imbalances within the system. One node may finish work quickly and post a communication. If the neighboring node has more work it may not respond to the communication quickly enough and an erroneous error will be reported. Future work may want to augment failure reporting to more than monitoring communication from the application. The communication layer would continue to monitor timeout violations for application communications, but would then double check if a timeout was detected. When a timeout violation is detected by a node, the node may send a separate query to the communication layer on the suspect remote node. The communication layer could be configured to respond quickly

when these verification queries were received. This system of "trust but verify" would continue to monitor communication sent by the application layer, but verify that the node has failed. That way false positives are not reported simply for work imbalances. This approach could be a hybrid between a timeout ticketing system and a heartbeat based failure detection system.

A current limitation of the messaging module is that rank 0 cannot be allowed to fail. If any other node dies the recovery operation is coordinated by rank 0. If rank 0 dies there is no succession plan for another rank to coordinate the recovery. A simple improvement could be made simply allowing another rank to take over the root node's responsibilities. If a failure is detected with node 0, all the surviving nodes could communicate with rank 1 to coordinate recovery, then rank 2, and so on.

Another potential failure which was not explored with this work is a "zombie" process. That is when a node is detected to have failed. But then starts communicating again at some later point. This type of failure was not explored in this work. Zombie processes may start send communication which would need to be ignored by the surviving nodes after a failure. `Messenger` could incorporate a epoch number into its current message header. If a node received a message with an invalid epoch number it could simply discard the message.

6.3.3 Checkpoint / Recovery

An optimal checkpoint interval was proposed by Young in 1974 based on average MTTF and time to create a checkpoint [98]. The interval is given by:

$$I = \sqrt{2 \times MTTF \times C}$$

where C is the time it takes to complete a checkpoint. Additionally, the temporal coupling in failures on large scale machines suggests that techniques which dynamically alter the time between checkpoints may be beneficial. Other fault tolerance researchers have explored "lazy-checkpointing" to exploit this temporal coupling. The current checkpointing strategy is rigid in its approach to taking checkpoints. `Vault` takes a full checkpoint each time one is requested. Future work with mesh generation resiliency may benefit from a more flexible way of determining when a machine is likely to fail, and to increase or decrease the checkpoint frequency accordingly.

6.4 Conclusion

High Performance Computing is not doomed to grind to a halt as computer systems increase in scale. However, software design strategies which are suitable on machines that fail only once a month are not suitable when failures occur on the order of hours or minutes. Precautions must be made to treat failures as a normal part of operating on large scale systems.

Large CFD problems run today require computation on the order of one hundred thousand node hours. This puts CFD at a tipping point since today's large systems are also reporting

mean failure rates on the order of one hundred thousand node hours. Over the next two decades, grand challenge problems in CFD are expected to require mesh sizes exceeding 10 billion control volumes. [21] At current failure rates, CFD workflows will be plagued by failures. The total number of node hours for large CFD jobs may increase while the average node hours to failure does not. Predictions for future compute systems do not appear to offer relief. [4] Over the next decade, CFD software efforts that hope to address the grand challenge problems described in the CFD 2030 report should incorporate significant fault tolerance techniques.

This work has demonstrated that resiliency can be integrated into a parallel mesh generation software package. Mesh generation is just one component required for an automated, large scale, CFD workflow. Hopefully, this work adds to the discussion of how to create robust, 21^{st} century, computational science tools.

REFERENCES

[1] Top500.org, "Top 500 List June 2016," 2016.

[2] Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., Hill, K., et al., "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[3] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al., "Exascale computing study: Technology challenges in achieving exascale systems," http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf, 2008.

[4] Dongarra, J., Herault, T., and Robert, Y., *Fault Tolerance Techniques for High-Performance Computing*, Springer International Publishing, 2015.

[5] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M., "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, Vol. 23, No. 4, 2009, pp. 374–388.

[6] Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., and Snir, M., "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, Vol. 1, No. 1, 2014, pp. 5–28.

[7] Sprague, M., Boldyrev, S., Chang, C.-S., Fischer, P. F., Grout, R., Gustafson, W. I., Hittinger, J. A., Merzari, E., and Moser, R., *Outcomes from the DOE Workshop on Turbulent Flow Simulation at the Exascale*, No. 2016–3321, American Institute of Aeronautics and Astronautics, June 2016.

[8] Schroeder, B. and Gibson, G. A., "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, Vol. 78, IOP Publishing, 2007.

[9] Vera, X., Unsal, O., Ergin, O., Abella, J., and González, A., "Enhancing reliability of a many-core processor," Dec. 6 2011, US Patent 8,074,110.

[10] Borkar, S., "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *Micro, IEEE*, Vol. 25, No. 6, 2005, pp. 10–16.

[11] Hazucha, P., Karnik, T., Maiz, J., Walstra, S., Bloechel, B., Tschanz, J., Dermer, G., Hareland, S., Armstrong, P., and Borkar, S., "Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-μ to 90-nm generation," *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*, IEEE, 2003, p. 012022.

[12] Michalak, S. E., Harris, K. W., Hengartner, N. W., Takala, B. E., Wender, S., et al., "Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, Vol. 5, No. 3, 2005, pp. 329–335.

[13] El-Sayed, N. and Schroeder, B., "Reading between the lines of failure logs: Understanding how HPC systems fail," *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, IEEE, 2013, pp. 1–12.

[14] Springmeyer, R., Still, C., Schulz, M., Ahrens, J., Hemmert, S., Minnich, R., McCormick, P., Ward, L., and Knoll, D., "From Petascale to Exascale: Eight Focus Areas of R&D Challenges for HPC Simulation Environments," Tech. Rep. LLNL-TR 474731, Lawrence Livermore National Laboratory, 2011.

[15] Härtig, H., Matsuoka, S., Mueller, F., and Reinefeld, A., "Resilience in Exascale Computing (Dagstuhl Seminar 14402)," *Dagstuhl Reports*, Vol. 4, No. 9, 2015.

[16] DeBardeleben, N., Laros, J., Daly, J., Scott, S., Engelmann, C., and Harrod, H., "High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development," http://www.csm.ornl.gov/ engelman/publications/debardeleben09high-end.pdf, 2009.

[17] Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al., "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, Vol. 28, May 2014, pp. 129–173.

[18] Dongarra, J. et al., "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, Vol. 25, February 2011, pp. 3–60.

[19] Zheng, G., Ni, X., and Kalé, L. V., "A scalable double in-memory checkpoint and restart scheme towards exascale," *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, IEEE, June 2012.

[20] Zheng, G., Shi, L., and Kalé, L. V., "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," *2004 IEEE International Conference on Cluster Computing*, IEEE, 2004, pp. 93–103.

[21] Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D., "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," NASA CR-2014-218178, Langley Research Center, March 2014.

[22] Sahoo, R. K., Squillante, M. S., Sivasubramaniam, A., and Zhang, Y., "Failure data analysis of a large-scale heterogeneous server environment," *Dependable Systems and Networks, 2004 International Conference on*, IEEE, 2004, pp. 772–781.

[23] Schroeder, B. and Gibson, G. A., "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, Vol. 7, No. 4, 2010, pp. 337–350.

[24] "Failure History of TSUBAME 2.5," mon.g.gsic.titech.ac.jp/trouble-list/index.htm.

[25] Tu, T., O'Hallaron, D. R., and Ghattas, O., "Scalable parallel octree meshing for terascale applications," *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, IEEE, 2005.

[26] Burstedde, C., Wilcox, L. C., and Ghattas, O., "`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees," *SIAM Journal on Scientific Computing*, Vol. 33, No. 3, 2011, pp. 1103–1133.

[27] Lintermann, A., Schlimpert, S., Grimmen, J., Günther, C., Meinke, M., and Schröder, W., "Massively parallel grid generation on HPC systems," *Computer Methods in Applied Mechanics and Engineering*, Vol. 277, 2014, pp. 131–153.

[28] Betro, V. C., "Fully Antisotropic Split-Tree Adaptive Refinement Mesh," AIAA Paper 2011–0895, 2011.

[29] Ishikawa, N., Sasaki, D., and Nakahashi, K., "Large-scale Distributed Computation Using Building-Cube Method," AIAA Paper 2011-0754, 2011.

[30] Dawes, W., Harvey, S., Fellows, S., Favaretto, C., and Vellivelli, A., "Viscous layer meshes from level sets on cartesian meshes," AIAA Paper 2007-0555, 2007.

[31] Dawes, W., Harvey, S., Fellows, S., Eccles, N., Jaeggi, D., and Kellar, W., "A Practical Demonstration of Scalable, Parallel Mesh Generation," AIAA Paper 2009–0981, 2009.

[32] Aftosmis, M. J., Berger, M. J., and Melton, J. E., "Robust and Effcient Cartesian Mesh Generation for Component-Based Geometry," *AIAA Journal*, Vol. 36, No. 6, 1998, pp. 952–960.

[33] Karman, Jr., S. L., "SPLITFLOW: A 3D unstructured Cartesian/prismatic grid CFD code for complex geometries," AIAA Paper 1995–0343, 1995.

[34] Lahur, P. R., "Automatic hexahedra grid generation method for component-based surface geometry," AIAA Paper 2005–5242, 2005.

[35] Maréchal, L., "Advances in octree-based all-hexahedral mesh generation: handling sharp features," *Proceedings of the 18th International Meshing Roundtable*, Springer, 2009, pp. 65–84.

[36] Qian, J. and Zhang, Y., "Sharp feature preservation in octree-based hexahedral mesh generation for CAD assembly models," *Proceedings of the 19th International Meshing Roundtable*, Springer, 2010, pp. 243–262.

[37] Gagvani, N. and Silver, D., "Parameter-controlled volume thinning," *Graphical Models and Image Processing*, Vol. 61, No. 3, 1999, pp. 149–164.

[38] Gagvani, N. and Silver, D., "Animating volumetric models," *Graphical Models*, Vol. 63, No. 6, 2001, pp. 443–458.

[39] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.1," June 4th 2015.

[40] Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V., "BEOWULF: A parallel workstation for scientific computation," *Proceedings, International Conference on Parallel Processing*, Vol. 95, 1995, pp. 11–14.

[41] Gropp, W., Lusk, E., Doss, N., and Skjellum, A., "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, Vol. 22, No. 6, 1996, pp. 789–828.

[42] Lauria, M. and Chien, A., "MPI-FM: High performance MPI on workstation clusters," *Journal of Parallel and Distributed Computing*, Vol. 40, No. 1, 1997, pp. 4–18.

[43] Aulwes, R. T., Daniel, D. J., Desai, N. N., Graham, R. L., Risinger, L. D., Taylor, M. A., Woodall, T. S., and Sukalski, M. W., "Architecture of LA-MPI, a network-fault-tolerant MPI," *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, IEEE, 2004, p. 15.

[44] Bagchi, S., Whisnant, K., Kalbarczyk, Z., and Iyer, R. K., "The chameleon infrastructure for adaptive, software implemented fault tolerance," *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, IEEE, 1998, pp. 261–267.

[45] Batchu, R., Dandass, Y. S., Skjellum, A., and Beddhu, M., "MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware," *Cluster Computing*, Vol. 7, No. 4, 2004, pp. 303–315.

[46] Chakravorty, S. and Kalé, L. V., "A fault tolerant protocol for massively parallel systems," *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, IEEE, 2004, p. 212.

[47] Meneses, E., Mendes, C. L., and Kalé, L. V., "Team-based message logging: Preliminary results," *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, IEEE, 2010, pp. 697–702.

[48] Meneses, E., Bronevetsky, G., and Kale, L. V., "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 1533–1540.

[49] Chakravorty, S., Mendes, C. L., and Kalé, L. V., "Proactive fault tolerance in MPI applications via task migration," *High Performance Computing-HiPC 2006*, Springer, 2006, pp. 485–496.

[50] Sahoo, R. K., Oliner, A. J., Rish, I., Gupta, M., Moreira, J. E., Ma, S., Vilalta, R., and Sivasubramaniam, A., "Critical event prediction for proactive management in large-scale computer clusters," *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2003, pp. 426–435.

[51] Jasper, D. P., "A discussion of checkpoint/restart," *Software Age*, October 1969, pp. 9–14.

[52] Biedron, R. T., Carlson, J.-R., Derlaga, J. M., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, B., Lee-Rausch, E. M., Nielsen, E. J., Park, M. A., Rumsey, C. L., Thomas, J. L., and Wood, W. A., "FUN3D Manual: 12.9," NASA TM-2016-219012, Langley Research Center, March 2016.

[53] Petrini, F., "Scaling to thousands of processors with buffered coscheduling," *Scaling to New Heights Workshop*, 2002.

[54] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S., "FTI: high performance fault tolerance interface for hybrid systems," *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011, p. 32.

[55] Cappello, F., "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, Vol. 23, No. 3, 2009, pp. 212–226.

[56] Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., and Engelmann, C., "Combining partial redundancy and checkpointing for HPC," *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, IEEE, 2012, pp. 615–626.

[57] Stellner, G., "CoCheck: Checkpointing and process migration for MPI," *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, IEEE, 1996, pp. 526–531.

[58] Hargrove, P. H. and Duell, J. C., "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, Vol. 46, IOP Publishing, 2006, p. 494.

[59] Bouteiller, A., Lemarinier, P., Krawezik, G., et al., "Coordinated checkpoint versus message log for fault tolerant MPI," *Proceedings. 2003 IEEE International Conference on Cluster Computing*, IEEE, 2003, pp. 242–250.

[60] Torell, W. and Avelar, V., "Mean time between failure: Explanation and standards," white paper 78, APC, 2004.

[61] Intel Corporation ®, "Calculated MTBF," http://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/s3420gpmtbfcalculationrev10.pdf, 2009.

[62] NVIDIA Corporation ®, "Tesla K40 GPU ACTIVE ACCELERATOR Board Specification," https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf, 2013.

[63] Kingston Technology Corporation ®, "SSD Now V300," https://www.kingston.com/datasheets/sv300s3_us.pdf, 2015.

[64] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 2.2," September 4th 2009.

[65] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," September 21st 2012.

[66] Zounmevo, J. A., Kimpe, D., Ross, R., and Afsahi, A., "Extreme-scale computing services over MPI: Experiences, observations and features proposal for next-generation message passing interface," *International Journal of High Performance Computing Applications*, Vol. 28, No. 4, 2014, pp. 435–449.

[67] Bland, W., Du, P., Bouteiller, A., Herault, T., Bosilca, G., and Dongarra, J., "A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI," *Euro-Par 2012 Parallel Processing*, Springer, 2012, pp. 477–488.

[68] Fagg, G. E. and Dongarra, J. J., "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *Recent advances in parallel virtual machine and message passing interface*, Springer, 2000, pp. 346–353.

[69] Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., et al., "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," *Supercomputing, ACM/IEEE 2002 Conference*, IEEE, 2002, pp. 29–29.

[70] Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., and Magniette, F., "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ACM, 2003, p. 25.

[71] Graham, R. L., "MPI 3.0 Fault Tolerance Working Group," http://meetings.mo pi-forum.org/mpi3.0_ft.php(August 2015).

[72] Izrailevsky, Y., "The Netflix Simian Army," http://techblog.netflix.com/2011/07/netflix-simian-army.html, June 2011.

[73] Chrisochoides, N., "Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications," AIAA Paper 2016–3181, 2016.

[74] Chrisochoides, N., *Numerical Solution of Partial Differential Equations on Parallel Computers*, chap. Parallel Mesh Generation, Lecture Notes in Computational Science and Engineering, Springer-Verlag, 2006.

[75] Loseille, A., Menier, V., and Alauzet, F., "Parallel Generation of Large-size Adapted Meshes," *Procedia Engineering*, 24th International Meshing Roundtable, Sandia National Laboratories, 2015, pp. 57–69.

[76] Campbell, C. H., Anderson, B., Bourland, G., Bouslog, S., Cassady, A., Horvath, T., Berry, S., Gnoffo, P., Wood, B., Reuther, J., et al., "Orbiter return to flight entry aeroheating," , No. 2006–2917, 2006.

[77] Libby, R., "Effective HPC hardware management and Failure prediction strategy using IPMI," *Proceedings of the 17th Annual Intl Symposium on High Performance Computing Systems and Applications & OSCAR Symposium*, NRC Research Press, 2003, pp. 295–302.

[78] Gu, J., Zheng, Z., Lan, Z., White, J., Hocks, E., and Park, B.-H., "Dynamic meta-learning for failure prediction in large-scale systems: A case study," *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, IEEE, 2008, pp. 157–164.

[79] Tiwari, D., Gupta, S., Rogers, J., Maxwell, D., Rech, P., Vazhkudai, S., Oliveira, D., Londo, D., DeBardeleben, N., Navaux, P., et al., "Understanding GPU errors on large-scale hpc systems and the implications for system design and operation," *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, IEEE, 2015, pp. 331–342.

[80] Xu, J., Kalbarczyk, Z., and Iyer, R. K., "Networked Windows NT system field failure data analysis," *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, IEEE, 1999, pp. 178–185.

[81] Di Martino, C., Kalbarczyk, Z., Iyer, R. K., Baccanico, F., Fullop, J., and Kramer, W., "Lessons learned from the analysis of system failures at petascale: The case of blue waters," *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, IEEE, 2014, pp. 610–621.

[82] Vatsa, V., Lockard, D., Khorrami, M., and Carlson, J.-R., "Aeroacoustic simulation of a nose landing gear in an open jet facility using FUN3D," AIAA Paper 2012–2280, 2012.

[83] Khorrami, M. R., Fares, E., and Casalino, D., "Towards Full-Aircraft Airframe Noise Prediction: Lattice-Boltzmann Simulations," AIAA Paper 2014–2481, 2014.

[84] Aftosmis, M. J., Berger, M. J., and Murman, S. M., "Applications of space-filling curves to Cartesian methods for CFD," AIAA Paper 2004–1232, 2004.

[85] Shewchuk, J. R., "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, Vol. 18, No. 3, Oct. 1997, pp. 305–363.

[86] Karamcheti, V., Li, C., Pechtchanski, I., and Yap, C., "A core library for robust numeric and geometric computation," *Proceedings of the fifteenth annual symposium on Computational geometry*, ACM, 1999, pp. 351–359.

[87] Woop, S., Benthin, C., and Wald, I., "Watertight ray/triangle intersection," *Journal of Computer Graphics Techniques (JCGT)*, Vol. 2, No. 1, 2013, pp. 65–82.

[88] Wang, Z. and Srinivasan, K., "An adaptive Cartesian grid generation method for 'Dirty' geometry," *International journal for numerical methods in fluids*, Vol. 39, No. 8, 2002, pp. 703–717.

[89] Lahur, P. R., Hashimoto, A., and Murakami, K., "Automatic Grid Generation Method with Direct Tretment of Defective STL Data," *Research Notes Proceedings of the 20th International Meshing Roundtable, Springer Verlag, Berlin Heidelberg*, 2011, pp. 71–75.

[90] Hashimoto, A., Ishiko, K., Lahur, P. R., Murakami, K., and Aoyama, T., "Validation of Fully Automatic Grid Generation Method on Aircraft Drag Prediction," AIAA Paper 2010–4669, 2010.

[91] Möller, T. and Trumbore, B., "Fast, minimum storage ray/triangle intersection," *ACM SIGGRAPH 2005 Courses*, ACM, 2005, p. 7.

[92] Möller, T., "A fast triangle-triangle intersection test," *Journal of graphics tools*, Vol. 2, No. 2, 1997, pp. 25–30.

[93] Pirzadeh, S. Z., "Advanced unstructured grid generation for complex aerodynamic applications," *AIAA journal*, Vol. 48, No. 5, 2010, pp. 904–915.

[94] Ewing, P., "Best Practices For Aerospace Aerodynamics," STAR East Asia Conference.

[95] "ZeroMQ," http://www.zeromq.org, Accessed: 2016-02-01.

[96] Association, I. T. et al., *InfiniBand Architecture Specification: Release 1.0*, InfiniBand Trade Association, 2000.

[97] Dompierre, J., Labbé, P., Vallet, M.-G., and Camarero, R., "How to Subdivide Pyramids, Prisms, and Hexahedra into Tetrahedra," *8th International Meshing Roundtable*, 1999.

[98] Young, J. W., "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, Vol. 17, No. 9, 1974, pp. 530–531.

VITA

Matthew O'Connell was born in Syracuse New York on March 1st, 1987. As a child, Matthew talked too much, rode the bus often, and spent far too many evenings watching episodes of NOVA and Nature with his parents.

Matt's enthusiasm for science lead him to Austin Peay State University in 2005, where he studied Physics, and learned the value of hard work. Four years later in 2009, Matt left Austin Peay with: a Bachelor's degree in Physics, the Robert F. Sears award, and Emily Stark.

Matt pursued a Masters degree in Mechanical Engineering at the University of Tennessee at Chattanooga: SimCenter. In 2011, Matt finished his thesis work, "A Comparison of Two Methods for Two Dimensional Mesh Adaptation with Elliptic Smoothing" under the supervision of Steve Karman. Matt immediately began his doctoral work.

Matt was wed to Emily on June 22nd, 2013. Mr. O'Connell began work at NASA Langley Research center four weeks later.

Mr. O'Connell continued to pursue his doctoral work on parallel mesh generation and fault tolerance techniques for high performance computing while at NASA Langley Research Center. In 2016 Mr. O'Connell completed his doctoral work , "A Fault Tolerant Grid Generation Technique," and graduated from The University of Tennessee at Chattanooga in August 2016.

Dr. O'Connell remains passionate for scientific exploration, and engineering applications. His research interests include mesh generation, manipulation, and adaptation.