Copyright
by
Alison Nicholas Norman
2010

The Dissertation Committee for Alison Nicholas Norman
certifies that this is the approved version of the following dissertation:

Compiler-Assisted Staggered Checkpointing

Committee:

Calvin Lin, Supervisor

Sung-Eun Choi

Lorenzo Alvisi

Kathryn S. McKinley

Keshav Pingali

Compiler-Assisted Staggered Checkpointing

by

Alison Nicholas Norman, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Dedicated to my family, especially Grandmother and GranMarge, who I miss very much

Acknowledgments

Many people have advised, guided, and encouraged me on my journey to achieving my doctoral degree.

I would like to begin by thanking my advisor, Calvin Lin, for all of the time he has spent on this research and the support he has shown me over the years. This research would not be the same without his advice. Sung-Eun Choi joined this project in its early stages and has volunteered her time and energy for this research despite it being terribly inconvenient for her. Sung has provided expertise and advice on many subjects, and I appreciate her guidance.

The remainder of my committee includes: Lorenzo Alvisi, Keshav Pingali, and Kathryn S. McKinley. It is an honor to have their minds considering this research. I would especially like to thank Kathryn, who has acted as a role model and mentor during my time in graduate school.

Many people outside the department also contributed to this research. I would like to thank the people at the Texas Advanced Computing Center, who supplied High Performance Computing resources and cheerfully shared information. Many thanks as well to Patty Hough, who provided the original funding for this work, and Greg Bronevetsky. Both Patty and Greg contributed immeasurably to my overall understanding of the real-world experi-

ence of checkpointing. Thank you also to Paul Tobias, who provided guidance on statistical methods with rapid response and quick turn-around while he was preparing to leave the country.

I have been inspired over the years by any number of good teachers and mentors. In particular, I would like to thank Carolyn Hansen, one of my middle-school teachers, who encouraged me to think about, write about, and present many different subjects, and who let me know that she believed I could do anything.

My first introduction to computers was in my parents' home, who allowed my preschool self enough access to the computer that I was able to spill paper clips into the keyboard. My first programming experience was with the University of Georgia Challenge program, where I spent Saturdays moving a turtle around the screen in LOGO. I distinctly remember learning that it was impossible to program the turtle to move in a perfect circle.

In high school, I was fortunate enough to intern with J. Scott Shaw in the University of Georgia Astrophysics department. Dr. Shaw provided me with a list of summer projects—one of these projects was working with code. Due to some confusion where I thought "code" meant the cryptographic form rather than the programming form, I spent the summer programming in FORTRAN. I loved it. The next summer I applied for the Program for Women in Science and Engineering at Iowa State, which is an eight-week summer internship program in Ames. I was accepted, but I balked at moving to Iowa rather than Galveston, Texas, where I had another summer opportunity. Due

in large part to the skilled persuasion of Charlie Wright, I moved to Iowa for the summer and worked with him in his robotics lab. There, I programmed LEGO Mindstorm robots in Interactive C. Thanks to these early experiences and these wonderful men, I had a deep love for the capabilities of computers and a solid introduction to programming before I ever graduated from high school.

During my college days at the Georgia Institute of Technology, I was guided and encouraged by Jim Greenlee, Kurt Eiselt, and Howard Karloff, for which I am so grateful. In addition, thank you to Jennifer, my college roommate and fellow computer scientist, with whom I learned to be confident in my knowledge.

In graduate school, I have been supported and guided by so many wonderful friends and colleagues: Serita, Adam, Heather, Raj, Jenny, Jeff, Shel, Paul, Greg, Angela, Harry, Jenn, Walter, Allen, Bob, Kartik, JP, Misha, Matt, Ben, Maria, Peter, Mike, Sam, Daniel, Teck Bok, Katie, Bert, and Curtis. Many, many thanks for technical discussions, advice, celebrations of the good, and commiseration about the bad.

Finally, I would like to thank my family. Thank you to Mom and Dad, for all their support, advice, and inspiration. They are both true role models. My mother is the first female computer scientist I ever met and is a huge reason why it has never occurred to me that I could not do whatever I wanted. My father has always encouraged me to do my best and provided help and advice along the way. He is the best teacher I have ever met, and he is the reason I

aspire to teach. Thank you to my sister, Abigail, who has encouraged me over the years and provided tangible support over the past few months during the last push to graduate. Thank you also to my brother, Justin, who has never let me forget exactly how long I have spent in graduate school.

I would also like to thank my family-by-marriage. The Normans have supported me throughout this process, beginning before I was officially a part of their family. I am forever grateful. Many thanks also to Beth and Paul, who might as well be family, for their guidance and encouragement.

Last of all, I would like to express my sincere gratitude, appreciation, and love for my husband, Michael, and my son, Jeffrey. Michael, who was with me when I began graduate school and married me in the middle of it, has supported me in more ways than I can count and enabled me to accomplish this dream of mine with far more serenity than I would have otherwise. Jeffrey joined our family more recently; his joy and enthusiasm have caused me to smile and remember there is a world more important than graduate school.

Compiler-Assisted Staggered Checkpointing

Publication No. _____

Alison Nicholas Norman, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Calvin Lin

To make progress in the face of failures, long-running parallel applications need to save their state, known as a checkpoint. Unfortunately, current checkpointing techniques are becoming untenable on large-scale supercomputers. Many applications checkpoint all processes simultaneously—a technique that is easy to implement but often saturates the network and file system, causing a significant increase in checkpoint overhead. This thesis introduces compiler-assisted staggered checkpointing, where processes checkpoint at different places in the application text, thereby reducing contention for the network and file system. This checkpointing technique is algorithmically challenging since the number of possible solutions is enormous and the number of desirable solutions is small, but we have developed a compiler algorithm that both places staggered checkpoints in an application and ensures that the solution is desirable. This algorithm successfully places staggered checkpoints in parallel applications configured to use tens of thousands of processes. For our benchmarks, this algorithm successfully finds and places useful recovery

lines that are up to 37% faster for all configurations than recovery lines where all processes write their data at approximately the same time.

We also analyze the success of staggered checkpointing by investigating sets of application and system characteristics for which it reduces network and file system contention. We find that for many configurations, staggered checkpointing reduces both checkpointing time and overall execution time.

To perform these analyses, we develop an event-driven simulator for large-scale systems that estimates the behavior of the network, global file system, and local hardware using predictive models. Our simulator allows us to accurately study applications that have thousands of processes; it on average predicts execution times as 83% of their measured value.

Table of Contents

Acknowledgments **v**

Abstract **ix**

List of Tables **xiv**

List of Figures **xxv**

Chapter 1. Introduction **1**

Chapter 2. Related Work **11**

2.1 Checkpointing Parallel Applications 11

2.1.1 Coordinated Checkpointing 12

2.1.2 Checkpointing Without Dynamic Coordination 13

2.1.3 Compiler-Assisted Checkpointing 15

2.1.4 Other Techniques 16

2.2 Simulation of Large-Scale Systems 17

2.3 Summary 19

Chapter 3. Simulation for Large-Scale Systems **21**

3.1 Overview of the Simulator 23

3.2 Translator 25

3.3 Simulator 31

3.3.1 Modeling System Components 32

3.3.1.1 Calculating Error 33

3.3.1.2 Local Hardware Model 35

3.3.1.3 Network Model 40

3.3.1.4 File System Model 43

3.3.2 Event Simulation 46

3.4 Evaluation 58
3.5 Conclusions 62

Chapter 4. Problem and Solution Spaces for Staggered Checkpointing 63
4.1 Methodology 63
4.2 Evaluation 65
4.2.1 Lonestar 69
4.2.2 Ranger 85
4.2.3 The Effects of Interval Size 101
4.2.4 Comparison Between Systems 102
4.2.5 Relationship to Application Characteristics 104
4.3 Conclusions 106

Chapter 5. Algorithm for Compiler-Assisted Staggered Checkpointing 107
5.1 Background 108
5.2 Algorithm Overview 110
5.3 Algorithm Phase One: Identifying Communication 111
5.4 Algorithm Phase Two: Determining Inter-Process Dependences 113
5.5 Algorithm Phase Three: Generating Useful Recovery Lines . . 117
5.5.1 Naïve Algorithm 117
5.5.2 Our Algorithm 118
5.5.2.1 Foundation Algorithm 120
5.5.2.2 Reducing L, the Number of Checkpoint Locations 121
5.5.2.3 Reducing P, the Number of Independent Processes 123
5.5.2.4 The Effects of Clumps on the Search Space . . . 125
5.5.2.5 Pruning Invalid Solutions 127
5.5.2.6 Optimizations 128
5.5.2.7 Branch-and-Bound Using the Wide and Flat Metric 129
5.5.2.8 Pruning Policy 131
5.5.2.9 Adaptive Bins 132
5.5.2.10 Correctness Considerations 134

5.6 Implementation 135
5.6.1 Benchmarks 135
5.7 Evaluation 136
5.7.1 Algorithm 136
5.7.1.1 Wide-and-Flat Metric 147
5.7.1.2 Foundation Algorithm Parameters 148
5.7.2 Pruning Policy Parameters 150
5.7.3 Staggered Checkpointing 157
5.8 Conclusions 164

Chapter 6. Conclusions and Future Work 172
6.1 Contributions 172
6.1.1 Algorithm for Compiler-Assisted Staggered Checkpointing 173
6.1.2 Problem and Solution Spaces for Staggered Checkpointing 174
6.1.3 Simulation for Large-Scale Systems 175
6.2 Future Work 176
6.3 Final Thoughts 177

Bibliography 180

Vita 221

List of Tables

3.1 An example of the representation of events in the trace file. . 25
3.2 Error for the local hardware models on Lonestar. 39
3.3 Error for the local hardware models on Ranger. 40
3.4 Error for the sub-models of the local hardware models. 40
3.5 Error for the network models. 43
3.6 Error for the file system models. 46
3.7 Error for the benchmarks with message passing on Lonestar. 60
3.8 Error for the benchmarks with message passing on Ranger. . 60
3.9 Error for the synthetic benchmarks with checkpointing on Lonestar. 61
3.10 Error for the synthetic benchmarks with checkpointing on Ranger. 61
3.11 Error for the benchmarks with message passing and checkpointing on Lonestar. 61
3.12 Error for the benchmarks with message passing and checkpointing on Ranger. 62

4.1 Lonestar Problem Space: This table identifies the problem space by comparing the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. So, this table shows that one process can checkpoint 16 MB 110 minutes faster than the 2,048 processes can each checkpoint 16 MB. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system. 69
4.2 Lonestar Problem Space: This table reports the number of minutes needed for a number of processes to each synchronously checkpoint an amount of data. So 2,048 processes synchronously checkpointing 16 MB of data requires approximately 111 minutes. Note that its configurations begin at 16 rather than 256. 70

4.3 Lonestar Problem Space, Statistical Information: This table identifies the problem space by comparing the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. So, this table shows that one process can checkpoint 16 MB 110 minutes faster than the 2,048 processes can each checkpoint 16 MB. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system. . . 71

4.4 Lonestar Solution Space (Checkpoint Time in 3 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately three minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint approximately 11 minutes faster using staggered checkpointing rather than using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 74

4.5 Lonestar Solution Space (Total Time in 3 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately three minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by nine minutes or 10% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 75

4.6 Lonestar Solution Space (Checkpoint Time in 15 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately fifteen minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 60 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. . . 78

4.7 Lonestar Solution Space (Total Time in 15 minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately fifteen minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 48 minutes or 40% using staggered checkpointing rather than using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 79

4.8 Lonestar Solution Space (Checkpoint Time in 30 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately thirty minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 109 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 82

4.9 Lonestar Solution Space (Total Time in 30 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately thirty minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 109 minutes or 80% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 83

4.10 Lonestar Solution Space (Staggered Checkpoint Time in 15 Minute Interval): This table shows the number of minutes spent checkpointing for the staggered policy with approximately fifteen minutes of local execution time. Note that the data in this table begins at 16 processes. 84

4.11 Ranger Problem Space: This table identifies the problem space and compares the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. So, one process can checkpoint 16 MB 11 minutes faster than the 2,048 processes can each checkpoint 16 MB. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system. 86

4.12 Ranger Problem Space, Statistical Information: This table identifies the problem space and compares the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. So, one process can checkpoint 16 MB 8 minutes faster than the 2,048 processes can each checkpoint 16 MB. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system. 87

4.13 Ranger Problem Space: This table reports the number of minutes needed for a number of processes to each synchronously checkpoint an amount of data. So 2,048 processes checkpointing 16 MB of data requires 13 minutes. 88

4.14 Ranger Solution Space (Checkpoint Time in 3 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately three minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 3 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 91

4.15 Ranger Solution Space (Total Time in 3 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately three minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 3 minutes or 20% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 92

4.16 Ranger Solution Space (Checkpoint Time in 15 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately fifteen minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 11 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. . . 95

4.17 Ranger Solution Space (Total Time in 15 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately fifteen minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 11 minutes or 40% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 96

4.18 Ranger Solution Space (Checkpoint Time in 30 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately thirty minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 11 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 99

4.19 Ranger Solution Space (Total Time in 30 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately thirty minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 11 minutes or 30% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively. 100

4.20 Liveness sizes and instructions in a single phase for a subset of the NAS parallel benchmarks [51], `BT`, `LU`, and `SP`, and a well-known CFD benchmark, `Ek-simple`. 105

4.21 Characteristics of Real-World Applications 105

5.1 Clumps for the example shown in Figure 5.7. 125
5.2 Clump sets for example in Figure 5.7. 125
5.3 Parameters of the Pruning Policy 133
5.4 Initial search space. The initial number of checkpoint locations and possible recovery lines without any of our reduction techniques, including phases. P represents the number of processes in the system, L represents the number of checkpoint locations. 140
5.5 Phase Results. Each benchmark consists of several phases; we chose one phase from each benchmark to report. The $P : L$ syntax shown in this table describes the number of processes that execute a particular number of checkpoint locations. Where P is absent, all processes execute that location. 141
5.6 Checkpoint Reduction Numbers. These numbers represent the checkpoint locations and search space that remain after our algorithm has merged any closely-spaced checkpoint locations. . 142
5.7 Clumps. This table reports the number of clumps for each benchmark, the size of those clumps, the number of checkpoint locations executed by each, and the resulting size of the search space. C represents clumps; the $C : P$ and $C : L$ notations shown in this table denote the number of clumps containing a particular number of processes and the number of clumps that execute a particular number of checkpoint locations, respectively. For brevity, we only report five most significant results in each column for `Ek-simple`. 143
5.8 Clump sets. S represents the number of sets. The $S : C$ syntax shown in this table denotes the number of sets containing a particular number of clumps. 144
5.9 Summary of search space reduction by each technique in our algorithm for the benchmarks configured to use five different process sizes. The second column displays the initial size of the search space. The next column reports the reduction realized when phases are introduced; this column reports the search space for a representative phase from each benchmark. The last three columns report the size of the search space after the checkpoint locations are merged, clumps are introduced, and clump sets are formed. 145

5.10 Actual search space reduction performed by our algorithm for the our benchmarks configured to use five different process sizes. The third column, "Skipped Initially", indicates the number of partial and complete recovery lines that are eliminated as invalid through a single validity check before the lines are generated. The fourth column, "Generated", reports the number of lines generated, and then the fifth column, "Investigated", displays the number of lines that are checked for validity after generation. The fifth column, "Leveraged", indicates the number of lines that are eliminated after generation but without validity checks: recovery lines constraints are strictly increasing, and our algorithm leverages that information. The last two columns, "Pruned by WAF" and "Valid Found", report the number of lines eliminated by our branch-and-bound methodology and the number of lines produced by the algorithm, respectively. . . . 146

5.11 The simulated checkpoint times on Ranger for recovery lines in the `BT` benchmark with 4,096 processes and local execution of approximately three minutes. 147

5.12 The results of our algorithm when run with varied values of j, the parameter the determines how many partitions are merged to form a new partition at each step. In this table, the number of processes in the initial partition, k, remains constant at 4. The first column reports the value of j, and the second column reports the number of recovery lines found by the algorithm for that configuration. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "10%" displays the percentage of lines found in the lowest 10% of the bins. The final column, "Compile Time", reports the execution time for our compiler. All numbers are for a single clump set of the SP benchmark with 4,096 processes and a 1,632 problem size. 149

5.13 The results of our algorithm when run with varied values of k, the parameter that determines how many processes comprise the initial partitions. In this table, the number of partitions merged to form a new partition at each step, j, remains constant at 2. The first column reports the value of k, and the second column reports the number of recovery lines found by the algorithm for that configuration. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "10%" displays the percentage of lines found in the lowest 10% of the bins. The final column, "Compile Time", reports the execution time for our compiler. All numbers are for a single clump set of the SP benchmark with 4,096 processes and a 1,632 problem size. 150

5.14 The results of our algorithm for `SP` when run with varied values of the pruning threshold. In this table, the first column reports the value of the pruning threshold. As the pruning threshold increases, more lines are stored before the next pruning pass. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes. 151

5.15 The results of our algorithm on the `LU` benchmark when run with varied values of the line check threshold. In this table, the first column reports the number of processes for that configuration, and the second column reports the value of the line check threshold. As the line check threshold increases, more lines are available to be leveraged by the algorithm. The third column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes. 153

5.16 The results of our algorithm for LU with 4,096 processes when run with varied values of Lines to Keep. In this table, the first column reports the value of the Lines to Keep parameter. This parameter is a divisor and the number of lines kept is equal to the pruning threshold divided by it. As parameter decreases, more lines are stored between pruning passes. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Algorithm Time" and "Memory Usage", report the execution time for the recovery line algorithm and the amount of memory it consumes. 154

5.17 The results of our algorithm for LU with 4,096 processes when run with varied values of Target Number of Bins. In this table, the first column reports the target number of bins. This parameter is a divisor and the number of lines kept is equal to the pruning threshold divided by it. As this parameter decreases, more lines are stored between pruning passes. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes. 155

5.18 The results of our algorithm for LU with 4,096 processes when executed with varied values of Minimum Lines in Each Bin. In this table, the first column reports the value for Minimum Lines in Each Bin as it relates to our initial parameter. As this parameter increases, a higher percentage of the lines to survive a pruning pass are kept in the lower bins. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes. 156

5.19 Ranger data for approximately three minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line places by our algorithm and the simultaneous one. 167

5.20 Lonestar data for approximately three minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line placed by our benchmark and the simultaneous one. 168

5.21 Ranger data for approximately fifteen minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line identified by our algorithm and the simultaneous one. 169

5.22 Lonestar data for approximately fifteen minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line identified by our algorithm and the simultaneous one. 170

5.23 Measured total execution time for one iteration of the `LU` benchmark on Lonestar for executions checkpointing with either staggered or simultaneous checkpointing 171

List of Figures

3.1 Example of a Typical Cluster 22

3.2 Simulation Infrastructure 23

3.3 `BT`'s local hardware model compared with its measured average cycles per operation for Lonestar. The dots represent the measured data and the line represents the generated model. . 39

3.4 Network Model: This graph plots the measured (points) and predicted (line) time (y-axis) for processes (x-axis) sending a total data size of 2GB on Lonestar. The measured data ends at 128 processes, but the predicted data continues to 256 processes. 43

3.5 File system model: This graph plots the measured (points) and predicted (line) amounts of time (y-axis) spent writing 64 MB messages by various numbers of processes (x-axis) on Lonestar. 45

3.6 Simulation Example: Slice of Execution 47

3.7 Simulation Example, Step 1: The state after p_0's local event at index 25 has been simulated. p_o does not execute the events at indices 23 and 24. Notice p_0's clock and index have been updated. 48

3.8 Simulation Example, Step 2: The state after p_2's network send at index 23 has been initiated. The send has been placed in the appropriate send queue and is waiting to be sent. 50

3.9 Simulation Example: Step 3 Network Receive 52

3.10 Simulation Example, Step 4: The state following the post of p_0's file system write at index 27. 53

3.11 Simulation Example, Step 5: The state following the simulation of p_1's local event at index 25. 54

3.12 Simulation Example, Step 6: Network Sends and File System Writes Progress 55

3.13 Simulation Example, Step 7: The state following the completion of p_3's network receive at index 24. 57

3.14 Simulation Example, Step 8: The state following the execution of p_2's local event at index 25. 58

4.1 Lonestar Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a three minute interval size. 72
4.2 Lonestar Solution Space (Total Time in 3 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a three minute interval size. 73
4.3 Lonestar Solution Space, 3 minutes, Total Time 73
4.4 Lonestar Solution Space (Checkpoint Time in 15 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size. 76
4.5 Lonestar Solution Space (Total Time in 15 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size. 77
4.6 Lonestar Solution Space (Checkpoint Time in 30 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size. 80
4.7 Lonestar Solution Space (Total Time in 30 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size. 81
4.8 Ranger Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a three minute interval size. 89
4.9 Ranger Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a three minute interval size. 90
4.10 Ranger Solution Space (Checkpoint Time in 15 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size. 93
4.11 Ranger Solution Space (Total Time in 15 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size. 94

4.12 Ranger Solution Space (Checkpoint Time in 30 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size. 97

4.13 Ranger Solution Space (Total Time in 30 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size. 98

5.1 Examples of invalid (a) and valid (b) recovery lines. The arrows represent message sends. The source is the sending process and the target is the receiving process. The **X**'s mark checkpoint locations. 109

5.2 A code example from the NAS parallel benchmark BT and its corresponding result from symbolic expression analysis. In this example, `node` represents the process identifier and `no_nodes` represents the number of processes executing the application. . 113

5.3 An example of Lamport's logical clocks. Each process's clock value is indicated by the [#], where # is the value of the clock. 114

5.4 An example of vector clocks. 114

5.5 Pseudocode for a naïve algorithm 119

5.6 An example of partitioning of processes. At the base of the diagram, each block represents a process and that process's checkpoint locations. The algorithm begins at the bottom, merging three blocks into a partition and finding all partial valid recovery lines for that partition. As the algorithm completes each level, the partitions of that level are merged to form the partitions of the next level. In this figure, each line in the merged partitions represents either a partial valid recovery line or a complete valid recovery line. 121

5.7 An example of process communication. Each ⓗ represents a checkpoint location. 124

Chapter 1

Introduction

Supercomputers are large-scale systems often composed of many thousands of processors. These machines support long-running parallel applications that use many—sometimes tens of thousands—processes; each processor can support one or more of these logical processes. Researchers create such applications as research tools to explore many phenomena, including climate change, protein folding, and the stability of the nuclear stockpile [171].

Unfortunately, these applications might encounter failures before they complete, due to problems with the application itself, system software errors, or hardware failures [125, 126, 185]. These hardware failures grow proportionally with the number of sockets[1] in a system [48]—a trend that will continue [48, 158]. In addition, supercomputing centers are encouraging capability computing [5, 71], which encourages developers to use more processes per application to more fully exploit the available resources on the supercomputer. The more processes an application uses, the greater the probability it will stop prematurely due to hardware and system software failures.

To mitigate the effect of these early failures, many applications have

[1] Sockets are the hardware mechanism that connects processors to motherboards.

each process periodically save an intermediate state, known as a *checkpoint*; the application then uses this state to resume execution after a failure. The saved state must be *consistent* [153], which means that it could have existed during the execution of the application. Saving a consistent state in a parallel application is difficult because parallel applications perform inter-process communication, and if the saved state reflects a message as being received but does not reflect that same message as being sent, the state could not have existed and thus is not consistent. For any message m from process p to process q, if q saves a checkpoint after m is received, then p must save its checkpoint after p has sent m. To save the state of a parallel application, it must be determined at what location each process should save its checkpoint and what data each process should save. Each of these things may either be done manually by the programmer or automatically by a software tool.

In many supercomputing applications today, programmer implement *manual checkpoints*: that is, the application programmer specifies the data to be saved, when it should be saved, and how the application should recover, and then implements those choices directly in the application source code. The advantage of this type of checkpoint is that it leverages the knowledge of the application programmer to save a minimum amount of state. However, this technique requires effort by the application programmer and can be error-prone, since testing is difficult and identifying necessary information can be complex [69]. Manually identifying the information to be saved will become even more difficult as application algorithms move towards implicit

and adaptive methods. In addition, to ensure that a consistent state is saved, processes typically synchronize immediately before and after the checkpoint, and synchronization is becoming increasingly expensive [65].

Automatic techniques either aid the programmer or relieve the programmer of these decisions. Such techniques are welcomed by the application programmers. A recent user survey at the Texas Advanced Computing Center (TACC) [2] showed that over 65% of respondents would be interested in completely automated checkpointing support [206]. However, current automated techniques result in large checkpoint sizes and every process writing its checkpoint at approximately the same time. As a result, they can suffer from overhead due to network and file system contention [146]. This contention can be reduced by writing the data somewhere other than the global file system, reducing the size of the data, or reducing the number of processes writing simultaneously. Unfortunately, the first two techniques have drawbacks, as we will now explain.

Checkpointing methods that do not store checkpoint data on the global file system typically write the data to either the memory of another process or local storage [52, 107, 199, 218]. The former class works well for applications that are not memory bound. However, most scientific applications use a large fraction of available memory [147], and some use all available memory and would like more [184]. The latter class of techniques applies to applications executing on supercomputers with local storage, and many supercomputers have no local storage. In addition, storing checkpoint data on local drives can

make recovery more difficult since the checkpoint data does not reside in a single location.

Some automatic checkpointing techniques reduce the size of the checkpoint. However, these checkpoints are likely larger than those resulting from manual checkpointing since they do not use the knowledge of the application programmer.

Another possibility is to allow each process to checkpoint at will and then require the recovery system to determine a correct state from the existing checkpoints. However, it is then possible that an intermediate correct state does not exist, and the application must restart from the beginning.

These stumbling blocks are among the myriad of reasons that only 2.5% of the respondents to the TACC survey [206] use an existing checkpoint library. Recall that 65% of the respondents are interested in completely automated checkpointing.

To investigate this problem and its scale, we evaluate synchronous checkpointing on two supercomputers, Lonestar [204] and Ranger [205], both located at TACC [2]. In our results, we analyze the benefit of our technique for five scientific applications: our results show all five of the applications could benefit from a new checkpointing technique.

Our Solution. Our approach is a compiler-assisted checkpointing technique that reduces programmer work, reduces network and file system contention,

and does not require process synchronization. Our solution is a form of staggered checkpointing that at compile-time places process checkpoint calls at different locations in the application text while guaranteeing that the saved checkpoints will form a consistent state. Our technique eliminates dynamic inter-process coordination and creates a separation in time of each process's checkpoint, which reduces contention at the file system. The file system on most supercomputers has some unreliability, so reducing its burden may result in more overall reliability.

Identifying desirable checkpoint locations in an application is difficult since there are many possible locations and the state saved must be consistent. To ensure a consistent state, our solution considers *recovery lines* [153], which are sets of application-level checkpoints that include exactly one checkpoint per process. A recovery line is *valid* only if it saves a consistent state and *invalid* if it does not. It is a challenge to statically differentiate the desired valid recovery lines from those that are invalid because the solution space is enormous. The number of possible recovery lines grows as L^P, where L is the number of possible checkpoint locations and P is the number of processes, and the number of valid recovery lines is typically less than 1% of the total. Recovery lines that are *useful*—a useful recovery line is both valid and sufficiently staggered to reduce contention—are even more rare. Thus, any algorithm that reduces the search space arbitrarily is unlikely to find enough useful recovery lines. Our algorithm uses a number of techniques to intelligently reduce the search space and introduces a static metric for evaluating the usefulness of a recovery

line; this metric allows us to choose and place recovery lines that will reduce contention at the file system when the processes save their checkpoints. It successfully identifies useful recovery lines in applications configured to use tens of thousands of processes.

By identifying valid recovery lines, our algorithm guarantees the saving of a consistent state even for applications that use shared memory or non-deterministic communication. In addition, this requirement ensures that our algorithm guarantees a correct state for applications using non-deterministic methods, such as many algorithms for partitioning, approximate matrix decompositions, and particle swarm optimization [111]. These algorithms include codes used in research regarding *ad hoc* wireless networks [81], and probabilistic search [216]. Allowing invalid recovery lines would significantly complicate both the runtime and recovery systems for these applications. The runtime system would need to log messages for each process from the time that process takes a checkpoint until all processes have checkpointed and dynamic coordination would be required to identify when every process has finished writing the checkpoint. For applications with large amounts of point-to-point communication, these message logs can become prohibitively large, especially if the application is memory-bound. Our solution does not require message logging or dynamic coordination at any time.

Scope. Our solution saves valid recovery lines, which give us the ability to allow non-determinism in both the communication and sequential code.

However, communication non-determinism causes our algorithm to identify more conservative communication dependences, which then limits its ability to stagger the checkpoint locations.

Our solution best applies to applications that have many processes and large checkpoints.

Our checkpointing approach reduces file system contention by separating process checkpoint locations among communication calls while considering the constraints introduced by that communication, it works well when applied to applications with large amounts of point-to-point communication. Staggered checkpointing is trivial for applications with large communication-free zones, and it not useful for applications with frequent synchronization or collective communication.

For applications that are memory-bound, it is best not to store the checkpoints in the memory of the local machines, but when there are enough processes writing large enough checkpoints to the global file system simultaneously, contention occurs at the network and filesystem [146]. Our checkpoint method reduces such contention. It also works well for applications executed on supercomputers with no local storage, since in such a system checkpoints must be stored on a global file system. For supercomputers with local storage, our solution may not be the best option, since checkpointing to another machine's hard drive removes contention from the global file system entirely. However, supercomputers are now often built without hard drives.

Evaluation by Simulation. We use simulation to evaluate our solution. We develop an event-driven simulator for large-scale systems that estimates the behavior of the network, global file system, and local hardware using predictive models. Currently, each model is a formula that encapsulates the performance of its respective component and thus provides the simulator with a fast way to estimate that component's behavior under a variety of workloads. Thus, the simulator efficiently estimates the system consequences of a change in checkpoint strategy and encourages experimentation with approaches to checkpointing.

Thesis statement. Compiler-assisted staggered checkpointing can reduce the burden of checkpoint implementation by the application developer, and decrease checkpoint runtime overhead to improve significantly the performance of parallel applications that checkpoint automatically.

Contributions. This dissertation makes three contributions:

- We investigate sets of application characteristics, including the number of processes and checkpoint sizes, and system characteristics, including network, file system, and processor speeds, where staggered checkpointing is needed and effective. We identify situations where staggered checkpointing is needed, which refer to as the *problem space*, by determining sets of application and system characteristics where synchronous checkpointing causes contention but the checkpointing of a single process does

not. We identify situations where staggered checkpointing is effective, or the *solution space*, by determining sets of characteristics for which staggered checkpointing reduces such contention. We also analyze how the solution space varies with machine characteristics such as the ratio of network and file system speeds to processor speeds.

We identify the problem and solution spaces for Ranger [205] and Lonestar [204], supercomputers at the Texas Advanced Computing Center [2].

- We design and implement a new compile-time algorithm to generate and place useful recovery lines in applications that use up to tens of thousands of processes. We also devise a new metric to statically estimate the usefulness of both complete recovery lines, those that include every process, and partial recovery lines, those that only include some of the processes. These estimates allow the algorithm to consider many fewer possible solutions, thus improving its performance.

  We show that our compilation system can place useful recovery lines in applications that use tens of thousands of processes.

- We develop an event-driven simulator that efficiently estimates the performance of large-scale parallel applications. In addition, it evaluates the effectiveness of our staggered checkpointing technique.

  We show that our simulator can simulate applications that use thousands of processes, and it can do so with sufficient accuracy to evaluate the effectiveness of our staggered checkpointing technique.

Our analysis of synchronous checkpointing shows that it can take up to an hour and 14 minutes for 8,192 processes on a particular supercomputer to checkpoint 4 MB of data each. With three minutes over which to separate the checkpoints, staggered checkpointing reduces this time by 12 minutes to just over an hour. With fifteen minutes, staggered checkpointing reduces this time to 1 minute. These results imply that staggered checkpointing is a useful technique.

Our algorithm reduces the search space for one of our application benchmarks from 38^{1024} to $3(5^{32}) + 2^1$, or by 1,594 orders of magnitude before it begins creating recovery lines. Our pruning policy enables the rapid sorting of the created recovery lines, and thus the algorithm can quickly identify lines that both save a consistent state and reduce network and file system contention.

For our benchmarks, our algorithm successfully finds and places useful recovery lines in applications that use up to 65,536 processes. The staggered recovery lines placed by our algorithm checkpoint an average of 37% faster for all configurations than a recovery line performing simultaneous checkpointing that writes an equivalent amount of data.

Our simulator on average predicts the execution time our application benchmarks with checkpointing as 83% of their measured performance, which provides sufficient accuracy to evaluate synchronous and staggered checkpointing.

Chapter 2

Related Work

This chapter reviews work that investigates checkpointing parallel applications and the simulation of large-scale systems. This chapter places our work in context and shows that previous work does not fully address key concerns.

2.1 Checkpointing Parallel Applications

In this section, we review recent work related to the checkpointing of parallel applications. For a full account of the evolution of checkpointing and other rollback recovery protocols please see the survey developed by Elnozahy *et al.* [66].

Much checkpointing research focuses on determining the location(s) where processes should save their checkpoints. Here, the main concern is to guarantee that the recovered state is consistent. There are two major approaches: 1) protocols that ensure a consistent state through dynamic coordination of the processes, and 2) those that ensure a consistent state without dynamic coordination. We discuss these general approaches in Sections 2.1.1 and 2.1.2.

Our work is distinguished by its ability to reduce file system contention, so throughout this section we pay particular attention to this aspect of other solutions. In addition, our checkpointing method guarantees the consistency of the checkpoint without: 1) dynamic coordination, which consumes execution time, 2) message logging, which consumes local resources, or 3) input from the user, which burdens the application programmer.

2.1.1 Coordinated Checkpointing

Coordinated checkpointing, in which processes dynamically coordinate to ensure consistency, is one of the most popular checkpointing techniques. Unfortunately, it results in all processes saving their checkpoints at approximately the same time; when large numbers of processes are involved and write their checkpoints to a central file system, this technique leads to high checkpoint overhead due to contention [146]. There are a few methods to reduce this contention: 1) save checkpoints somewhere other than the global file system, 2) reduce the size of the checkpoints, and 3) reduce the number of processes checkpointing simultaneously. The first approach is taken by, for example, an extension [218] of the Berkeley Lab Checkpoint/Restart (BLCR) mechanism [181], which is implemented within the some MPI libraries; it saves each checkpoint to the hard drive of another machine. While this approach does eliminate contention at the file system, it does not work on the many supercomputers without local storage.

The second approach, for example DejaVu [176], reduces the size of the

checkpoint. Techniques that use this approach are typically unable to reduce the size enough to avoid contention for large numbers of processes.

The third approach to reducing checkpoint overhead is to reduce the number of processes that checkpoint simultaneously; this approach is the one used by our staggered checkpointing. Our solution, however, scales to large numbers of processes; other solutions that separate checkpoint locations do not. For instance, Plank's staggered checkpointing algorithm [159] reduces the numbers of processes saving their checkpoints simultaneously but results in limited separation of the checkpoint writes. Staggered consistent checkpointing [211], separates checkpoints by only allowing a single process to write its checkpoint at a time. While this protocol reduces file system contention, it is inappropriate for large-scale applications because it linearizes checkpointing, lengthening the time from the beginning of an application's checkpoint to the end. Other protocols reduce the number of simultaneously writing processes by partitioning processes and then performing coordinated checkpointing within the partitions [82, 98]; these protocols still suffer from dynamic coordination and require message logging for communication between partitions. The drawbacks of message logging are discussed in the next section, Section 2.1.2.

2.1.2 Checkpointing Without Dynamic Coordination

Some protocols aim to reduce checkpointing overhead by eliminating dynamic coordination amongst processes. Since the coordination ensures a consistent state, these protocols must do so through other means, such as

through message logging or requiring the user to ensure correctness. We discuss the drawbacks of these techniques next.

In message logging, processes write their messages to stable storage so that, if necessary, the messages can be replayed during recovery to create a consistent state. Checkpointing techniques that use message logging assume that the application has enough unused memory on the local hardware to store message logs, an assumption that is not true for many applications [158]. For instance, both Adaptive MPI [52] and the protocol integrated with MPICH-V2 [36] allow processes to checkpoint at any time but also require processes to log their sent messages. Both the checkpoints and logs are initially saved in memory.

Protocols that require the user to guarantee correctness place an undue burden on the application programmer, who often resorts to synchronous checkpointing, during which all processes checkpoint at the same time and thus generate file system contention. Many of the techniques that depend on the user then try to reduce this contention through other means. For instance, some of these techniques without dynamic coordination try to reduce this contention by saving each process's checkpoint to the hard drive of another machine, as is done by the coordinated protocol BLCR. The protocol developed by the Pittsburgh Supercomputing Center [199], for example, uses this methodology. Such techniques are not applicable to systems without local storage.

Another protocol, *cooperative checkpointing* [152], concentrates on re-

ducing file system contention across all applications executing on the system, by preventing an application from checkpointing when traffic on the network or at the file system is already high. It allows the runtime system to skip the checkpoints requested by the application based on various criteria, including network and file system usage. Cooperative checkpointing does not, however, reduce the contention caused by all processes checkpointing synchronously; it makes no attempt to stagger the checkpoints.

2.1.3 Compiler-Assisted Checkpointing

Compiler-assisted checkpointing techniques move some burden from the user and/or the runtime environment to the compiler. This burden can include: reducing the size of the checkpoint, identifying checkpoint locations, or ensuring that a consistent state is saved. Many techniques exist to reduce the size of the checkpoint, but they are largely orthogonal to the research discussed in this section and thus are covered in Section 2.1.4. Other approaches, including ours, move the latter two tasks to the compiler. For instance, compilers are used to identify communication-free areas for processes to checkpoint [58, 174], which both ensures a consistent state and identifies checkpoint locations. For these checkpointing techniques to reduce contention at the file system, applications must have communication-free zones with sufficient room to separate the process checkpoint locations in time. However, many applications do not have sufficient communication-free zones.

Another checkpointing protocol, application-level coordinated non-blocking

checkpointing [43], requires the user to choose potential checkpoint locations, but the user can do so without regard to the consistency of the resulting state. The chosen locations are annotated by the compiler so that a consistent state is ensured at runtime through message logging. An extension to this approach also identifies the data to be saved and saves only live data [228]. Regardless, this approach requires message logging and results in all processes checkpointing at approximately the same time, which can lead to contention at the file system [146].

Compilers can also ensure that the checkpoints from a consistent state in the face of communication, our solution takes this approach. Another such method uses the compiler to analyze application communication and then adjusts the previously placed checkpoint locations to ensure a consistent state is saved [14], but it makes no attempt to stagger the checkpoint locations and results in all processes checkpointing at approximately the same time.

2.1.4 Other Techniques

There are some software packages that help the programmer save the checkpoint data without addressing at what location(s) the processes should save its checkpoint; they can be used in conjunction with techniques that identify checkpoint locations. They include DMTCP [22], TICK [79], and adaptive incremental checkpointing [12]. One such technique does reduce the likelihood of contention at the file system, but it requires a lightweight storage architecture and overlay network [147]; our method does not require particular

hardware.

2.2 Simulation of Large-Scale Systems

In this section, we review work related to simulation of large-scale parallel applications. Our infrastructure simulates long-running parallel applications that use many processes. In addition, our simulator shows the effects of network and file system contention, which we care about since our goal is to evaluate staggered checkpointing. It is unique in its ability to efficiently simulate applications using thousands of processes without detailed knowledge of the execution environment or application. In this section, we evaluate other solutions based on their ability to accomplish these goals.

There are simulators that can also simulate parallel applications, but they are not scalable. For instance, the developers of the Structured Simulation Toolkit [209] intend to allow the simulation of a system at many levels of granularity, but its current implementation uses SimpleScalar for cycle-accurate simulation, which is too detailed for our goals. Susukita et al. [202] recently introduced a simulator that can rapidly simulate applications on large-scale systems using a network model, a cycle-accurate simulator, and detailed knowledge of the application. PHANTOM [230] simulates applications a process at a time by replaying message logs on a single node of the target machine—a technique that does not scale well to applications using large numbers of processes.

Many other simulators of large-scale systems are scalable but require

intimate knowledge of the application [38], the target system [84, 168, 188], or both [212]. For instance, simulators that use the LogGP framework to estimate network performance require much detail about the network [27, 104].

Other simulators use models to estimate application performance, but they use a single analytical model to predict the performance of the entire application [28, 93, 112, 134]. This method is not appropriate for evaluating checkpointing policies, since the interaction of the application, local hardware, file system, and network is not exposed.

Parallel Simulators. Some simulators use parallelism to speed up simulation. These parallel simulators require a large number of processors and low memory usage per application process. For instance, MPI-SIM [165], a parallel simulator, requires the privatization of global variables, which can use a large amount of memory and thus limits the number of simulated processes that can be mapped to each processor being used by the simulator. A simulator based on MPI-SIM [11] has successfully reduced this memory usage through static analysis techniques that eliminate some data structures, but this technique still requires a large number of processors. Parallel simulators built on Charm++ [177, 234] estimate application performance by assigning many processes to each node, thus reducing the number of processors necessary but still requiring low memory usage. Another existing parallel simulator [154] can simulate large-scale parallel applications but its simulation time is slower than the application's execution time.

Table Lookups. Prior simulators estimate the performance of the supercomputing system inexpensively by using tables of collected data. Table lookup limits the ability of the simulators to extrapolate for other configurations and predict the effects of contention, unlike our method, which quickly approximates the timing of application operations using models of the network, file system, and local hardware that encapsulate the performance of each component.

For instance, ChronosMix [198], a simulator built to estimate parallel application performance on networks of workstations, uses table lookup to approximate local machine and network performance. (It does not model the file system.) Another such simulator is the one introduced by Snavely et al. [193]. It also uses table lookups to estimate both local machine and network performance and does not include file system performance. This simulator also requires communication traces from the application, and those traces must result from executions using the same number of processes as the simulation.

2.3 Summary

To summarize, many checkpointing techniques exist but none both statically guarantee a correct state in the face of communication and separate the process checkpoints to reduce network and file system contention. Our compiler-assisted staggered checkpointing technique accomplishes both of these goals.

In addition, many simulators address one or more of our concerns, but

none addresses the simulation of large-scale applications with checkpointing on supercomputers without user knowledge. Our simulator is capable of simulating these applications.

Chapter 3

Simulation for Large-Scale Systems

The key to evaluating staggered checkpointing is the ability to simulate long-running applications executing on large-scale supercomputing systems. Simulation gives us the ability to assess staggered checkpointing on many systems, including those which have not yet been implemented and those to which we do not have access or have limited access.

To evaluate staggered checkpointing, the simulator must model the entire system, which includes the local hardware, network, and global file system; a representation of a supercomputing system is shown in Figure 3.1. In addition, we want a simulator that requires little to no information about the application other than the source code itself. Existing simulators do not meet these criteria and are thus unsuitable for our evaluation of staggered checkpointing. For instance, cycle-accurate simulators, such as SimpleScalar [26] and M5 [178], are appropriate for microarchitecture design of small applications but would require too much execution time for our analysis at our scale. SimpleScalar requires approximately an hour to simulate one second of execution for a single process, and M5 needs approximately three hours to simulate

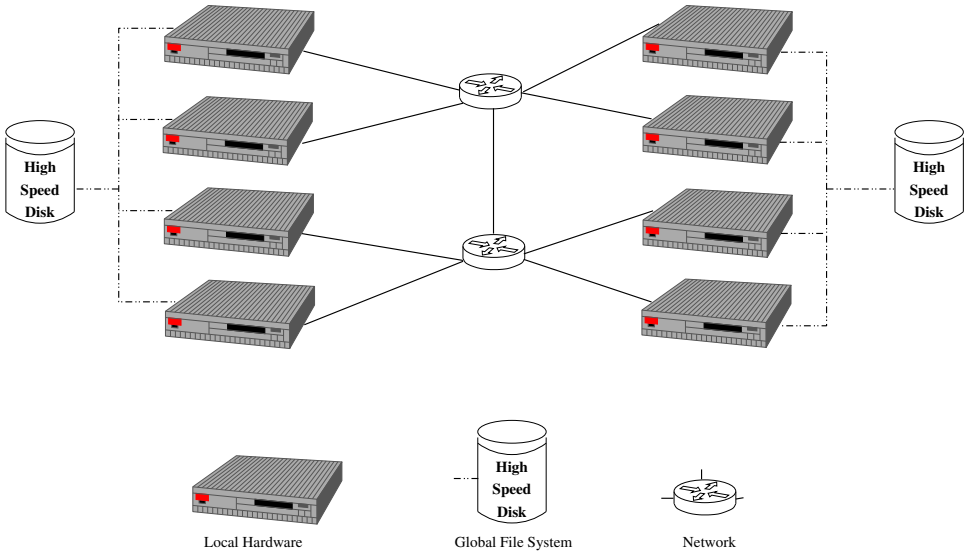


Figure 3.1: Example of a Typical Cluster

that single second.[1] The candidate that comes closest to meeting our needs is a recently published simulator from the Institute of Systems, Information Technologies, and Nanotechnologies in Japan [202]. This system can rapidly simulate applications configured to use thousands of processes, but it requires the user to identify the kernel of the application being simulated. In addition, it does not model either the file system or contention on the network, both of which must be considered to evaluate staggered checkpointing. Since no existing simulator can perform our desired analysis, we must develop our own.

[1]We estimated these numbers using SimpleScalar's simulation time for an instruction profiling simulation and M5's simulation time for a detailed, single core simulation.

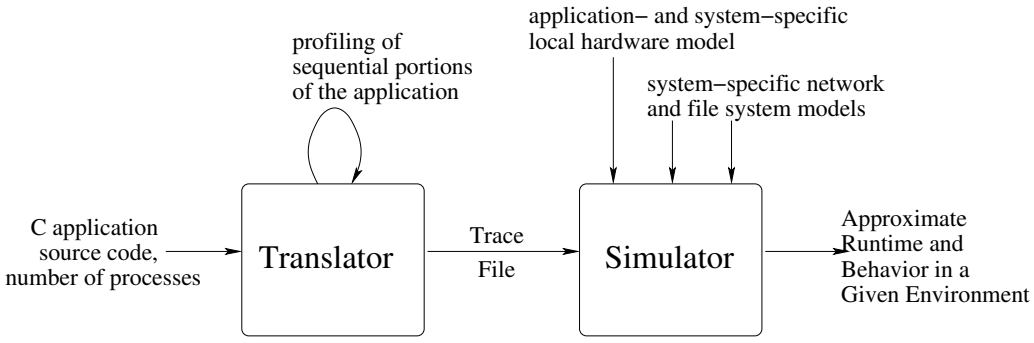


Figure 3.2: Simulation Infrastructure

Solution. Our simulator uses models of the network, file system, and local hardware to quickly approximate the timing of application operations. These models are key to the scalability of our simulator: they encapsulate the performance of these system components, including the effects of contention, into a single calculation the provides a fast answer regarding the cost of an event. Thus, our simulator estimates the system consequences of a change in checkpoint strategy efficiently and encourages experimentation with approaches to checkpointing. In addition, it can be used to analyze parallel program performance and scalability on large parallel computing systems.

The remainder of this chapter further describes our simulator and why it works. We present our design and implementation decisions and analyze its efficiency and accuracy.

3.1 Overview of the Simulator

As shown in Figure 3.2, our simulation system has two parts:

1. the *translator*, which translates the application source code into a sequence of events, and

2. the *simulator*, which accepts traces of events and simulates the application at event-level granularity.

Our simulator estimates the execution time of an application by simulating three pieces of the computing system: the file system, the network, and the local hardware. Quickly approximating the performance of these components is challenging because they perform complex tasks in a complex execution environment; we use closed-form expressions to represent the behaviors of these components in the simulator. These expressions encapsulate each component's complexity into a single calculation: a formula that calculates the performance of that component under a given workload. For instance, the network model provides the simulator with performance information for the entire network and thus eliminates the need to separately consider the performance of each cable, switch, and router. Similarly, the local hardware model encapsulates the performance of the processor(s), their cores, the memory subsystem, and all other pieces of the local hardware. The local hardware model represents the behavior of a particular application on that hardware, so a new model is required for each application, though not for each configuration of that application. These models are further discussed in Section 3.3.1.

Our translator is implemented in Broadway [90], a source-to-source C compiler. It converts applications that are written in C and use the Message

Event	Arg1	Arg2	Arg3	Explanation
Send	p2	p3	1MB	p2 sends a 1 MB message to p3
Receive	p3	p2		p3 receives a message from p2
Local	All	12		All processes execute 12 local operations
Local	p1	45		p1 executes 45 local operations
File System Write	p0	4MB		p0 writes 4 MB of data to the file system

Table 3.1: An example of the representation of events in the trace file.

Passing Interface (MPI) [196] for all communication into representative traces. Although our translator assumes C and MPI, our algorithms are extensible to other languages and other explicit communication libraries. However, the applications do need to be written using a Single Process, Multiple Data (SPMD) model. Our simulator is independent of the translation process and simulates any application in the recognized trace format.

The next sections explore the translator and the simulator in detail.

3.2 Translator

The translator accepts the source code of an application written in C, analyzes it, and converts it into a trace accepted by our simulation; the generated trace is specific to that application and to the given number of processes. Each event in the trace is specified by type, the processes that execute it, and any additional necessary information, such as data size or the number of instructions represented. Event types fall into three categories:

1. network events, which are any type of communication event;

2. file system events, which are currently limited to writes since we focus on the writing of checkpoints; and

3. local events, which consist of all other operations.

An example of a trace file can be seen in Table 3.1.

To convert an application to a trace, the translator performs a context-sensitive analysis of the application source code. This analysis includes: categorizing each operation as a network, file system, or local event; identifying the process(es) that executes each operation; and calculating how many times each operation executes in each context. For this analysis, the translator requires the application source code and the number of processes that are to be used in the execution of that application. It represents the application internally as a control flow graph, so every operation is in a basic block; in addition, the operations in the blocks are represented in our compiler's low-level intermediate language. Each pass of our translator is over this control flow graph.

We now explain each step further.

Step One: Categorizing Operations. As the translator performs a context-sensitive pass over the application source code, it categorizes each operation into one of the three event types recognized by the simulator: 1) network, 2) file system, or 3) local. Any MPI-like call, such as communication or process synchronization, is classified as a network event. All checkpoint or file write

operations are file system events, and all other operations are local events—including all computation, library calls, and function calls.

Step Two: Identifying the Executing Process. As the translator analyzes each operation, it must also determine which processes execute that operation; in the parallel applications we consider, any number of processes could execute each operation. Which processes execute an operation depends on the control flow directing that operation. Thus, our translator identifies each operation's executing process by analyzing the control flow directing that operation. Our translator collects all conditional statements that both rely on the process identifier and affect a particular block. It symbolically executes these conditions to determine which processes execute that block. It performs this analysis for each block in each context; context is determined by the path the execution takes to arrive at that block.

For any communication operations, the translator also uses symbolic execution to determine any processes on the opposite end of the communication.

For both of these symbolic analyses, our translator is limited to knowledge it can determine statically. Thus, our translator assumes that values resulting from system calls or from communication do not appear in the control flow.

Step Three: Calculating the Execution Count. To create an accurate trace, the translator calculates the execution count for each operation in each context. The analysis must account for loops, conditional statements, and variation amongst different processes in the system. Like the analysis used to identify executing processes, this analysis uses a form of symbolic execution. Since this analysis is time and space intensive, we reduce the number of times it must execute by dividing the blocks into two categories: those that are *process-specific*, or whose execution count depends on which process performs the execution, and those blocks that are *process-independent*, or whose execution count is the same for all blocks. Only the former category requires the expensive algorithm to determine the execution count; we use profiling to calculate this information for the latter category. We next discuss the symbolic execution and profiling stages in more detail.

Symbolic Execution. Symbolic execution is used for all process-specific blocks. The symbolic execution algorithm we use here is similar to the algorithm used to identify the executing processes for each block. However, unlike the executing process algorithm, all control flow affecting a block must be collected and analyzed, not just the control flow that relies on the process identifier. For each process-specific block, then, the translator gathers all affecting control flow and performs symbolic execution to determine the execution count. It assumes that all loops in the gathered control flow have a definite trip count.

Since the control-flow affecting the block relies on the process identifier, it is likely that the execution count of the block is dependent on the process executing it. To increase our accuracy while preserving our efficiency, the translator randomly selects some number (*e.g.*, 5) of the processes that execute that block and determines that block's execution count for each of the processes. It keeps a running average and, if at the end of the calculation for that number of processes the average has a stable value, that average is considered the execution count for the block. If the average is not stable, more processes are added to the average until either the average is stable or all processes have been included in the average.

Profiling. The translator uses a single profiling run to determine the execution count for all process-independent blocks: since the execution count for these blocks is independent of the process performing the execution, a single execution that mimics that of a single process provides the execution count for all processes. To conduct the profiling, the translator instruments each process-independent block with code to track the number of times it executes. It also analyzes the initial, end, and step conditions of each loop and, if possible, increases the step to reduce the number of times the loop executes. It writes all the instrumented blocks to a file. The translator then compiles the file, executes it on the local machine, and collects the profiling information. For each block, the execution count is divided by the number of contexts in which that block executes, enabling the translator to determine

loop trip counts and how often each branch of a conditional statement executes.

Step Four: Creating the Trace. After calculating the execution count for each block, our translator then performs a second context-sensitive pass over the application source code, converting each operation into an event using the information gathered during the first pass. The translator writes each event to a trace file. This process is basically straight-forward, but we do employ some optimizations, which we now describe.

Aggregating Local Operations. The translator aggregates consecutive local operations executed by the same process into one event representing many local operations. In fact, the translator eliminates function boundaries; thus, it can aggregate local operations from a larger span of source code—and the resulting trace is context-insensitive, which reduces the amount of work our simulator must perform and increases its efficiency. This collapse reduces the size of the trace file and maintains accuracy since each local operation is still represented.

Optimizing Loops. Our translator recognizes two types of loops and handles each differently: (1) loops containing process-specific operations *or* operations of any type other than local and (2) those containing only process-independent local operations. Loops of the former type are maintained and represented in the trace file. Loops of the latter type are collapsed into a

single local event representing all operations contained in the loop and their execution count; the resulting event may be further aggregated with other consecutive local operations. Recall that our translator determines the trip counts for the loops when it calculates the execution count for each block in its first pass.

Handling Conditionals. Blocks affected by conditional statements will be handled in the same manner as other blocks: the events contained in the affected blocks are weighted based on the execution count of that block. For example, if the *true* block of an `if` statement executes 60 out of the 100 times the condition is executed and the *false* block executes the other 40, the true events are weighted by 60, the false events by 40, and the condition itself is weighted by 100.

3.3 Simulator

The simulator, which executes with a single thread of control, estimates the execution time for an application using a trace of that application and models that estimate the performance of the network, global file system, and local hardware for that application in the execution environment being simulated. In this section, we present the simulator design and its implementation, beginning with the models.

3.3.1 Modeling System Components

Our simulator uses models to approximate the costs of network, global file system, and local hardware events. These models, which are closed-form expressions, allow our simulator to perform coarse-grained simulations and still approximate the complex behavior of the components of the execution environment—including such behaviors as contention at the network, file system, and memory hierarchy. In addition, these models provide the ability to approximate the behavior under varying and unanticipated workloads.

Each model does represent all costs of the execution of an event of its type. For example, the network model includes all interactions of the network and the cost of any buffering on the local hardware. Similarly, the local hardware model encapsulates the performance of the processor(s), any cores, the memory subsystem, and all other pieces of the local hardware involved in the execution of a local event.

These models are unique to their represented component, so if that component is replaced, the model should be changed, either by the creation of a new model or by modification of the existing model. The local hardware model is also unique to a particular application on a particular hardware, so a new model should be provided for each application on each local hardware configuration.

We develop these models from data generated by performing experiments measuring each event's behavior under a wide range of workloads on

a parallel machine. The experiments reported here are performed on two systems, both at the Texas Advanced Computing Center (TACC): Lonestar [204], which uses Dell PowerEdge 1955 blades for its local hardware, and Ranger [205], which uses SunBlade x6420 as its local hardware. Both systems have Infiniband interconnects [203] and use Lustre [144] for the global file system.

We use regression analysis to separately analyze the data resulting from each event type. We use Mathematica 7.0 [172] to determine the constant values for a function we supply. We develop each function by identifying likely variables and examining the resulting values to determine the significant ones, and then we remove the insignificant ones and model the data again with the new variables. The final function is the model we use for that event type.

Below, we further discuss the local hardware, network, and file system models. We present an error analysis with each model, so we first discuss how we calculate error.

3.3.1.1 Calculating Error

To assess the accuracy of our simulator, we use accepted statistical methods for data distributed normally. We first calculate the ratio of the time predicted by our simulator to the time we measure; this ratio is less than 1 for times that are underpredicted and greater than 1 for numbers our simulator overpredicts. These ratios are not normally distributed, but their natural logs have a typical normal histogram, closely follow a straight line on

normal probability paper, and easily pass goodness of fit tests for normality. Thus, we use statistical methods regarding normal data on the natural logs of these ratios.

For each data set, we calculate the average, standard deviation, 95% confidence interval around the mean, and a tolerance interval indicating the range for values of 95% of the population with 95% certainty. This accuracy information applies to the natural logs of our ratios, so we take the anti-log of each result. These anti-logs now represent the accuracy of our simulator in the form of ratios.

When we report simulated times, we apply the average error to each time; this number indicates the likely time given the error of that data set. In addition, we calculate the confidence interval around each time using the lower and upper bound ratios for confidence. We perform the same calculation with the lower and upper bound ratios for tolerance to get the tolerance interval around each time.

When we calculate the difference between two simulated times, we also calculate the difference between the times with the average error applied, and then we calculate the confidence interval. Our original confidence interval applies to the simulated times and not the difference, so we calculate a new confidence interval by deriving the standard deviations from the confidence intervals surrounding the two times being compared, σ_1 and σ_2. Using these standard deviations, we calculate the standard deviation of the difference, σ_3, using the following formula:

$\sigma_3 = \sqrt{\sigma_1^2 + \sigma_2^2}$.

We multiply σ_3 with the factor for the 95% confidence interval, which is typically 1.96 for our sample sizes, to find the distance of the interval on either side of the mean. Subtracting this distance from the mean provides the confidence interval lower bound and adding it to the mean provides the confidence interval upper bound. We calculate the tolerance interval similarly, but we replace the confidence interval factor with the tolerance interval factor. The tolerance interval factor varies for each data set based on the number of elements, the average, and the standard deviation.

3.3.1.2 Local Hardware Model

Simulating local hardware performance is made challenging by the need to simulate the memory system. Memory systems have complex behavior that is dependent on the memory footprint and access actions of each application. For example, an application whose memory references are mostly found in the L1 cache has a different performance profile than an application whose memory references are mostly found in the L2 cache, even if the number of local operations is identical. Here we make a tradeoff: to accurately represent such behavior across applications, our simulator would need to track the state of the memory system—a level of detail that would dramatically increase simulation time. Instead, we create application-specific local hardware models that estimate costs of local events, including those associated with the memory system.

The local hardware model is developed from experiments measuring the performance of many data configurations of an application using a small number of processes (say, 4). The model's parameter is the application data configuration. For many applications, the data configuration is related to the problem size, but it can be any number that reflects the size of the data set of each process; this number should relate to the memory footprint of the application. In our methodology, we discuss how we determine this value for our benchmarks. Using this parameter, the local hardware model calculates an average cycle cost per local operation for a particular application; this result is applied to all local operations in that application.

Assumptions. We assume that averaging the cycle cost for all local operations in the application reasonably estimates the cycle cost per local operation for smaller segments of the application, which implies that the local hardware performance for different segments of computation are uniform. This assumption results in sufficient accuracy even though the dominant type of operation may vary between different program segments since memory latency dominates performance for these applications and overwhelms performance differences between other hardware operations

Methodology. To create the local hardware model, we modify the application source code to include only local operations and exclude any operations that would be considered file system or network events, such as communication

calls or file system writes. The modified source code is input to the translator, which creates the corresponding trace, from which we tally the number of local operations executed by the application. The modified source code is also compiled and executed on the target system, and the number of cycles required for the execution to complete is measured. Using the number of local operations and the number of cycles they require to execute, we calculate the average number of cycles consumed for each operation. We gather this data using a range of data configurations with a single system size, and we then use regression analysis to generate a closed-form expression that correlates application data configuration information with the average number of cycles per operation.

For this methodology to succeed, it is important to correctly represent the data configuration of the application. We determine this number for each of our benchmarks by understanding how the major data structures scale as the data configuration changes. For the NAS parallel benchmark `LU` [51], the data structures scale directly with problem size. For `BT` and `SP`, also NAS parallel benchmarks, and `Ek-Simple`, an application benchmark, the data structures scale with the problem size divided by the square root of the number of processes. These data configurations are used as parameters to their respective models.

In addition, error can be introduced if the appropriate data configurations are not considered in the model. The data configurations should represent the likely use of the application. For instance, if the data configurations

are too small, and they do not properly tax the memory subsystem, then the prediction will be less accurate for larger data configurations.

Resulting Model. For Lonestar and Ranger, the local hardware performance data for our application benchmarks shows the functional form of exponential rise to a plateau—so the user-supplied seed equation to Mathematica uses exponential rise to a plateau. To further aid Mathematica, we determined the correct signs and perturbations for the equation to be in the correct quadrant and form the right shape. The resulting equation has the form:

$$a^2 * e^{-b*data_configuration+c} + d$$

where $data_configuration$ is a parameter to the model, and a, b, c, and d are values determined by Mathematica. An example model can be seen in Figure 3.3, which shows the local hardware model we derived for the NAS parallel benchmark `BT` [51] on Lonestar. This model has an average error of 1.02 for configurations requiring at least thirty seconds to execute.

For the local hardware models for our benchmarks on Lonestar, the average error is 1.02. On Ranger, the average error is 1.0. A full reporting of the errors for both Lonestar and Ranger is included in Tables 3.2 and 3.3. These tables also report the confidence and tolerance intervals.

Predictive Power. To better understand the predictive power of our models, we use a subset of our measured data on Lonestar to form a model, and

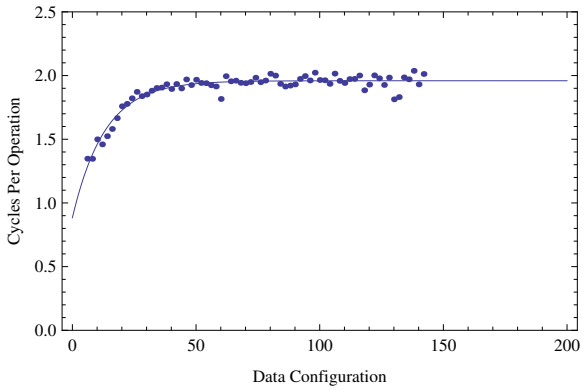


Figure 3.3: `BT`'s local hardware model compared with its measured average cycles per operation for Lonestar. The dots represent the measured data and the line represents the generated model.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.02	1.01-1.02	0.97-1.07
LU	1.01	1.00-1.01	0.99-1.03
SP	1.03	1.03-1.03	0.99-1.08
Ek-simple	1.03	1.03-1.04	0.99-1.08
Synthetic	1.02	1.02-1.02	0.98-1.07

Table 3.2: Error for the local hardware models on Lonestar.

then use that model to predict the application performance for the rest of the measured data. For instance, for the benchmark `BT`, we use the first three-quarters of configurations to create a model and then predict the last quarter. The resulting average error is 1.02, and can be seen in Table 3.4. This low error demonstrates the effectiveness our methodology.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.00	1.00-1.01	0.95-1.05
LU	1.00	1.00-1.00	0.96-1.05
SP	1.02	1.01-1.03	0.91-1.15
Ek-simple	1.00	1.00-1.00	0.98-1.02
Synthetic	1.00	1.00-1.00	1.00-1.00

Table 3.3: Error for the local hardware models on Ranger.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.02	1.01-1.03	0.96-1.08
LU	1.00	1.00-1.00	1.00-1.00
SP	1.03	1.02-1.04	0.98-1.09
Ek-simple	1.02	1.02-1.03	0.99-1.05
Synthetic	1.07	1.06-1.10	0.97-0.99

Table 3.4: Error for the sub-models of the local hardware models.

3.3.1.3 Network Model

The simulator uses the network model to calculate the bandwidth available to each process for message sends. The model requires the specification of number of processes sending messages simultaneously and the total amount of data to be sent in order to calculate the bandwidth.

Assumptions. The network model assumes that all costs of a network send can be calculated with the total amount of send data and the number of processes sending. It also makes the simplifying assumption that bandwidth is affected by the variance of those parameters regardless of how that data is distributed amongst the processes. The latter assumption results in the same

approximation for 16 processes sending 2 MB of data each as for 15 of those 16 processes sending 1 MB of data each and the single process sending 17 MB of data.

The network model also assumes the contention model is asymptotic: processes always make some amount of progress even when the network is saturated.

Methodology. The network model is created using experimental data resulting from the Intel MPI Benchmark Multi-PingPong [101]. We execute this benchmark with many configurations: each configuration measures the number of microseconds required for a particular number of processes to send and receive a single data size. Each process has a communication partner process; as each primary process sends its data, its partnering process receives that data and then returns it, allowing the primary process to complete its receive. At the end of each experiment, the results from the individual process pairs are gathered and the average and standard deviation calculated. If the standard deviation is larger than a percentage of the average, any outlier results are discarded and the average is recalculated until it converges. If the number of results falls too low, the experiments are performed again. These experiments are performed on a system simultaneously in use by other researchers, which helps us ensure that our models reflect typical performance.

Resulting Models. To model each network, we use two sub-models. For each machine, one sub-model represents all message sizes less than or equal to 16KB and the other represents the message sizes larger than that. This division represents the change in MVAPICH [7], which is the form of MPI used on Lonestar and Ranger, from the Eager protocol to the Rendezvous protocol. The equations representing the four sub-models are unique, but each uses num_sends and $totalSize$ as parameters, where num_sends represents the number of sending processes, and $totalSize$ represents the total amount of data to be sent. The constant values are determined by Mathematica. An example of the performance of our network models is shown in Figure 3.4, and the error for these models can be seen in Table 3.5. This figure represents the network model for Lonestar.

Predictive Power. The Lonestar network model is created using 128 process configurations, each with many data sizes. We create a sub-model using 96 of those process configurations and use it to predict the remaining configurations. This results in an average error of 1.03, a confidence interval of 1.00-1.06, and a tolerance interval of .70-1.39. The sub-model typically over-predicts the measured values, but its low error demonstrates the effectiveness our methodology.

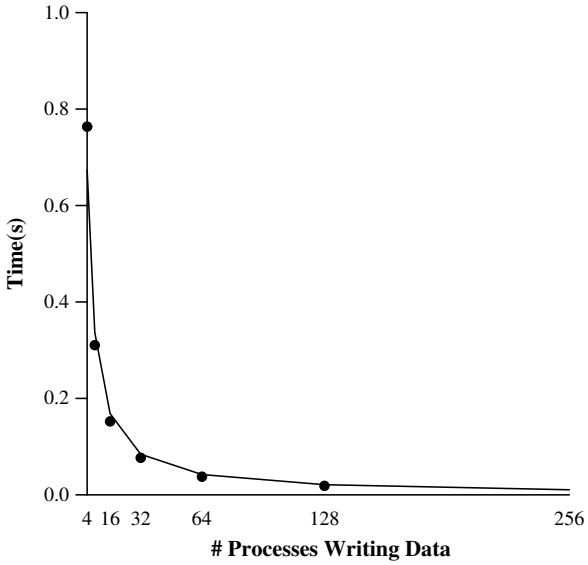


Figure 3.4: Network Model: This graph plots the measured (points) and predicted (line) time (y-axis) for processes (x-axis) sending a total data size of 2GB on Lonestar. The measured data ends at 128 processes, but the predicted data continues to 256 processes.

3.3.1.4 File System Model

The simulator calculates the file system bandwidth available to each process using the file system model, which uses the number of processes writing simultaneously and the total amount of data as input parameters.

Machine	Average Error	Confidence Interval	Tolerance Interval
Lonestar	0.99	0.97-1.01	0.70-1.39
Ranger	1.16	1.12-1.20	0.70-1.94

Table 3.5: Error for the network models.

Assumptions. Our file system model assumes that all costs of a file system operation can be modeled using the total amount of data being written and the total number of processes writing that data. It also assumes that the file system performance is independent of how the data is distributed among the processes.

Like the network model, the file system model assumes the contention model is asymptotic.

Methodology. The file system model results from the analysis of experimental data collected using a benchmark that timed each process packing and writing some amount of data to the global file system. In this benchmark, each process writes to a unique file during each test. The rest of the methodology follows that of the network model.

Resulting Models. The file system models each use two sub-models, one for writes less than or equal to 4MB, and the other for larger writes. 4MB is the measured optimal transfer size on the system and the transfer size we used during the file system benchmark tests. As a result, all writes less than 4MB require a single transfer and writes above that size require more transfers. During simulation, the simulator uses the file system sub-model that represents the maximum data size being written by a single process. The equations are different for each sub-model, but all function on the input parameters num_writes and $totalSize$, where num_writes represents the number of writing processes,

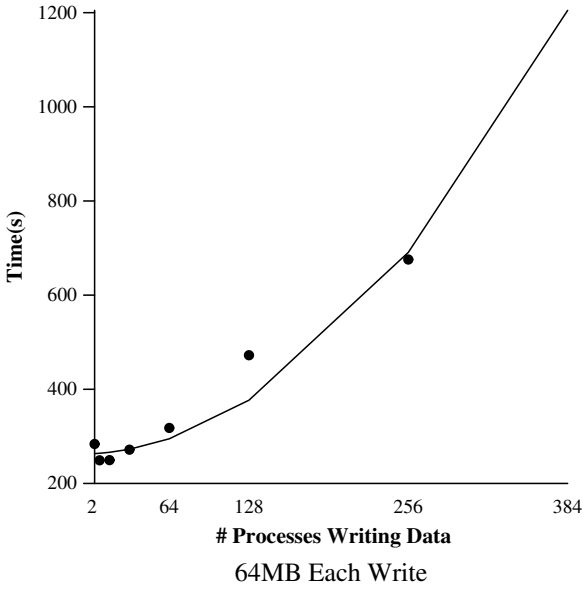


Figure 3.5: File system model: This graph plots the measured (points) and predicted (line) amounts of time (y-axis) spent writing 64 MB messages by various numbers of processes (x-axis) on Lonestar.

totalSize represents the total amount of data being written. The constant values are determined by Mathematica.

Figure 3.5 shows the performance function of our large data size sub-model for Lonestar. In this figure, the message size, 64 MB, remains constant as the number of processes increases, so the total amount of data increases as the number of processes increases. It indicates the accuracy of our file system model, which is less accurate for smaller numbers of processes than for larger but is quite accurate overall.

The average error for the Lonestar file system models is 0.97 over all data and process sizes. On Ranger, the average error is 1.03. A full reporting

Machine	Average Error	Confidence Interval	Tolerance Interval
Lonestar	0.97	0.92-1.01	0.70-1.34
Ranger	1.03	1.00-1.06	0.98-1.26

Table 3.6: Error for the file system models.

of the errors can be seen in Table 3.6.

Predictive Power. We use the measured data on Lonestar up to and including 160 processes to develop a model, and then we predict the performance of the file system up to and including 256 processes. The average prediction error is 0.84. The confidence interval is 0.76-0.94, and the tolerance interval is 0.39-1.83. This manageable error gives us confidence in our methodology.

3.3.2 Event Simulation

Here, we present the simulation details of local events, some message sends and receives, file system writes, and process synchronization.

Initialization. At the beginning of a simulation, the simulator reads the events in the trace file into an event queue. The simulator also maintains state throughout the simulation for each simulated process. Each process state includes: a clock, which represents elapsed cycles in the simulated execution; an index into the event queue, which acts as a program counter; and a status, which is either `ready` or `waiting`. `Ready` indicates the process's next event can be simulated. `Waiting` indicates the process is waiting for a network or

Index:	23	24	25	26	27
Event type:	Send	Receive	Local	Local	Filesystem Write
Executed by:	p2 –> p3	p3 <– p2	All	p1	p0
Number/Size:	1 MB		12	45	4MB

Partial Event Queue (showing events 23–27)

Send queue

Write queue

Status Information

process	clock	index	status	Incoming msg queue
0	22	23	ready	
1	94	23	ready	
2	56	23	ready	
3	73	23	ready	

Figure 3.6: Simulation Example: Slice of Execution

file system event to complete or on an event from another process.

For each simulation step, the simulator identifies the process with the earliest clock value that has a `ready` status. It finds the next event that process executes and simulates it. The engine's next action depends on the type of event to be simulated.

Example: Initial State. Figure 3.6 presents the state for an example simulation. It shows a representative slice of an application's event queue, beginning at event 23, and the state of each of four simulated processes. In this example, all processes are ready to execute event 23, where the slice of the event queue shown begins. Process 0's clock is less than 23, so perhaps it

Index:	23	24	25	26	27
Event type:	Send	Receive	Local	Local	Filesystem Write
Executed by:	p2 –> p3	p3 <– p2	All	p1	p0
Number/Size:	1 MB		12	45	4MB

Partial Event Queue (showing events 23–27)

Status Information

process	clock	index	status	Incoming msg queue
0	82	26	ready	
1	94	23	ready	
2	56	23	ready	
3	73	23	ready	

Send queue

Write queue

Figure 3.7: Simulation Example, Step 1: The state after p_0's local event at index 25 has been simulated. p_o does not execute the events at indices 23 and 24. Notice p_0's clock and index have been updated.

did not execute all the preceding events.

Local Events. Local events are the easiest events to simulate, since their execution is local to and thus only affects the simulated process. To simulate a local event, the simulator uses the local hardware model, which is explained is Section 3.3.1.2, to calculate the cycles spent executing that event. It adds that number of cycles to the executing process's clock and increments its index.

Example: Local Event. In our example in Figure 3.6, *process 0*, or p_0, has the lowest clock value and its status is `ready`. So the next event

executed by p_0 is simulated now. For this example, it is the local event at index 25, because p_0 is not involved in the events at indices 23 and 24. The event at index 25 is executed by all processes and represents 12 local operations from the application source code. However, at this time, p_0's execution is the only execution being simulated.

For the purposes of this example, we assume that each local operation averages five cycles of execution time. So, for 12 local operations, the local hardware model returns 60 cycles. Those 60 cycles are added to p_0's clock, so it becomes 82. p_0's index is incremented to 26. This new state is represented in Figure 3.7.

Network and File System Events. Network events represent any action that takes place over the network. These actions include message sends, receives, and process synchronization. File system events, which represent actions on the global file system, are currently limited to writes. This choice assumes that reads equally affect all checkpointing policies and thus do not affect the evaluation of those policies. Here, we present the simulation details of message sends and receives, file system writes, and process synchronization events.

Network Event: Send. When the simulator encounters a network send event, it places the message in a global send queue. For simulation efficiency, we only simulate message transmission periodically, since that limits

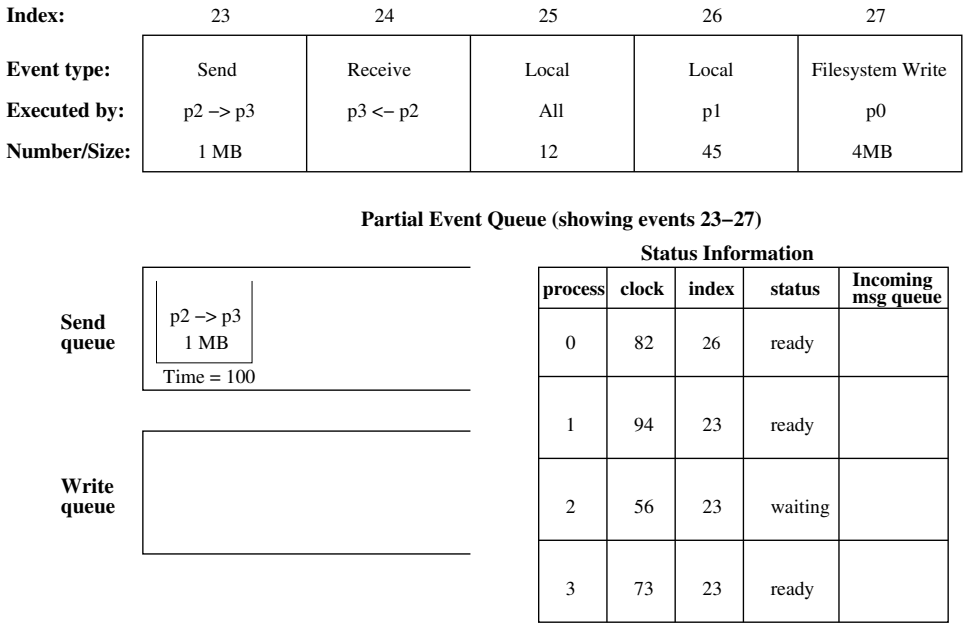


Figure 3.8: Simulation Example, Step 2: The state after p_2's network send at index 23 has been initiated. The send has been placed in the appropriate send queue and is waiting to be sent.

how often the bandwidth available to each process varies and allows sends to make progress with a minimal number of bandwidth recalculations. The global send queue is a collection of buffers, each associated with a single send time. The simulator adds the message to be sent to the buffer representing the next send opportunity, which is calculated based on the sending process's clock. If the send is a blocking send, the sending process's state is changed to `waiting`; otherwise, the process's index is incremented and the effect of the send on the sending process is complete.

Each send buffer's messages are sent when every process has either ad-

vanced past the associated time or has a status of `waiting`. When the message send completes, the simulator places the message receipt information, which includes the sender and the time of receipt, in the receiving process's incoming queue, and the receiving process's state is set to `ready`. For a blocking send, the sending process's state is returned to `ready`, its index is incremented, and its clock is adjusted to reflect the time the message finished sending.

Example: Encountering a Network Send. In our example simulation in Figure 3.7, the simulator next executes p_2's event 23, a blocking network send of a 1 MB message to p_3. When the simulator encounters this event, it calculates the next send time and places p_2's message in the appropriate send buffer. The simulator then changes p_2's status to `waiting`. In this example, the network and file system queues are being processed every 100 cycles. Since p_2's clock is 56, the next opportunity to send will be at time 100, so p_2's message is placed in the appropriate buffer. The new state is shown in Figure 3.8.

Network Event: Receive. When the simulator encounters a message receive event, it checks the receiving process's incoming message queue for the expected message. If the message is not present, the process's index remains unchanged, its status is changed to `waiting`, and the receive simulation is attempted again the next time that process's execution is simulated. If the message is present in the incoming queue, then the receive completes. To

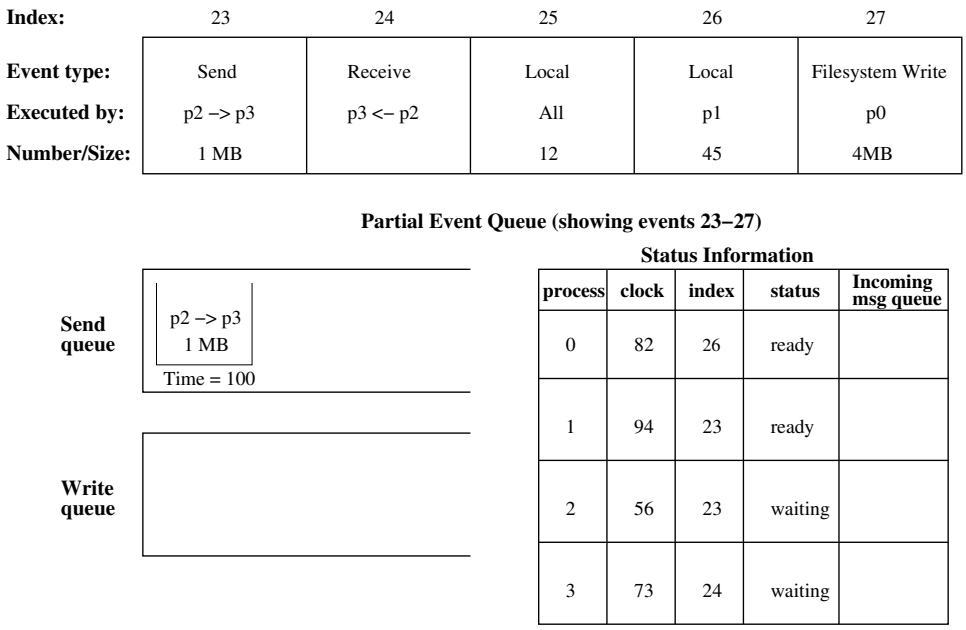


Figure 3.9: Simulation Example: Step 3 Network Receive

complete the receive, the simulator increments the receiving process's index and sets that process's clock to the maximum of its current time and the time of the message receipt.

Example: Encountering a Network Receive. In Figure 3.8, p_3 is the ready process with the lowest time clock, so its execution is simulated next. p_3's next event is a blocking message receive from p_2 at index 24. The simulator executes this event by checking p_3's message queue, which, as seen in the status of Figure 3.8, is empty. So the simulator changes p_3's status to `waiting` (Figure 3.9) and continues the simulation.

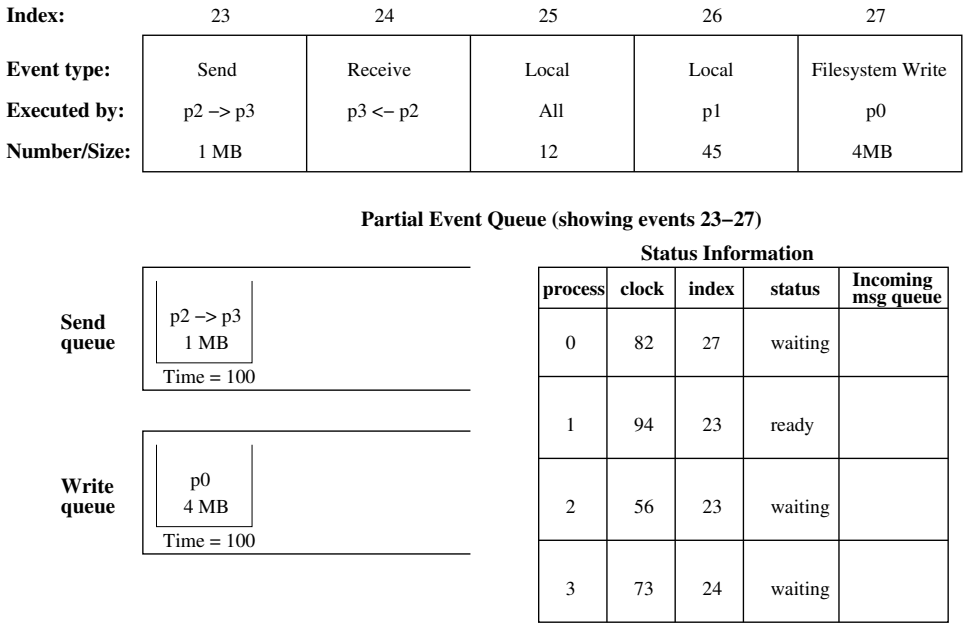


Figure 3.10: Simulation Example, Step 4: The state following the post of p_0's file system write at index 27.

File System Event: Write. The simulator treats file system writes similarly to network sends: it periodically allows writes to proceed and the global write queue is also a collection of buffers representing potential write times. When a write is encountered, the simulator places it in the appropriate write buffer based on the writing process's clock and simulates the write when all processes have either advanced past the time associated with that buffer or are `waiting`.

Example: Encountering a File System Write. In Figure 3.9, p_0's execution of event 27, a file system write, is ready to be simulated: event

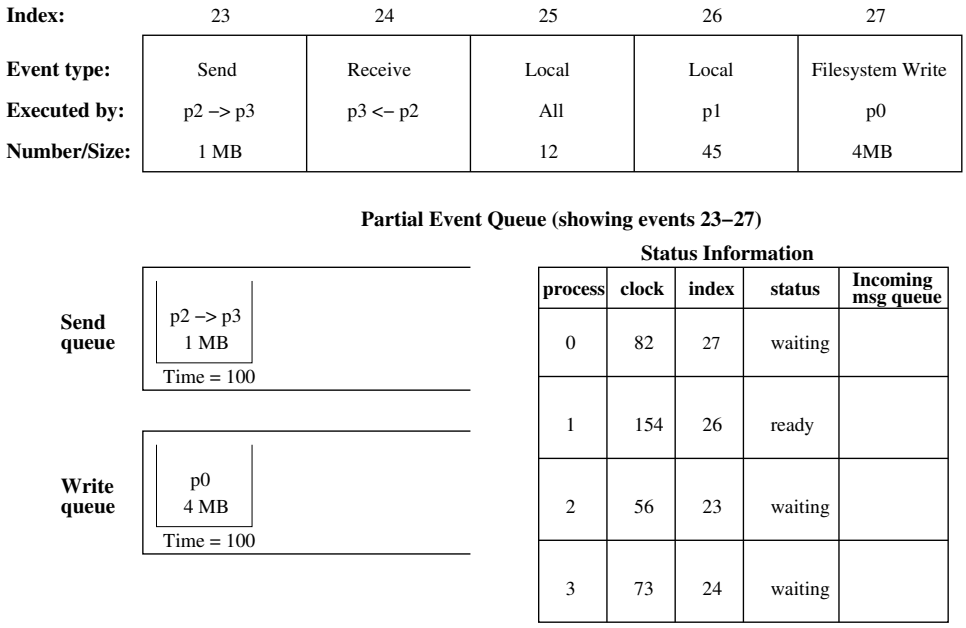


Figure 3.11: Simulation Example, Step 5: The state following the simulation of p_1's local event at index 25.

26 is skipped because p_0 does not execute it. The simulator places the write in the write buffer associated with time 100, since p_0's clock is 82, and changes p_0's status to `waiting`. Figure 3.10 displays the resulting state.

The simulation continues with p_1's execution of event 25; p_1's clock is adjusted to reflect the number of cycles spent, as seen in Figure 3.11.

Queue Processing. When all processes have either a simulated execution time exceeding the next buffer processing time or a status of `waiting`, the simulator performs any network sends or file system writes contained in the buffers associated with that processing time. Here we explain how that simu-

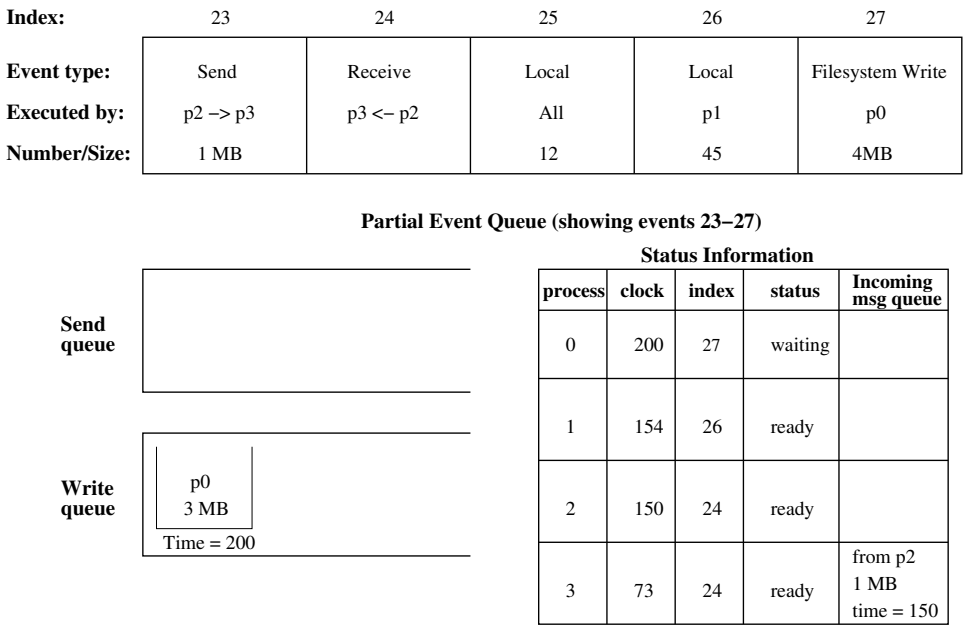


Figure 3.12: Simulation Example, Step 6: Network Sends and File System Writes Progress

lation occurs.

Send Queue. To simulate message sends waiting in a send buffer, the simulator determines the number of messages to be sent and the total amount of data. It then uses the network model to calculate the number of cycles needed for each send to complete. Any of the waiting messages that can complete before the time associated with the next send buffer do so and are added to the incoming message queues of the receiving process(es). Messages that require more cycles to complete progress as far as possible and then the remaining data is sent at the next send time.

Write Queue. The write queue is processed identically to the send queue, except the bandwidth available to each process is determined by the file system model and not the network model.

Example: Queue Processing. In the example simulation, the current system state (Figure 3.11) reflects three waiting processes and one ready process with a clock over 100, which indicates that any network sends and file system writes in the buffers associated with time 100 are ready to be simulated. This example shows one pending message send, a 1 MB message from p_2 to p_3, and one pending file system write, a 4 MB write from p_0.

Suppose the network model calculates 50 cycles to send a 1 MB message, so in this case the message from p_2 to p_3 takes 50 cycles. Since the send is a blocking send, the simulator updates p_2's clock to 150, increments p_2's index, and changes p_2's status to `ready`. The simulator places the message receive information in p_3's incoming message queue and changes p_3's state to `ready`.

For the write queue, let's suppose that the file system model approximates that this write takes 400 cycles, or 100 cycles per megabyte. The write, then, progresses for 100 cycles and the remaining 3 MB of data must be processed with any other writes pending in the buffer for time step 200. p_0's status remains as `waiting`, but its clock is updated to 200, as seen in Figure 3.12.

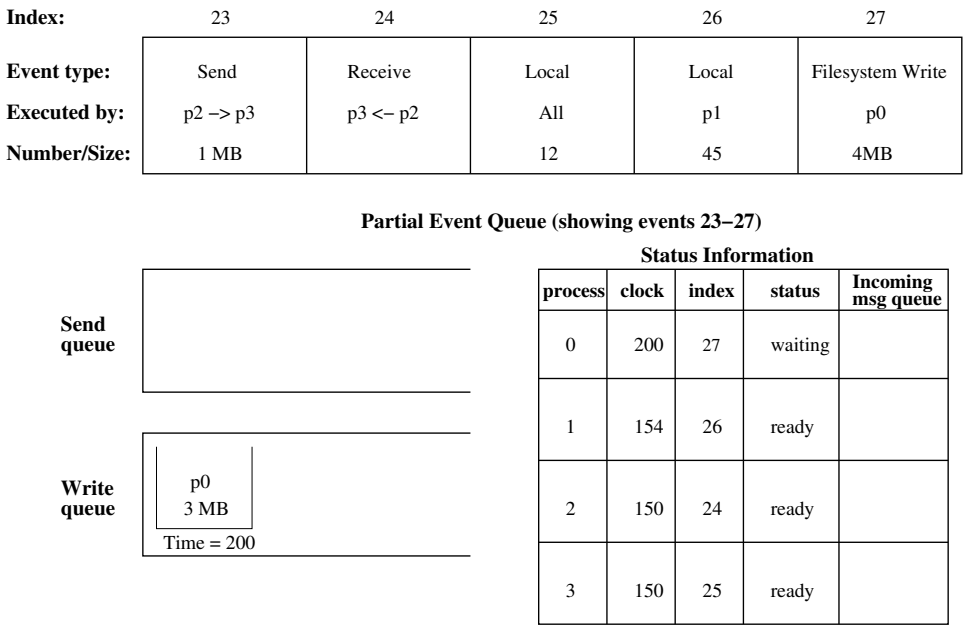
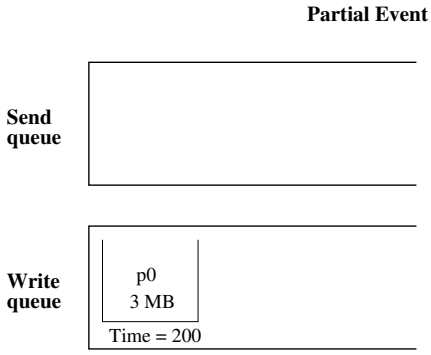


Figure 3.13: Simulation Example, Step 7: The state following the completion of p_3's network receive at index 24.

Example: Network Receive Completes. As the queue processing finishes, p_3 is `ready` and has the lowest clock value. Since p_3 has yet to complete event 24, the simulator executes that message receive by removing the message from p_3's incoming message queue, updating p_3's clock to the time of the message receipt, changing p_3's state to `ready`, and incrementing p_3's index (Figure 3.13).

As the example execution proceeds, p_2's event 25 is simulated, its clock updated, and its index incremented (Figure 3.14).

Index:	23	24	25	26	27
Event type:	Send	Receive	Local	Local	Filesystem Write
Executed by:	p2 –> p3	p3 <– p2	All	p1	p0
Number/Size:	1 MB		12	45	4MB



Partial Event Queue (showing events 23–27)

Status Information

process	clock	index	status	Incoming msg queue
0	200	27	waiting	
1	154	26	ready	
2	162	26	ready	
3	150	25	ready	

Figure 3.14: Simulation Example, Step 8: The state following the execution of p_2's local event at index 25.

Synchronization Events. A synchronization event is simulated as a send from every process to every other process. When all sends have been received, all process clocks are updated to the time of the last receipt, and all processes continue to the next event.

3.4 Evaluation

Our simulator can convert an application using thousands of processes into a trace and then estimate its execution time. To better appreciate the results generated by our simulator, we must understand how the components interact and how the error from each model affects the results. We have

presented the error for each model in isolation, and now we analyze how the models perform together.

Our simulator accurately predicts the performance of our application benchmarks (with no checkpointing). To derive this conclusion, we compare each benchmark's measured execution times on the target system with its simulated execution times. This analysis includes the local hardware and the network models but not the file system model as it does not include checkpointing, so these results show how the local hardware model works with the network model. The full results can be seen in Tables 3.7 and 3.8; all errors are reported for configurations using greater than 16 processes and executing for longer than thirty seconds. In these results, the source of the error is the local hardware model. The effects of the network error are negligible.

We also use our sub-models, which we create from a smaller set of experimental results than our other models, to predict the performance of each component and for `BT` in its entirety. These results show low error when we compare the resulting predictions to our measured times; this result gives us confidence in our methodology. The `BT` result was, for instance, an average error of 0.94.

The simulator predicts the performance of our synthetic benchmarks with low error, which shows how the local hardware model interacts with the file system model. To arrive at this result, we compare the measured execution time of the synthetic benchmark with two checkpointing policies to the simulated execution time. The average error for both benchmarks on both

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.00	0.98-1.02	0.83-1.21
LU	0.77	0.73-0.80	0.59-0.98
SP	0.74	0.68-0.81	0.27-2.06
Ek-simple	0.93	0.89-0.97	0.65-1.34

Table 3.7: Error for the benchmarks with message passing on Lonestar.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.23	1.21-1.25	1.09-1.39
LU	0.97	0.91-1.03	0.69-1.37
SP	0.71	0.70-0.73	0.57-0.88
Ek-simple	1.14	1.09-1.19	0.62-2.10

Table 3.8: Error for the benchmarks with message passing on Ranger.

systems is small: 1.27 for Staggered and 1.00 for Synchronous on Lonestar, and 1.0 for Staggered and 0.98 for Synchronous on Ranger. The full results can be seen in Tables 3.9 and 3.10. The errors on Lonestar are much higher than those on Ranger; this difference stems from the variability of the Lonestar file system and our model creation methodology. To create the model, we eliminate the results of any experiments that are too far from the average. We then eliminate any averages that appear high in relation to the other configurations. These decisions are the best way to represent the file system since otherwise we would represent a falsely slow file system; but it does result in a model that can underpredict the effects of contention. The Ranger file system is faster than the Lonestar file system and thus the same phenomenon is not visible in those results.

Checkpointing Policy	Average Error	Confidence Interval	Tolerance Interval
Staggered	1.27	1.21-1.33	0.80-2.01
Synchronous	1.00	0.89-1.12	0.40-2.51

Table 3.9: Error for the synthetic benchmarks with checkpointing on Lonestar.

Checkpointing Policy	Average Error	Confidence Interval	Tolerance Interval
Staggered	1.00	0.97-1.02	0.73-1.38
Synchronous	0.98	0.96-1.01	0.71-1.35

Table 3.10: Error for the synthetic benchmarks with checkpointing on Ranger.

In addition, we compare the measured execution times of the benchmarks with checkpointing to their simulated execution times, which provides information about how the simulator works as a whole. The overall error for the application benchmarks with checkpointing is also small: the average error is .85 for Lonestar and .80 for Ranger. The errors are reported in Tables 3.11 and 3.12.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	1.03	1.02-1.04	0.62-1.71
LU	0.81	0.81-0.82	0.55-1.19
SP	0.89	0.89-0.90	0.64-1.24
Ek-simple	0.67	0.66-0.68	0.36-1.25

Table 3.11: Error for the benchmarks with message passing and checkpointing on Lonestar.

Benchmark	Average Error	Confidence Interval	Tolerance Interval
BT	0.87	0.85-0.90	0.67-1.12
LU	0.87	0.83-0.91	0.59-1.27
SP	0.83	0.79-0.88	0.54-1.26
Ek-simple	0.64	0.60-0.69	0.37-1.10

Table 3.12: Error for the benchmarks with message passing and checkpointing on Ranger.

3.5 Conclusions

We have developed a scalable simulator that successfully simulates checkpointing parallel applications configured to use many processes. Our simulator estimates the system consequences of a change in checkpoint strategy efficiently and thus allows us to study staggered checkpointing. Our simulator consistently results in low error, which shows that our methodology is sound. In addition, when we created sub-models to test the predictive performance of our simulator, we found that these models predicted our measured data with low error. Thus, our simulator demonstrates predictive ability.

Chapter 4

Problem and Solution Spaces for Staggered Checkpointing

In this chapter, we identify the parameters under which staggered checkpointing is useful. We define a *problem space* where synchronous checkpointing causes *avoidable* file system and network contention. Avoidable contention occurs when all processes checkpointing b bytes simultaneously consume more time than a single process writing b bytes. We also define a *solution space*, which is where staggered checkpointing reduces or alleviates the avoidable contention resulting from synchronous checkpointing. We use two example supercomputing systems, Lonestar [204] and Ranger [205], both located at the Texas Advanced Computing Center (TACC) [2].

As we identify the problem and solution spaces, we also describe how file system performance and application characteristics, such as the number of processes, checkpoint sizes, and number of local operations, affect these spaces.

4.1 Methodology

To identify the problem and solution spaces for staggered checkpointing, we develop a synthetic benchmark that simulates a parallel application

executing local operations. This synthetic benchmark contains no communication, so it tests the checkpointing policy without the effects of communication.

We implement two checkpointing polices: *staggered* and *synchronous*. Our synchronous policy forces all processes to synchronize and then checkpoint. Our staggered policy allows N checkpointing locations; these locations are placed evenly throughout the local operations. An approximately equivalent number of processes checkpoints at each location.

To define these spaces, we simulate our synthetic benchmark with a variety of configurations:

- on each machine, Ranger and Lonestar
- with each checkpointing policy, staggered and synchronous,
- from 16 to 16,384 processes,
- with per process checkpoint sizes of 4 MB to 1 GB, and
- for local instruction execution times from 1 second to several days.

N is currently set to 12, a number chosen based on the properties of the application benchmarks (Table 5.6). For the staggered policy, the processes checkpoint throughout the local execution time, so the local execution time represents the interval during which the process checkpoint locations can be staggered.

Our synthetic benchmark can be related to real-world application benchmarks through the number of processes used by the application. In particular, we relate these results to five real-world applications that investigate tides [114], turbulence [6, 23], earthquakes [60], and combustion [55].

In our analysis, we indicate when staggered checkpointing makes checkpointing *feasible*. When we conduct experiments at TACC, we are only able to write to the file system for fifteen minutes of time, so we consider checkpointing *feasible* when it requires less than fifteen minutes of time. Staggered checkpointing causes checkpointing a configuration to become feasible when the synchronous checkpoint time is over 15 minutes but the staggered checkpoint time is below fifteen minutes.

4.2 Evaluation

In this section, we first identify the problem and solution spaces for Lonestar and Ranger in detail and perform preliminary analysis. We draw larger conclusions at the end of the chapter. In particular, we explore how these spaces change as the characteristics of the systems and applications change, and what these trends imply for staggered checkpointing in the future.

We present two problem definition tables for each machine. In the first problem definition table, the numbers represent the time difference in minutes between the time required for a single process to write a certain amount of data and multiple processes to each synchronously write that amount of data. This difference represents the maximum improvement that staggered checkpointing

can achieve over synchronous checkpointing: staggered checkpointing cannot enable multiple processes to write a checkpoint faster than a single process can write a checkpoint of the same size. The second problem definition table reports the number of minutes required for many different numbers of processes to synchronously checkpoint various amounts of data.

The solution space is presented as a series of graphs, where each graph represents the results for a range of process sizes, with each process writing several amounts of data over a particular interval size. For both machines, for interval sizes of 20 seconds and smaller, staggered checkpointing consumes essentially the same amount of time as synchronous checkpointing: the interval size is too small for the checkpoints to be sufficiently staggered to noticeably reduce file system contention. Increasing the interval size allows the processes to checkpoint farther apart and thus increases the benefits of staggering. To maximize the benefits of staggering, checkpoint locations must be separated by enough time for the checkpoint data to be fully written before the next set of checkpoints begins. The number of seconds necessary for a perfectly staggered line to achieve maximum benefit can be calculated with the following formula:

$$\frac{ceiling(\frac{number_of_processes}{checkpoint_locations}) * data_per_checkpoint}{maximum_write_rate_of_the_system}$$

Although our experiments include a large range of interval sizes and process sizes as small as 16, here we present data for only the larger process sizes three interval sizes. For Lonestar, we present process sizes of 256 processes and larger with the exception of Tables 4.2 and 4.10, which present data

beginning at 16 processes. For Ranger, the problem space results also present process sizes of 256 processes and larger; the solution space results omit the 256 and 512 process cases. For these cases, synchronous checkpointing performs very well and staggered checkpointing is uninteresting.

In this chapter, we present results and analysis for interval sizes of three, fifteen, and thirty minutes. For our synthetic benchmark, this interval represents the total amount of local execution time. In applications, this interval is determined by the time between barriers and collective communication. In today's applications, three minutes is closest to the expected available interval, though the intervals are often much smaller. Fifteen minutes represents what we consider a reasonable amount of time for a process to be checkpointing; thirty minutes is a large amount of time in which to checkpoint, but we include the results here to better understand the trends of the solution spaces.

Our simulator performs optimistically in the face of network and file system saturation, so we do not present the results for configurations that cause both staggered and simultaneous checkpointing to saturate the file system. The configurations that are presented in the figures and tables are configurations that are inside the fidelity of our simulator. Empty spaces in our figures represent configurations removed for this reason.

In the solution space graphs displaying checkpoint times, each stacked bar reflects three times: 1) the amount of time required for one process to write that amount of data, which represents the ceiling of improvement for staggered checkpointing; 2) the amount of time required for that amount of

checkpoint data to be written by that number of processes using the staggered checkpointing policy; 3) the amount of time required for that amount of checkpoint data to be written by that number of processes using the synchronous checkpointing policy. The graphs displaying total execution times report similarly, but their data represents the total execution time rather than the checkpoint time. Since our synthetic benchmark does not contain communication, the total time improvement typically reflect the improvement in checkpointing time. For both sets of graphs, the exposed synchronous time indicates the benefit of staggered checkpointing.

Each graph representing checkpoint time displays time ranging from zero to just over an hour; bars that extend into the top of the graph indicate total checkpointing time of longer than the time shown. In addition, each graph displays horizontal lines at 15 minutes, 30 minutes, and an hour. The total execution time graphs display times from just below the expected execution time without checkpointing to just over an hour after that.

We also include statistical analysis tables. The tables displaying checkpoint time improvement highlight the *sweet spot*, or the configurations for which synchronous checkpointing consumes more than five minutes, staggered checkpointing reduces checkpoint time to 30 minutes or less, and the overall reduction is at least 25%. The numbers in these tables are the raw improvement of staggered checkpointing in light of the accuracy of our simulator. The bold numbers indicate configurations made feasible by staggered checkpointing. In these tables, removed configurations are indicated in tables by a "–".

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0	1	2	7	24	90	349
8 MB	1	4	15	57	221	868	3,444
16 MB	2	8	29	110	434	1,722	6,858
32 MB	4	14	55	217	861	3,429	13,686
64 MB	7	28	109	430	1,714	6,843	27,342
128 MB	14	54	215	857	3,421	13,671	54,654
256 MB	27	108	429	1,711	6,836	27,327	109,279
512 MB	54	214	855	3,418	13,664	54,639	218,528
1 GB	107	428	1,709	6,832	27,320	109,264	437,026

Table 4.1: Lonestar Problem Space: This table identifies the problem space by comparing the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. So, this table shows that one process can checkpoint 16 MB 110 minutes faster than the 2,048 processes can each checkpoint 16 MB. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system.

4.2.1 Lonestar

Lonestar is a supercomputer that was installed in 2004 and is currently ranked number 123 in the Top 500 list [8]. It includes local hardware with 2.66 GHz processors, a Myrinet network for communication, and a Lustre file system featuring 4.6 GB/s throughput [204]. Our simulator assumes that applications executing on Lonestar use one process per local hardware machine, or *node*.

Problem Space. The Lonestar problem space shows that staggered checkpointing has the potential to greatly reduce the overhead caused by synchronous checkpointing. In the problem definition table for Lonestar, Ta-

Checkpoint	Number of Processes										
Size	**16**	**32**	**64**	**128**	**256**	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0	0	0	0	0	0	2	7	24	90	349
8 MB	0	0	0	1	1	4	15	57	221	868	3,444
16 MB	1	1	1	1	3	8	29	111	435	1,723	6,858
32 MB	2	2	2	3	6	16	57	219	863	3,431	13,688
64 MB	4	4	4	6	11	31	112	434	1,718	6,847	27,346
128 MB	8	8	9	12	22	63	223	865	3,430	13,679	54,662
256 MB	17	17	19	24	44	125	446	1,728	6,852	27,344	109,296
512 MB	35	35	38	48	88	249	890	3,452	13,698	54,674	218,562
1 GB	70	71	76	96	176	497	1,778	6,901	27,389	109,333	437,095

Table 4.2: Lonestar Problem Space: This table reports the number of minutes needed for a number of processes to each synchronously checkpoint an amount of data. So 2,048 processes synchronously checkpointing 16 MB of data requires approximately 111 minutes. Note that its configurations begin at 16 rather than 256.

ble 4.1, we see that for the smaller configurations in the upper left hand corner, such as 256 processes each writing 4 MB of data, there is no room for improvement since synchronous checkpointing performs well. For the larger configurations on the right hand side of the table, such as 16,384 processes each writing 4 MB of data, and even the middle-sized configurations in the middle of the table, such as 2,048 processes each writing 16 MB of data, there is much room for improvement because synchronous checkpointing causes contention at the file system. Table 4.3 displays the problem space with statistical information.

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(50%) 0(50%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	2(90%) 2(90%) c:2-2 t:2-2	7(100%) 7(100%) c:7-7 t:7-7	24(100%) 25(100%) c:24-26 t:23-26	90(100%) 93(100%) c:89-97 t:87-99	349(100%) 360(100%) c:343-377 t:338-381
8 MB	1(70%) 1(70%) c:1-1 t:1-1	4(90%) 4(90%) c:4-5 t:4-5	15(100%) 16(100%) c:15-16 t:15-17	57(100%) 59(100%) c:56-62 t:55-62	221(100%) 228(100%) c:217-238 t:214-241	868(100%) 895(100%) c:853-937 t:842-949	3,444(100%) 3,550(100%) c:3,383-3717 t:3,338-3,762
16 MB	2(70%) 2(70%) c:2-2 t:2-2	8(90%) 8(90%) c:7-8 t:7-8	29(100%) 29(100%) c:28-31 t:28-31	110(100%) 114(100%) c:108-119 t:107-121	434(100%) 448(100%) c:427-469 t:421-474	1,722(100%) 1,775(100%) c:1,692-1,859 t:1,669-1,881	6,858(100%) 7,070(100%) c:6,738-7,402 t:6,648-7,492
32 MB	4(60%) 4(60%) c:4-4 t:4-4	14(90%) 15(90%) c:14-16 t:14-16	55(100%) 57(100%) c:54-60 t:53-60	217(100%) 224(100%) c:213-234 t:210-237	861(100%) 888(100%) c:846-929 t:834-941	3,429(100%) 3,535(100%) c:3,369-3,701 t:3,324-3,746	13,686(100%) 14,109(100%) c:13,446-14,772 t:13,267-14,951
64 MB	7(60%) 7(60%) c:7-8 t:7-8	28(90%) 28(90%) c:27-30 t:26-30	109(100%) 112(100%) c:106-117 t:105-119	430(100%) 444(100%) c:423-465 t:417-471	1,714(100%) 1,767(100%) c:1,684-1,851 t:1,662-1,873	6,843(100%) 7,055(100%) c:6,723-7,386 t:6,633-7,476	27,342(100%) 28,188(100%) c:26,863-29,512 t:26,505-29,870
128 MB	14(60%) 14(60%) c:13-15 t:13-16	54(90%) 56(90%) c:53-59 t:52-60	215(100%) 222(100%) c:211-233 t:208-236	857(100%) 884(100%) c:842-926 t:830-937	3,421(100%) 3,527(100%) c:3,361-3,693 t:3,316-3,738	13,671(100%) 14,094(100%) c:13,431-14,756 t:13,252-14,935	54,654(100%) 56,345(100%) c:53,697-58,992 t:52,981-59,708
256 MB	27(60%) 28(60%) c:26-30 t:25-31	108(90%) 111(90%) c:105-117 t:103-119	429(100%) 442(100%) c:420-463 t:414-469	1,711(100%) 1,764(100%) c:1,680-1,847 t:1,657-1,870	6,836(100%) 7,047(100%) c:6,715-7,379 t:6,625-7,469	27,327(100%) 28,172(100%) c:26,848-29,497 t:26,490-29,855	109,279(100%) 112,658(100%) c:107,365-117,952 t:105,934-119,383
512 MB	54(60%) 55(60%) c:51-60 t:50-61	214(90%) 221(90%) c:209-233 t:205-236	855(100%) 882(100%) c:839-925 t:827-937	3,418(100%) 3,523(100%) c:3356-3691 t:3311-3736	13,664(100%) 14,086(100%) c:13,423-14,750 t:13,243-14,929	54,639(100%) 56,329(100%) c:53,681-58,977 t:52,965-59,693	218,528(100%) 225,286(100%) c:214,702-235,871 t:211,838-238,734
1 GB	107(60%) 110(60%) c:101-120 t:99-122	428(90%) 441(90%) c:417-465 t:410-472	1,709(100%) 1,762(100%) c:1,676-1,848 t:1,652-1,871	6,832(100%) 7,043(100%) c:6,709-7,377 t:6,618-7,468	27,320(100%) 28,165(100%) c:26,838-29,491 t:26,479-29,850	109,264(100%) 112,643(100%) c:107,348-117,938 t:105,916-119,370	437,026(100%) 450,542(100%) c:429,374-471,710 t:423,648-477,436

Table 4.3: Lonestar Problem Space, Statistical Information: This table identifies the problem space by comparing the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. So, this table shows that one process can checkpoint 16 MB 110 minutes faster than the 2,048 processes can each checkpoint 16 MB. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system.

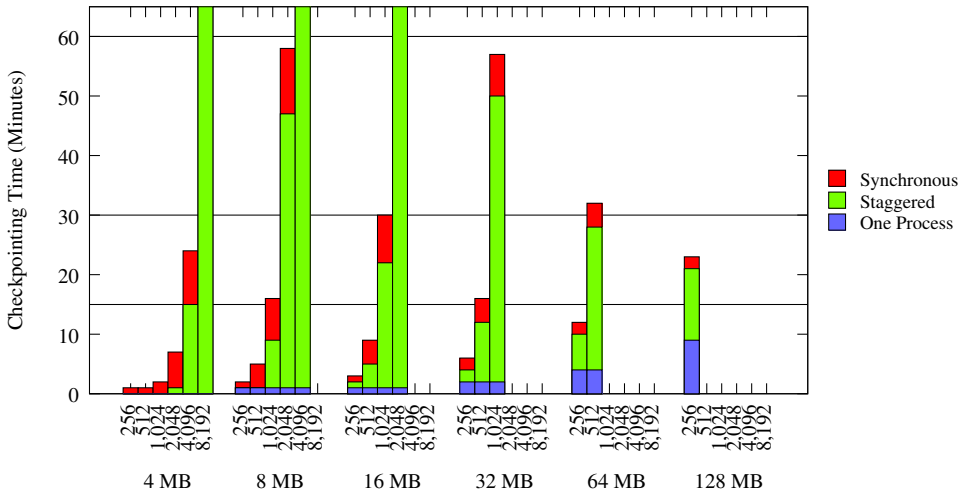


Figure 4.1: Lonestar Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a three minute interval size.

Solution Space. Our Lonestar solution space results show that staggered checkpointing reduces the checkpoint and total execution times over those of synchronous checkpointing for many configurations.

With just a three minute interval size, staggered checkpointing begins to reduce checkpoint time. Three minutes is sufficient for improvement since the three minute interval separates the checkpoints such that a new set of checkpoints begins every fifteen seconds. However, at this interval size, staggered checkpointing only shows large percentages of improvement for small numbers of processes checkpointing small amounts of data, as we see in Figure 4.1 and Table 4.4. Staggered checkpointing reduces the checkpoint time by at least 50% for the following configurations: up to 2,048 processes writing

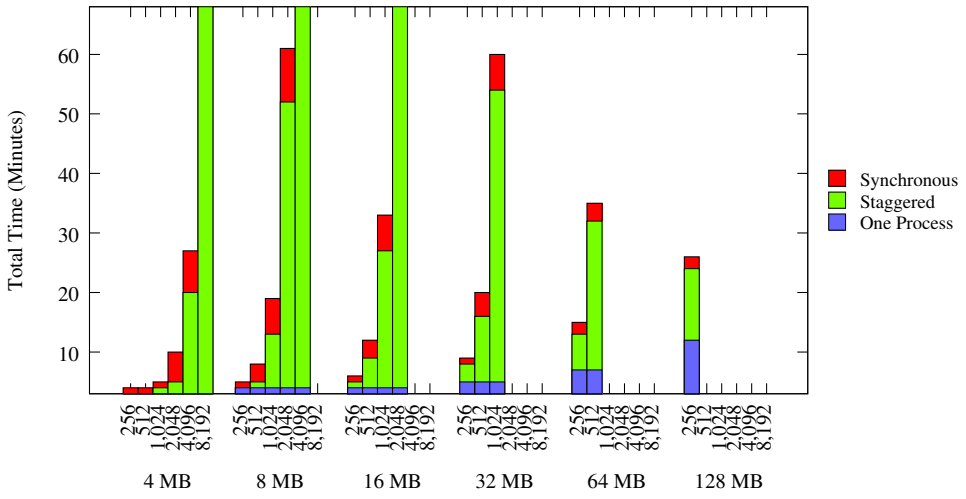


Figure 4.2: Lonestar Solution Space (Total Time in 3 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a three minute interval size.

Figure 4.3: Lonestar Solution Space, 3 minutes, Total Time

4 MB each, and up to 512 processes writing up to 16 MB each. Since these are smaller configurations, the time consumed by synchronous checkpointing is smaller, so the absolute savings is smaller. For the larger configurations, staggered checkpointing shows a small amount of improvement.

The three minute interval has a sweet spot of six configurations, and it converts two configurations from infeasible to feasible: 1,024 processes writing 8 MB of data each and 512 processes writing 32 MB of data each.

Figure 4.3 and Table 4.5 display the total execution time information for the three minute interval size.

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(50%) 0(50%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	2(80%) 2(80%) c:2-2 t:2-2	6(80%) 6(80%) c:5-6 t:5-6	9(40%) 9(40%) c:8-11 t:8-11	13(10%) 13(10%) c:7-19 t:6-20	–
8 MB	1(60%) 1(60%) c:1-1 t:1-1	3(70%) 4(70%) c:3-4 t:3-4	**7(40%)** **7(40%)** **c:6-8** **t:6-8**	11(20%) 11(20%) c:7-15 t:6-16	14(10%) 15(10%) c:0-29 t:-4-33	–	–
16 MB	2(50%) 2(50%) c:1-2 t:1-2	4(50%) 4(50%) c:4-5 t:3-5	7(30%) 8(30%) c:6-9 t:5-10	12(10%) 12(10%) c:5-19 t:3-21	–	–	–
32 MB	2(30%) 2(30%) c:1-2 t:1-2	**4(20%)** **4(20%)** **c:3-5** **t:3-5**	8(10%) 8(10%) c:4-12 t:3-13	–	–	–	–
64 MB	2(10%) 2(10%) c:1-3 t:1-3	4(10%) 4(10%) c:2-6 t:2-7	–	–	–	–	–
128 MB	2(10%) 2(10%) c:0-3 t:0-4	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.4: Lonestar Solution Space (Checkpoint Time in 3 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately three minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint approximately 11 minutes faster using staggered checkpointing rather than using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(10%) 1(30%) c:1-1 t:0-2	1(20%) 1(30%) c:1-2 t:1-2	2(40%) 3(50%) c:2-3 t:2-3	5(50%) 6(60%) c:5-7 t:5-8	7(30%) 11(40%) c:8-15 t:7-15	10(10%) 28(30%) c:16-39 t:14-41	–
8 MB	1(20%) 2(40%) c:1-3 t:1-3	3(40%) 4(50%) c:3-5 t:3-5	5(30%) 8(40%) c:6-11 t:6-11	8(10%) 20(30%) c:12-27 t:11-28	11(0%) 56(30%) c:29-84 t:23-90	–	–
16 MB	1(20%) 2(40%) c:2-3 t:2-3	3(30%) 5(40%) c:4-6 t:3-7	6(20%) 12(40%) c:8-16 t:7-16	9(10%) 32(30%) c:18-45 t:14-49	–	–	–
32 MB	1(20%) 3(30%) c:2-4 t:2-4	3(20%) 7(30%) c:4-9 t:4-10	6(10%) 18(30%) c:11-25 t:9-27	–	–	–	–
64 MB	1(10%) 4(30%) c:2-6 t:2-6	3(10%) 10(30%) c:6-14 t:5-15	–	–	–	–	–
128 MB	1(10%) 7(30%) c:3-10 t:3-10	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.5: Lonestar Solution Space (Total Time in 3 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately three minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by nine minutes or 10% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

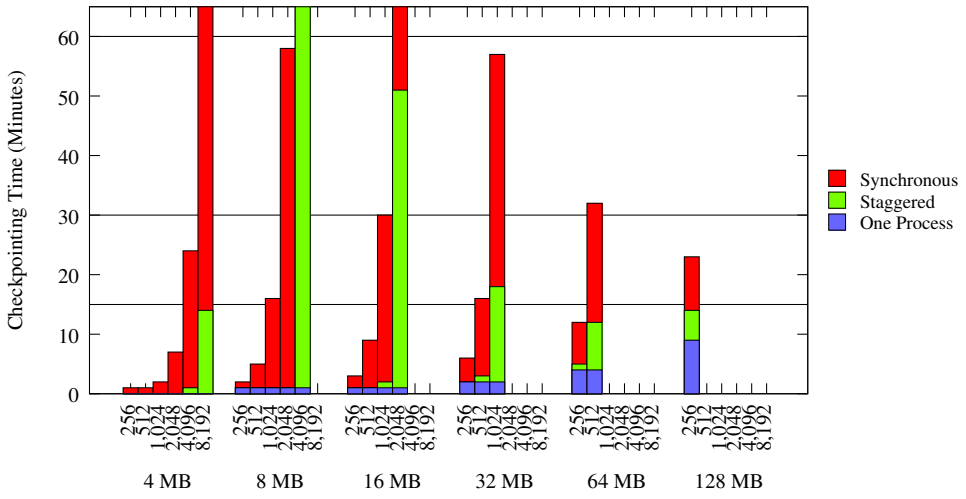


Figure 4.4: Lonestar Solution Space (Checkpoint Time in 15 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size.

Staggered checkpointing performs well when the interval size increases to fifteen minutes. In Figures 4.4 and 4.5, the interval size is approximately fifteen minutes. In comparison to Figure 4.1, the area of improvement grows to include larger numbers of processes and larger data sizes. In fact, staggered checkpointing achieves maximum improvement in all process configurations with 2,048 processes or fewer writing 4 MB of data each, with 512 processes or fewer writing 8 MB of data, and with 256 processes or fewer writing 32 MB of data. Staggered checkpointing now reduces checkpointing time by at least 50% for all configurations included in the graph except 256 processes writing 128 MB of data and 4,096 processes writing 8 MB of data. The improvements in those configurations are 40% and 30%, respectively.

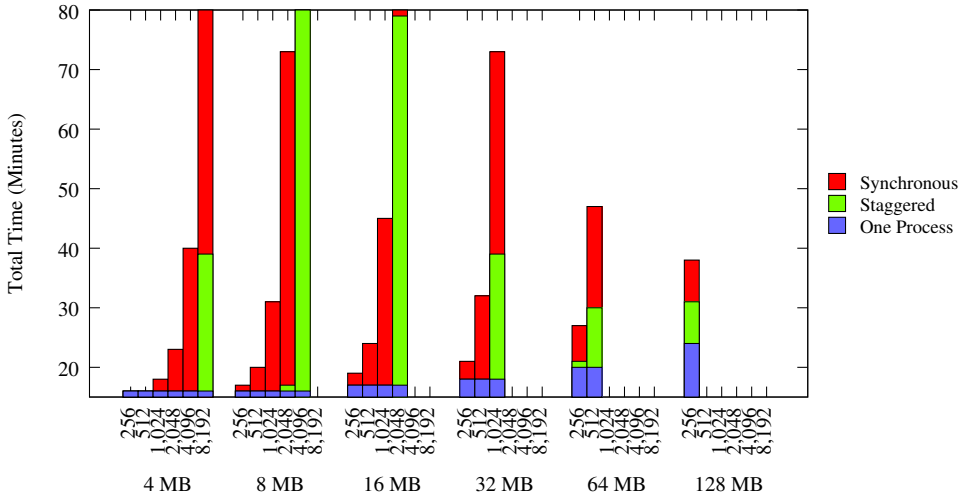


Figure 4.5: Lonestar Solution Space (Total Time in 15 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size.

As we can see in Figure 4.4 and the sweet spot displayed in Table 4.6, for many of the configurations this percent savings also represents a significant time savings—only three of our configurations are not yet feasible. The table shows that the sweet spot has increased to include larger process and data sizes and is now thirteen configurations.

A concern is how checkpointing influences the overall execution time. Figure 4.5 shows that total execution time is less than twenty minutes in the staggered case for up to 4,096 processes writing 4 MB, 2,048 processes writing 8 MB, 1,024 processes writing 16 MB, and 512 processes writing 32 MB. For these fourteen configurations, then, checkpointing adds less than five minutes of execution time. The total execution time only remains under twenty minutes

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(50%) 0(50%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	2(80%) 2(90%) c:2-2 t:2-2	7(90%) 7(90%) c:6-7 t:6-7	**24(100%)** **24(100%)** **c:23-26** **t:23-26**	**77(80%)** **79(80%)** **c:74-83** **t:73-85**	–
8 MB	1(70%) 1(70%) c:1-1 t:1-1	4(90%) 4(90%) c:4-5 t:4-5	**15(90%)** **15(90%)** **c:15-16** **t:14-16**	**56(100%)** **58(100%)** **c:55-61** **t:55-62**	72(30%) 74(30%) c:61-87 t:58-91	–	–
16 MB	2(60%) 2(60%) c:2-2 t:2-2	7(90%) 8(90%) c:7-8 t:7-8	**28(90%)** **29(90%)** **c:27-30** **t:27-31**	60(50%) 62(50%) c:56-68 t:55-70	–	–	–
32 MB	4(60%) 4(60%) c:3-4 t:3-4	**14(80%)** **14(80%)** **c:13-15** **t:13-15**	39(70%) 41(70%) c:38-44 t:37-44	–	–	–	–
64 MB	6(60%) 7(60%) c:6-7 t:6-7	**20(60%)** **21(60%)** **c:19-23** **t:19-23**	–	–	–	–	–
128 MB	**8(40%)** **9(40%)** **c:7-10** **t:7-10**	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.6: Lonestar Solution Space (Checkpoint Time in 15 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately fifteen minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 60 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(0%) 4(20%) c:2-6 t:1-6	1(0%) 4(20%) c:2-6 t:2-6	2(10%) 5(30%) c:3-7 t:3-8	7(30%) 10(40%) c:7-13 t:7-13	24(60%) 27(70%) c:22-32 t:21-33	67(60%) 75(70%) c:63-88 t:60-91	–
8 MB	1(10%) 5(30%) c:3-7 t:2-7	4(20%) 8(40%) c:5-10 t:5-11	15(50%) 18(60%) c:15-22 t:14-23	56(80%) 60(80%) c:51-68 t:49-70	55(20%) 94(40%) c:66-122 t:59-129	–	–
16 MB	2(10%) 6(30%) c:3-8 t:3-8	7(30%) 11(50%) c:8-14 t:7-15	28(60%) 32(70%) c:26-37 t:25-38	49(40%) 65(50%) c:50-80 t:47-84	–	–	–
32 MB	4(20%) 7(30%) c:5-10 t:4-11	14(40%) 17(50%) c:14-21 t:13-22	34(50%) 42(60%) c:33-51 t:31-53	–	–	–	–
64 MB	6(20%) 11(40%) c:7-14 t:7-15	18(40%) 24(50%) c:18-30 t:17-31	–	–	–	–	–
128 MB	7(20%) 14(40%) c:9-18 t:8-19	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.7: Lonestar Solution Space (Total Time in 15 minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately fifteen minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 48 minutes or 40% using staggered checkpointing rather than using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

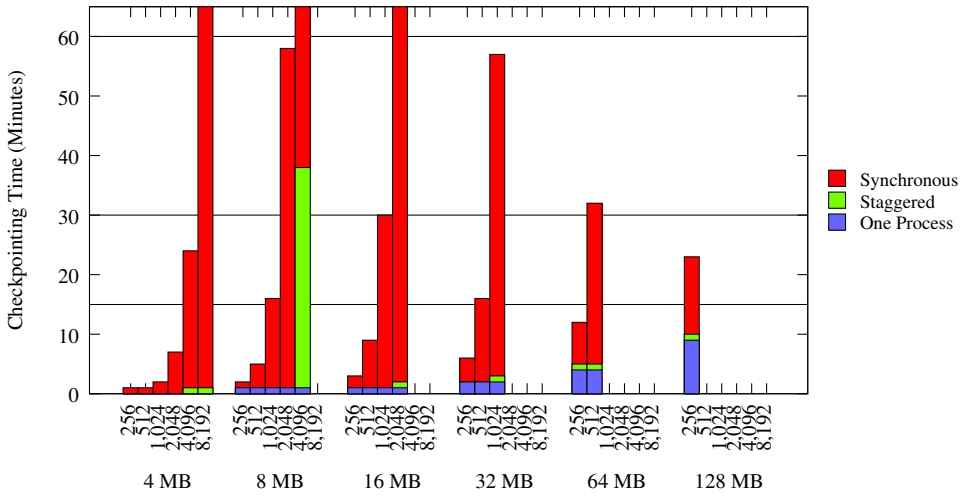


Figure 4.6: Lonestar Solution Space (Checkpoint Time in 30 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size.

in the synchronous case for five configurations. Table 4.7 presents the numbers related to the total execution time.

As the interval size increases to thirty minutes, the space where staggered checkpointing achieves maximum benefit expands to even larger configurations. In Figure 4.6, we see that all configurations are now feasible except for one—4,096 processes writing 8 MB of data each. This configuration is also not included in the sweet spot shown in Table 4.8. This results seems odd given the improvement in 8,192 processes checkpointing 4 MB each and 2,048 processes checkpointing 16 MB each. But in fact, we see similar jumps in the other configurations as the amount of data increases from 4 MB to 8 MB per process. This effect is most likely an artifact of the file system, which uses a

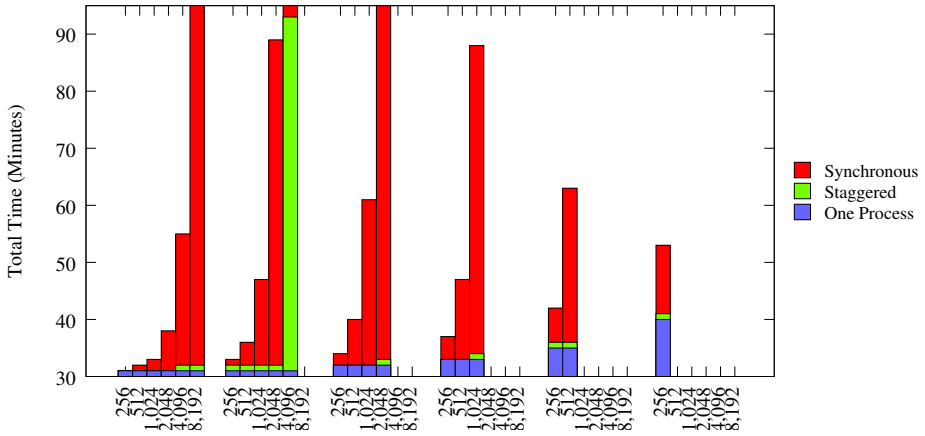


Figure 4.7: Lonestar Solution Space (Total Time in 30 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size.

more efficient protocol for sizes less than or equal to 4 MB.

In Table 4.8, we see that the sweet spot expands to include one configuration.

Table 4.9 presents the numbers related to the total execution time.

Checkpoint Locations. Recall that our synthetic benchmark allows N checkpoint locations for the staggered policy. Effectively, then, in the staggered policy $1/N^{th}$ of the total processes checkpoint at each location. So when N is set to 12, as it is for these experiments, and the total number of processes is 1024, there are 85 or 86 processes checkpointing at each location, effectively making the staggered case equivalent to the synchronous case for 85 or 86 processes. We can see this effect by comparing Table 4.2 and Table 4.10, where

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(50%) 0(50%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	2(80%) 2(90%) c:2-2 t:2-2	7(90%) 7(90%) c:6-7 t:6-7	**24(100%)** **24(100%)** **c:23-26** **t:23-26**	**89(100%)** **92(100%)** **c:88-96** **t:86-97**	–
8 MB	1(70%) 1(70%) c:1-1 t:1-1	4(90%) 4(90%) c:4-5 t:4-5	**15(90%)** **15(90%)** **c:15-16** **t:14-16**	**56(100%)** **58(100%)** **c:55-61** **t:55-62**	184(80%) 189(80%) c:178-200 t:176-203	–	–
16 MB	2(60%) 2(60%) c:2-2 t:2-2	7(90%) 8(90%) c:7-8 t:7-8	**28(100%)** **29(100%)** **c:28-30** **t:27-31**	**109(100%)** **113(100%)** **c:107-118** **t:106-120**	–	–	–
32 MB	4(60%) 4(60%) c:4-4 t:3-4	**14(90%)** **15(90%)** **c:14-15** **t:14-16**	**54(90%)** **56(90%)** **c:53-59** **t:53-60**	–	–	–	–
64 MB	7(60%) 7(60%) c:7-8 t:6-8	**27(80%)** **28(80%)** **c:26-29** **t:26-30**	–	–	–	–	–
128 MB	**12(60%)** **13(60%)** **c:12-14** **t:11-14**	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.8: Lonestar Solution Space (Checkpoint Time in 30 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately thirty minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 109 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

1024 processes writing 16 MB in the staggered case require the same amount of checkpointing time, 1 minute, as 85 processes writing 16 MB in the syn-

Checkpoint Size	Number of Processes						
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0(0%) 7(20%) c:3-11 t:2-12	1(0%) 7(20%) c:3-11 t:3-12	2(10%) 9(30%) c:5-13 t:4-14	7(20%) 13(30%) c:9-18 t:8-19	24(40%) 30(50%) c:24-37 t:22-38	89(70%) 96(80%) c:82-110 t:79-113	–
8 MB	1(0%) 8(20%) c:4-12 t:3-13	4(10%) 11(30%) c:7-15 t:6-16	15(30%) 22(50%) c:16-27 t:15-29	56(60%) 63(70%) c:53-74 t:51-76	159(60%) 179(70%) c:150-208 t:143-215	–	–
16 MB	2(10%) 9(30%) c:5-13 t:4-14	7(20%) 14(40%) c:10-19 t:8-20	28(50%) 35(60%) c:28-42 t:26-44	109(80%) 116(80%) c:100-133 t:96-137	–	–	–
32 MB	4(10%) 11(30%) c:6-15 t:5-16	14(30%) 21(40%) c:16-27 t:14-28	54(60%) 62(70%) c:51-72 t:49-74	–	–	–	–
64 MB	7(20%) 14(30%) c:9-20 t:8-21	27(40%) 34(50%) c:27-42 t:25-43	–	–	–	–	–
128 MB	12(20%) 21(40%) c:15-27 t:13-29	–	–	–	–	–	–
256 MB	–	–	–	–	–	–	–
512 MB	–	–	–	–	–	–	–
1 GB	–	–	–	–	–	–	–

Table 4.9: Lonestar Solution Space (Total Time in 30 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately thirty minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 109 minutes or 80% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

chronous case, which would fall between 64 processes and 128 processes. As long as each set of checkpoints finishes before the next set starts, this effect will continue. When a set of checkpoints does not complete before the next set

Checkpoint	Number of Processes										
Size	**16**	**32**	**64**	**128**	**256**	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**
4 MB	0	0	0	0	0	0	0	0	0	13	271
8 MB	0	0	0	0	0	0	0	1	149	784	3,352
16 MB	1	1	1	1	1	1	1	51	358	1,634	6,765
32 MB	2	2	2	2	2	2	18	158	782	3,340	13,592
64 MB	4	4	4	4	5	11	72	372	1,636	6,755	27,209
128 MB	8	8	8	9	14	41	183	802	3,347	13,579	54,526
256 MB	17	17	18	21	36	103	405	1,664	6,767	27,243	109,159
512 MB	35	35	37	45	80	227	849	3,387	13,612	54,573	218,425
1 GB	70	71	75	93	168	476	1,737	6,836	27,303	109,232	436,958

Table 4.10: Lonestar Solution Space (Staggered Checkpoint Time in 15 Minute Interval): This table shows the number of minutes spent checkpointing for the staggered policy with approximately fifteen minutes of local execution time. Note that the data in this table begins at 16 processes.

starts, staggered checkpointing does not achieve the maximum improvement, but it can still improve the checkpointing time. For instance, when we compare 1,024 processes writing 32 MB in Table 4.10 and 1,024 processes writing 32 MB in Table 4.2, the staggered case, which requires 18 minutes, is much faster than the synchronous case, which requires 57 minutes. However, the staggered case is significantly slower than the 85 process synchronous case, which only requires 2-3 minutes. If our synthetic benchmark were to allow more checkpoint locations, we would likely see more improvement at larger interval sizes.

Error Analysis. Our results for Lonestar show that staggered checkpointing is still beneficial even when we consider the error margins for our simulator. Let 1.0 represent the measured execution time of our synthetic benchmarks with checkpointing. Then from Chapter 3, we know that the av-

erage error for our synthetic benchmark with synchronous checkpointing is a factor of 1.0 of the measured execution time, and the average error for the synthetic benchmark with staggered checkpointing is 1.27. These errors mean that the simulator consistently overpredicts the execution time of staggered checkpointing on Lonestar. Thus, our results are conservative, and staggered checkpointing should produce more benefit than we see here.

4.2.2 Ranger

Ranger [205] consists of local hardware with four 2.3 GHz processors with four cores each, an Infiniband network for communication, and a Lustre file system featuring 40 GB/s throughput [186]. Ranger is a newer, more modern supercomputer than Lonestar. It was installed in 2008 and is currently ranked 11 in the Top 500 list [8]. We use Ranger differently than Lonestar: our simulator assumes that each application on Ranger executes with sixteen processes per node, which is quite different from the one process per node configuration we use on Lonestar.

Problem Space. The Ranger problem space results show that staggered checkpointing has great potential to reduce the overhead caused by synchronous checkpointing. In the problem definition table for Ranger, Table 4.11, smaller configurations, such as 256 processes each writing 4 MB of data, and even up to 1,024 processes each writing 16 MB of data, show little to no room for improvement; synchronous checkpointing performs well for these configurations.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0	0	1	5	19	74	296	1,184	4,736
8 MB	0	0	1	6	23	89	356	1,420	5,673
16 MB	0	1	3	11	45	178	710	2,837	11,340
32 MB	0	1	6	22	89	355	1,418	5,670	22,673
64 MB	1	3	11	44	178	709	2,835	11,338	45,341
128 MB	1	6	22	89	355	1,418	5,669	22,672	90,675
256 MB	3	11	44	177	709	2,834	11,336	45,341	181,343
512 MB	6	22	89	354	1,417	5,668	22,671	90,679	362,680
1 GB	11	44	177	709	2,834	11,335	45,340	181,355	725,354

Table 4.11: Ranger Problem Space: This table identifies the problem space and compares the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. So, one process can checkpoint 16 MB 11 minutes faster than the 2,048 processes can each checkpoint 16 MB. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system.

For larger configurations such as 16,384 processes each writing 4 MB of data, there is much room for improvement. For the middle-sized configurations, such as 2,048 processes writing 32 MB of data, staggered checkpointing can improve checkpointing time, but the maximum improvement is a small number of minutes. Ranger's file system can write 40 GB/s, so the time needed for processes to synchronously checkpoint is smaller than that of Lonestar, and thus the potential benefit of staggering is smaller.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(10%)	0(40%)	1(70%)	5(90%)	19(100%)	74(100%)	296(100%)	1,184(100%)	4,736(100%)
	0(10%)	0(40%)	1(70%)	4(90%)	18(100%)	72(100%)	287(100%)	1,150(100%)	4,598(100%)
	c:0-0	c:0-0	c:1-1	c:4-5	c:17-19	c:70-74	c:279-296	c:1,116-1,183	c:4,464-4,732
	t:0-0	t:0-0	t:1-1	t:4-5	t:17-19	t:69-75	t:277-298	t:1,106-1,193	t:4,425-4,771
8 MB	0(10%)	0(30%)	1(60%)	6(80%)	23(100%)	89(100%)	356(100%)	1,420(100%)	5,673(100%)
	0(10%)	0(30%)	1(60%)	6(80%)	22(100%)	87(100%)	345(100%)	1,379(100%)	5,508(100%)
	c:0-0	c:0-0	c:1-2	c:5-6	c:21-23	c:84-89	c:335-356	c:1,338-1,419	c:5,347-5,668
	t:0-0	t:0-0	t:1-2	t:5-6	t:21-23	t:83-90	t:332-358	t:1,327-1,430	t:5,301-5,715
16 MB	0(10%)	1(20%)	3(60%)	11(80%)	45(100%)	178(100%)	710(100%)	2,837(100%)	11,340(100%)
	0(10%)	1(20%)	3(60%)	11(80%)	43(100%)	173(100%)	689(100%)	2,754(100%)	11,010(100%)
	c:0-0	c:1-1	c:3-3	c:11-11	c:42-45	c:168-178	c:669-709	c:2,674-2,835	c:10,689-11,331
	t:0-0	t:1-1	t:3-3	t:10-11	t:42-45	t:166-179	t:663-715	t:2,651-2,858	t:10,596-11,423
32 MB	0(10%)	1(20%)	6(60%)	22(80%)	89(100%)	355(100%)	1,418(100%)	5,670(100%)	22,673(100%)
	0(10%)	1(20%)	5(60%)	22(80%)	86(100%)	345(100%)	1,377(100%)	5,505(100%)	22,013(100%)
	c:0-1	c:1-2	c:5-6	c:21-22	c:84-89	c:334-355	c:1,337-1,417	c:5,345-5,666	c:21,371-22,655
	t:0-1	t:1-2	t:5-6	t:21-23	t:83-90	t:332-358	t:1,325-1,429	t:5,298-5,712	t:21,187-22,840
64 MB	1(10%)	3(20%)	11(50%)	44(80%)	178(100%)	709(100%)	2,835(100%)	11,338(100%)	45,341(100%)
	1(10%)	3(20%)	11(50%)	43(80%)	172(100%)	689(100%)	2,753(100%)	11,007(100%)	44,020(100%)
	c:0-1	c:2-3	c:10-11	c:42-45	c:167-178	c:668-709	c:2,672-2,833	c:10,686-11,329	c:42,737-45,303
	t:0-1	t:2-3	t:10-12	t:41-45	t:166-179	t:662-715	t:2,649-2,856	t:10,594-11,421	t:42,367-45,673
128 MB	1(10%)	6(20%)	22(50%)	89(80%)	355(100%)	1,418(100%)	5,669(100%)	22,672(100%)	90,675(100%)
	1(10%)	5(20%)	22(50%)	86(80%)	344(100%)	1,376(100%)	5,504(100%)	22,012(100%)	88,034(100%)
	c:1-2	c:5-6	c:20-23	c:83-89	c:334-355	c:1,336-1,417	c:5,343-5,665	c:21,370-22,654	c:85,467-90,601
	t:0-2	t:4-7	t:20-23	t:82-90	t:331-358	t:1,324-1,429	t:5,297-5,711	t:21,185-22,839	t:84,729-91,339
256 MB	3(10%)	11(20%)	44(50%)	177(80%)	709(100%)	2,834(100%)	11,336(100%)	45,341(100%)	181,343(100%)
	3(10%)	11(20%)	43(50%)	172(80%)	688(100%)	2,752(100%)	11,006(100%)	44,020(100%)	176,062(100%)
	c:1-4	c:9-12	c:41-46	c:166-178	c:667-709	c:2,671-2,833	c:10,684-11,328	c:42,736-45,305	c:170,928-181,195
	t:1-5	t:9-13	t:40-46	t:164-180	t:661-715	t:2,647-2,857	t:10,592-11,421	t:42,367-45,674	t:169,451-182,672
512 MB	6(10%)	22(20%)	89(50%)	354(80%)	1,417(100%)	5,668(100%)	22,671(100%)	90,679(100%)	362,680(100%)
	5(10%)	22(20%)	86(50%)	344(80%)	1,376(100%)	5,503(100%)	22,011(100%)	88,038(100%)	352,117(100%)
	c:2-8	c:18-25	c:81-91	c:332-356	c:1,334-1,418	c:5,340-5,666	c:21,367-22,654	c:85,469-90,606	c:341,850-362,383
	t:1-9	t:17-26	t:80-93	t:328-360	t:1,322-1,430	t:5,294-5,712	t:21,182-22,839	t:84,731-91,345	t:338,897-365,337
1 GB	11(10%)	44(20%)	177(50%)	709(80%)	2,834(100%)	11,335(100%)	45,340(100%)	181,355(100%)	725,354(100%)
	11(10%)	43(20%)	172(50%)	688(80%)	2,751(100%)	11,005(100%)	44,020(100%)	176,073(100%)	704,228(100%)
	c:5-17	c:36-50	c:162-182	c:663-713	c:2,667-2,836	c:10,680-11,330	c:42,732-45,307	c:170,936-181,210	c:683,694-724,761
	t:3-19	t:34-52	t:159-185	t:656-720	t:2,643-2,860	t:10,587-11,424	t:42,362-45,677	t:169,458-182,687	t:677,788-730,667

Table 4.12: Ranger Problem Space, Statistical Information: This table identifies the problem space and compares the performance of synchronous checkpointing to the performance of a single process checkpointing the same amount of data. So, one process can checkpoint 16 MB 8 minutes faster than the 2,048 processes can each checkpoint 16 MB. The numbers reflect the difference in minutes between one process checkpointing and multiple processes checkpointing and show the upper bound for improvement by staggered checkpointing. Below the red (or dark gray) line synchronous checkpointing saturates the file system. One process writing the amounts of data shown in this table does not saturate the file system.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0	0	1	5	19	74	296	1,184	4,736
8 MB	1	1	2	6	23	90	356	1,421	5,674
16 MB	2	3	5	13	46	180	712	2,839	11,342
32 MB	4	6	10	26	93	359	1,423	5,675	22,678
64 MB	9	12	20	53	186	718	2,844	11,346	45,349
128 MB	19	23	40	107	373	1,436	5,687	22,690	90,693
256 MB	39	47	81	214	745	2,871	11,373	45,377	181,380
512 MB	79	95	162	428	1,490	5,741	22,744	90,752	362,753
1 GB	158	191	324	855	2,981	11,482	45,487	181,502	725,501

Table 4.13: Ranger Problem Space: This table reports the number of minutes needed for a number of processes to each synchronously checkpoint an amount of data. So 2,048 processes checkpointing 16 MB of data requires 13 minutes.

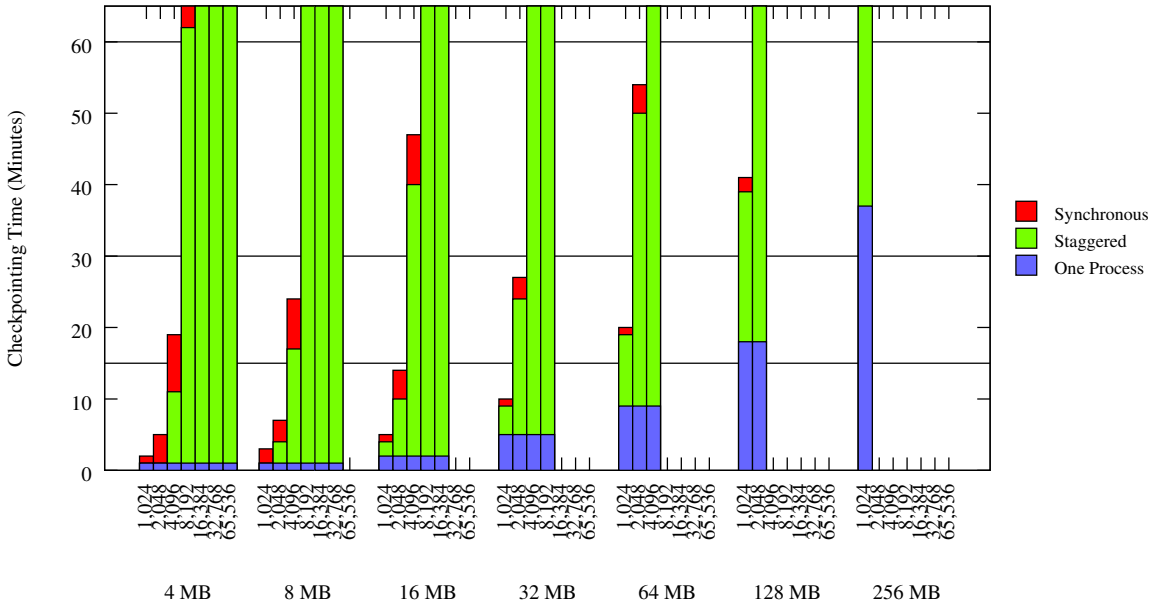


Figure 4.8: Ranger Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a three minute interval size.

Solution Space. As in our Lonestar analysis, our solution space results show that staggered checkpointing reduces checkpoint overhead for many configurations. Since the Ranger file system is faster than that of Lonestar, staggered checkpointing is able to benefit a larger number of process configurations and data sizes than it could on Lonestar. However, it has less benefit for the small configurations. This result applies to every interval size.

On Ranger, staggering the checkpoints over even a small interval size can improve checkpoint time, but the time is only improved by a small amount. For example, Figures 4.8 and 4.9 and Tables 4.14 and 4.15, show improvement

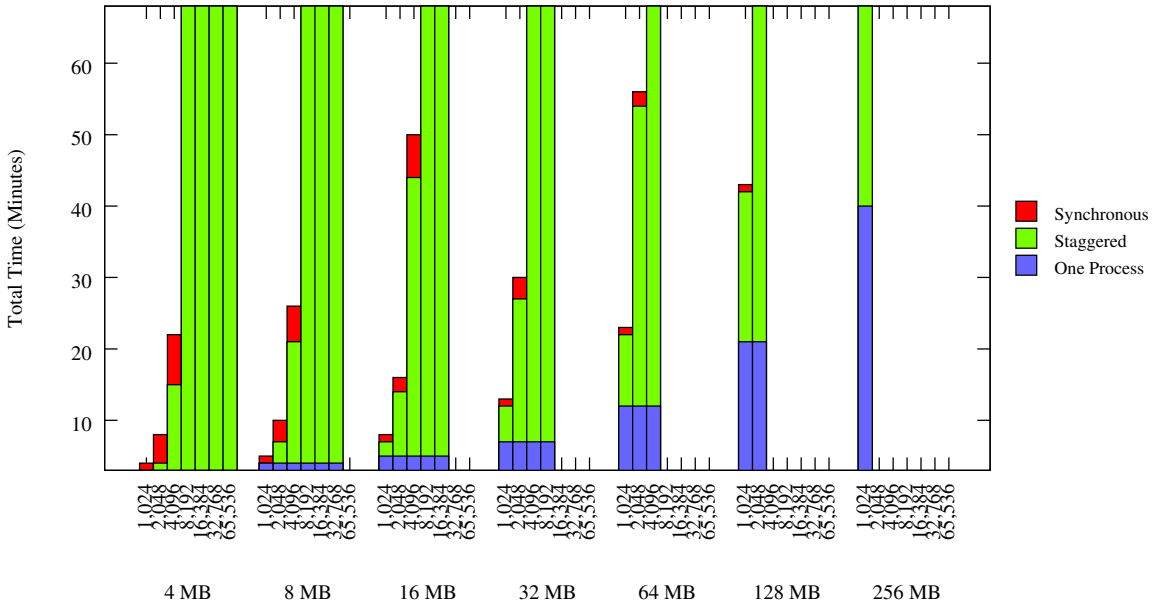


Figure 4.9: Ranger Solution Space (Checkpoint Time in 3 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a three minute interval size.

for checkpoints staggered within the three minute interval size. Staggered checkpointing does improve the checkpointing time by at least 50% for three configurations: 1,024 processes and 2,048 processes each writing 4 MB of data, and 2,048 processes writing 8 MB of data. However, in each case, the total improvement is four minutes or less. For these small configurations in this small interval size, staggered checkpointing is useful but shows small absolute savings. For larger configurations, the improvement is negligible.

The sweet spot, shown in Table 4.14, only contains four configurations for this interval size. Only a single configuration, 4,096 processes writing 4 MB of data each, is converted from infeasible to feasible; its checkpoint time

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(10%) 0(10%) c:0-0 t:0-0	0(40%) 0(40%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	4(80%) 4(80%) c:4-4 t:4-4	**8(40%)** **8(40%)** **c:7-9** **t:7-9**	12(20%) 12(20%) c:9-15 t:9-16	15(10%) 15(10%) c:3-27 t:0-30	17(0%) 16(0%) c:-31-63 t:-44-77	17(0%) 16(0%) c:-173-205 t:-227-260
8 MB	0(10%) 0(10%) c:0-0 t:0-0	0(20%) 0(20%) c:0-0 t:0-0	1(40%) 1(40%) c:1-1 t:1-1	3(50%) 3(50%) c:3-3 t:3-3	6(30%) 6(30%) c:5-7 t:5-7	10(10%) 10(10%) c:7-14 t:6-14	14(0%) 14(0%) c:0-28 t:-4-32	16(0%) 16(0%) c:-41-72 t:-57-88	–
16 MB	0(0%) 0(0%) c:0-0 t:0-0	0(10%) 0(10%) c:0-0 t:0-1	1(20%) 1(20%) c:1-1 t:1-1	3(20%) 3(20%) c:3-4 t:3-4	6(10%) 6(10%) c:5-8 t:4-9	11(10%) 10(10%) c:3-17 t:1-19	14(0%) 14(0%) c:-14-42 t:-23-50	–	–
32 MB	0(0%) 0(0%) c:0-0 t:0-0	0(10%) 0(10%) c:0-1 t:0-1	1(10%) 1(10%) c:1-2 t:1-2	3(10%) 3(10%) c:2-4 t:2-4	7(10%) 6(10%) c:3-10 t:2-11	11(0%) 10(0%) c:-4-24 t:-8-29	–	–	–
64 MB	0(0%) 0(0%) c:0-0 t:0-1	0(0%) 0(0%) c:0-1 t:0-1	1(10%) 1(10%) c:0-2 t:0-2	3(10%) 3(10%) c:1-5 t:0-6	7(0%) 6(0%) c:-1-14 t:-3-16	–	–	–	–
128 MB	0(0%) 0(0%) c:-1-1 t:-1-1	0(0%) 0(0%) c:-1-1 t:-1-2	1(0%) 1(0%) c:0-3 t:-1-3	3(0%) 3(0%) c:-1-7 t:-2-9	–	–	–	–	–
256 MB	0(0%) 0(0%) c:-1-2 t:-2-2	0(0%) 0(0%) c:-2-2 t:-2-3	1(0%) 1(0%) c:-2-4 t:-3-5	–	–	–	–	–	–
512 MB	0(0%) 0(0%) c:-3-3 t:-4-4	0(0%) 0(0%) c:-3-4 t:-5-5	–	–	–	–	–	–	–
1 GB	0(0%) 0(0%) c:-6-6 t:-8-8	–	–	–	–	–	–	–	–

Table 4.14: Ranger Solution Space (Checkpoint Time in 3 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately three minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 3 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(0%) 0(0%) c:0-0 t:0-0	0(10%) 0(10%) c:0-0 t:0-0	1(20%) 1(30%) c:1-1 t:1-1	4(50%) 4(50%) c:4-4 t:4-4	7(30%) 7(30%) c:6-8 t:6-8	10(10%) 11(10%) c:9-14 t:9-14	12(0%) 18(10%) c:8-27 t:7-28	13(0%) 37(0%) c:-2-76 t:-5-79	13(0%) 109(0%) c:-46-264 t:-60-279
8 MB	0(0%) 0(0%) c:0-0 t:0-0	0(10%) 0(10%) c:0-1 t:0-1	1(20%) 1(20%) c:1-1 t:1-1	3(30%) 3(30%) c:3-3 t:3-3	5(20%) 6(20%) c:5-7 t:5-7	8(10%) 10(10%) c:7-13 t:7-13	11(0%) 18(0%) c:7-30 t:6-31	12(0%) 41(0%) c:-5-88 t:-9-92	–
16 MB	0(0%) 0(0%) c:0-0 t:0-0	0(10%) 0(10%) c:0-1 t:0-1	1(10%) 1(20%) c:1-1 t:1-1	3(20%) 3(20%) c:3-4 t:2-4	5(10%) 6(10%) c:5-8 t:5-8	8(0%) 12(10%) c:6-18 t:6-18	11(0%) 25(0%) c:2-49 t:0-51	–	–
32 MB	0(0%) 0(0%) c:0-0 t:0-1	0(0%) 0(10%) c:0-1 t:0-1	1(10%) 1(10%) c:1-2 t:1-2	3(10%) 3(10%) c:2-4 t:2-4	5(10%) 7(10%) c:4-10 t:4-11	8(0%) 16(0%) c:4-27 t:3-28	–	–	–
64 MB	0(0%) 0(0%) c:0-1 t:0-1	0(0%) 1(0%) c:0-1 t:0-1	1(0%) 2(10%) c:1-2 t:1-2	3(0%) 4(10%) c:2-6 t:2-6	5(0%) 9(0%) c:3-15 t:2-16	–	–	–	–
128 MB	0(0%) 1(0%) c:0-1 t:0-1	0(0%) 1(0%) c:0-2 t:0-2	1(0%) 2(0%) c:1-3 t:0-3	3(0%) 5(0%) c:1-8 t:1-9	–	–	–	–	–
256 MB	0(0%) 1(0%) c:0-2 t:-1-2	0(0%) 1(0%) c:0-3 t:0-3	1(0%) 3(0%) c:0-5 t:0-6	–	–	–	–	–	–
512 MB	0(0%) 2(0%) c:-1-4 t:-1-5	0(0%) 2(0%) c:-1-6 t:-1-6	–	–	–	–	–	–	–
1 GB	0(0%) 3(0%) c:-2-9 t:-2-9	–	–	–	–	–	–	–	–

Table 4.15: Ranger Solution Space (Total Time in 3 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately three minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 3 minutes or 20% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

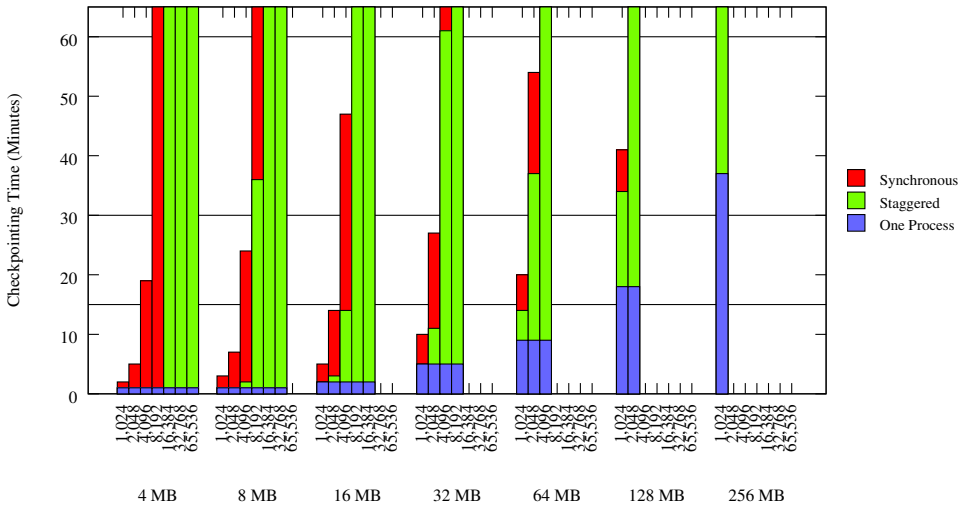


Figure 4.10: Ranger Solution Space (Checkpoint Time in 15 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size.

is reduced from 19 minutes to 10 minutes.

In Figures 4.10 and 4.11 and Tables 4.16 and 4.17, the interval size increases to approximately fifteen minutes and we see improvement in a larger number of configurations with larger numbers of processes and larger data sizes than those that were improved in the three minute interval size. In Table 4.16, the number of configurations in the sweet spot has almost tripled to 11. In addition, staggered checkpointing exhibits significant time savings. It achieves maximum improvement six configurations: up to 2,048 processes writing up to 16 MB of data each. In addition, it reduces checkpointing time by at least 50% for an additional six configurations: 1,024 and 2,048 processes writing 32 MB, 4,096 processes writing up to 16 MB, and 8,192 processes writing up to

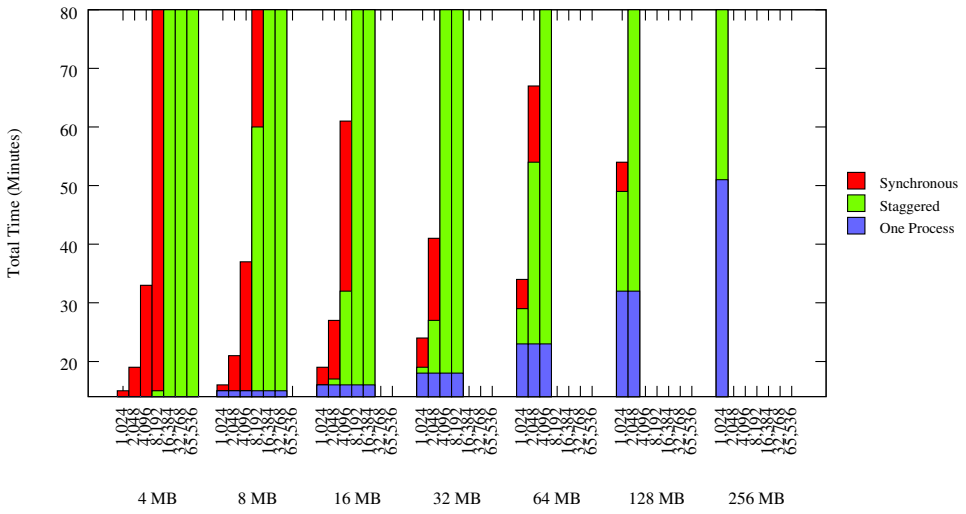


Figure 4.11: Ranger Solution Space (Total Time in 15 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a fifteen minute interval size.

8 MB. As Figures 4.10 and 4.11 show, many of these configurations show a significant absolute time improvement as well.

Five configurations are converted from infeasible to feasible in this interval size. This small number of conversions represents a significant drop from the Lonestar results, where all but three configurations were feasible in the fifteen minute interval size. However, our Ranger simulations are able to consider much larger configurations, and all the configurations represented in the Lonestar results are feasible with staggered checkpointing on Ranger in the fifteen minute interval size.

In Figure 4.11, total execution time is less than twenty minutes in the staggered case for ten configurations; the total execution time only remains

Checkpoint Size	Number of Processes 256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
	0(10%)	0(40%)	1(70%)	5(90%)	**18(100%**	**73(100%)**	77(30%)	84(10%)	85(0%)
4	0(10%)	0(40%)	1(70%)	4(90%)	**18(100%**	**71(100%)**	75(30%)	81(10%)	83(0%)
MB	c:0-0	c:0-0	c:1-1	c:4-5	**c:17-18**	**c:69-73**	c:65-86	c:36-127	c:-105-271
	t:0-0	t:0-0	t:1-1	t:4-5	**t:17-19**	**t:69-74**	t:62-89	t:23-140	t:-159-325
	0(10%)	0(30%)	1(60%)	6(80%)	**22(90%)**	54(60%)	70(20%)	81(10%)	
8	0(10%)	0(30%)	1(60%)	5(80%)	**21(90%)**	53(60%)	68(20%)	78(10%)	_
MB	c:0-0	c:0-0	c:1-1	c:5-6	**c:21-22**	c:50-56	c:55-81	c:23-134	
	t:0-0	t:0-0	t:1-2	t:5-6	**t:21-22**	t:49-56	t:52-85	t:7-150	
	0(10%)	1(20%)	3(50%)	11(80%)	**33(70%)**	53(30%)	71(10%)		
16	0(10%)	1(20%)	3(50%)	10(80%)	**32(70%)**	51(30%)	69(10%)	_	_
MB	c:0-0	c:1-1	c:3-3	c:10-11	**c:30-33**	c:45-57	c:42-96		
	t:0-0	t:1-1	t:2-3	t:10-11	**t:30-34**	t:43-59	t:34-104		
	0(10%)	1(20%)	5(50%)	**16(60%)**	33(30%)	53(10%)			
32	0(10%)	1(20%)	5(50%)	**15(60%)**	32(30%)	51(10%)	_	_	_
MB	c:0-1	c:1-1	c:4-5	**c:15-16**	c:29-35	c:38-65			
	t:0-1	t:1-2	t:4-5	**t:14-17**	t:28-36	t:34-69			
	0(0%)	2(20%)	**6(30%)**	16(30%)	33(20%)				
64	0(0%)	2(20%)	**6(30%)**	16(30%)	32(20%)	_	_	_	_
MB	c:0-1	c:1-2	**c:5-7**	c:14-18	c:25-39				
	t:0-1	t:1-2	**t:5-7**	t:13-18	t:23-41				
	1(0%)	2(10%)	6(20%)	16(20%)					
128	0(0%)	2(10%)	6(20%)	16(20%)	_	_	_	_	_
MB	c:0-1	c:1-3	c:5-8	c:12-20					
	t:-1-2	t:1-3	t:4-8	t:11-21					
	0(0%)	2(0%)	6(10%)						
256	0(0%)	2(0%)	6(10%)	_	_	_	_	_	_
MB	c:-1-2	c:0-4	c:3-9						
	t:-2-3	t:-1-4	t:2-10						
	0(0%)	2(0%)							
512	0(0%)	2(0%)	_	_	_	_	_	_	_
MB	c:-3-4	c:-2-6							
	t:-4-5	t:-3-7							
	1(0%)								
1	0(0%)	_	_	_	_	_	_	_	_
GB	c:-6-7								
	t:-8-9								

Table 4.16: Ranger Solution Space (Checkpoint Time in 15 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately fifteen minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 11 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(0%) 0(0%) c:0-1 t:0-1	0(0%) 1(0%) c:0-1 t:0-1	1(10%) 1(10%) c:1-2 t:1-2	5(20%) 5(30%) c:4-6 t:4-6	18(60%) 19(60%) c:18-20 t:18-20	73(80%) 75(80%) c:73-77 t:73-77	59(20%) 65(20%) c:56-75 t:56-75	64(10%) 88(10%) c:50-126 t:47-130	65(0%) 162(0%) c:7-316 t:-7-330
8 MB	0(0%) 0(0%) c:0-1 t:0-1	0(0%) 1(0%) c:0-1 t:0-1	1(10%) 2(10%) c:1-2 t:1-2	6(30%) 6(30%) c:6-7 t:5-7	22(60%) 23(60%) c:22-24 t:22-24	44(40%) 46(40%) c:44-49 t:43-49	54(10%) 62(20%) c:51-73 t:50-74	61(0%) 91(10%) c:45-137 t:41-141	–
16 MB	0(0%) 1(0%) c:0-1 t:0-1	1(0%) 1(10%) c:1-2 t:0-2	3(10%) 3(20%) c:3-4 t:3-4	11(40%) 11(40%) c:11-12 t:10-12	28(50%) 29(50%) c:28-31 t:28-31	41(20%) 45(20%) c:40-51 t:39-51	55(10%) 69(10%) c:47-92 t:45-94	–	–
32 MB	0(0%) 1(0%) c:0-1 t:0-1	1(10%) 2(10%) c:1-2 t:1-2	5(20%) 5(20%) c:4-6 t:4-6	14(30%) 15(40%) c:14-16 t:13-16	26(20%) 29(30%) c:26-32 t:25-32	42(10%) 49(10%) c:38-61 t:37-62	–	–	–
64 MB	0(0%) 1(0%) c:0-2 t:0-2	2(10%) 2(10%) c:1-3 t:1-3	5(20%) 6(20%) c:5-7 t:5-7	14(20%) 15(20%) c:13-17 t:13-17	27(10%) 31(10%) c:25-37 t:24-37	–	–	–	–
128 MB	0(0%) 1(0%) c:0-2 t:0-2	2(0%) 2(10%) c:1-4 t:1-4	5(10%) 6(10%) c:5-8 t:5-8	14(10%) 16(10%) c:12-20 t:12-20	–	–	–	–	–
256 MB	0(0%) 2(0%) c:0-3 t:0-3	2(0%) 3(0%) c:1-5 t:1-5	5(10%) 7(10%) c:4-10 t:4-10	–	–	–	–	–	–
512 MB	0(0%) 2(0%) c:-1-5 t:-1-6	2(0%) 4(0%) c:0-7 t:0-8	–	–	–	–	–	–	–
1 GB	0(0%) 4(0%) c:-2-10 t:-2-10	–	–	–	–	–	–	–	–

Table 4.17: Ranger Solution Space (Total Time in 15 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately fifteen minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 11 minutes or 40% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

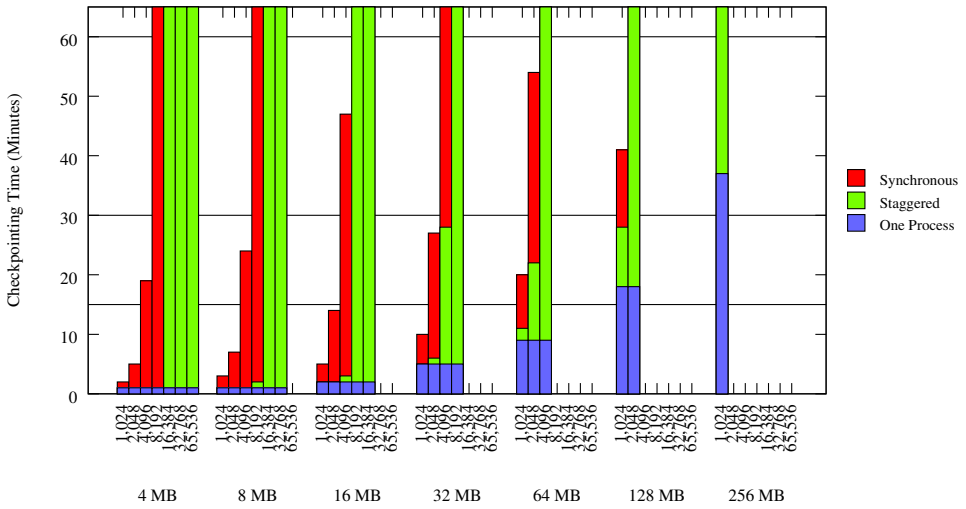


Figure 4.12: Ranger Solution Space (Checkpoint Time in 30 Minute Interval). This figure displays the time spent checkpointing for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size.

under twenty minutes in the synchronous case for four configurations. The configurations under twenty minutes for staggered checkpointing are: up to 8,192 processes writing 4 MB, 4,096 processes writing 8 MB, 2,048 processes writing 16 MB, and 1,024 processes writing 32 MB. For these configurations, checkpointing adds less than five minutes of execution time.

Figures 4.12 and 4.13 and Tables 4.18 and 4.19 show a larger benefit than the previous figures and tables, and the sweet spot has further expanded to include sixteen configurations. However, staggered checkpointing is still unable to reduce the checkpointing time to under an hour for the largest configurations. This result shows that, for checkpointing these large configurations to be viable, staggered checkpointing will need to be combined with other tech-

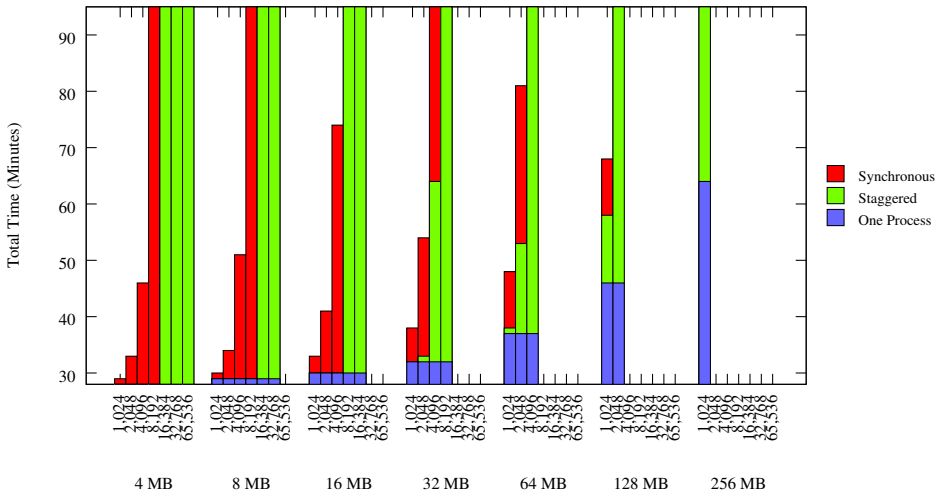


Figure 4.13: Ranger Solution Space (Total Time in 30 Minute Interval). This figure displays the total execution time for various numbers of processes writing various per process checkpoint sizes in a thirty minute interval size.

niques, such as those that reduce checkpoint size through compression or by careful choice of file format.

Error Analysis. Our results for Ranger show that staggered checkpointing is still beneficial even when we consider the error margins for our simulator. Let 1.0 represent the measured execution time of our synthetic benchmarks with checkpointing. Then from Chapter 3, we know that the average predicted error for our synthetic benchmark with synchronous checkpointing is a factor of 0.98 of the measured execution time, and the average predicted error for the synthetic benchmark with staggered checkpointing is 1.0. These errors mean that the simulator consistently underpredicts the execution time of synchronous checkpointing on Ranger. Thus, our results are

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(10%) 0(10%) c:0-0 t:0-0	0(40%) 0(40%) c:0-0 t:0-0	1(70%) 1(70%) c:1-1 t:1-1	5(90%) 4(90%) c:4-5 t:4-5	**18(100%** **18(100%** **c:17-18** **t:17-19**	**73(100%)** **71(100%)** **c:69-73** **t:69-74**	180(60%) 175(60%) c:166-184 t:163-186	167(10%) 163(10%) c:118-207 t:106-219	171(0%) 166(0%) c:-20-352 t:-74-406
8 MB	0(10%) 0(10%) c:0-0 t:0-0	0(30%) 0(30%) c:0-0 t:0-0	1(60%) 1(60%) c:1-1 t:1-2	6(80%) 6(80%) c:5-6 t:5-6	**22(90%)** **22(90%)** **c:21-22** **t:21-23**	**89(100%)** **86(100%)** **c:84-89** **t:83-89**	142(40%) 138(40%) c:126-149 t:122-153	161(10%) 156(10%) c:103-210 t:87-226	–
16 MB	0(10%) 0(10%) c:0-0 t:0-0	1(20%) 1(20%) c:1-1 t:1-1	3(50%) 3(50%) c:3-3 t:3-3	11(80%) 11(80%) c:10-11 t:10-11	**44(90%)** **43(90%)** **c:41-44** **t:41-44**	110(60%) 107(60%) c:101-112 t:100-114	142(20%) 138(20%) c:112-164 t:105-171	–	–
32 MB	0(10%) 0(10%) c:0-1 t:0-1	1(20%) 1(20%) c:1-2 t:1-2	5(50%) 5(50%) c:5-6 t:5-6	**21(80%** **21(80%** **c:20-22** **t:20-22**	66(70%) 64(70%) c:61-66 t:60-67	106(30%) 103(30%) c:90-115 t:87-119	–	–	–
64 MB	1(10%) 1(10%) c:0-1 t:0-1	3(20%) 2(20%) c:2-3 t:2-3	**10(50%** **9(50%)** **c:9-10** **t:9-10**	32(60%) 31(60%) c:29-33 t:29-33	65(40%) 64(40%) c:57-70 t:55-72	–	–	–	–
128 MB	1(0%) 1(0%) c:0-2 t:0-2	4(20%) 4(20%) c:3-4 t:2-5	12(30%) 12(30%) c:11-14 t:10-14	33(30%) 32(30%) c:28-35 t:27-36	–	–	–	–	–
256 MB	1(0%) 1(0%) c:-1-3 t:-1-3	4(10%) 4(10%) c:2-5 t:1-6	12(20%) 12(20%) c:9-15 t:8-16	–	–	–	–	–	–
512 MB	1(0%) 1(0%) c:-2-4 t:-3-5	4(0%) 4(0%) c:0-7 t:-1-8	–	–	–	–	–	–	–
1 GB	1(0%) 1(0%) c:-5-7 t:-7-9	–	–	–	–	–	–	–	–

Table 4.18: Ranger Solution Space (Checkpoint Time in 30 Minute Interval), Statistical Information: This table shows the improvement in time spent checkpointing between staggered checkpointing and synchronous checkpointing in minutes for checkpoints staggered across approximately thirty minutes of local execution. So 2,048 processes checkpointing 16 MB of data can checkpoint 11 minutes faster using staggered checkpointing than they can using synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. A bold number indicates that staggered checkpointing makes checkpointing that configuration feasible, or reduces checkpoint time to under 15 minutes. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

Checkpoint Size	Number of Processes								
	256	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**
4 MB	0(0%) 1(0%) c:0-2 t:0-2	0(0%) 1(0%) c:0-2 t:0-2	1(0%) 2(10%) c:1-3 t:1-3	5(10%) 5(20%) c:4-6 t:4-6	18(40%) 19(40%) c:18-21 t:18-21	73(70%) 76(70%) c:73-78 t:73-78	145(40%) 152(50%) c:144-160 t:143-161	127(10%) 152(10%) c:115-189 t:111-193	129(0%) 227(0%) c:73-380 t:59-394
8 MB	0(0%) 1(0%) c:0-2 t:0-2	0(0%) 1(0%) c:0-2 t:0-2	1(0%) 2(10%) c:1-3 t:1-3	6(20%) 6(20%) c:5-7 t:5-7	22(40%) 23(40%) c:22-25 t:22-25	89(80%) 91(80%) c:89-94 t:88-94	111(30%) 118(30%) c:108-129 t:107-130	123(10%) 152(10%) c:107-197 t:103-201	–
16 MB	0(0%) 1(0%) c:0-2 t:0-2	1(0%) 1(0%) c:0-2 t:0-2	3(10%) 3(10%) c:2-5 t:2-5	11(30%) 12(30%) c:11-13 t:11-13	44(60%) 45(60%) c:44-47 t:44-47	90(40%) 94(40%) c:89-99 t:88-100	109(10%) 124(20%) c:102-146 t:100-148	–	–
32 MB	0(0%) 1(0%) c:0-2 t:0-2	1(0%) 2(10%) c:1-3 t:1-3	5(10%) 6(20%) c:5-7 t:5-7	21(40%) 22(40%) c:21-24 t:21-24	57(50%) 59(50%) c:56-62 t:56-62	83(20%) 91(20%) c:80-102 t:79-103	–	–	–
64 MB	1(0%) 1(0%) c:0-3 t:0-3	2(10%) 3(10%) c:2-4 t:2-5	9(20%) 10(20%) c:9-12 t:9-12	28(30%) 29(40%) c:27-31 t:27-32	53(20%) 57(30%) c:51-63 t:51-64	–	–	–	–
128 MB	1(0%) 2(0%) c:0-3 t:0-3	3(10%) 4(10%) c:3-6 t:2-6	10(20%) 12(20%) c:10-14 t:10-14	27(20%) 30(20%) c:26-34 t:26-34	–	–	–	–	–
256 MB	1(0%) 2(0%) c:0-4 t:0-5	3(0%) 5(10%) c:2-7 t:2-7	10(10%) 13(10%) c:9-16 t:9-16	–	–	–	–	–	–
512 MB	1(0%) 3(0%) c:0-6 t:-1-7	3(0%) 6(0%) c:2-10 t:1-10	–	–	–	–	–	–	–
1 GB	1(0%) 5(0%) c:-1-11 t:-2-11	–	–	–	–	–	–	–	–

Table 4.19: Ranger Solution Space (Total Time in 30 Minute Interval), Statistical Information: This table shows the execution time improvement in minutes between staggered checkpointing and synchronous checkpointing for processes executing approximately thirty minutes of local instructions. The number in parentheses represents the percentage improvement. The table shows that for 2,048 processes checkpointing 16 MB of data the execution time is reduced by 11 minutes or 30% using staggered checkpointing rather than synchronous checkpointing. Below the green (or light gray) synchronous checkpointing saturates the file system. Below the red (or dark gray) line staggered checkpointing saturates the file system. "c:" and "t:" indicate the confidence and tolerance intervals, respectively.

conservative, and staggered checkpointing should produce more benefit than we see here.

4.2.3 The Effects of Interval Size

In our results, we identify the sweet spot for each interval size. As the interval size increases, the sweet spot expands so that it covers configurations with larger numbers of processes checkpointing larger amounts of data. Configurations above and to the left of the sweet spot typically have a very fast synchronous checkpoint time and staggered checkpointing is not necessary. Configurations below and to the right of the sweet spot are so large that staggered checkpointing cannot reduce checkpoint time to under 30 minutes.

As the interval size increases, the space where staggered checkpointing achieves maximum benefit expands to include larger configurations. On Ranger, we see that only four configurations are in the sweet spot for the three minute interval size. This number almost triples for the fifteen minute interval size, and then increases by just over a third for the thirty minute configuration. On Lonestar, we see that the three minute interval size provides a sweet spot of six configurations and that the size of the sweet spot doubles for the fifteen minute interval size. However, the size of the sweet spot has stabilized for the sixteen minute interval size. This effect occurs since once the processes at one checkpoint location can complete their checkpoints before the processes at the next checkpoint location start, more time separating the checkpoints does not provide more opportunity to reduce contention. For instance, on Lonestar

for 256 processes writing 8 MB we achieve maximum benefit staggering over just three minutes of local execution time (Figure 4.1). For larger configurations, such as 256 processes writing 32 MB and 512 processes writing 16 MB, we achieve maximum benefit when checkpoints are staggered through fifteen minutes of local execution time (Figure 4.4), and 1,024 processes writing 16 MB need thirty minutes of local execution time to achieve maximum benefit (Figure 4.6). We also see this effect in Table **??**, where the sweet spot only increases by one configuration from the 15 minute interval.

4.2.4 Comparison Between Systems

The Ranger file system is faster than that of Lonestar, so the Ranger results show that improvement is realized in larger configurations at smaller time scales. However, the effect of contention from synchronous checkpointing on Ranger is greatly reduced from its effect on Lonestar, so there is less room for improvement through staggered checkpointing. Thus, Lonestar shows more improvement in these figures and tables, but if we compare the amount of checkpointing time required for staggered checkpointing to write the checkpoints for each configuration, staggered checkpointing on Ranger makes checkpointing viable for more, and larger, configurations than staggered checkpointing on Lonestar.

We can use the staggered checkpointing performance trends between Lonestar, the older supercomputer, and Ranger, the newer supercomputer, to predict how staggered checkpointing will behave on future systems. We see

that better file system performance delays the need for staggered checkpointing, but, once staggered checkpointing is necessary, it also extends the area where it is beneficial. We can see this effect in the sweet spots of our tables. While the sweet spots do not differ much in terms of size, the Ranger sweet spots include many more configurations with at least 1,024 processes. As file systems become faster, staggered checkpointing will become more beneficial to larger numbers of processes each writing larger amounts of data than it is currently.

As systems grow larger, the applications running on them will also grow larger, so staggered checkpointing will continue to be useful. We expect the applications will continue to grow larger due to the current support for capability computing, which encourages application developers to use more processes per application. For instance, the original RedStorm allocation policies set aside 80% of the compute-node hours to be used by applications using a minimum of 40% of the computer nodes [147]. RedStorm initially had 10,880 processors, and now has 38,400. In addition, many large-scale applications use weak scaling, so enough work exists for each process even when process counts are large. However, our results show that as staggered checkpointing is applied to larger configurations, it also needs a larger interval size to show improvement. In addition, to checkpoint large configurations such as 16,384 processes writing 4 MB of data each in a reasonable amount of time, staggered checkpointing will need to be combined with other techniques, such as data compression.

4.2.5 Relationship to Application Characteristics

Using the approximate local execution time and the number of processes, we can relate these spaces to our application benchmarks, `BT`, `SP`, `LU`, which are NAS parallel benchmarks [51], and `Ek-simple`, a well-known CFD benchmark, for various checkpoint sizes. The characteristics for a single phase of each of our benchmarks are reported in Table 4.20. Recall that the processes checkpoint throughout the execution time, so the local execution time corresponds with the times represented by our tables. Comparing these characteristics to our results, we see that the benchmarks most closely relate to the results presented for the three minute interval size (Figures 4.1, 4.3), 4.10, and 4.11). To achieve these per-phase execution times, which are much longer than the standard execution times for these benchmarks, we assume very high problem sizes. As a result, the live data sizes for these benchmarks with these configurations are higher than the expected amount of checkpoint data and, for `BT` and `Ek-Simple`, higher than the amount of memory available on most systems.

In addition, Table 4.21 displays the characteristics of five scientific applications. Comparing these applications to our problem and solution spaces, we see that the first two applications, modeling tides [114] and supersonic turbulence [6], could perform synchronous checkpointing on Ranger for the data sizes we investigate. On Lonestar, both applications are likely to benefit from staggered checkpointing for data sizes over 64 MB for the tide modeling application and 32 MB of the supersonic turbulence application. The earthquake

Benchmark	Processes	Approximate Local Execution Time (s)		Liveness
		Lonestar	Ranger	Size
BT	1,024	3m20s	2m21s	62 GB
	4,096	5m47s	4m6s	98 GB
	16,384	4m32s	3m13s	114 GB
Ek-simple	1,024	3m6s	2m44s	534 GB
	4,096	3m6s	2m44s	534 GB
	16,384	3m6s	2m44s	534 GB
LU	1,024	3m31s	3m11s	14 GB
	4,096	2m55s	2m38s	15 GB
	16,384	4m6s	3m43s	23 GB
SP	1,024	2m41s	3m4s	21 GB
	4,096	2m57s	3m20s	26 GB
	16,384	2m50s	3m18s	31 GB

Table 4.20: Liveness sizes and instructions in a single phase for a subset of the NAS parallel benchmarks [51], `BT`, `LU`, and `SP`, and a well-known CFD benchmark, `Ek-simple`.

Application Subject	Processes	Approximate Execution Time
Modeling Tides [114]	256	45 days
Supersonic Turbulence [6]	512	41 days
Earthquakes [60]	2,000	35 hours
Turbulent Combustion [55]	10,000	10 days
Turbulence [23]	65,536	1 week

Table 4.21: Characteristics of Real-World Applications

simulation [60] would benefit from staggered checkpointing for checkpoint sizes over 32 MB on Ranger and 4 MB on Lonestar. The fourth application, which studies turbulent combustion [55], would benefit from staggered checkpointing on Ranger, but for the checkpointing time to be feasible it would need to checkpoint less than 8 MB of data in at least a fifteen minute interval. The last application also could benefit from staggered checkpointing on Ranger, but staggered checkpointing cannot decrease the checkpoint time below and hour with an interval size of thirty minutes or less.

4.3 Conclusions

Within the solution spaces, staggered checkpointing reduces both checkpointing time and overall execution time. The solution spaces as presented are significant for both Lonestar and Ranger; staggered checkpointing is a useful technique for both these systems. Checkpoint locations only need to be separated by approximately three minutes for improvements to be realized in some configurations on both Lonestar and Ranger. As the time separating the checkpoints increases, the amount of improvement shown increases for both machines.

An important result of staggered checkpointing is its ability to make checkpointing feasible for some configurations when it was not before. As expected, the number of converted configurations increases as the amount of time in which to stagger increases. However, some configurations require over an hour to checkpoint for even the thirty minute interval size. To checkpoint these configurations, staggered checkpointing should be combined with other techniques, such as data compression.

Since Ranger is ranked 11 in the Top 500 Supercomputer Sites [8] and Lonestar is ranked 123, these techniques will extend to a large range of supercomputers.

Chapter 5

Algorithm for Compiler-Assisted Staggered Checkpointing

This chapter describes our algorithm and set of heuristics that allow the compiler to place staggered checkpoints in parallel applications. Placing staggered checkpoints is challenging because of the large solution search space, which grows as L^P, where L is the number of possible checkpoint locations and P is the number of processes executing the checkpoint locations. Moreover, we have observed that the number of valid solutions grows more slowly than the number of invalid solutions, making the search space extremely sparse.

Since the search space is large and sparse, our algorithm must efficiently prune the parts of the space that are least likely to contain valid solutions. Thus, our algorithm performs a constrained search [208] that exploits dependences among processes to prune large portions of the search space that cannot contain valid solutions. Our algorithm reduces the space by grouping processes into sets that checkpoint together; the sets are determined by dependences introduced by the application's communication. The use of these sets may eliminate some valid solutions from the search space, but we explain how it will eliminate orders of magnitude more invalid solutions. Our algorithm

quantifies the quality of a solution using the *Wide and Flat* (WAF) metric, which identifies the interval of a solution's checkpoint locations (width) and the number of processes that checkpoint at each (flatness). As we'll see, it also applies this metric to partial solutions so that undesired ones may be pruned earlier in the algorithm. This pruning further reduces the search space but often eliminates both valid and invalid solutions.

We evaluate our techniques on a set of four commonly used parallel benchmarks: `BT`, `SP`, and `LU` from the NAS Parallel Benchmark Suite [51], and `Ek-simple` a well-known computational fluid dynamics benchmark. For 16,384 processes, our simulations show that our compiler-based staggered checkpointing reduces checkpoint time by up to 52% when compared against the time necessary for that number of processes to simultaneously write an equivalent amount of data. We also show that our WAF metric correlates well with actual checkpoint overhead and overall execution time.

5.1 Background

In this section, we define terms and concepts that we need to explain our algorithm.

A *recovery line* is a set of checkpoints, one checkpoint per process. A *valid recovery line* represents a consistent state, or a state that could have existed in the execution of the program [67], so it disallows the save of a message *receive* if the corresponding *send* of the message is not also saved. By ensuring that data shared amongst processes is the same data everywhere,

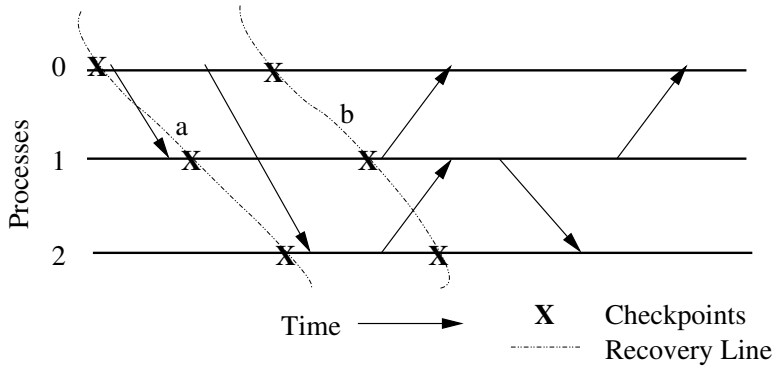


Figure 5.1: Examples of invalid (a) and valid (b) recovery lines. The arrows represent message sends. The source is the sending process and the target is the receiving process. The **X**'s mark checkpoint locations.

valid recovery lines save a consistent state even in the face of non-determinism in applications. For example, recovery line a in Figure 5.1 is invalid because it saves a message as received on process 1 while process 0 does not save the corresponding send. Thus, the state represented by a could not exist in an actual execution of the system. By contrast, recovery line b is valid because it could have existed in a possible execution.

Our algorithm specifies a recovery line as set of {*process, checkpoint location*} pairs; it has a pair for each process executing the application. Since our algorithm builds each recovery line over time, we define a *partial recovery line* as a set of {*process, checkpoint location*} pairs for a subset of the processes in the system; a *partial valid recovery line* is a partial recovery line that represents a state that could have existed in the execution of the program when only the represented processes are considered.

Valid recovery lines can differ in the number of processes that checkpoint at approximately the same time, and thus they differ in terms of the network and file system congestion that they induce. Thus, we refer to one recovery line as being better than another if it reduces that contention and thus has lower checkpoint overhead.

5.2 Algorithm Overview

The goal of our algorithm is to stagger the checkpoints and thus create a *useful* recovery line, which is a line that is both valid and sufficiently staggered to reduce contention. However, you could imagine a case where process dependences interact to disallow staggered checkpoints. Thus, our algorithm guarantees the placement of a valid recovery line, but it does not guarantee that the checkpoints will be staggered. Our algorithm, has three phases: the first identifies all communication, the second determines inter-process dependences, and the third generates the recovery lines. All phases perform a context-sensitive analysis.

In this chapter, we present the components of our algorithm in the order in which they analyze the application. Phase One is a straightforward application of known technology. Phase Two applies our novel use of vector clocks to statically track process dependences. However, our most novel and important contributions, such as the techniques we use to prune the search space and our metric for identifying useful recovery lines, occur in Phase Three and are presented in Section 5.5.

5.3 Algorithm Phase One: Identifying Communication

This phase of our algorithm identifies an application's communication events, identifies each event's communicating processes, and matches the sends with their corresponding receives. Identifying communication events is straightforward; identifying the communicating processes is more complex.

Two factors determine which processes are communicating: the process(es) that executes the communication call and the neighbor process(es) that is indicated in the arguments to that call. The algorithm uses *symbolic expression analysis* to identify these processes. Symbolic expression analysis is a form of backwards constant propagation where values are expressed in terms of the variables, constants, and operations that created them. We assume that these expressions consist of arithmetic operations on constants, the number of processes executing the application, and the identifier of the process where the call is occurring. For now, the latter two do not need to be constants.

Figure 5.2 shows a snippet of code including a communication event, `MPI_Irecv()` and the resulting symbolic expression analysis performed on the argument to the call that identifies the communicating process, `from_process`. In this example, `node` represents the process identifier and `no_nodes` represents the number of processes executing the application. The symbolic expression analysis first identifies the definition of `from_process` immediately preceding the call; in this example, the definition occurs in line 6. We see that the arithmetic expression on the right hand side of the definition includes the variables `i`, `p`, and `j`. The analysis next identifies the most recent definitions of those

variables (found in lines 5, 1, and 4, respectively). The analysis continues until each variable's value can be represented by arithmetic operations on `node` or `no_nodes`. It then combines these definitions into a single arithmetic expression representing the value of `from_process`. The final expression can be seen in the lower box of Figure 5.2. The compiler also performs a control-dependence analysis to detect when a communication event's execution or communicating processes depend on a preceding control structure(s); this information is used to further identify the communicating processes.

To match the sends to their respective receives and the non-blocking calls to their respective waits, we assume that the number of processes the application will use to execute is statically known. For each communication call, our algorithm evaluates the control-dependence information for each process identifier and, in doing so, determines which processes execute the call. For each process that executes the call, the compiler evaluates the relevant symbolic expressions for each process identifier and then matches the calls based on their communicating processes, message identifiers, and location in the application text. Previous research has presented more complex algorithms that allow for loop unrolling to match calls and other complications [117, 118], but we have implemented a simpler algorithm that is successful for our benchmarks.

```
1. p = sqrt(no_nodes)
2. cell_coord[0][0] = node % p
3. cell_coord[1][0] = node /p
4. j = cell_coord[0][0] − 1
5. i = cell_coord[1][0] − 1
6. from_process = (i −1 + p) % p + p * j
7. MPI_Irecv( x, x, x, from_process, ...)
```

```
from_process = (node / (sqrt(no_nodes)) −1 −1 +
                sqrt(no_nodes)) % sqrt(no_nodes) +
                sqrt(no_nodes) * (node % sqrt(no_nodes) −1)
```

Figure 5.2: A code example from the NAS parallel benchmark BT and its corresponding result from symbolic expression analysis. In this example, `node` represents the process identifier and `no_nodes` represents the number of processes executing the application.

5.4 Algorithm Phase Two: Determining Inter-Process Dependences

Our algorithm identifies inter-process dependences using *vector clocks* [72, 136]. To do so, it uses the communication information generated in the first phase and assumes that the number of processes the application will use is statically known. The vector clocks are generated and maintained statically; to our knowledge, we are the first to apply them in this manner.

Vector clocks extend Lamport's logical clocks [121] in which each process maintains its own clock, so that each process maintains its logical clock as well as a vector representing the events in the system on which it depends. The vector has an element for each process in the system, and when a process receives a message, it sets each of the vector elements to the maximum

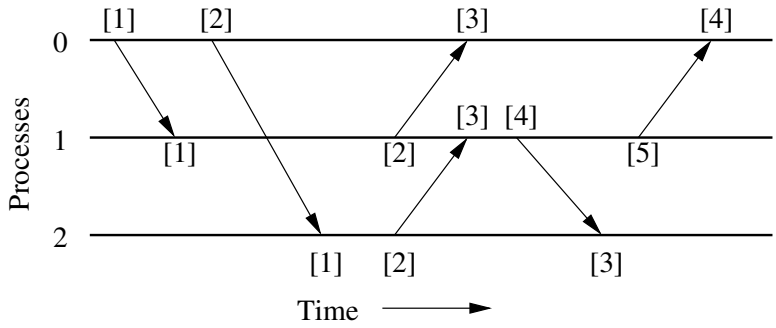


Figure 5.3: An example of Lamport's logical clocks. Each process's clock value is indicated by the [#], where # is the value of the clock.

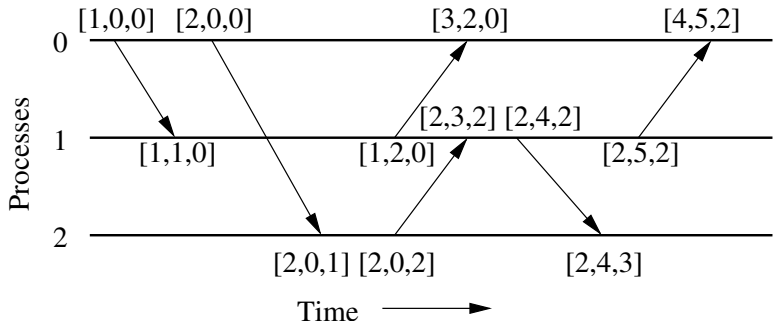


Figure 5.4: An example of vector clocks.

of that element's present value and the value of the corresponding element in the sending process's vector clock; in this way, the process's vector clock represents the events on which it relies. A vector clock is maintained locally by each process i as follows:

$VC(e_i)[i] := VC[i] + 1$
if e_i is an internal or send event

$VC(e_i) := max\{VC(e_i), VC(e_j)\}$
$VC(e_i)[i] := VC[i] + 1$
if e_i is receive from process j
where the send was event e_j

where e_i is an event e on process i, $VC(e_i)$ is its vector clock, and $VC(e_i)[i]$ is the element for process i in that vector clock.

In our algorithm, each process increments its clock for dependence-generating events and only those events. An example showing each process's logical clock can be seen in Figure 5.3; in this example, each process increments its clock at a send or a receive event. Figure 5.4 extends this example to show the resulting vector clocks. In this figure, process 0 begins by sending two messages. For each, it increments its element in its vector clock, so after sending the message to process 2, its vector clock is $[2, 0, 0]$. When process 2 receives the message from process 0, it updates its vector clock to reflect both that an event occurred (the receive) and that it is now dependent on process 0 executing at least two events. So, process 2's clock becomes $[2, 0, 1]$. When process 2 then sends a message to process 1, it increments its vector clock to $[2, 0, 2]$ to reflect another event (the send). Immediately before process 1 receives the message, its vector clock is $[1, 2, 0]$. Upon receiving the message, process 1 increments its element in its vector clock to reflect the receive, and then sets its element for process 2 to 2 to show that it depends on process 2 executing at least two events. Process 1 also sets its element for process 0 to

the maximum value of its element for process 0 and that of process 2. Since process 1 had previously only depended on the first event of process 0 and process 2 depends on the first two, then process 1 updates its clock to reflect a dependence on the first two. Therefore, process 1's vector clock becomes $[2, 3, 2]$.

Given vector clocks, a valid recovery line can be determined using the following formula [153], which states that the recovery line is *not* valid if process j requires more events on process i than process i's clock reflects.

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VC(e_i)[i] \geq VC(e_j)[i]$$

Using communication information gleaned in the previous phase, our compiler creates a vector clock for each node. For each communication call, our compiler iterates through each process that executes that call and updates that process's vector clock. We assume that each non-blocking receive occurs at its associated wait; all other events occur at their calls. To assist the recovery line generation algorithm, the state of each process's vector clock is saved at each communication call for the algorithm to reference. Unfortunately, this storage is expensive, sized $O(P^2)$ at each communication call, where P is the number of processes. These vector clocks are sparsely populated, so we represent them as sparse matrices to reduce their size, reducing the memory footprint of our compiler when analyzing the NAS parallel benchmark `BT` [51] configured to use 64 processes by 75%.

5.5 Algorithm Phase Three: Generating Useful Recovery Lines

The third phase of our algorithm generates recovery lines; we carefully design this phase to eliminate redundant work and reduce the search space. To appreciate the algorithm used by this phase, it is important to first understand why a more obvious *naïve* algorithm does not work efficiently. Thus, we begin by explaining a naïve algorithm.

In all of the algorithms that we discuss, checkpoint locations are limited to dependence-generating events. Adjusting the checkpoint locations in the generated recovery lines relative to those checkpoint locations we consider may both increase the separation of the checkpoints and reduce the amount of data checkpointed—both of which can further reduce network and file system contention. However, we defer such considerations to future work.

5.5.1 Naïve Algorithm

A naïve algorithm, which is depicted in Figure 5.5, uses exhaustive depth-first search and employs backtracking when a recovery line is found to be invalid.

Since the search space is so large, this algorithm, which has a complexity of $O(P^2 * L^P)$, where P is the number of processes and L is the number of checkpoint locations, is infeasible for even small values of P. Even though it performs simple pruning by backtracking, this algorithm also performs a large amount of redundant work: any time the same two {*process, checkpoint*

location} pairs are assigned to a recovery line, the validity of that combination of pairs is checked. The combination of those pairs is considered valid if the two pairs could exist in a valid recovery line. For example, assume that the combination of pairs $\{P_0, L_3\}$ and $\{P_3, L_0\}$ is invalid. When the algorithm is ready to assign a location for P_3, P_0's checkpoint location is already set, as are the checkpoint locations of P_1 and P_2. The algorithm attempts to assign P_3 to checkpoint at L_0 and it finds that the line is invalid; the algorithm proceeds to another checkpoint location for P_3 (if one exists) or returns to P_2. Either way, the algorithm will eventually return to P_2 since this is an exhaustive depth-first search. When it returns to P_2, assume that it finds another checkpoint location that is valid with those of P_0 and P_1. Then the algorithm continues to P_3. It will first attempt to let P_3 checkpoint at L_0—even though the checkpoint location for P_0 has not changed and that combination has already tested invalid once. Hence, this algorithm is inefficient [146].

5.5.2 Our Algorithm

We now describe our algorithm, which employs several techniques to reduce the search space. We explain each technique separately, beginning with our foundation algorithm, which eliminates portions of the search space that contain no valid recovery lines. Then we present techniques used to reduce the number of considered checkpoint locations, L. Next, we describe two heuristics. The first reduces the number of considered processes, P, and prunes large numbers of invalid recovery lines and also may eliminate some

```
straight_rl_gen(0, NULL);

//recurse for every process in the system
straight_rl_gen(process,recovery_line) {

  //location is valid for every process

  if(process == system_size) //valid found!
    add recovery_line to set of valid recovery lines

  //for each checkpoint location
  for(loc = 0; loc < location_num; loc++)
  {
    //add process checkpointing at
    //loc to recovery_line
    if(line is still valid)
      straight_rl_gen(process + 1, recovery_line);

    //remove addition from recovery_line
  }
}
```

Figure 5.5: Pseudocode for a naïve algorithm

valid recovery lines. The second heuristic produces recovery lines that yield lower checkpoint overhead; this heuristic may prune both valid and invalid recovery lines from consideration.

5.5.2.1 Foundation Algorithm

Our algorithm is a constraint-satisfaction solution synthesis algorithm[1] that builds valid recovery lines by combining valid {*process, checkpoint location*} pairs into ever larger partial valid recovery lines until a valid recovery line is generated. This technique eliminates redundant comparisons of invalid {*process, checkpoint location*} pair combinations. Each invalid combination is immediately pruned from further consideration and, once pruned, is not regenerated.

Our algorithm begins by placing processes into partitions of size k. For each partition, the algorithm generates all partial valid recovery lines. It then merges j partitions to create a new partition. It combines the partial valid recovery lines from each source partition to form new recovery lines, storing the valid ones. The algorithm repeats these steps until all processes are in the same partition and the generated valid recovery lines are complete. k and j are parameters to the algorithm. In our experiments, we found the algorithm to be fastest and generate the best results for $k = 2$ and $j = 2$.

Figure 5.6 shows an example of how processes are placed into partitions that are merged; the example reflects $k = 3$ and $j = 2$. At the lowest level are the processes themselves, which are then placed into partitions of size 3. For each partition, the algorithm generates all partial valid recovery lines then merges it with another partition to create a new partition. At the top

[1]It is based on the basic Essex solution synthesis algorithm [208]. It uses upward constraint-propagation to build a lattice representing the minimal problem.

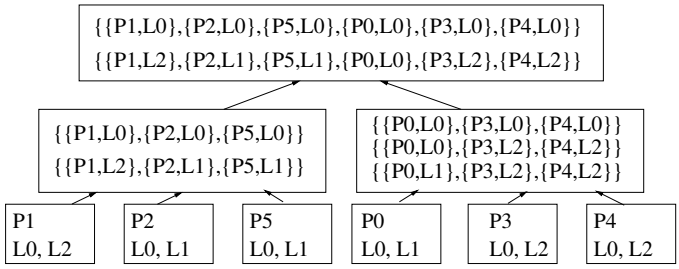


Figure 5.6: An example of partitioning of processes. At the base of the diagram, each block represents a process and that process's checkpoint locations. The algorithm begins at the bottom, merging three blocks into a partition and finding all partial valid recovery lines for that partition. As the algorithm completes each level, the partitions of that level are merged to form the partitions of the next level. In this figure, each line in the merged partitions represents either a partial valid recovery line or a complete valid recovery line.

partition level, all four processes are in the same partition and the generated valid recovery lines are complete.

The execution complexity of this algorithm is $O(log(P) * L^{P-2})$. In the next section, we discuss steps taken to reduce the value of L; then we discuss steps taken to reduce the value of P.

5.5.2.2 Reducing L, the Number of Checkpoint Locations

Since the search space is sized L^P, we can reduce the search space by reducing L, the number of checkpoint locations considered. To do this, we divide the application program into *phases*, which are sets of checkpoint locations delimited by events that synchronize all processes, such as barriers or collective communication. Since valid recovery lines cannot straddle such events, our algorithm can search within each phase without missing any valid recovery lines.

The algorithm now must execute once per phase, but L is greatly reduced for each algorithmic pass. Any application without any synchronizing events is considered a single phase.

Additionally, in some applications so few local operations, or operations that execute entirely on the local machine, separate some of the dependence-generating events that these events will likely occur in close temporal proximity. Recall that our algorithm considers these events to be checkpoint locations; separating process checkpoint locations across events that are spaced so closely will minimize any ability they have to reduce contention and thus checkpoint overhead. We therefore merge any checkpoint locations that are not separated by some minimum amount of local operations. The choice of this amount is important, as it needs to reduce the search space but still allow enough locations for our algorithm to find useful recovery lines. Merging too many checkpoint locations can cause so few checkpoint locations a useful line cannot be found but not merging checkpoint locations that are too close together to successfully reduce contention needlessly increases the search space. The minimum number of instructions that must separate two locations is a parameter to our algorithm. Increasing the number of instructions results in fewer possible checkpoint locations and a faster algorithm execution time. Decreasing the number provides more possible valid recovery lines. For our results, our algorithm merges any locations not separated by more than a two million local operations. We discuss this choice in the evaluation of our algorithm.

The execution complexity of the algorithm remains the same as that of

the foundation algorithm $O(log(P) * L^{P-2}$, but the value of L is now reduced.

5.5.2.3 Reducing P, the Number of Independent Processes

Since the number of checkpoint locations in a given phase is typically much smaller than the number of processes, multiple processes will have to checkpoint at the same checkpoint location. If we intentionally select the processes that checkpoint together to also be the ones that communicate, then, since recovery line constraints are generated by communication, the number of invalid recovery lines will be reduced. Thus, our algorithm reduces the search space by generating *clumps*: clumps are sets of processes that communicate between two consecutive checkpoint locations. Our algorithm considers each clump as a single entity when it generates a recovery line, and so every process in a clump checkpoints at the same checkpoint location.

Any processes that communicate between two consecutive checkpoint locations form a clump. For example, in Figure 5.7, the clump between checkpoint locations 0 and 1 is `(process 0, process 1)`. All processes are in the same clump between checkpoint locations 1 and 2. Between checkpoint locations 2 and 3, the clumps are `(process 0, process 1)` and `(process 2, process 3)`. In the algorithm, all clumps are associated with their defining checkpoint locations. The recovery line algorithm considers clumps as indivisible, and thus the processes within a clump checkpoint at a single location.

Once all clumps have been formed, *alike* clumps, or clumps that have the same member processes, are identified and combined across checkpoint

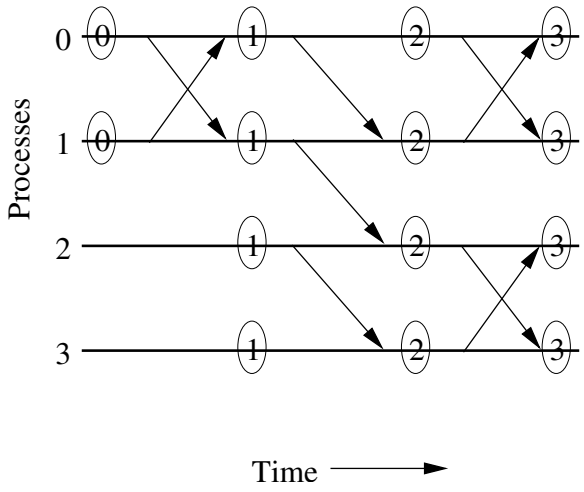


Figure 5.7: An example of process communication. Each ⓗ represents a checkpoint location.

intervals. In other words, only one clump with a particular member set will be saved, but that clump can checkpoint at any of the checkpoint locations between which any of its like clumps were found. So, in the example in Figure 5.7, the clump containing `(process 0, process 1)` between checkpoint locations 0 and 1, and the clump containing `(process 0, process 1)` between checkpoint locations 2 and 3 will be combined into a single clump with member processes 0 and 1; this clump will checkpoint a locations 0, 1, 2, and 3. Table 5.1 shows the clumps relating to the example in Figure 5.7.

Once the clumps have been consolidated, they are further combined into *clump sets*. A clump set is a group of clumps where each process is represented exactly once, such that when a recovery line is formed using every clump in a clump set that recovery line is complete. Our recovery line algorithm operates

Clump	Processes	Checkpoint Locations	Characteristic
a	(process 0, process 1)	0,1,2,3	strongly connected
b	(process 2, process 3)	2,3	strongly connected
c	(process 0, process 1, process 2, process 3)	1,2	connected

Table 5.1: Clumps for the example shown in Figure 5.7.

Clump Set	Clumps
0	clump a, clump b
1	clump c

Table 5.2: Clump sets for example in Figure 5.7.

on each clump set, and it places each of its member clumps in a recovery line, reducing P by treating each clump as a process. Clump constraints are represented in the same way as process constraints, and clumps are partitioned in the same way as processes in the basic algorithm. Table 5.2 shows the clumps sets resulting from the example in Figure 5.7.

Using clumps and clump sets reduces algorithmic complexity to $O(log(C) * L^{C-2})$, where C is the number of clumps in a clump set. The algorithm does now execute multiple times—once per clump set for each phase—but that only affects the coefficients of the reduced complexity.

5.5.2.4 The Effects of Clumps on the Search Space

By carefully determining which processes checkpoint together, clumping eliminates large numbers of invalid recovery lines. It may also eliminate

some (but fewer) valid ones. To understand how this effect occurs, we consider clumps as directed graphs, and then we identify clumps whose members are strongly connected and those whose members are not. In such a graph, the processes in the clump are the graph nodes and the communication between them forms the directed edges. If every process in the clump sends and receives between two consecutive checkpoint locations, then the graph (or clump) is strongly connected in that interval. In this case, clumping only eliminates invalid recovery lines because each process is constrained by each other process. In Table 5.1, the clumps are identified as either connected or strongly connected based on the communication in Figure 5.7.

However, if at least one process does not send *and* receive in a particular checkpoint interval, then the clump is not strongly connected, and clumping eliminates both valid and invalid recovery lines. Consider the example in Figure 5.7: with the communication shown, processes 0, 1, 2, and 3 will be clumped between checkpoint locations 1 and 2. However, the checkpoint location of process 0 does not completely constrain the possible checkpoint locations of processes 1, 2, and 3. For instance, if process 0 saves its checkpoint at location 2, then we know processes 1,2, and 3 *can* save their checkpoints at location 2, but they can also save their checkpoints at location 1. So, by forcing the processes to checkpoint at the same location, the algorithm eliminates valid recovery lines.

Fortunately, the recovery lines eliminated by this method are still largely invalid since the processes in the clump are still connected. To understand this,

let's look again at the example from Figure 5.7. This time, let us begin by considering the possible checkpoint locations of process 2. If process 2 saves its checkpoint at location 2, then processes 0 and 1 must save their checkpoints at location 2 or later, since process 2 relies on the message from process 1, forcing process 1 to save its checkpoint at location 2, and then process 1 relies on the message from process 0, forcing process 0 to save its checkpoint at location 2. If process 2 saves its checkpoint at location 1, then process 3 must also save its checkpoint at location 1, since if process 3 were to save its checkpoint at locations 2 or 3, it would save the message from process 2 that happens after location 1 and is thus not saved by process 2. By forcing these processes to save their checkpoints at the same location, the algorithm never considers the invalid recovery lines that could result when they save their checkpoints separately.

5.5.2.5 Pruning Invalid Solutions

Although our aforementioned techniques greatly reduce the number of invalid partial recovery lines in the search space, some still exist. We would like to prune these as quickly as possible.

The dependences that are generated by an application's communication are strictly increasing, so when the algorithm finds that two clumps form an invalid partial recovery line at a combination of checkpoint locations, it has information that can be used to prune other lines without additional dependence checks. Since the line is invalid, one clump must rely on more events

at its checkpoint location than the other clump is saving at its checkpoint location. So if clump a is saving fewer events than clump b needs, then for any partial recovery line where clump a is checkpointing at that checkpoint location, clump b cannot checkpoint at its current location or at any later location.

For instance, combining Figure 5.7 with the clump information from Tables 5.1 and 5.2, if clump b tries to checkpoint at location 2, while clump a is checkpointing at location 0, that is an invalid recovery line. The algorithm can use this information to also prune lines where clump a is checkpointing at location 0 and clump b is checkpointing at location 3. Our algorithm uses this technique, which leverages the strictly increasing nature of the vector clock values, to reduce the search space.

5.5.2.6 Optimizations

Although checking recovery line constraints is time consuming and storing generated recovery lines is memory intensive, our algorithm is fast and bounds memory usage. For speed, our algorithm performs only one validity check before generating a line; it compares the first checkpoint locations for the first clump in each of the source partial valid recovery lines—if they are compatible, then the line is generated. Our algorithm generates many recovery lines for each partition and then performs the complete validity checks. This method allows the algorithm to leverage the constraint information described above: once a recovery line has proven invalid, the algorithm can delete

all other recovery lines in the list that exhibit the invalidating characteristic. However, to bound memory usage, it does not necessarily generate all recovery lines from a partition simultaneously. Instead, the algorithm generates lines until the number generated reaches a threshold, then the algorithm prunes the invalid ones. This process is iterative and continues until all lines have been generated: once the algorithm prunes a set of lines, it generates more lines until the threshold is reached again, and then it initiates another pruning pass.

5.5.2.7 Branch-and-Bound Using the Wide and Flat Metric

As the algorithm merges partitions, we want it to assemble partial valid recovery lines that will result in complete valid recovery lines that are staggered. But the more a line is staggered, the more communication happens between each checkpoint location in the recovery line—and, thus, the more constraints are introduced. Hence, earlier in the algorithm, when the partitions are smaller, the lines that are less staggered have a better chance of forming valid recovery lines as they are merged to a complete line. However, to bound memory usage and execution time, the algorithm must limit the number of partial valid recovery lines it keeps for each partition. Given this tension between generating valid recovery lines and generating useful recovery lines, we must be careful which partial valid recovery lines are kept and which are discarded.

To help the algorithm choose lines, we introduce a metric to help us

evaluate the amount of staggering contained in each line. The *Wide and Flat* (WAF) metric statically estimates the usefulness of both partial and complete recovery lines by quantifying the interval of the checkpoint locations (width) and the number of processes that checkpoint at each (flatness). Currently, this metric is a simplified form of the circular string edit distance [88] from the generated line to a perfectly staggered one, and our goal is to find solutions with low WAF values, which indicate that fewer edits are needed to convert that solution to the perfectly staggered line, and thus that the solution is more staggered.

We use the *branch-and-bound* search strategy [29] in conjunction with our WAF metric to find useful recovery lines. In this strategy, heuristics are used to prune preliminary results based on the likelihood that the partial result will yield the desired solution—those recovery lines that are both valid and staggered. At each partition level, the algorithm determines which lines to prune by separating the recovery lines, or partial recovery lines, into *bins* based on their WAF values. During pruning, a minimum number of lines is saved from each of the bins; this number decreases as the algorithm nears completion. The details of this policy are discussed in the next section.

Until now, our techniques have primarily pruned large amounts of invalid recovery lines. The branch-and-bound search strategy may also eliminate arbitrary numbers of valid recovery lines as it attempts to identify lines that lead to low checkpoint overhead.

5.5.2.8 Pruning Policy

Our pruning policy is designed to reduce the memory usage and execution time of our algorithm. There are several parameters that affect its performance—here we discuss each. The settings we use for our results are displayed in Table 5.3. These settings are based on the memory available on the machines on which the compiler executes. Those settings are the same for all results we present in this chapter. Quantitative results and analysis for this parameter are included in Section 5.7.2.

As explained in Section 5.5.2.5, the algorithm generates some threshold number of recovery lines, the Line Check Threshold, before performing validity checks, eliminating invalid lines, and evaluating the remaining ones. This number is important since the higher it is, and thus the more lines that are checked for validity at once, the more potential exists for the algorithm to leverage information from lines that prove invalid. However, a higher threshold also means more lines must be stored, because when this threshold is higher it reduces the number of pruning interruptions. Reducing the number of interruptions decreases the algorithm execution time. When this threshold is lower, it reduces the algorithm's memory usage.

After the invalid lines are eliminated, if the number of remaining lines is above the Pruning Threshold, the algorithm begins deleting lines. To choose which lines to prune, the algorithm calculates each line's WAF value and places it into the bin representing that value. (Section 5.5.2.9 discusses these bins in detail.) All existing lines are evaluated and sorted before the algorithm

begins pruning. When pruning begins, it proceeds from the bin representing the highest WAF value to that representing the lowest.

The algorithm prunes lines until the total number of lines is some factor of the pruning threshold, called Lines to Keep. Reducing the lines below the pruning threshold ensures that when the algorithm resumes the number of lines does not immediately reach the pruning threshold again. The algorithm leaves a minimum number of lines in each bin (Minimum Lines in Each Bin) and deletes the rest; this minimum helps ensure that valid lines are found, since it preserves lines in the higher bins and those lines are more likely to form complete valid lines. This minimum is determined by a formula that weighs how close the algorithm is to complete, and it approaches zero as the algorithm nears completion.

5.5.2.9 Adaptive Bins

The bins in our algorithm are used in two places: 1) in recovery line development, where their function is to save a particular distribution of lines with a variety of WAF values during each pruning stage, and 2) during the final line choice, where their function is to identify which line(s) to place in the application. In both places, the lines are mapped to bins by the lines' WAF values. Since the WAF values are simplified circular string edit distances, their magnitude varies based on the application and the number of processes; it also increases as the algorithm progresses and there are more processes in each line. So we use an algorithm that adapts the mapping of the lines to the bins to

Variable	Value	Effect
Pruning Threshold	$\frac{32{,}768}{(sizeofcurrentlines)}$	**increase** to generate more recovery lines and reduce execution time; **decrease** to reduce memory
Line Check Threshold	$2048 * (pruningthreshold)$	**increase** to maximize leveraging and reduce execution time; **decrease** to reduce memory
Lines to Keep	$\frac{(pruningthreshold)}{5}$	**increase** to keep more lines between pruning phases; **decrease** to reduce number of times pruning is initiated and thus time
Target Number of Bins	100	**increase** to keep more lines with higher WAF and have more differentiation between WAF values; **decrease** to keep more lines with lower WAF and collapse WAF values into larger ranges
Minimum Lines in Each Bin	$\frac{Closeness*(linestokeep)}{(numberofbins)}$ where $Closeness < 1$	**increase** to keep more lines with less staggering; **decrease** to keep more staggered lines

Table 5.3: Parameters of the Pruning Policy

the changing WAF values.

We map WAF values to bins as follows. We set a target number of

bins (called Target Number of Bins). Then, we divide the WAF value of the first line to be evaluated during that pruning pass by ten until its magnitude is less than the magnitude of the target number of bins. Every other line in that pruning pass has the magnitude of its WAF value reduced by the same amount, even if that means we exceed our target number of bins. Setting the magnitude reduction this way ensures an even comparison between the lines.

The number of bins affects how the lines are pruned: more bins results in a larger number of lines left in bins representing higher WAF values, since more bins represent those values and a minimum number of lines are left in each bin. Our algorithm currently sets the target number of bins to 100.

5.5.2.10 Correctness Considerations

Some programs delegate communication to wrapper functions, so the context of a function call determines which processes execute the communication. Thus, to ensure correctness, our algorithm requires the context-sensitive placement of recovery lines. To place context-sensitive recovery lines, the compiler inserts a single integer variable in the application text. This variable is modified to produce a unique value for each context during execution [225]. Process checkpoints are guarded by a condition checking for the appropriate context.

Additionally, a recovery line must either reside entirely inside a loop body or entirely outside of a loop body.

5.6 Implementation

This algorithm has been implemented using the `Broadway` [91] source-to-source ANSI C compiler. `Broadway` performs context-sensitive interprocedural pointer analysis and thus provides a powerful tool for our analysis.

Our implementation makes more assumptions than our algorithm. Our implementation assumes the application is written in C and uses the MPI communication library. It also assumes that any variables analyzed during the symbolic expression or control-dependence analyses are regular variables or single-dimension arrays. The variables that are analyzed must not be not pointers, multi-dimensional arrays, or loop induction variables. Loops that must be evaluated by our analysis must have a fixed trip count.

This implementation assumes that the number of processes the application uses is statically known and that all communication is *deterministic*. Deterministic communication is communication that depends only on input values or the process's rank: it disallows the use of `MPI_ANY_SOURCE`. The algorithm allows for the relaxing of this condition.

5.6.1 Benchmarks

For our results, we are using application benchmarks to illustrate the potential for staggered checkpointing. `BT`, `SP`, and `LU` are FORTRAN codes from the NAS Parallel Benchmark Suite [51] that we convert to C using `for_c` [4]. `Ek-simple` is a well-known computational fluid dynamics benchmark.

These example applications are simplified—for example, command-line arguments are set to be constants and function pointers are replaced with static function calls. These simplifications do not affect the overall behavior of the benchmark.

5.7 Evaluation

To evaluate our algorithm, we consider many facets of our solution. We begin by presenting an evaluation of our algorithm, including its scalability, ability to find useful recovery lines, and sensitivity to its parameters. An evaluation of the the *Wide-and-Flat* metric is also included. We finish by presenting a performance evaluation of recovery lines identified by our algorithm.

5.7.1 Algorithm

In this section, we report the results for our algorithm for our four application benchmarks, each using several process sizes. We report the effects of each optimization technique, which in addition to reducing algorithm complexity, significantly reduce the search space. The execution times for the results presented here are on the order of minutes for 1,024 and 4,096 processes, hours for 16,384 processes, and days for 65,536 processes. Since our algorithm is context-sensitive, the recovery lines reported here are dynamically unique but may not be statically unique.

Generating useful recovery lines for large parallel applications is challenging as the execution complexity of the naive algorithm shows: its com-

plexity is $O(L^P)$, where P is the number of processes and L is the number of checkpoint locations. We design a new algorithm that uses a number of techniques to reduce this complexity to: $O(log(C) * L^{C-2})$, where L represents the number of merged checkpoint locations in a phase, and C represents the number of clumps in a clump set and C is necessarily less than or equal to P. This complexity shows that our algorithm's execution time is not directly related to the number of processes or the problem sizes, and is thus scalable. Our results show that algorithm successfully places recovery lines in all four benchmarks when they are configured to use 16,384 processes, and in most of them when they are configured to use 65,536 processes.

Our algorithm is able to identify useful recovery lines in such large applications because it intelligently reduces the very large search space, which begins as L^P. The large numbers in Table 5.4 define the original valid recovery line search space, as this table reports the number of checkpoint locations and potential recovery lines existing in the benchmarks without any of our reduction techniques. In addition, this table shows the problem size used for each benchmark. In the reporting of these results, the problem size only effects the number of checkpoint locations that are merged and the WAF execution times. The problem sizes reflect approximately fifteen minutes of local execution time for each chosen phase.

As phases are introduced, the search space begins to narrow. In Table 5.5, we report the number of phases discovered for each benchmark. The rest of our results are presented for a single phase: the phase chosen is ei-

ther representative of the phases for that benchmark or is the phase with the majority of the communication.

Our algorithm reduces L, the number of considered checkpoint locations and the base of our search space calculation, by merging checkpoint locations that are too closely spaced for our goals. These results are shown in Table 5.6. For the problem sizes shown in these tables, most checkpoint locations are separated by a sufficient number of local operations to remain individual locations. However, `Ek-simple`'s results show several merged locations.

While reducing L does reduce the search space, reducing P, the number of considered processes and the exponent of our search space calculation, helps much more. In Table 5.7, we report on the results of dividing the processes into clumps and subsequently treating each clump as a process in the recovery line algorithm. Clump sizes are relatively even within a configuration, though there is often one clump containing all processes (usually the result of a collective communication call). Clumps reduce the search space dramatically; the space for `LU` with 1,024 processes is reduced by over 800 orders of magnitude. Also, the number of clumps does not increase linearly with the number of processes but at a slower rate, indicating that the clump concept is scalable in practice. Once the clumps are combined into clump sets, as we see in Table 5.8, the search space is reduced even further. The number of clump sets is factored out of the exponent of the search space and has instead become a multiplier: the number of possible recovery lines for `LU` with 1,024 processes has now been

reduced by another 25 orders of magnitude.

Although we have significantly reduced the search space (a summary of the reductions can be seen in Table 5.9), it is still very large; `LU` with 1,024 processes still has a possible $1.6 * 10^{25}$ recovery lines. In this space, however, our synthesis algorithm can find useful recovery lines. By leveraging constraint information, first by not propagating invalid constraints in the synthesis algorithm, then by reducing the number of checks required to determine if a recovery line is valid (usually an average of $P^2/2$), and finally by pruning recovery lines less likely to reduce contention, we convert an originally intractable problem into manageable one. Although our branch-and-bound tactic prunes valid and invalid lines, we still find valid recovery lines with low contention. Table 5.10 reports the results for each reduction technique employed by our synthesis algorithm. The first column represents those lines eliminated through our initial validity check performed between just two clumps before the line is generated. The next three columns respectively report the number of recovery lines generated, the number eliminated as invalid after being fully investigated for validity, and those eliminated as invalid without checking by leveraging the strictly increasing nature of the recovery line constraints. Last, the table shows the number of lines pruned by our branch and bound strategy and the number of valid recovery lines produced by our algorithm. Notice that for `Ek-simple` the number of lines is reduced by exactly half as the number of processes increases; this phenomena is a result of our pruning policy which adjusts for the amount of memory used to store each line.

Benchmark	Processes	Problem Size	Checkpoint Locations	Possible Recovery Lines
BT	1,024	6,528	38	38^{1024}
	4,096	10,771	38	38^{4096}
	16,384	17,772	38	38^{16384}
	65,536	29,324	38	38^{65536}
Ek-simple	1,024	2,125,200	$1024P : 6L$, $992P : 22L$, $961P : 5L$, $31P : 12L$	$6^{1024} + 22^{992} + 5^{961} + 12^{31}$
	4,096	4,250,400	$4096P : 6L$, $4032P : 22L$, $3969P : 5L$, $63P : 12L$	$6^{4096} + 22^{4032} + 5^{3969} + 12^{63}$
	16,384	8,500,800	$16384P : 6L$, $16256P : 22L$, $16129P : 5L$, $127P : 12L$	$6^{16384} + 22^{16256} + 5^{16129} + 12^{127}$
	65,536	17,001,600	$65536P : 6L$, $65280P : 22L$, $65025P : 5L$, $255P : 12L$	$6^{65536} + 22^{65280} + 5^{65025} + 12^{255}$
LU	1,024	6,528	$1024P : 16L$, $992P : 18L$, $31P : 4L$	$16^{1024} + 18^{992} + 4^{31}$
	4,096	10,771	$4096P : 16L$, $4032P : 18L$, $63P : 4L$	$16^{4096} + 18^{4032} + 4^{63}$
	16,384	17,772	$16384P : 16L$, $16256P : 18L$, $127P : 4L$	$16^{16384} + 18^{16256} + 4^{127}$
	65,536	29,324	$65536P : 16L$, $65280P : 18L$, $255P : 4L$	$16^{65536} + 18^{65280} + 4^{255}$
SP	1,024	5,480	37	37^{1024}
	4,096	9,042	37	37^{4096}
	16,384	14,919	37	37^{16384}
	65,536	24,616	37	37^{65536}

Table 5.4: Initial search space. The initial number of checkpoint locations and possible recovery lines without any of our reduction techniques, including phases. P represents the number of processes in the system, L represents the number of checkpoint locations.

Benchmark	Processes	Number of Phases	Checkpoint Locations	Possible Recovery Lines
BT	1,024	11	$14L$	14^{1024}
	4,096	11	$14L$	14^{4096}
	16,384	11	$14L$	14^{16384}
	65,536	11	$14L$	14^{65536}
Ek-simple	1,024	7	$992P : 20L$, $961P : 4L$, $31P : 8L$	$20^{992} + 4^{961} + 8^{31}$
	4,096	7	$4032P : 20L$, $3969P : 4L$, $63P : 8L$	$20^{4032} + 4^{3969} + 8^{63}$
	16,384	7	$16256P : 20L$, $16129P : 4L$, $127 : 8L$	$20^{16256} + 4^{16129} + 8^{127}$
	65,536	7	$65280P : 20L$, $65025P : 4L$, $255P : 8L$	$20^{65280} + 4^{65025} + 8^{255}$
LU	1,024	17	$1024P : 1L$, $992P : 8L$	$1^{1024} + 8^{992}$
	4,096	17	$4096P : 1L$, $4032P : 8L$	$1^{4096} + 8^{4032}$
	16,384	17	$16384P : 1L$, $16256P : 8L$	$1^{16384} + 8^{16256}$
	65,536	17	$65536P : 1L$, $65280P : 8L$	$1^{65536} + 8^{65280}$
SP	1,024	9	$14L$	14^{1024}
	4,096	9	$14L$	14^{4096}
	16,384	9	$14L$	14^{16384}
	65,536	9	$14L$	14^{65536}

Table 5.5: Phase Results. Each benchmark consists of several phases; we chose one phase from each benchmark to report. The $P : L$ syntax shown in this table describes the number of processes that execute a particular number of checkpoint locations. Where P is absent, all processes execute that location.

Benchmark	Processes	Checkpoint Locations	Possible Recovery Lines
BT	1,024	$14L$	14^{1024}
	4,096	$14L$	14^{4096}
	16,384	$14L$	14^{16384}
	65,536	$14L$	14^{65536}
Ek-simple	1,024	$992P:13L$, $961P:3L$, $31P:6L$	$13^{992}+3^{961}+6^{31}$
	4,096	$4032P:13L$, $3969P:3L$, $63P:6L$	$13^{4032}+3^{3969}+6^{63}$
	16,384	$16256P:13L$, $16129P:3L$, $127P:6L$	$13^{16256}+3^{16129}+6^{127}$
	65,536	$65280P:13L$, $65025:3L$, $255P:6L$	$13^{65280}+3^{65025}+6^{255}$
LU	1,024	$1024P:1L$, $992P:8L$	$1^{1024}+8^{992}$
	4,096	$4096P:1L$, $4032P:8L$	$1^{4096}+8^{4032}$
	16,384	$16384P:1L$, $16256P:8L$	$1^{16384}+8^{16256}$
	65,536	$65536P:1L$, $65280P:8L$	$1^{655536}+8^{65280}$
SP	1,024	$14L$	14^{1024}
	4,096	$14L$	14^{4096}
	16,384	$14L$	14^{16384}
	65,536	$14L$	14^{65536}

Table 5.6: Checkpoint Reduction Numbers. These numbers represent the checkpoint locations and search space that remain after our algorithm has merged any closely-spaced checkpoint locations.

Benchmark	Processes	Clumps	Clump Size	Checkpoint Locations	Possible Recovery Lines
BT	1,024	97	$96C:32P$, $1C:1024P$	$96C:5L$, $1C:2L$	$5^{96}+2^1$
	4,096	193	$192C:64P$, $1C:4096P$	$192C:5L$, $1C:2L$	$5^{192}+2^1$
	16,384	385	$384C:128P$, $1C:16384P$	$384C:5L$, $1C:2L$	$5^{384}+2^1$
	65,536	769	$768C:256P$, $1C:65536P$	$768C:5L$, $1C:2L$	$5^{768}+2^1$
Ek-simple	1,024	125	$65C:32P$, $2C:31P$, $2C:30P$, $2C:29P$, $2C:28P$	$1C:12L$, $2C:11L$, $31C:9L$, $30C:5L$, $61C:3L$	$12^1+11^2+9^{31}+5^{30}+3^{61}$
	4,096	253	$129C:64P$, $2C:63P$, $2C:62P$, $2C:61P$, $2C:60P$	$1C:12L$, $2C:11L$, $63C:10L$, $62C:5L$, $125C:3L$	$12^1+11^2+10^{63}+5^{62}+3^{125}$
	16,384	509	$257C:128P$, $2C:127P$, $2C:126P$, $2C:125P$, $2C:124P$	$1C:12L$, $2C:11L$, $127C:9L$, $126C:5L$, $252C:3L$	$12^1+12^2+9^{127}+5^{126}+3^{252}$
	65,536	1,021	$513C:256P$, $2C:255P$, $2C:254P$, $2C:253P$, $2C:252P$	$1C:12L$, $2C:11L$, $255C:9L$, $254C:5L$, $509C:3L$	$12^1+11^2+9^{255}+5^{254}+3^{509}$
LU	1,024	64	$64C:32P$	$64C:6L$	6^{64}
	4,096	128	$128C:64P$	$128C:6L$	6^{128}
	16,384	256	$256C:128P$	$256C:6L$	6^{256}
	65,536	512	$512C:256P$	$512C:6L$	6^{512}
SP	1,024	97	$96C:32P$, $1C:1024P$	$96C:5L$, $1C:1L$	$5^{96}+1^1$
	4,096	193	$192C:64P$, $1C:4096P$	$192C:5L$, $1C:2L$	$5^{192}+2^1$
	16,384	385	$384C:128P$, $1C:16384P$	$384C:5L$, $1C:2L$	$5^{384}+2^1$
	65,536	769	$768C:256P$, $1C:65536P$	$768C:5L$, $1C:2L$	$5^{768}+2^1$

Table 5.7: Clumps. This table reports the number of clumps for each benchmark, the size of those clumps, the number of checkpoint locations executed by each, and the resulting size of the search space. C represents clumps; the $C:P$ and $C:L$ notations shown in this table denote the number of clumps containing a particular number of processes and the number of clumps that execute a particular number of checkpoint locations, respectively. For brevity, we only report five most significant results in each column for `Ek-simple`.

Benchmark	Processes	Clump Sets	Set Size	Possible Recovery Lines
BT	1,024	4	$3S:32C,\ 1S:1C$	$3(5^{32})+2^1$
	4,096	4	$3S:64C,\ 1S:1C$	$3(5^{64})+2^1$
	16,384	4	$3S:128C,\ 1S:1C$	$3(5^{128})+2^1$
	65,536	4	$3S:256C,\ 1S:1C$	$3(5^{256})+2^1$
Ek-simple	1,024	2	$2S:32C$	$12^1+11^2+10^1+9^{30}+6^{30}$
	4,096	2	$2S:64C$	$12^1+11^2+11^1+10^{62}+6^{62}$
	16,384	2	$2S:128C$	$12^1+11^2+10^1+9^{126}+6^{126}$
	65,536	2	$2S:256C$	$12^1+11^2+10^1+9^{254}+6^{254}$
LU	1,024	2	$2S:32C$	$2(6^{32})$
	4,096	2	$2S:64C$	$2(6^{64})$
	16,384	2	$2S:128C$	$2(6^{128})$
	65,536	2	$2S:256C$	$2(6^{256})$
SP	1,024	4	$3S:32C,\ 1S:1C$	$3(5^{32})+1^1$
	4,096	4	$3S:64C,\ 1S:1C$	$3(5^{64})+2^1$
	16,384	4	$3S,128C,\ 1S:1C$	$3(5^{128})+2^1$
	65,536	4	$3S:256C,\ 1S:1C$	$3(5^{256})+2^1$

Table 5.8: Clump sets. S represents the number of sets. The $S:C$ syntax shown in this table denotes the number of sets containing a particular number of clumps.

Benchmark	Processes	Possible Recovery Lines				
		Initial	Phases	Checkpoint Reduction	Clumps	Clump Sets
BT	1,024	38^{1024}	14^{1024}	14^{1024}	$5^{96} + 2^1$	$3(5^{32}) + 2^1$
	4,096	38^{4096}	14^{4096}	14^{4096}	$5^{192} + 2^1$	$3(5^{64}) + 2^1$
	16,384	38^{16384}	14^{16384}	14^{16384}	$5^{384} + 2^1$	$3(5^{128}) + 2^1$
	65,536	38^{65536}	14^{65536}	14^{65536}	$5^{768} + 2^1$	$3(5^{256}) + 2^1$
Ek-simple	1,024	$6^{1024} + 22^{992} + 5^{961} + 12^{31}$	$20^{992} + 4^{961} + 8^{31}$	$13^{992} + 3^{961} + 6^{31}$	$12^1 + 11^2 + 9^{31} + 5^{30} + 3^{61}$	$12^1 + 11^2 + 10^1 + 9^{30} + 6^{30}$
	4,096	$6^{4096} + 22^{4032} + 5^{3969} + 12^{63}$	$20^{4032} + 4^{3969} + 8^{63}$	$13^{4032} + 3^{3969} + 6^{63}$	$12^1 + 11^2 + 10^{63} + 5^{62} + 3^{125}$	$12^1 + 11^2 + 11^1 + 10^{62} + 6^{62}$
	16,384	$6^{16384} + 22^{16256} + 5^{16129} + 12^{127}$	$20^{16256} + 4^{16129} + 8^{127}$	$13^{16256} + 3^{16129} + 6^{127}$	$12^1 + 11^2 + 9^{127} + 5^{126} + 3^{252}$	$12^1 + 11^2 + 10^1 + 9^{126} + 6^{126}$
	65,536	$6^{65536} + 22^{65280} + 5^{65025} + 12^{255}$	$20^{65280} + 4^{16129} + 8^{255}$	$13^{65280} + 3^{65025} + 6^{255}$	$12^1 + 11^2 + 9^{255} + 5^{254} + 3^{509}$	$12^1 + 11^2 + 10^1 + 9^{254} + 6^{254}$
LU	1,024	$16^{1024} + 18^{992} + 4^{31}$	$1^{1024} + 8^{992}$	$1^{1024} + 8^{992}$	6^{64}	$2(6^{32})$
	4,096	$16^{4096} + 18^{4032} + 4^{63}$	$1^{4096} + 8^{4032}$	$1^{4096} + 8^{4032}$	6^{128}	$2(6^{64})$
	16,384	$16^{16384} + 18^{16256} + 4^{127}$	$1^{16384} + 8^{16256}$	$1^{16384} + 8^{16256}$	6^{256}	$2(6^{128})$
	65,536	$16^{65536} + 18^{65280} + 4^{255}$	$1^{65536} + 8^{65280}$	$1^{65536} + 8^{65280}$	6^{512}	$2(6^{256})$
SP	1,024	37^{1024}	14^{1024}	14^{1024}	$5^{96} + 1^1$	$3(5^{32}) + 1^1$
	4,096	37^{4096}	14^{4096}	14^{4096}	$5^{192} + 2^1$	$3(5^{64}) + 2^1$
	16,384	37^{16384}	14^{16384}	14^{16384}	$5^{384} + 2^1$	$3(5^{128}) + 2^1$
	65,536	37^{65536}	14^{65536}	14^{65536}	$5^{768} + 2^1$	$3(5^{256}) + 2^1$

Table 5.9: Summary of search space reduction by each technique in our algorithm for the benchmarks configured to use five different process sizes. The second column displays the initial size of the search space. The next column reports the reduction realized when phases are introduced; this column reports the search space for a representative phase from each benchmark. The last three columns report the size of the search space after the checkpoint locations are merged, clumps are introduced, and clump sets are formed.

Benchmark	Processes	Recovery Lines					
		Skipped Initially	Generated	Investigated	Leveraged	Pruned by WAF	Valid Found
BT	1,024	18,606	4,603,597	4,603,597	0	4,588,183	620
	4,096	50,262	9,325,092	9,325,092	0	9,293,840	432
	16,384	77,206	18,731,170	18,731,170	0	18,668,236	442
	65,536	156,414	37,759,736	37,759,736	0	37,632,812	408
Ek-simple	1,024	401,967	2,465,869	924,627	1,541,242	193,270	408
	4,096	1,187,522	8,243,283	2,850,232	5,393,051	2,819,754	204
	16,384	3,598,557	10,667,118	3,620,224	7,046,894	3,571,133	102
	65,536	3,889,674	22,989,972	8,860,009	14,129,963	8,763,342	50
LU	1,024	1,056	3,233,318	2,615,253	618,065	2,603,302	409
	4,096	2,046	6,603,040	5,604,553	998,487	5,580,103	203
	16,384	128,655	13,129,553	10,591,018	2,538,535	10,542,142	103
	65,536	8,382	26,464,194	19,406,505	7,057,689	19,308,542	191
SP	1,024	22,573	4,600,329	4,600,329	0	4,584,915	614
	4,096	45,352	9,331,318	9,331,318	0	9 299,935	538
	16,384	91,860	18,927,310	18,927,310	0	18,863,945	188
	65,536	179,544	39,437,768	39,437,768	0	39,308,070	169

Table 5.10: Actual search space reduction performed by our algorithm for the our benchmarks configured to use five different process sizes. The third column, "Skipped Initially", indicates the number of partial and complete recovery lines that are eliminated as invalid through a single validity check before the lines are generated. The fourth column, "Generated", reports the number of lines generated, and then the fifth column, "Investigated", displays the number of lines that are checked for validity after generation. The fifth column, "Leveraged", indicates the number of lines that are eliminated after generation but without validity checks: recovery lines constraints are strictly increasing, and our algorithm leverages that information. The last two columns, "Pruned by WAF" and "Valid Found", report the number of lines eliminated by our branch-and-bound methodology and the number of lines produced by the algorithm, respectively.

WAF value	Checkpoint Time 4 MB	32 MB
418	698s	4,026s
3,958	249s	4,689s
7,008	1,080s	5,561s
7,361	1,141s	5,613s

Table 5.11: The simulated checkpoint times on Ranger for recovery lines in the `BT` benchmark with 4,096 processes and local execution of approximately three minutes.

5.7.1.1 Wide-and-Flat Metric

Our WAF metric separates the useful recovery lines from the other generated lines. We first evaluate the WAF metric by placing recovery lines in our synthetic benchmark and estimating their performance in our simulator. When we consider total execution time, the WAF metric ranks 83% of the fastest third of the lines into the smallest third of the bins. While the metric, which is designed to reduce checkpointing time, does not perfectly predict the relative execution times of the lines, it still strongly correlates with execution time. The WAF metric is not as effective when the application does not execute enough instructions to sufficiently separate the number of processes writing an amount of data. We examine this circumstance in Chapter 4.

To test the usefulness of our algorithm, we compare lines with a few WAF values for several configurations of each benchmark. The WAF metric works best for larger configurations. Even for smaller configurations, the WAF metric always identifies a line faster than the line in which every process checkpoints at approximately the same time. However, the line from the lowest bin may not be the fastest. In smaller configurations, less separation of checkpoint

locations can still reduce contention for the file system sufficiently such that the file system does not saturate. We see this phenomenon in Table 5.11, which reports the simulated checkpointing times on Ranger for recovery lines with 4,096 processes in a phase of the `BT` benchmark with a local execution time of approximately three minutes. In this table, we see that the second line is faster than the first for 4 MB of data. However, when the size of the checkpoint data is increased to 32 MB, the first line is faster than the second.

To better handle smaller configurations, the WAF metric could be modified to account for both the amount of live data and the maximum throughput of the file system. However, this change would result in a system-dependent metric and assumes that the file system is not in use by any other application also executing on the system.

Since the WAF metric varies each pruning pass, WAF values cannot be compared across lines from different benchmarks, phases, or pruning passes.

5.7.1.2 Foundation Algorithm Parameters

In addition to testing the overall algorithm, we also tested the behavior of the foundation algorithm as the parameters that affect the creation of the partitions vary. Recall that we have identified two parameters: one, j, controls the number of partitions merged to form a new partition, and the other, k, controls the number of processes included in the initial partition. Table 5.12 displays how our algorithm performs with values of j from 2 to 8 while k remains constant. The reported results are for a single clump set in the `SP`

j	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time
2	1,516	8.45%	24.11%	32.81%	49.81%	1h20m
4	3,001	2.75%	15.76%	24.89%	43.47%	2h30m
8	3,133	2.30%	12.41%	22.39%	41.75%	6h38m

Table 5.12: The results of our algorithm when run with varied values of j, the parameter the determines how many partitions are merged to form a new partition at each step. In this table, the number of processes in the initial partition, k, remains constant at 4. The first column reports the value of j, and the second column reports the number of recovery lines found by the algorithm for that configuration. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "10%" displays the percentage of lines found in the lowest 10% of the bins. The final column, "Compile Time", reports the execution time for our compiler. All numbers are for a single clump set of the SP benchmark with 4,096 processes and a 1,632 problem size.

benchmark, but these results are representative. The results show that the algorithm finds more complete recovery lines for higher values of j. However, the algorithm also then requires more time. When j is smaller, the algorithm finds fewer complete recovery lines but the number of useful recovery lines is larger. In addition, the lines that are useful are more staggered. Thus, the algorithm performs more effectively when j is smaller.

In Table 5.13, we report results gathered with varied values of k and $j = 2$. Again, the results are reported for a single clump set, but are indicative of the overall results. The results show that while all configurations find a similar number of useful recovery lines, the algorithm takes significantly longer as k increases. We conclude that the algorithm performs more effectively when k is smaller.

k	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time
2	1,568	9.05%	24.53%	33.25%	49.93%	53m
4	1,516	8.45%	24.11%	32.81%	49.81%	1h20m
8	2,244	9.01%	24.42%	33.14%	50.29%	3h55m
16	2,998	4.96%	20.32%	29.46%	47.47%	14h37m

Table 5.13: The results of our algorithm when run with varied values of k, the parameter that determines how many processes comprise the initial partitions. In this table, the number of partitions merged to form a new partition at each step, j, remains constant at 2. The first column reports the value of k, and the second column reports the number of recovery lines found by the algorithm for that configuration. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "10%" displays the percentage of lines found in the lowest 10% of the bins. The final column, "Compile Time", reports the execution time for our compiler. All numbers are for a single clump set of the SP benchmark with 4,096 processes and a 1,632 problem size.

5.7.2 Pruning Policy Parameters

Here we present a quantitative analysis of each of the parameters of our pruning policy. Our results show that most of the parameters affect memory usage and compiler execution time without greatly affecting the WAF distribution. However, we see the parameter that affects the number of lines to survive a pruning pass (Lines to Keep) also affects the WAF distribution due to its interaction with the minimum number of lines that are saved in each bin (Minimum Lines in Each Bin). Since these parameters interact, a change to one can be offset by a change in the other. This interaction also applies to the Target Number of Bins and Minimum Lines in Each Bin parameters. If the algorithm is unable to identify useful recovery lines and a constant memory usage and execution time are desired, the Minimum Lines in Each Bin

Pruning Threshold	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time	Memory Usage
16,384	162	62.11%	73.29%	76.40%	81.99%	10m34s	1.0 GB
32,768	321	62.50%	73.44%	76.56%	82.19%	14m38s	1.1 GB
65,336	614	70.80%	79.28%	81.73%	86.13%	35m6s	1.2 GB

Table 5.14: The results of our algorithm for SP when run with varied values of the pruning threshold. In this table, the first column reports the value of the pruning threshold. As the pruning threshold increases, more lines are stored before the next pruning pass. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes.

parameter should be increased.

Recall that our initial settings are based on the amount of memory available on the machines we use to execute the compiler. We explore the space on either side of our initial parameters. In all tables, the initial parameter is indicated in bold.

We conduct these experiments using the SP benchmark with 4,096 processes and a problem size of 1,632, and the LU benchmark with 4,096 processes and a problem size of 6,528 and 256 processes with a problem size of 816. In each experiment, we vary only the parameter being discussed and the other parameters are set to their initial values. We report the results that best illustrate the effects of the parameters.

Pruning Threshold. The pruning threshold parameter controls how many valid recovery lines are stored before the algorithm performs a pruning pass. As the parameter increases, more lines are stored before the pass begins, resulting in more memory usage but reducing the number of pruning passes that must be performed and thus execution time. In Table 5.14, we see that as the parameter is increased, the compiler execution time and memory usage increase, as expected. The compiler execution time increases by 142% when our initial setting is doubled, and the memory usage increases by 9%. In addition, we see that as the parameter increases more recovery lines are identified and they are more useful, as more lines are found in the lower bins. The number of lines increases because the pruning threshold is higher, which also increases the number of lines that survive each pass. Thus, the algorithm can keep more lines. The distribution is better since our pruning policy keeps a minimum number of lines in each bin that is a percentage of the lines to survive each pass, and more lines implies that more lines may be kept in lower bins.

Line Check Threshold. The line check threshold parameter controls how many lines are generated before the algorithm checks them for validity. Its value is a multiple of the pruning threshold. For example, in Table 5.15 we see that the initial value for this threshold is 2,048, indicating that the algorithm will generate 2,048 times more lines than the pruning threshold before it checks them for validity. This initial value is high—the machines on which the compiler executes have lots of memory, and the higher the line check thresh-

Number of Processes	Line Check Threshold	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time	Memory Usage
	1	819	38.07%	39.90%	39.90%	42.35%	2m22s	270 MB
256	4	819	38.07%	39.90%	39.90%	42.35%	2m20s	271 MB
	2,048	819	38.07%	39.90%	39.90%	42.35%	2m53s	310 MB
	1	278	5.42%	44.40%	49.56%	59.57%	20m37s	3.0 GB
4,096	4	281	7.17%	45.88%	50.90%	60.93%	19m55s	3.0 GB
	2,048	280	7.17%	44.09%	50.54%	61.29%	13m50s	3.0 GB

Table 5.15: The results of our algorithm on the `LU` benchmark when run with varied values of the line check threshold. In this table, the first column reports the number of processes for that configuration, and the second column reports the value of the line check threshold. As the line check threshold increases, more lines are available to be leveraged by the algorithm. The third column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes.

old, the more lines are available to be leveraged. As is shown in the 4,096 process case in the table, as the threshold increases the execution time of the algorithm decreases. However, as we see in the 256 process case, the memory usage increases significantly as this parameter increases. The 4,096 process case uses too much memory to see this effect; the 256 process case executes too quickly to see the execution time change. The WAF distribution remains essentially the same, as this parameter only controls the pruning of invalid recovery lines. The distribution varies a little since the pruning threshold is only checked after the lines have been checked for validity.

Lines To Keep	Number of Lines	WAF percentages 10%	25%	33%	50%	Algorithm Time	Memory Usage
10	141	7.14%	43.57%	50.00%	60.71%	12m35s	2.9 GB
5	280	7.17%	44.09%	50.54%	61.29%	13m50s	3.0 GB
1	1,042	5.76%	55.43%	60.61%	69.26%	1h12m	3.0 GB

Table 5.16: The results of our algorithm for `LU` with 4,096 processes when run with varied values of Lines to Keep. In this table, the first column reports the value of the Lines to Keep parameter. This parameter is a divisor and the number of lines kept is equal to the pruning threshold divided by it. As parameter decreases, more lines are stored between pruning passes. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Algorithm Time" and "Memory Usage", report the execution time for the recovery line algorithm and the amount of memory it consumes.

Lines To Keep. The Lines to Keep parameter controls the number of lines that survive a pruning pass. It is a factor of the pruning threshold and is controlled by the divisor displayed in Table 5.16. Since this number is a divisor, as it decreases more lines survive the pruning pass and as it increases fewer lines survive. The more lines that are kept between pruning passes result in the pruning passes being triggered more often, and thus increase execution time, as we see in the table. Execution time is drastically increased if the number of lines kept after each pass is equivalent to the pruning threshold. However, the more lines that are kept each time, the better the WAF distribution should look since more lines will be kept in lower bins. In our example, this is not true for the lower 10% of the bins, but it can be clearly seen in the other three divisions. The number of lines found also increases as the divisor decreases

Lines To Keep	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time	Memory Usage
10	247	8.94%	22.36%	63.82%	63.82%	13m53s	3.0 GB
100	280	7.17%	44.09%	50.54%	61.29%	13m50s	3.0 GB
1,000	232	6.06%	22.94%	31.60%	52.38%	14m	3.0 GB

Table 5.17: The results of our algorithm for `LU` with 4,096 processes when run with varied values of Target Number of Bins. In this table, the first column reports the target number of bins. This parameter is a divisor and the number of lines kept is equal to the pruning threshold divided by it. As this parameter decreases, more lines are stored between pruning passes. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes.

since more lines can survive a pruning pass.

Target Number of Bins. The target number of bins gives our algorithm a finer granularity for separating the WAF values of the lines. A larger number of bins will result in more lines with higher WAF values surviving, as more bins mean more lines are kept under the Minimum Lines in Each Bin parameter. As we see in Table 5.17, the target number does not affect compiler execution time or memory usage. Our initial value works well in our experiments, and we expect there is little reason to vary this parameter.

Minimum Lines in Each Bin. Minimum Lines in Each Bin controls the number of lines that must be left in each higher bin. In our settings, this value is determined by how close the algorithm is to completion, such that

Minimum Lines in Each Bin	Number of Lines	WAF percentages 10%	25%	33%	50%	Compile Time	Memory Usage
Half	205	11.27%	61.27%	65.69%	73.04%	14m17s	3.0 GB
Initial	280	7.17%	44.09%	50.54%	61.29%	13m50s	3.0 GB
Double	368	8.17%	36.51%	43.87%	56.13%	14m16s	3.0 GB

Table 5.18: The results of our algorithm for `LU` with 4,096 processes when executed with varied values of Minimum Lines in Each Bin. In this table, the first column reports the value for Minimum Lines in Each Bin as it relates to our initial parameter. As this parameter increases, a higher percentage of the lines to survive a pruning pass are kept in the lower bins. The second column shows the number of recovery lines found by the algorithm. The "WAF percentages" columns report the percentage of found lines that are in a particular percentage of the lowest WAF bins. So, for instance, the column labelled "25%" displays the percentage of lines found in the lowest 25% of the bins. The final columns, "Compile Time" and "Memory Usage", report the execution time for our compiler and the amount of memory it consumes.

the percentage of lines stored in higher bins is higher at the beginning of the algorithm and lower at the end. We tested this parameter by adding a coefficient that either halved or doubled the percentage. Increasing this parameter increases the percentage of the lines to survive a pruning pass that will be in higher bins, and this parameter should be increased to increase the algorithm's ability to find valid lines and decreased to increase the algorithm's ability to find staggered lines. Since this value is a percentage of Lines to Keep, changing this value does not effect the algorithm execution time or the memory usage, as we see in Table 5.18. However, this value greatly affects the WAF distribution.

Minimum Number of Instructions that Must Separate Checkpoint Locations. Another parameter to our algorithm is the minimum number of

instructions that must separate two checkpoint locations for those locations to be considered independently in the algorithm. Otherwise, the locations are merged. Although this parameter is not really a parameter to our pruning policy but is instead part of our search space reduction, we include it here since it affects the algorithm's ability to find useful recovery lines. In the experiments presented here, the value is two million instructions. Given the average number of cycles per instruction in our benchmarks, this value translates to approximately .001 seconds of real time, which is a small amount of time in which to separate checkpoints. As this parameter is increased it affects the algorithm's ability to stagger. As the parameter is decreased, it can affect the WAF metric's ability to determine if the line is staggered. For our benchmarks, however, we did not see that the algorithm was particularly sensitive to its exact value. The separation of checkpoint locations typically varies by orders of magnitude.

5.7.3 Staggered Checkpointing

We evaluate lines identified by our algorithm using both simulated and measured performance. Our numbers are for two supercomputers located at the Texas Advanced Computing Center [2], Ranger [205] and Lonestar [204]. Lonestar is older, as it was installed in 2004 and is currently number 123 in the Top 500 list [8]. It consists of local hardware with 2.66 GHz processors, a Myrinet network for communication, and a Lustre file system featuring 4.6 GB/s throughput. Ranger is the newer, more modern supercomputer. It was

installed in 2008 and is currently ranked 11 in the Top 500 list. It consists of local hardware with four 2.3 GHz processes with four cores each, an Infiniband network for communication, and a Lustre file system featuring 40 GB/s throughput.

We test lines identified by our algorithm using our four application benchmarks. These benchmarks are indicative of the larger applications in use today and evaluating the performance of staggered checkpointing in the face of their communication is useful. These benchmarks typically have short execution times, so we perform experiments with larger than typical problem sizes. These problem sizes result in live data sizes that are very large, so we evaluate staggered checkpointing based on checkpoint sizes that our simulator can evaluate. We limit the execution to one iteration for three of our benchmarks to better understand how an iteration is affected by staggered checkpointing. For `Ek-simple`, we report results with 100 iterations. To perform these tests, we modify the line placement in our algorithm so that it places a line from both the lowest bin and a line from the highest bin.

We evaluate a line identified by our algorithm as having a low WAF value, or a staggered line, against a line identified by our algorithm as having a high WAF value. The lines with high WAF values typically perform *simultaneous checkpointing*, which occurs when all processes write their checkpoints at the same location in the application text. This technique differs from synchronous checkpointing since it does not require synchronization.

Our simulator performs optimistically in the face of network and file

system saturation, so we do not present the results for configurations that cause both staggered and simultaneous checkpointing to saturate the file system. The configurations that are presented in the tables are configurations that are inside the fidelity of our simulator. For Ranger, we present configurations of 1,024 processes each writing up to 256 MB, 4,096 processes each writing 32 MB, and 16,384 processes each writing 4 MB. For Lonestar, these configurations are smaller: 1,024 processes each writing 32 MB of data, and 4,096 processes writing 4 MB of data. Unlike our algorithm, our simulator scale with the number of processes. As such, our simulator cannot simulate in a reasonable amount of time two of our application benchmarks with 16,384 processes or any of the application benchmarks with 65,536 processes. Thus, these results are also not presented.

Our results show that our WAF metric works well when simultaneous checkpointing saturates the file system. In addition, when a sufficiently small number of processes write a sufficiently small amount of data such that the file system is not saturated, simultaneous checkpointing is better than staggered checkpointing, since staggered checkpointing can disrupt communication.

Simulated Results. In Tables 5.19-5.22, we present data supporting both our claim that staggered checkpointing improves performance and our claim that our WAF metric identifies useful recovery lines. Each of these tables presents the improvement in checkpointing time realized by the staggered line over the simultaneous line. This improvement is adjusted for the average

error of our simulator, and these tables also display the confidence interval around the mean and the tolerance interval. In addition, each table displays the adjusted improvement in the total execution time and the corresponding statistical information. We apply the error of the appropriate file system to the checkpoint times and the error of the appropriate benchmark to the total execution times.

Since we increase the local execution time of the phase of interest by scaling the problem size of the benchmark, the total execution time of the benchmark increases as well. Thus the total execution time improvement percentage is typically less for the larger interval size.

The results shown are for each machine when the checkpoint locations are staggered within three and fifteen minutes of local execution time. These time periods translate to three or fifteen minutes without a synchronization point or a collective communication, respectively.

Checkpoint Time. When checkpoint locations are separated within approximately three minutes of local execution, the staggered lines placed by our algorithm reduce checkpoint time by an average of 26% on Ranger (Table 5.19) and 35% on Lonestar (Table 5.20). Lonestar's percentage is higher in part because the highest improvements tend to be in the smaller configurations, and the Lonestar results have fewer large configurations. In these results, two Ranger configurations, `BT` and `Ek-simple` with 4,096 processes writing 4 MB of checkpoint data, become feasible. Two Lonestar configura-

tions also become feasible: 1,024 processes writing 32 MB of data each and 4,096 processes writing 4 MB of data each, both with `Ek-simple`. In general, the checkpoint time results are unsurprising and are expected from our solution space analysis. Here, we can conclude that the WAF metric can identify useful recovery lines even at this small interval size.

If we increase the amount of local execution time in which the checkpoint locations can be staggered to approximately fifteen minutes (Tables 5.21 and 5.22), we see that the staggered lines now show improvement in larger process configurations and almost all configurations show a large increase in the percentage improvement in checkpointing time over that shown in the three minute case.

In addition, the staggered lines placed by our algorithm make checkpointing some configurations feasible that are not with the simultaneous line, pointing to the usefulness of both staggered checkpointing and our WAF metric. On Ranger, simultaneous checkpointing is infeasible for 1,024 processes writing 256 MB (requires 1 hour 21 minutes), 4,096 processes writing 4 MB (requires 19 minutes) and 32 MB (requires 1 hour 34 minutes), and 16,384 processes writing 4 MB (requires 4 hours 57 minutes). For `BT` with 4,096 processes writing 4 MB, checkpointing using the staggered line requires only 12 minutes, an improvement of 7 minutes. For `Ek-simple` with 4,096 processes writing 4 MB, the staggered line reduces the checkpoint time to 38 seconds, an improvement of over 18 minutes; when this configuration writes 32 MB, the checkpointing time is reduced to 5 minutes from over an hour and a half.

On Lonestar, simultaneous checkpointing requires over fifteen minutes for the following configurations that we consider: 1,024 processes writing 32 MB (57 minutes) and 4,096 processes writing 4 MB (24 minutes). Staggered checkpointing makes checkpointing feasible in the former configuration for `BT` (11 minutes) and `Ek-simple` (4 minutes), and in the latter configuration for `BT` (5 minutes), `Ek-simple` (18 seconds), and `SP` (13 minutes).

In results not presented here, we see that the good lines identified by our algorithm are useful with a time interval in which to stagger that is much smaller than three minutes. The results show large percentages but small absolute times, so staggered checkpointing will be useful in these cases only if the application checkpoints frequently. For instance, with approximately 20 seconds in which to stagger in a phase in `BT`, the line with staggered checkpointing reduces checkpointing time by 30 seconds for 1,024 processes writing 4 MB of data, which is 28% of the checkpoint time. LU shows even more improvement for just over one minute local execution time: 51% or 36 seconds. The total execution time for these configurations, which with simultaneous checkpointing is approximately five minutes in the `BT` case and ten minutes for the `LU` example, is reduced by 9% for `BT` and 7% for `LU`.

Total Execution Time. The results that are particularly interesting in these tables are the total execution time results. The total execution time is important because staggered checkpointing can disrupt communication: our results show that the staggered line is more likely to improve performance even

in that case. However, the execution time is improved by a smaller number of seconds than the checkpointing time, a possible side-effect of communication disruption. Unfortunately, improving the execution time by a significant percentage is difficult since that time includes the entire benchmark. Thus, the percentage change is often not large enough to still exist once the simulator error is applied. However, we can glean interesting insights by evaluating the predicted improvement from the configurations that do show statistically significant results.

When checkpoints are staggered within a three minute interval size during communication, we would expect the total execution time to suffer since the communication is being disrupted during so compact an interval. However, in the Ranger results (Table 5.19), we see that our simulator typically predicts an improvement in total execution time, though that improvement is often not enough to place the lower end of the confidence and tolerance intervals in the positive. The configurations with a change large enough to be statistically significant are for `BT` with 4,096 or 16,384 processes and all amounts of data, and for 1,024 processes with 32 MB of data. In these cases, the staggered line exhibits a large improvement in checkpointing time and that improvement is reflected in the total execution time.

Although our simulator usually predicts 0% change for `Ek-simple` in the three minute case, for `Ek-simple` with 1,024 processes checkpointing 256 MB, our simulator predicts a 23% *gain* in total execution time. A possible cause is that `Ek-simple` has more irregular communication that the rest of

the benchmarks, so its clumps tend to not be strongly connected, making `Ek-simple` more susceptible to communication disruption.

In the fifteen minute case, our simulator generally predicts an improvement in total execution time with the staggered line for three of our benchmarks: `BT`, `LU`, and `SP`. On Ranger, these benchmarks typically show improvement in process sizes greater than 1,024: a finding in keeping with our solution space results which show little improvement on Ranger for process sizes less than or equal to 1,024. On Lonestar, these benchmarks always show improvement. Unlike the other benchmarks, `Ek-simple` either exhibits no change or a small increase in the total execution time. However, this tradeoff is reasonable since the percentage change is generally small and in many cases checkpointing with the staggered line causes checkpointing `Ek-simple` to be feasible.

Measured Results. Our measured results show that we can expect an improvement in the total execution time. Table 5.23 shows that staggered checkpointing improves the execution time of one iteration of the `LU` benchmark by an average of 41%, even for these small configurations.

5.8 Conclusions

We show that staggered checkpointing is an effective checkpointing technique and that it is most effective for large numbers of processes. As file systems become faster, staggered checkpointing will become effective for even larger numbers of processes and sizes of checkpoint data; we expect the

size of applications will grow due to the current support for capability computing, which encourages application developers to use more processes per application. For instance, the original RedStorm allocation policies set aside 80% of the compute-node hours to be used by applications using a minimum of 40% of the computer nodes [147]: RedStorm initially had 10,880 processors, and now has 38,400. As the applications use more processes, the need for staggered checkpointing will increase as will its effectiveness.

We develop a scalable and efficient recovery line generation algorithm that enables staggered checkpointing and thus relieves programmer burden. This algorithm uses a constraint-based search to eliminate redundant work and several techniques to reduce the search space, including the WAF metric. For our benchmarks, our algorithm successfully finds and places useful recovery lines in applications configured to use 16,384 processes that are up to 52% faster than the equivalent numbers of processes simultaneously writing an equivalent amount of data. Our algorithm is the most useful for applications with large amounts of point-to-point communication. If the application instead consists of local execution separated by collective communication, staggered checkpointing is trivial and our algorithm is less effective.

Currently the algorithm analyzes and places a recovery line in every phase of an application. A possible extension would consider the length of each phase and only place a recovery line in the longest phase. This extension would reduce execution time.

The WAF metric is currently a simplified circular string edit distance

and it works well. However, it could be extended to also consider the amount of data that would need to be saved at each location. This extension could potentially reduce checkpoint size.

In addition, checkpoint locations are currently limited to dependence-generating events. Due to this constraint, it is possible to further optimize the recovery line by adjusting the checkpoint locations in the generated recovery lines relative to the checkpoint locations we consider. Such adjustments may both increase the separation of the checkpoints and reduce the amount of data checkpointed—both of which will further reduce network and file system contention.

Benchmark	Processes	Checkpoint Data Per Process	Improvement in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval	Improvement in Total Execution Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)			Sec	(%)		
BT	1,024	4MB	54s	(56%)	53%-59%	52%-60%	59s	(3%)	-1%-7%	-2%-8%
		32MB	135s	(23%)	19%-26%	18%-27%	141s	(6%)	2%-10%	1%-10%
		256MB	95s	(2%)	-2%-6%	-3%-7%	140s	(2%)	-2%-6%	-3%-7%
	4,096	4MB	430s	(39%)	35%-42%	34%-43%	415s	(9%)	5%-13%	4%-13%
		32MB	1,365s	(25%)	21%-29%s	20%-30%	1,211s	(12%)	8%-16%	8%-17%
	16,384	4MB	8,952s	(52%)	49%-55%	48%-56%	9,795s	(41%)	37%-44%	37%-44%
Ek-simple	1,024	4MB	60s	(62%)	59%-65%	58%-66%	-879s	(0%)	-10%-10%	-13%-12%
		32MB	293s	(49%)	46%-52%	45%-53%	-11,938s	(-3%)	-13%-7%	-15%-10%
		256MB	2,127s	(45%)	42%-48%	41%-49%	-99,888s	(-23%)	-34%–12%	-36%–9%
	4,096	4MB	493s	(45%)	41%-48%	40%-49%	949s	(0%)	-10%-10%	-12%-12%
		32MB	2,342s	(43%)	40%-46%	39%-47%	2,153s	(0%)	-9%-10%	-12%-13%
LU	1,024	4MB	50s	(51%)	48%-54%	47%-55%	54s	(4%)	-3%-10%	-4%-11%
		32MB	65s	(11%)	7%-15%	6%-16%	71s	(3%)	-3%-10%	-4%-11%
		256MB	66s	(1%)	-3%-5%	-4%-7%	71s	(1%)	-5%-8%	-7%-9%
	4,096	4MB	222s	(20%)	16%-24%	15%-25%	244s	(7%)	1%-13%	-1%-15%
		32MB	207s	(4%)	0%-8%	-1%-9%	227s	(3%)	-4%-9%	-5%-10%
SP	1,024	4MB	37s	(38%)	35%-41%	34%-42%	42s	(6%)	-2%-13%	-4%-15%
		32MB	23s	(4%)	0%-8%	-1%-9%	27s	(2%)	-6%-10%	-7%-11%
		256MB	23s	(0%)	-4%-5%	-5%-6%	27s	(0%)	-7%-8%	-9%-10%
	4,096	4MB	94s	(9%)	5%-12%	3%-14%	108s	(5%)	-2%-13%	-4%-15%
		32MB	84s	(2%)	-3%-6%	-4%-7%	97s	(1%)	-6%-9%	-8%-11%
	16,384	4MB	74s	(0%)	-4%-5%	-5%-6%	82s	(0%)	-7%-8%	-9%-10%

Table 5.19: Ranger data for approximately three minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line places by our algorithm and the simultaneous one.

Benchmark	Processes	Checkpoint Data Per Process	Improvement in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval	Improvement in Total Execution Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)			Sec	(%)		
BT	1,024	4MB	94s	(65%)	60%-70%	59%-71%	79s	(5%)	4%-6%	3%-6%
		32MB	702s	(20%)	14%-26%	12%-27%	582s	(12%)	11%-13%	11%-14%
	4,096	4MB	520s	(35%)	29%-40%	27%-42%	391s	(9%)	8%-10%	7%-10%
Ek-simple	1,024	4MB	108s	(75%)	70%-80%	69%-81%	-249s	(0%)	-2%-2%	-3%-3%
		32MB	2,927s	(82%)	78%-87%	76%-88%	-1,339s	(0%)	-3%-2%	-3%-2%
	4,096	4MB	653s	(43%)	38%-49%	37%-50%	966s	(0%)	-2%-2%	-2%-3%
LU	1,024	4MB	82s	(57%)	52%-62%	51%-64%	90s	(7%)	6%-8%	6%-8%
		32MB	171s	(5%)	-2%-11%	-3%-13%	190s	(4%)	3%-4%	2%-5%
	4,096	4MB	173s	(12%)	5%-18%	4%-19%	191s	(5%)	4%-6%	4%-6%
SP	1,024	4MB	33s	(23%)	17%-29%	15%-30%	32s	(6%)	5%-7%	5%-7%
		32MB	62s	(2%)	-5%-8%	-7%-10%	62s	(1%)	1%-2%	1%-2%
	4,096	4MB	57s	(4%)	-3%-10%	-5%-12%	57s	(3%)	2%-4%	2%-4%

Table 5.20: Lonestar data for approximately three minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line placed by our benchmark and the simultaneous one.

Benchmark	Processes	Checkpoint Data Per Process	Improvement in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval	Improvement in Total Execution Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)			Sec	(%)		
BT	1,024	4MB	56s	(75%)	55%-61%	54%-62%	62s	(0%)	-4%-4%	-4%-5%
		32MB	266s	(54%)	41%-48%	40%-49%	288s	(2%)	-2%-6%	-3%-7%
		256MB	1,028s	(24%)	18%-25%	17%-27%	1,071s	(6%)	2%-10%	1%-10%
	4,096	4MB	381s	(34%)	31%-38%	30%-39%	342s	(2%)	-2%-6%	-3%-7%
		32MB	1,153s	(21%)	17%-25%s	16%-26%	997s	(5%)	1%-8%	0%-9%
	16,384	4MB	9,064s	(52%)	49%-56%	48%-57%	9,934s	(27%)	23%-30%	22%-31%
Ek-simple	1,024	4MB	47s	(48%)	45%-51%	44%-52%	108s	(0%)	-10%-10%	-12%-12%
		32MB	221s	(37%)	34%-41%	33%-42%	-8,661s	(0%)	-10%-10%	-13%-12%
		256MB	1,734s	(37%)	33%-40%	32%-41%	-95,664s	(-4%)	-14%-6%	-16%-9%
	4,096	4MB	1,071s	(97%)	94%-100%	93%-100%	-1,287s	(0%)	-7%-7%	-9%-9%
		32MB	5,137s	(94%)	91%-97%	91%-98%	-18,438s	(0%)	-10%-9%	-13%-12%
LU	1,024	4MB	50s	(51%)	48%-54%	47%-55%	54s	(1%)	-5%-7%	-7%-9%
		32MB	241s	(40%)	37%-44%	36%-45%	263s	(4%)	-2%-11%	-3%-12%
		256MB	267s	(6%)	2%-10%	0%-11%	292s	(3%)	-4%-9%	-5%-11%
	4,096	4MB	788s	(71%)	68%-74%	67%-75%	849s	(9%)	3%-15%	1%-17%
		32MB	876s	(16%)	12%-20%	11%-21%	942s	(6%)	0%-13%	-1%-14%
SP	1,024	4MB	56s	(58%)	55%-61%	54%-62%	68s	(2%)	-5%-10%	-7%-12%
		32MB	119s	(20%)	16%-24%	15%-25%	137s	(4%)	-4%-11%	-6%-13%
		256MB	119s	(3%)	-2%-7%	-3%-8%	137s	(2%)	-6%-9%	-8%-11%
	4,096	4MB	616s	(56%)	52%-59%	51%-60%	704s	(14%)	7%-22%	6%-23%
		32MB	556s	(10%)	6%-14%	5%-15%	635s	(6%)	-1%-14%	-3%-15%
	16,384	4MB	814s	(5%)	1%-9%	0%-10%	948s	(4%)	-4%-11%	-6%-13%

Table 5.21: Ranger data for approximately fifteen minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line identified by our algorithm and the simultaneous one.

Benchmark	Processes	Checkpoint Data Per Process	Improvement in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval	Improvement in Total Execution Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)			Sec	(%)		
BT	1,024	4MB	97s	(67%)	62%-72%	61%-73%	59s	(1%)	-1%-2%	-1%-2%
		32MB	1,696s	(71%)	44%-51%	44%-52%	1,701s	(12%)	10%-13%	10%-13%
	4,096	4MB	472s	(31%)	26%-37%	24%-39%	336s	(2%)	1%-4%	1%-4%
Ek-simple	1,024	4MB	79s	(55%)	50%-60%	48%-62%	183s	(0%)	-2%-2%	-3%-3%
		32MB	2,278s	(64%)	59%-69%	58%-71%	1,845s	(0%)	-2%-2%	-3%-3%
	4,096	4MB	1,469s	(98%)	93%-102%	92%-104%	426s	(0%)	-2%-2%	-3%-3%
LU	1,024	4MB	82s	(58%)	52%-63%	51%-64%	90s	(2%)	1%-3%	1%-3%
		32MB	702s	(20%)	14%-26%	12%-27%	776s	(9%)	8%-10%	8%-10%
	4,096	4MB	731s	(49%)	43%-54%	42%-55%	792s	(9%)	8%-10%	8%-10%
SP	1,024	4MB	96s	(66%)	61%-71%	60%-73%	100s	(5%)	4%-6%	4%-6%
		32MB	319s	(9%)	3%-15%	1%-17%	321s	(6%)	5%-6%	5%-7%
	4,096	4MB	375s	(25%)	19%-31%	18%-32%	375s	(10%)	9%-11%	9%-11%

Table 5.22: Lonestar data for approximately fifteen minutes of local execution: This table presents the improvement shown by staggered checkpointing on Lonestar when the checkpoint locations are staggered within approximately three minutes of local execution. The first and second columns report the benchmarks and number of processes under consideration. The third column, "Checkpoint Data Per Process", reports the amount of data being written by each process. The remaining eight columns show realized improvement between the staggered line identified by our algorithm and the simultaneous one.

Processes	Problem Size	Data Per Process	Total Execution Time with Staggered Checkpointing	Total Execution Time with Simultaneous Checkpointing	Percent Improvement
16	204	4MB	29s	46s	37%
		32MB	145s	322s	55%
64	408	4MB	40s	59s	38%
		32MB	209s	323s	35%

Table 5.23: Measured total execution time for one iteration of the LU benchmark on Lonestar for executions checkpointing with either staggered or simultaneous checkpointing

Chapter 6

Conclusions and Future Work

In this thesis, we have provided a compiler-assisted checkpointing technique that reduces programmer work, reduces network and file system contention, and does not require process synchronization. Our solution is a form of staggered checkpointing that places process checkpoint calls at different locations in the application text at compile-time while guaranteeing that the saved checkpoints will form a consistent state. Our technique eliminates dynamic inter-process coordination and creates a separation in time of each process's checkpoint, which reduces contention at the file system.

6.1 Contributions

In this dissertation we have presented an efficient and scalable compile-time algorithm that identifies staggered checkpoint locations in parallel applications. We have also identified application and system characteristics where staggered checkpointing reduces checkpoint overhead, and presented a simulator that efficiently simulates parallel applications. We have shown that staggered checkpointing is effective and that is is poised to become more effective in the future as applications use more processes and file systems become

faster. We now review our contributions.

6.1.1 Algorithm for Compiler-Assisted Staggered Checkpointing

Identifying desirable checkpoint locations in an application is difficult since there are many possible locations and the state saved must be consistent. To ensure a consistent state, our solution identifies valid *recovery lines* [153]. It is a challenge to statically differentiate the desired valid recovery lines from those that are invalid because the number of possible solutions is enormous. This number grows as L^P, where L is the number of possible checkpoint locations and P is the number of processes, and the number of valid recovery lines is typically less than 1% of the total. Recovery lines that are *useful*—a useful recovery line is both valid and sufficiently staggered to reduce contention—are even more rare.

We designed and implemented a new compile-time algorithm that generates and places useful recovery lines in applications that use up to tens of thousands of processes. This algorithm uses a constraint-based search to eliminate redundant work and reduces the search space by constraining the checkpoint locations, considering clumps of processes rather than independent processes, and performing the search for a useful recovery lines within sets of clumps where each process is represented exactly once. These techniques reduce the search space for `BT` with 1,024 processes from 38^{1024} to $3(5^{32}) + 2^1$, or by 1,594 orders of magnitude. Our pruning policy enables the rapid sorting of the created recovery lines, and a metric we introduced, the *Wide-and-Flat*

(WAF) metric, statically estimates the usefulness of both complete recovery lines, those that include every process, and partial recovery lines, those that only include some of the processes.

For our benchmarks, our algorithm successfully finds and places useful recovery lines in applications that use up to 65,536 processes. The staggered recovery lines placed by our algorithm checkpoint an average of 37% faster for all configurations than a recovery line performing simultaneous checkpointing that writes an equivalent amount of data.

6.1.2 Problem and Solution Spaces for Staggered Checkpointing

We used our simulator to investigate sets of application characteristics, including the number of processes and checkpoint sizes, and system characteristics, including network, file system, and processor speeds, where staggered checkpointing is needed and effective. We identified situations where staggered checkpointing is needed, which we refer to as the *problem space*, by determining sets of application and system characteristics where synchronous checkpointing causes contention but the checkpointing of a single process does not. We identified situations where staggered checkpointing is effective, or the *solution space*, by determining sets of characteristics for which staggered checkpointing reduces such contention. We also analyzed how the solution space varies with machine characteristics such as the ratio of network and file system speeds to processor speeds.

Within the solution spaces, staggered checkpointing reduces both check-

pointing time and overall execution time. The solution spaces as presented are significant spaces for both Lonestar and Ranger; staggered checkpointing is a useful technique for both these systems. Checkpoint locations only need to be separated by approximately three minutes for improvements to be realized in some configurations on both machines. As the time separating the checkpoints increases, the amount of improvement shown increases.

Our analysis of synchronous checkpointing shows that it can take up to an hour and 14 minutes for 8,192 processes on Ranger to checkpoint 4 MB of data each. With three minutes over which to separate the checkpoints, staggered checkpointing reduces this time by 12 minutes to just over an hour. With fifteen minutes, staggered checkpointing reduces this time to 1 minute. The results on Lonestar are similar. These results imply that staggered checkpointing is a useful technique. Since Ranger is ranked 11 in the Top 500 Supercomputer Sites [8] and Lonestar is ranked 123, staggered checkpointing is applicable to a large range of supercomputers.

6.1.3 Simulation for Large-Scale Systems

Evaluating large-scale parallel applications in the presence of checkpointing is difficult to accomplish with both low overhead and good accuracy. To do so, we developed an event-driven simulator for large-scale systems that estimates the behavior of the network, global file system, and local hardware using predictive models. Currently, each model is a formula that encapsulates the performance of its respective component and thus provides the simula-

tor with a fast way to estimate that component's behavior under a variety of workloads.

Our simulator can simulate applications that use thousands of processes. On average, it predicts the execution time our application benchmarks with checkpointing as 83% of their measured performance, which provides sufficient accuracy to evaluate synchronous and staggered checkpointing.

6.2 Future Work

Our work provides a technique for identifying and placing useful recovery lines in large-scale applications. To enable this work to be tested more thoroughly and possibly used in a production environment, it should be combined with a tool that collects and writes the checkpoint data. In addition, a recovery algorithm needs to be designed and implemented.

Our algorithm currently limits checkpoint locations to dependence-generating events. Due to this constraint, it is possible to further optimize the recovery line by adjusting the checkpoint locations in the generated recovery lines relative to these dependence-generating events. Such adjustments may both increase the separation of the checkpoints and reduce the amount of data checkpointed—both of which will further reduce network and file system contention.

Our algorithm identifies and places recovery lines in every phase of an application. The algorithm could be extended to detect the best phase

for checkpointing and only analyze and place a recovery line in that phase. Currently, any placed recovery line will checkpoint every time the application reaches its context. The algorithm could be extended to only allow each placed line to checkpoint at some interval.

Wide-and-Flat Metric. The WAF metric is currently a simplified circular string edit distance [88] and it works well. However, it could be extended to also consider the amount of data that would need to be saved at each location. This extension could potentially reduce checkpoint size.

In Chapter 5, we showed that our WAF metric performs better when the checkpoint data size is larger. This phenomena occurs since the WAF metric assumes that the more staggered a line is, the more desirable it is. If we modified the metric to consider both checkpoint size and the characteristics of a system, it could instead identify lines that are sufficiently staggered to reduce the amount of data being written at any single location to below the network and file system saturation points. Lines that are staggered more than that amount can disrupt communication unnecessarily.

6.3 Final Thoughts

Staggered checkpointing extends the viability of checkpointing past the current state of the art to larger numbers of processes writing larger amounts of data. Our results on Lonestar and Ranger showed that as we move toward bigger and faster machines, staggered checkpointing will continue to extend the

viability of checkpointing. In addition, as applications grow larger, they will benefit more from staggered checkpointing. However, staggered checkpointing only extend the viability of checkpointing, so some applications will still not be able to checkpointing a reasonable amount of time. There are many reasons why staggered checkpointing might not make checkpointing some applications viable.

Some applications will need to checkpoint too much data for staggered checkpointing to sufficiently reduce checkpoint time. Those applications should use techniques to reduce checkpoint data size in addition to staggered checkpointing. These techniques include carefully identifying the data to be saved, data compression, and carefully choosing a file format, such as a high-density one.

Some applications will not have a long enough interval without a collective communication or synchronization point for the checkpoints to be sufficiently separated: staggered checkpointing works best for applications in which these intervals execute for minutes at a time. Unfortunately, many current applications have more closely spaced synchronization and collective communication. Although as file systems become faster, staggered checkpointing will be more effective with smaller intervals, it will always be more beneficial with more time over which to separate the checkpoints. To gain a larger benefit from staggered checkpointing, application developers would need to carefully design their applications to limit the use of synchronization and collective communication.

Some applications will have too many constraints for staggered checkpointing. Our algorithm statically guarantees a consistent state by analyzing the dependences introduced by an application's communication and then placing the checkpoints in such a way that the state saved is consistent. We can imagine that so many dependences could be introduced that the checkpoints could not be staggered. Though this case is possible with any communication, it is more likely to occur when the application performs non-deterministic communication, and our algorithm is forced to conservatively estimate the dependences.

Checkpointing large parallel applications still faces these and other challenges. This thesis contributes to the understanding of these challenges and takes steps to address some of them. In particular, it introduces compiler-assisted staggered checkpointing which extends the viability of checkpointing past the current state of the art.

Bibliography

[1] ASCI Sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/.

[2] Texas Advanced Computing Center (TACC). The University of Texas at Austin.

[3] *Texas Advanced Computing Center Usage Policies*.

[4] FOR_C: Fortran 77 to C translator. http://www.cobalt-blue.com, 1988.

[5] *Capability Compute System Scheduling Governance Model, Advanced Simulation and Computing Program*, April 2006.

[6] News from the San Diego Supercomputer Center. `http://www.sdsc.edu/nuggets/200706_nuggets.html`, June 2007.

[7] MVAPICH 1.0 user and tuning guide. `http://mvapich.cse.ohio-state.edu/support/mvapich_user_guide.html`, May 2008.

[8] Top 500 list. http://www.top500.org/list/2010/06/200, June 2010.

[9] Vikram Adve and Rizos Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *Int. J. High Perform. Comput. Appl.*, 14(4):304–316, 2000.

[10] Vikram S. Adve, Rajive Bagrodia, Ewa Deelman, Thomas Phan, and Rizos Sakellariou. Compiler-supported simulation of highly scalable parallel applications. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[11] Vikram S. Adve, Rajive Bagrodia, Ewa Deelman, and Rizos Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *Journal of Parallel and Distributed Computing*, 62(3):393–426, 2002.

[12] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 277–286, 2004.

[13] Adnan Agbaria, Ari Freund, and Roy Friedman. Evaluating distributed checkpointing protocols. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 266–273, 2003.

[14] Adnan Agbaria and William H. Sanders. Application-driven coordination-free distributed checkpointing. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 177–186, 2005.

[15] J. Ahn and P.B. Danzig. Speedup vs. simulation granularity. *IEEE/ACM Transactions on Networking*, 4(5):743–757, October 1996.

[16] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of IBM Blue Gene/P. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

[17] Sadaf R. Alam, Richard F. Barrett, Mark R. Fahey, Jeffery A. Kuehn, O.E. Bronson Messer, Richard T. Mills, Philip C. Roth, Jeffrey S. Vetter, and Patrick H. Worley. An evaluation of the Oak Ridge National Laboratory Cray XT3. *International Journal of High Performance Computing Applications*, 22(1):52–80, 2008.

[18] Sadaf R. Alam, Jeffery A. Kuehn, Richard F. Barrett, Jeff M. Larkin, Mark R. Fahey, Ramanan Sankaran, and Patrick H. Worley. Cray XT4: an early evaluation for petascale scientific simulation. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[19] S.R. Alam, R.F. Barrett, J.A. Kuehn, P.C. Roth, and J.S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *IEEE International Symposium on Workload Characterization*, pages 225–236, October 2006.

[20] George Almasi, Gyan Bhanot, Alan Gara, Manish Gupta, James Sexton, Bob Walkup, Vasily V. Bulatov, Andrew W. Cook, Bronis R. de Supinski, James N. Glosli, Jeffrey A. Greenough, Francois Gygi, Alison Kubota, Steve Louis, Thomas E. Spelce, Frederick H. Streitz, Peter L. Williams, Robert K. Yates, Charles Archer, Jose Moreira, and Charles Rendleman.

Scaling physics and material science applications on a massively parallel Blue Gene/L system. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 246–252, 2005.

[21] Lorenzo Alvisi, Elmootazbellah N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka de Mel. An analysis of communication-induced checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 242–249, June 1999.

[22] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *International Parallel and Distributed Processing Symposium*, pages 1–12, Rome, Italy, May 2009.

[23] K. Antypas, A. C. Calder, A. Dubey, R. Fisher, M. K. Ganapathy, J. B. Gallagher, L. B. Reid, K. Riley, D. Sheeler, and N. Taylor. Scientific applications on the massively parallel BG/L machines. In *PDPTA*, pages 292–298, 2006.

[24] Sarala Arunagiri, Seetharami Seelam, Ron A. Oldfield, Maria Ruiz Varela, Patricia J. Teller, and Rolf Riesen. Impact of checkpoint latency on the optimal checkpoint interval and execution time. Technical Report UTEP-CS-07-55, The University of Texas at El Paso, 2007.

[25] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W.

Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[26] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[27] A. I. Avetisyan, S. S. Gaisaryan, V. P. Ivannikov, and V. A. Padaryan. Productivity prediction of MPI programs based on models. *Automation and Remote Control*, 68(5):750–759, 2007.

[28] David H. Bailey and Allan Snavely. Performance modeling: Understanding the present and predicting the future. Technical report, Lawrence Berkeley National Laboratory, November 2005.

[29] Stephen Beale. *Hunter-Gatherer: Applying Constraint Satisfaction, Branch-and-Bound and Solution Synthesis to Computational Semantics*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1997.

[30] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, University of Tennessee, December 1994.

[31] Toine Beckers. High performance storage solutions. DataDirect Networks Presentation, December 2006.

[32] Mesfin Belachew, A. Gandhi, S. Gandwala, and R.K. Shyamasundar. MSC+ generalized hierarchical message sequence charts. In R. K. Ghosh

and Durgamadhab Misra, editors, *The Third International Conference on Information Technology (CIT2000)*, pages 183–189, Bhubaneswar, India, December 2000. Tata McGraw-Hill Publishing Company Limited.

[33] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[34] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007.

[35] Julien Bourgeois and François Spies. Performance prediction of an NAS benchmark program with ChronosMix environment. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 208–216, London, UK, 2000. Springer-Verlag.

[36] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, 2003.

[37] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *Proceedings of the 1996 ACM SIGMETRICS Interna-*

tional Conference on Measurement and Modeling of Computer Systems, pages 80–90, 1996.

[38] Jürgen Brehm, Manish Madhukar, Evgenia Smirni, and Lawrence W. Dowdy. PerPreT - a performance prediction tool for massive parallel systems. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 284–298, London, UK, 1995. Springer-Verlag.

[39] Jürgen Brehm, Patrick H. Worley, and Manish Madhukar. Performance modeling for SPMD message-passing programs. *Concurrency: Practice and Experience*, 10(5):333–357, April 1998.

[40] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[41] Ron Brightwell, Lee Ann Fisk, David S. Greenberg, Tramm Hudson, Mike Levenhagen, Arthur B. MacCabe, and Rolf Riesen. Massively parallel computing using commodity components. *Parallel Computing*, 26(2-3):243–266, 2000.

[42] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Radu Rugina, and Sally A. McKee. Compiler-enhanced incremental checkpointing for OpenMP applications. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, pages 275–276, 2008.

[43] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 84–94, 2003.

[44] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in application-level fault-tolerant MPI. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 234–243, 2003.

[45] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *ACM SIGOPS Operating Systems Review*, 38(5):235–247, 2004.

[46] Shawn T. Brown. Extreme scaling. Presentation at Pittsburgh Supercomputing Center's Optimizing Applications for Highly Scalable Multicore Architectures Workshop, August 2007.

[47] Rajkumar Buyya and Manzur Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. In *Journal of Concurrency and Computation: Practice and Experience*, pages 1175–1220. Wiley Press, 2002.

[48] Franck Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[49] Laura Carrington, Nicole Wolter, and Allan Snavely. A framework for application performance prediction to enable scalability understanding. In *Proceedings of the Scaling to New Heights Workshop*, Pittsburgh, May 2002.

[50] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430, 2001.

[51] NASA Ames Research Center. NAS parallel benchmarks. `http://www.nas.nasa.gov/Software/NPB`.

[52] Sayantan Chakravorty and Laxmikant V. Kalé. A fault tolerance protocol with fast fault recovery. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 120, 2007.

[53] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[54] K. Mani Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*,

C-21(6):546–556, June 1972.

[55] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):015001, 2009.

[56] Yuqun Chen, James S. Plank, and Kai Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–11, 1997.

[57] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 2005.

[58] Sung-Eun Choi and Steven J. Deitz. Compiler support for automatic checkpointing. In *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 213–220, June 2002.

[59] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI.

In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[60] Yifeng Cui, Regan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, and Thomas Jordan. Toward petascale earthquake simulations. *Acta Geotechnica*, 4(2):1861–1125, July 2009.

[61] John Daly. *Computational Science — ICCS 2006*, volume 2660/2003 of *Lecture Notes in Computer Science*, chapter A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps, pages 3–12. Springer Berlin/Heidelberg, January 2003.

[62] Laura Davey, Abdy Martinez, Georgia Pedicini, and Phil Salazar. Reliability issues on ASCI Blue Mountain and ASCI Q. Presentation at the Workshop on Scalable Fault Tolerance for Distributed Computing, Computer Science Research Institute, Sandia National Labs, June 2002.

[63] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, 2001.

[64] H. DeSterck, R. D. Falgout, J. W. Nolting, and U. M. Yang. Distance-two interpolation for parallel algebraic multigrid. *Journal of Physics Conference Series*, 78(1):12–17, July 2007.

[65] Elmootazbellah N. Elnozahy. Rollback-recovery in the petascale era. Talk given at the 2009 Fault Tolerance for Extreme-Scale Computing Workshop, March 2009.

[66] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[67] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[68] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The performance of consistent checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 39–47, Houston, Texas, 1992.

[69] Elmootazbellah N. Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[70] Environment Molecular Sciences Lab, Pacific Northwest National Laboratory. *Getting a Refund of Allocated Time*.

[71] Environmental Molecular Sciences Lab, Pacific Northwest National Laboratory. *Job Policies*.

[72] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

[73] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, November 1978.

[74] Richard M. Fujimoto, Kalyan Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-scale network simulation: How big? How fast? In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 116–123, October 2003.

[75] Sergey Gaissaryan, Arutyun Avetisyan, Oleg Samovarov, and Dmitry Grushin. Comparative analysis of high-performance clusters' communication environments using HPL test. In *Proceedings of the High Performance Computing and Grid in Asia Pacific Region*, pages 473–476, 2004.

[76] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, 2006.

[77] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the*

20th Annual International Conference on Supercomputing, pages 269–277, 2006.

[78] Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, and Roger D. Chamberlain. A federated simulation environment for hybrid systems. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, pages 198–210, 2007.

[79] Roberto Gioiosa, José Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, 2005.

[80] TJ Giuli and Mary Baker. Narses: A scalable flow-based network simulator. Technical report, Stanford University, November 2003.

[81] Michael T. Goodrich and Daniel S. Hirschberg. Efficient parallel algorithms for dead sensor diagnosis and multiple access channels. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 118–127, 2006.

[82] N.P. Gopalan and K. Nagarajan. *Distributed Computing—IWDC*, volume 3741/2005 of *Lecture Notes in Computer Science*, chapter Self-refined Fault Tolerance in HPC Using Dynamic Dependent Process Groups, pages 153–158. Springer Berlin / Heidelberg, 2005.

[83] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *Proceedings of the 16th International Conference on Supercomputing*, pages 77–83, 2002.

[84] Eric Grobelny, David Bueno, Ian Troxel, Alan D. George, and Jeffrey S. Vetter. FASE: A framework for scalable performance prediction of HPC systems and applications. *Simulation*, 83(10):721–745, 2007.

[85] Divya Gulati, Madhu Saravana, and Sibi Govindan. MPI communication analysis. Project Report, May 2005.

[86] Divya Gulati, Madhu Saravana, and Sibi Govindan. Parallel systems. Project Report, May 2005.

[87] Yang Guo, Weibo Gong, and Don Towsley. Time-stepped hybrid simulation (TSHS) for large scale networks. In *Proceedings of IEEE Infocom*, pages 441–450, Tel Aviv, March 2000.

[88] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* The Press Syndicate of The University of Cambridge, 1997.

[89] Samuel Z. Guyer, Daniel A. Jiménez, and Calvin Lin. The C-Breeze compiler infrastructure. Technical Report TR 01-43, Dept. of Computer Sciences, University of Texas at Austin, November 2001.

[90] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd conference on Domain-Specific Languages*, pages 39–52, 1999.

[91] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In R.F. Boisvert and P.T. P. Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, 2000.

[92] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions. *ArXiv e-prints*, September 2009.

[93] S. D. Hammond, J. A. Smith, G. R. Mudalige, and S. A. Jarvis. Predictive simulation of HPC applications. In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications*, pages 33–40, 2009.

[94] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 40:494–499, 2006.

[95] John W. Haskins, Jr. and Kevin Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. Technical report, University of Virginia, Charlottesville, VA, USA, 2002.

[96] John Heidemann, Nirupama Bulusu, Jeremy Elson, Chalermek Intanagonwiwat, Kun chan Lan, Ya Xu, Wei Ye, Deborah Estrin, and Ramesh Govin-

dan. Effects of detail in wireless network simulation. In *SCS Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 1–10, Phoenix, AZ, January 2001.

[97] Mark Hereld, Rick Stevens, Justin Teller, Wim Van Drongelen, and Hyong Lee. Large neural simulations on large parallel computers. *International Journal of Bioelectromagnetism*, 7:44–46, 2005.

[98] Justin C. Y. Ho, Cho-Li Wang, and Francis C. M. Lau. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, April 2008.

[99] Patricia D. Hough and Victoria E. Howle. *Parallel Processing for Scientific Computing*, chapter Fault Tolerance in Large-Scale Scientific Computing, pages 203–220. Society for Industrial and Applied Mathematics, Philadelphia, 2006.

[100] James H. Laros III, Lee Ward, Ruth Klundt, Sue Kelly, James L. Tomkins, and Brian R. Kellogg. Red Storm IO performance analysis. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 50–57, September 2007.

[101] Intel. Intel MPI benchmarks.

[102] V. P. Ivannikov, S. S. Gaisaryan, A. I. Avetisyan, and V. A. Padaryan. Estimation of dynamical characteristics of a parallel program on a model.

Programming and Computing Software, 32(4):203–214, 2006.

[103] Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, and Vartan Padaryan. Development of scalable parallel programs in ParJava environment. In *Proceedings of the 15th Parallel Computational Fluid Dynamics Meeting*, pages 291–293, Moscow, May 2003.

[104] Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, and Vartan Padaryan. *Parallel Computing Technologies*, volume 4671/2007 of *Lecture Notes in Computer Science*, chapter Model Based Performance Evaluation for MPI Programs, pages 537–543. Springer Berlin/Heidelberg, 2007.

[105] Tor E. Jeremiassen and Susan J. Eggers. Computing per-process summary side-effect information. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 175–191, London, UK, 1993. Springer-Verlag.

[106] Tor E. Jeremiassen and Susan J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 171–180, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[107] Qiangfeng Jiang and D. Manivannan. An optimistic checkpointing and selective message logging approach for consistent global checkpoint col-

lection in distributed systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[108] Hyungsoo Jung, Dongin Shin, Hyuck Han, Jai W. Kim, Heon Y. Yeom, and Jongsuk Lee. Design and implementation of multiple fault-tolerant MPI over Myrinet (M^3). In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 32, 2005.

[109] Laxmikant V. Kalé, Mark Hills, and Chao Huang. An orchestration language for parallel objects. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems*, pages 1–6, 2004.

[110] Joost-Pieter Katoen and L. Lambert. Pomsets for message sequence charts. In H. König and P. Langendörfer, editors, *Formale Beschreibungstechniken für Verteilte Systeme*, pages 197–207, Berlin, Germany, June 1998. GI/ITG, Shaker Verlag.

[111] J Kennedy and R Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, August 1995.

[112] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, page 37, 2001.

[113] Boris Koldehofe, Marina Papatriantafilou, Philippas Tsigas, and Phuong Hoai Ha. LYDIAN: User's guide. Technical report, Chalmers University of Technology and Göteborg University, 2004.

[114] Ethan J. Kubatko, Shintaro Bunya, Clint Dawson, Joannes J. Westerink, and Chris Mirabito. A performance comparison of continuous and discontinuous finite element shallow water models. *J. Sci. Comput.*, 40(1-3):315–339, 2009.

[115] Lawrence Livermore National Lab. Interleaved or Random I/O benchmark.

[116] Peter B. Ladkin and Stefan Leue. Four issues concerning the semantics of message flow graphs. In *Proceedings of the Seventh International Conference on Formal Description Techniques*, IFIP. Chapman and Hall, 1995.

[117] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[118] Peter B. Ladkin and Barbara B. Simons. Compile-time analysis of communicating processes. In *Proceedings of the Sixth International Conference on Supercomputing*, pages 248–259, July 1992.

[119] Peter B. Ladkin and Barbara B. Simons. *Lecture Notes in Computer Science*, chapter Static Analysis of Interprocess Communication. Springer-Verlag, 1994.

[120] Don Lamb. Insights into fault tolerance from FLASH production runs on Top 3 supercomputing platforms. Talk given at the 2009 Fault Tolerance for Extreme-Scale Computing Workshop, March 2009.

[121] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[122] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–258, 2007.

[123] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software—Practice and Experience*, 24(10):871–886, October 1994.

[124] Chung-Chi Jun Li and W. Kent Fuchs. CATCH–compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 74–81, Newcastle Upon Tyne, UK, June 1990.

[125] Yinglung Liang, Yanyong Zhang, Morris Jette, Anand Sivasubramaniam, and Ramendra Sahoo. Blue Gene/L failure analysis and prediction models. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 425–434, June 2006.

[126] Yinglung Liang, Yoanyong Zhang, Anand Sivasubramaniam, Ramendra K. Sahoo, Jose Moreira, and Manish Gupta. Filtering failure logs for a Blue Gene/L prototype. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 476–485, 2005.

[127] Yibei Ling, Jie Mi, and Xiaola Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Transactions on Computers*, 50(7):699–708, 2001.

[128] Benyuan Liu, Yang Guo, Jim Kurose, Don Towsley, and Weibo Gong. Fluid simulation of large scale networks: Issues and tradeoffs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2136–2142, 1999.

[129] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D K. Panda. Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 58, 2003.

[130] Yudan Liu, Raja Nassar, Chokchai (Box) Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–9, April 2008.

[131] Junsheng Long, W. Kent Fuchs, and Jacob A. Abraham. Compiler-assisted static checkpoint insertion. In *The Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, pages 58–65, Boston, Massachusetts, July 1992.

[132] Qingda Lu, Jiesheng Wu, Dhabaleswar Panda, and P. Sadayappan. Applying MPI derived datatypes to the NAS benchmarks: A case study. In *Proceedings of the 2004 International Conference on Parallel Processing Workshops*, pages 538–545, 2004.

[133] James E. Lumpp, Jr., Thomas L. Casavant, Julie A. Gannon, Kyle J. Williams, and Mark S. Andersland. Analysis of communication patterns for modeling message passing programs. In *Proceedings of the International Workshop on Principles of Parallel Computing*, pages 249–258, Lacanau, France, November 1993.

[134] Diego R. Martinez, Vicente Blanco, Marcos Boullón, José carlos Cabaleiro, and Tomás F. Pena. Analytical performance models of parallel programs in clusters. In *Parallel Computing: Architectures, Algorithms, and Applications*, volume 38 of *NIC*, pages 99–106, September 2007.

[135] Antonio S. Martins and R.A.L. Gonçalves. Implementing and evaluating automatic checkpointing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[136] Friedemann Mattern. Virtual time and global states of distributed

systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, October 1988.

[137] Sjouke Mauw and Michel A. Reniers. High-level message sequence charts. In *Time for Testing—SDL, MSC and Trends, Proceedings of the Eighth SDL Forum*, Evry, France, 1997.

[138] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, Sept 2005.

[139] Portolan Michele and Leveugle Régis. Effective checkpoint and rollback using hardware/OS collaboration. In *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 370–378, 2007.

[140] Ronald G. Minnich, Matthew J. Sottile, Sung-Eun Choi, Erik Hendriks, and Jim McKie. Right-weight kernels: An off-the-shelf alternative to custom light-weight kernels. *ACM SIGOPS Operating Systems Review*, 40(2):22–28, 2006.

[141] Adam Moody. The scalable checkpoint/restart (SCR) library: Approaching file I/O bandwidth of 1 TB/s. Talk given at the 2009 Fault Tolerance for Extreme-Scale Computing Workshop, March 2009.

[142] José E. Moreira, Valentina Salapura, George Almasi, Charles Archer, Ralph Bellofatto, Peter Bergner, Randy Bickford, Mathias Blumrich, José R. Brunheroto, Arthur A. Bright, Michael Brutman, José G. Casta nos, Dong Chen, Paul Coteus, Paul Crumley, Sam Ellis, Thomas Engelsiepen, Alan Gara, Mark Giampapa, Tom Gooding, Shawn Hall, Ruud A. Haring, Roger Haskin, Philip Heidelberger, Dirk Hoenicke, Todd Inglett, Gerrard V. Kopcsay, Derek Lieber, David Limpert, Pat McCarthy, Mark Megerian, Mike Mundy, Martin Ohmacht, Jeff Parker, Rick A. Rand, Don Reed, Ramendra Sahoo, Alda Sanomiya, Richard Shok, Brian Smith, Gordon G. Stewart, Todd Takken, Pavlos Vranas, Brian Wallenfelt, Michael Blocksome, and Joe Ratterman. The Blue Gene/L supercomputer: A hardware and software story. *International Journal of Parallel Programming*, 35(3):181–206, 2007.

[143] DataDirect Networks. DataDirect networks case study: Terascale simulation facility employs S2A storage system to deliver a scalable, high performance, high capacity, open-system solution for high speed scientific simulation, data analysis, and visualization. Technical report, 2007.

[144] DataDirect Networks. Best practices for architecting a Lustre^{TM}-based storage environment. Technical report, 2008.

[145] Keigo Nitadori, Junichiro Makino, and George Abe. High-performance small-scale simulation of star clusters evolution on Cray XD1. *ArXiv Astrophysics e-prints*, June 2006.

[146] Alison N. Norman, Sung-Eun Choi, and Calvin Lin. Compiler-generated staggered checkpointing. In *7th ACM Workshop on Languages, Compilers, and Runtime Support for Scalable Systems*, pages 1–8, Houston, Texas, October 2004.

[147] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, 2007.

[148] Leonid Oliker, Andrew Canning, Jonathan Carter, Costin Iancu, Michael Lijewski, Shoaib Kamil, John Shalf, Hongzhang Shan, Erich Strohmaier, Stephane Ethier, and Tom Goodale. Scientific application performance on candidate petascale platforms. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, page 69, 2007.

[149] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[150] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for Blue Gene/L systems. In *Proceedings of the 18th International Parallel and Distributed Systems Processing Symposium*, April 2004.

[151] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 14–23, 2006.

[152] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing theory. In *Proceedings of the Parallel and Distributed Processing Symposium*, April 2006.

[153] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, Laboratory for Computer Science, University of Bologna, Italy, January 1993.

[154] Kalyan S. Perumalla, Richard M. Fujimoto, Prashant J. Thakare, Santosh Pande, Homa Karimabadi, Yuri Omelchenko, and Jonathan Driscoll. Performance prediction of large-scale parallel discrete event models of physical systems. In *Proceedings of the 37th Conference on Winter Simulation*, pages 356–364, 2005.

[155] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 209b, April 2004.

[156] Fabrizio Petrini, Jarek Nieplocha, and Vinod Tipparaju. SFT: Scalable

fault tolerance. *ACM SIGOPS Operating Systems Review*, 40(2):55–62, 2006.

[157] Wayne Pfeiffer and Nicholas J. Wright. Modeling and predicting application performance on parallel computers using HPC challenge benchmarks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–12,14–18, April 2008.

[158] Ian R. Philp. Software failures and the road to a petaflop machine. In *Proceedings of the 1st Workshop on High Performance Computing Reliability Issues*, February 2005.

[159] James S. Plank. *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton University, Princeton, NJ, USA, 1993.

[160] James S. Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclustion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*, Winter 1995.

[161] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointingsystems. Technical report, University of Tennessee, Knoxville, TN, USA, 1996.

[162] James S. Plank and Michael G. Thomason. The average availability of parallel checkpointing systems and its importance in selecting run-

time parameters. In *29th International Symposium on Fault-Tolerant Computing*, pages 250–257, June 1999.

[163] James S. Plank and Michael G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, November 2001.

[164] Sundeep Prakash and Rajive L. Bagrodia. MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Proceedings of the 30th conference on Winter Simulation*, pages 467–474, 1998.

[165] Sundeep Prakash, Ewa Deelman, and Rajive Bagrodia. Asynchronous parallel simulation of parallel programs. *IEEE Transactions on Software Engineering*, 26(5):385–400, 2000.

[166] Jim Pruyne and Miron Livny. Managing checkpoints for parallel programs. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 140–154, London, UK, 1996. Springer-Verlag.

[167] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[168] Dhananjai M. Rao and Philip A. Wilsey. An ultra-large-scale simulation framework. *Journal of Parallel and Distributed Computing*, 62(11):1670–1693, 2002.

[169] Felix Rauch. Comprehensive throughput evaluation of LANs in clusters of PCs with SwitchBench—or how to bring your switch to its knees. In *Proceedings of the IEEE International Workload Characterization Symposium*, pages 155–162, October 2005.

[170] Felix Rauch, Christian Kurmann, and Thomas M. Stricker. Optimizing the distribution of large data sets in theory and practice. *Concurrency and Computation: Practice and Experience*, 14(3):165–181, March 2002.

[171] Gabriele Rennie. Supercomputers in support of science. LLNL-WEB-409815 Rev. 1, https://www-pls.llnl.gov/?url=science_and_technology-materials-thunder, November 2007.

[172] Wolfram Research. Wolfram Mathematica 7. www.wolfram.com, 2009.

[173] Gabrial Rodríguez, Patricia María J. Martín González, and Juan Touriño. Safe point detection for distributed checkpoint and restart. 2007.

[174] Gabriel Rodríguez, María J. Martín, Patricia González, and Juan Touriño. Controller/precompiler for portable checkpointing. *Transactions on Information and Systems*, E89-D(2):408–417, 2006.

[175] Rob Ross, Jose Moreira, Kim Cupps, and Wayne Pfeiffer. Parallel I/O on the IBM Blue Gene/L system. Blue Gene/L Consortium Quarterly Newsletter, Argonne National Laboratory, 1st Quarter 2006.

[176] Joseph F. Ruscio, Michael A. Heffner, and Srinidhi Varadarajan. DejaVu: Transparent user-level checkpointing, migration and recovery for

distributed systems. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 158, 2006.

[177] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. Emulating PetaFLOPS machines and Blue Gene. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, page 195, 2001.

[178] Ali G. Saidi, Nathan L. Binkert, Lisa R. Hsu, and Steven K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *Proceedings of the Workshop on Interaction between Operating System and Computer Architecture*, pages 33–38, 2005.

[179] Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[180] José Carlos Sancho, Fabrizio Petrini, Greg Johnson, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[181] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, and Andrew Lumsdaine. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[182] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Languages and Compilers for Parallel Computing*, pages 94–113, August 1997.

[183] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 633–655, Portland, OR, August 1993.

[184] Erik Schnetter. Between application and system: Fault tolerance mechanisms for the Cactus software framework. Talk given at the 2009 Fault Tolerance for Extreme-Scale Computing Workshop, March 2009.

[185] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *International Conference on Dependable Systems and Networks*, pages 249–258, June 2006.

[186] Karl W. Schulz. TACC overview & Lustre experiences. Presentation, Lustre User Group Meeting, April 2007.

[187] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[188] Ilya Sharapov, Robert Kroeger, Guy Delamarter, Razvan Cheveresan, and Matthew Ramsay. A case study in top-down performance estimation for a large-scale parallel application. In *Proceedings of the Eleventh*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 81–89, 2006.

[189] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[190] Luis M. Silva, João G. Silva, Simon Chapple, and Lyndon Clarke. Portable checkpointing and recovery. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, page 188, 1995.

[191] Luìs Moura E Silva and Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 2, chapter Parallel Programming Models and Paradigms, pages 4–27. Printice Hall, New Jersey, 1999.

[192] Warren Smith. Prediction services for distributed computing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[193] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, 2002.

[194] Allan Snavely, Xiaofeng Gao, Cynthia Lee, Laura Carrington, Nicole Wolter, Jesus Labarta, Jusit Gimenez, and Philip Jones. Performance modeling of HPC applications. In *Proceedings of the International Conference on Parallel Computing*, Dresden, October 2003.

[195] Allan Snavely, Nicole Wolter, and Laura Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of the IEEE International Workshop on Workload Characterization*, pages 149–156, December 2001.

[196] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[197] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: a scientific tool for modeling computational grids. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 53, 2000.

[198] Francois Spies, Julien Bourgeois, and Herve Guyennet. Performance prediction of parallel programs using ChronosMix environment. In Daniel M. Dubois, editor, *Fifth International Conference on Computing Anticipatory Systems*, pages 581–588. AIP, 2002.

[199] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for

the Pittsburgh Supercomputing Center Terascale Computing System. Technical report, Pittsburgh Supercomputing Center, November 2001. Presented at Supercomputing 2001.

[200] Volker Strumpen and Balkrishna Ramkumar. Portable checkpointing and recovery in heterogeneous environments. In *Proceesings of the 27th International Symposium on Fault Tolerant Computing*, 1996.

[201] Jaspal Subhlok, Shreenivasa Venkataramaiah, and Amitoj Singh. Characterizing NAS benchmark performance on shared heterogeneous networks. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 91, 2002.

[202] Ryutaro Susukita, Hisashige Ando, Mutsumi Aoyagi, Hiroaki Honda, Yuichi Inadomi, Koji Inoue, Shigeru Ishizuki, Yasunori Kimura, Hidemi Komatsu, Motoyoshi Kurokawa, Kazuaki J. Murakami, Hidetomo Shibamura, Shuji Yamamura, and Yunqing Yu. Performance prediction of large-scale parallell system and application using macro-level simulation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, 2008.

[203] Mellanox Technologies. Introduction to InfiniBand. Technical Report 2003WP, 2003.

[204] Texas Advanced Computing Center. *Lonestar User Guide*.

[205] Texas Advanced Computing Center. *Ranger User Guide*.

[206] Texas Advanced Computing Center. User survey. Fall 2009.

[207] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Stephen L. Scott, and Chokchai Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 303–311, September 2007.

[208] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, Inc., San Diego, California, 1993.

[209] Keith D. Underwood, Michael Levenhagen, and Arun Rodriguez. Simulating Red Storm: Challenges and successes in building a system simulation. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–10,26–30, March 2007.

[210] Nitin H. Vaidya. On staggered checkpointing. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, page 572, 1996.

[211] Nitin H. Vaidya. Staggered consistent checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):694–702, 1999.

[212] Arjan J.C. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):154–165, May 2003.

[213] Eric Van Hensbergen, Charles Forsyth, Jim McKie, and Ron Minnich. Holistic aggregate resource environment. *ACM SIGOPS Operating Systems Review*, 42(1):85–91, 2008.

[214] Ana Lucia Varbanescu, Henk Sips, and Arjan van Gemund. *Euro-Par 2006 Parallel Processing*, volume 4128/2006 of *Lecture Notes in Computer Science*, chapter PAM-SoC: A Toolchain for Predicting MPSoC Performance, pages 111–123. Springer Berlin/Heidelberg, November 2006.

[215] Ana Lucia Varbanescu, Henk J. Sips, and Arjan J.C. van Gemund. Semi-static performance prediction for MPSoC platforms. In *Proceedings of Compilers for Parallel Computers*, pages 1–15, January 2006.

[216] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *6th World Congresses of Structural and Multidisciplinary Optimization*, 2005.

[217] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems*, pages 240–250, 2002.

[218] John Paul Walters and Vipin Chaudhary. *High Performance Computing*, volume 4873/2007 of *Lecture Notes in Computer Science*, chapter

A Scalable Asynchronous Replication-Based Strategy for Fault Tolerant MPI Applications, pages 257–268. Springer Berlin / Heidelberg, 2007.

[219] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[220] Long Wang, Karthik Pattabiraman, Christopher Vick, and Alan Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 812–821, 2005.

[221] Panfeng Wang, Zhiyuan Wang, Yunfei Du, Xuejun Yang, and Haifang Zhou. Optimal placement of application-level checkpoints. In *10th IEEE International Conference on High Performance Computing and Communications*, pages 853–858, Sept 2008.

[222] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007.

[223] D. Brent Weatherly, David K. Lowenthal, Mario Nakazawa, and Franklin Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 5, 2003.

[224] Songjie Wei and Jelena Mirkovic. A realistic simulation of internet-scale events. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, page 28, 2006.

[225] Ben Wiedermann. Know your place: Selectively executing statements based on context. Technical Report TR-07-38, The University of Texas at Austin, Department of Computer Sciences, July 2007.

[226] Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad, and Laxmikant V. Kale. Performance prediction using simulation of large-scale interconnection networks in POSE. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 109–118, 2005.

[227] Anlu Yan and Wei-Bo Gong. Time-driven fluid simulation for high-speed networks. *IEEE Transactions on Information Theory*, 45(5):1588–1599, July 1999.

[228] Xuejun Yang, Panfeng Wang, Hongyi Fu, Yunfei Du, Zhiyuan Wang, and Jia Jia. Compiler-assisted application-level checkpointing for MPI programs. In *The 28th International Conference on Distributed Computing Systems*, pages 251–259, June 2008.

[229] Weikuan Yu, Jeffrey S. Vetter, and H. Sarp Oral. Performance characterization and optimization of parallel I/O on the cray XT. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008.

[230] Jidong Zhai, Wenguang Chen, and Weimin Zheng. PHANTOM: Predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[231] Yanyong Zhang, Mark S. Squillante, Anand Sivasubrmaniam, and Ramendra K. Sahoo. *Job Scheduling Strategies for Parallel Processing*, volume 3277/2005 of *Lecture Notes in Computer Science*, chapter Performance Implications of Failures in Large-Scale Cluster Scheduling, pages 233–252. Springer Berlin/Heidelberg, 2005.

[232] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level checkpoint and recovery for LAM/MPI. *SIGOPS Oper. Syst. Rev.*, 39(3):72–81, 2005.

[233] Gengbin Zheng, Gunavadhan Kakulapati, and Laxmikant V. Kalé. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *Procesedings of the 18th International Parallel and Distributed Processing Symposium*, page 78, April 2004.

[234] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. *International Journal of Parallel Programming*, 33(2):183–207, 2005.

[235] Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, and David Padua. Performance modeling and programming envornments for petaflops computers and the Blue Gene machine. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 197, April 2004.

Vita

Alison Nicholas Norman attended Oconee County High School, Watkinsville, Georgia. In 1996 she entered Georgia Institute of Technology in Atlanta, Georgia, and she received the degree of Bachelor of Science in May, 2000. In December 2005, she received her Master of Science from the University of Texas at Austin.

Permanent address: **6215 Oliver Loving Trail, Austin, Texas**

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.