ALGORITHMIC APPROACHES FOR ENHANCING SPEEDUP, ENERGY
AND RESILIENCY MEASURES OF SPARSE SCIENTIFIC
COMPUTATIONS

A Dissertation in
Computer Science and Engineering
by
Manu Shantharam

© 2012 Manu Shantharam

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2012
The dissertation of Manu Shantharam was reviewed and approved* by the following:

Padma Raghavan  
Distinguished Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Mahmut Kandemir  
Professor of Computer Science and Engineering

Mary Jane Irwin  
Robert E. Noll Professor, Evan Pugh Professor

Jai Li  
Professor of Statistics

Lee Coraor  
Associate Professor of Computer Science and Engineering  
Graduate Officer

*Signatures are on file in the Graduate School.
Abstract

High performance computing systems have increasingly complex node and network architectures including non-uniform memory subsystems, heterogeneous processors and hierarchical interconnects. The performance of scientific applications that run on such systems depends on several factors including memory access pattern, memory bandwidth, load balancing and resiliency. Consequently, optimizing the performance of scientific applications for high performance computing systems is challenging. We seek to address this challenge for applications involving sparse scientific computations because such computations form the basis for solving many large-scale models of physical phenomena. In this thesis, we seek to understand the interplay between sparse scientific applications and hardware, and develop algorithmic approaches to improve performance measures and resiliency for sparse scientific computations.

We organize the thesis into two parts. The first part concerns developing algorithmic approaches to enhance the performance of sparse scientific computations and has two main contributions: (i) a new sparse matrix representation and a corresponding sparse matrix vector multiplication (SpMV) algorithm that enhances performance of the SpMV operation and (ii) speedup-aware processor partitioning algorithms to manage sparse scientific workloads efficiently. The second part concerns analyzing the impact of transient errors on sparse scientific computing and developing a fault tolerant algorithm, and has two main results: (i) characterizing the impact of a single transient error on iterative methods and (ii) a new sparse checksum encoded algorithm-based fault tolerance technique for the preconditioned conjugate gradients method.

In Chapter 2, we focus on SpMV, which is at the heart of many scientific applications involving sparse linear system solution. We develop a new sparse matrix representation and a corresponding SpMV algorithm that exploits the dense substructures that are inherently present in many sparse matrices derived from partial differential equation models. We show that our SpMV algorithm reduces the total number of load operations and enhances locality in accesses to the vector, consequently, improving the SpMV performance on average by a third compared to the traditional compressed sparse row scheme on the Intel Nehalem processor.

In Chapter 3, we consider improving the performance sparse scientific workloads that are commonly executed on high performance computing systems. We observe many ap-
Applications in such workloads do not scale linearly with the number of cores, providing diminishing gains in execution time for larger numbers of cores. For a workload comprising multiple such applications, it is beneficial from system perspective to reduce system energy consumption and decrease workload completion time. We develop speedup-aware processor partitioning algorithms that exploit individual application scaling features and optimize processor allocations per application. Our results indicate that the speedup-aware algorithms can reduce workload completion time by as much as half and decrease the total energy consumption of the workload by more than half on 128 cores of the Intel Nehalem processor, compared to executing each application within the workload on all 128 cores one after the other.

In Chapter 4, we analyze the impact of silent data corruption due to a single transient error on sparse scientific computations, in particular, on sparse linear system solution. Transient errors result in bit flips in memory and errors in logic circuit output, leaving the computing system state corrupt. We provide a theoretical analysis of the impact of a single transient error on sparse linear system solution. Our analysis indicates that a single transient error during an SpMV operation can corrupt the entire resultant vector in a relatively short sequence of SpMV operations. Furthermore, our evaluations show that execution time of sparse linear system solution could increase by a factor as high as 200 or more, in the event of a transient error.

In Chapter 5, we focus on enhancing the resiliency of sparse linear system solution. We develop a new sparse checksum encoded algorithm-based fault tolerance technique for the preconditioned conjugate gradients (PCG) method, a widely used method for sparse linear system solution. We prove that our technique detects a single error in all the key operations within the method, including SpMV, vector operations and the application of a preconditioner through sparse triangular solution, when the linear system has a coefficient matrix that is symmetric positive definite and strictly diagonally dominant. Additionally, the overheads of using our fault tolerance technique are low, eleven percent on average in the event of no error and three percent in the event of a single error within PCG, when compared to having no fault tolerance for the PCG method.

Our thesis demonstrates an opportunity space to optimize sparse scientific applications for performance and resiliency on HPC systems. We develop new algorithmic formulations that enhance performance and resiliency of sparse scientific computations. Looking forward, we expect the opportunity space to grow with evolving applications and hardware, and much work remains to be done in optimizing scientific applications for various performance measures.
# Table of Contents

List of Figures vii

List of Tables x

Acknowledgments xi

**Chapter 1 Introduction**

1.1 Background and Motivation ........................................ 3
1.2 Contributions ...................................................... 6

**Chapter 2 Enhancing Performance Measures: Sparse Matrix Vector Multiplication**

2.1 Introduction ........................................................... 9
2.2 Background and Related Work ....................................... 11
2.3 Sparsity-Exploiting Effectively Dense Sparse Matrix Vector Multiplication (SpEED-SMV) ........................................ 14
2.3.1 Overheads of SpEED-SMV ......................................... 18
2.4 Experimental Evaluation ........................................... 19
2.4.1 Experimental Setup ................................................ 19
2.4.2 Evaluation and Discussion ....................................... 23
2.5 Conclusions and Future Work ....................................... 29

**Chapter 3 Enhancing Performance Measures: Sparse Scientific Workloads** 32

3.1 Introduction ........................................................... 32
3.2 Related Work .......................................................... 34
3.3 Speedup Aware Processor Partitioning ............................... 36
3.3.1 Construction of an optimal schedule for two-way processor partitioning 42
3.3.2 Heuristics ............................................................ 45
3.4 Experimental Evaluation ........................................... 46
3.4.1 Experimental Setup ................................................ 47
3.4.2 Node-Level Evaluation .......................................... 48
3.4.3 Large-Scale Evaluation .......................................... 50
3.5 Conclusion .............................................................. 52
List of Figures

1.1 Performance comparison of sparse (SpMV) and dense (DGEMM) benchmarks performing similar number of floating point operations. 4
1.2 Observed energy consumption for 4 applications for various schedules. High variation is observed. 5
1.3 Maximum observed performance degradation in PCG, in terms of number of iterations to convergence, due to a single transient error. 5

2.1 (a) CSR and (b) BCSR storage formats. 12
2.2 SpEED sparse matrix vector multiplication. 15
2.3 An example of reduced CSR representation. 15
2.4 SpEED sparse matrix vector multiplication with RCM ordering. 18
2.5 Illustration of the Intel Nehalem (a), and the AMD Shanghai (b) architecture. 21
2.6 Performance improvement of SMV optimization techniques relative to the base case on 8-cores of the Intel Nehalem processor for the test (a) Suite 1 matrices and (b) Suite 2 matrices. 23
2.7 Reduction in the energy consumption of the SMV optimization techniques relative to the base case on 8-cores of the Intel Nehalem processor for the test (a) Suite 1 matrices and (b) Suite 2 matrices. 24
2.8 (a) Performance improvement from using dense substructures in SpEED-SMV on 8-cores of the Intel Nehalem processor, (b) Performance comparison between R×C SMV using RCM and natural orderings of matrices on 8-cores of the Intel Nehalem processor. 27
2.9 Performance of SpEED-SMV with respect to the R×C-SMV on (a) Intel Nehalem, and (b) AMD Shanghai, two socket quad-core processors. 28
2.10 (a) fill-in ratio, (b) decrease in load operations, and (c) effective density on Intel Nehalem and AMD Shanghai processors. 30

3.1 Observed energy consumption for 4 applications for various schedules. High variation is observed. 33
3.2 Application speedup profiles for 1-8 cores. 36
3.3 (a) Speedup-profiles of two applications showing deviation from the linear speedup. Note that the gap between the ideal and actual speedup increases with core count. (b) Makespan of two applications with different schedules. The black line is the makespan when each application is run on all cores individually \((T(1,8) + T(2,8))\).

3.4 Approximate and actual speedup plots.

3.5 An example of the optimal schedule for a two-way processor partitioning scheme for set a 6 applications on 8 cores. (a) a set of best pairwise schedule for each pair of applications, (b) graph formulation of the set from (a) and (c) minimum weight perfect matching co-schedule. Note that \(A(i,j)\) represents Application ‘i’ running with ‘j’ threads.

3.6 Percentage reduction in the makespan for three different schemes with respect to the makespan of the base case for workloads of (a) 4 applications and (b) 6 applications. (c) and (d) show corresponding energy benefits.

3.7 Comparison between the SAPP-O and SAPP-G schemes.

3.8 Benefits of our schemes for large-scale systems. (a) and (b) show reduction in makespan for 10 different workloads of 4 applications each. (c) and (d) show the energy benefits of using of our schemes on the same set of workloads.

4.1 One dimensional heat flow problem.

4.2 Sparse matrix \(A\), vector \(y\), and the sequence of SpMV operations on \(A\) and \(y\).

4.3 Intermediate tree representations (0-3 iterations).

4.4 Final BFS tree representation.

4.5 Relative errors in the solution of a model heat flow problem using an explicit method.

4.6 Correlation between row 2-norm values of a matrix and iteration count when a soft error is inserted in the corresponding position of the vector.

4.7 Relative errors in (a) Krylov subspace, (b) A-norm error during CG iterations. Note that red, black and blue colors indicate that a soft error was inserted in a vector position corresponding to the highest, median and lowest row norm positions, respectively.

4.8 Sensitivity to error insertion position in SOR and PCG.

4.9 Performance impact of soft errors on PCG: (a) maximum performance degradation, and (b) average performance degradation.

4.10 Performance tradeoffs of PCG and SOR in the presence of soft errors. Average and maximum error corresponds to the average increase and highest iteration count observed in PCG and SOR on soft error insertion.

5.1 The C-ABFT technique.

5.2 C-ABFT technique and its overheads for (a) dense matrix-matrix multiplication, (b) sparse matrix-vector multiplication.

5.3 An illustration of the ABFT technique when there is an error in the metadata (a) column index, (b) row index.
5.4 S-ABFT technique and its corresponding overhead for an SpMV operation. 84
5.5 An example triangular solve system. The graph on the right represents the dependence among the solution elements. 86
5.6 The effect of the error occurrence position on the solution vector. The grey elements are corrupt. (a) FS = \{y_3, y_4, y_5, y_6\}, (b) FS = \{y_5\}, (c) FS = \{y_2, y_6, y_7\}. 87
5.7 S-ABFT technique for a sparse triangular solve. 87
5.8 Performance degradation due to overheads of ABFT, in terms of the percentage increase in execution time to convergence. 92
5.9 Percentage increase in execution times of PCG due to a single error in sparse triangular solution. The increase in execution time is from a corresponding increase in the number of iterations to convergence. 94
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Description of important variables used in our algorithm.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>A suite of smaller test matrices (Suite 1).</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>A suite of larger test matrices (Suite 2).</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Distribution of supernode sizes: Suite 1.</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Distribution of supernode sizes: Suite 2.</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>SMV optimization techniques.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>A list of variables used in Algorithm 3.1 and their descriptions.</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Different scheduling techniques and their descriptions.</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>The UFL benchmark matrices with their dimension (N) and number of nonzeroes (NNZ).</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>The UFL benchmark matrices with their dimension (N) and number of non-zeroes (NNZ).</td>
<td>93</td>
</tr>
</tbody>
</table>
Acknowledgments

I am immensely grateful to my adviser and mentor, Dr. Padma Raghavan, for her guidance and encouragement throughout my stay at Penn State. I thank my committee members, Dr. Mary Jane Irwin, Dr. Mahmut Kandemir and Dr. Jia Li, for their insightful commentary on my work. I also want to thank my wife and friend Sowmya, my parents, and my brother Meru, for their constant support and encouragement that has kept me going.
Chapter 1

Introduction

Today’s high performance computing (HPC) systems have complex node and network architectures including non-uniform memory subsystems, heterogeneous processors, and hierarchical interconnects [1, 2, 3, 4, 5]. Many scientific applications that run on such systems have irregular memory accesses, varying amounts of parallelism and different bandwidth requirements. As a consequence of complex hardware and scientific application features, optimizing the performance of scientific applications for HPC systems is challenging. We seek to address this challenge by first understanding the interactions between application and hardware in the domain of sparse scientific computations and then utilizing the insights gained from these interactions.

In this thesis, we focus on sparse scientific applications that involve solving partial differential equation (PDE) based systems, such as those found in heat diffusion, computational fluid dynamics and structural mechanics [6]. The computational and storage costs of these applications scale linearly with problem size, which makes them an attractive choice to solve large-scale models of physical phenomena. We seek to enhance these applications for a range of performance measures and resilience, which we organize into two parts. The first concerns developing algorithmic approaches to adapt sparse scientific computations to enhance speedup, throughput, and energy measures [7]. The second concerns analyzing the impact of transient errors on the resiliency of sparse linear system solution, and developing a new fault tolerance technique to improve its resiliency [8, 9].

In the first part of this thesis, we focus on sparse scientific computations resulting from (i) sparse matrix vector operation (in Chapter 2) and (ii) sparse scientific workload (in Chapter 3). We develop algorithmic approaches to enhance the performance of sparse scientific computations defined by metrics such as execution time, throughput and energy costs.
In Chapter 2, we consider the SpMV operation, which is at the heart of many scientific applications involving sparse linear system solution. We develop a new compressed sparse matrix representation that exploits the dense substructures that are inherently present in many sparse matrices derived from PDE models [7]. We adapt SpMV to utilize our sparse matrix representation. We show that our adaptation not only reduces the total number of load operations, but also enhances locality in accesses to the vector, consequently, improving the execution time of SpMV on multicores. We evaluate our scheme on Intel Nehalem and AMD Shanghai processors. We observe that for larger matrices on the Intel Nehalem processor, our method improves performance on average by 37.35% compared to the traditional compressed sparse row scheme (a blocked compressed form improves performance on average by 30.27%). Benefits of our new format are similar for the AMD processor.

In Chapter 3, we seek to improve the performance of workloads comprising sparse scientific applications. We observe many sparse applications have sub-linear fixed-problem-speedup on increasing processor counts. For a workload comprising such applications, we show scheduling opportunities that could potentially reduce workload completion time and decrease system energy consumption. Towards finding such a schedule, we develop speedup-aware processor assignment algorithms that exploit individual application scaling features and optimize processor allocations per application. The schedules predicted by our speedup-aware algorithms reduce workload completion time by as much as 31.5% on an 8-core node of the Intel Nehalem processors, and by as much as 49.3% on 16 such nodes when compared to executing each application within the workload on all the available cores one after the other (also known as the first-come-first-serve algorithm). Additionally, on the 16 node configuration, our algorithms decrease the total energy consumption by as much as 57.0% compared to the first-come-first-serve algorithm.

In the second part, we analyze the impact of silent data corruption due to a single transient error\(^1\) on the resiliency of sparse linear system solution in Chapter 4, and present an algorithm for fault tolerant sparse linear system solution in Chapter 5.

In Chapter 4, we analyze the impact of transient errors on the resiliency of sparse linear system solution. In particular, we consider silent data corruption that is hard to detect using traditional fault tolerance techniques. Our theoretical analysis indicates that a single transient error in a component of the solution vector can corrupt the entire resultant vector in a relatively short sequence of SpMV operations [8]. Additionally, our analysis shows that the magnitude of the error grows non-linearly as \((\|A_{s+}\|_2)^k\), where \(k\) is the

\(^1\) also referred to as soft errors in this thesis
number of SpMV operations and $||A_i||_2$ is the 2-norm of the i-th row of the matrix, $A$. Our evaluations indicate that the performance degradation of sparse linear system solution through a method such as preconditioned conjugate gradients (PCG) could be as high as a factor of 200 or more in the event of a transient error.

In Chapter 5, we develop a new sparse checksum encoded algorithm-based fault tolerance technique for the PCG method, to enhance the resiliency of sparse linear system solution. We prove that our technique detects a single error in all the key operations within the method, including SpMV, vector operations and the application of a preconditioner through sparse triangular solution. Therefore, our fault tolerant PCG can potentially reduce the execution time of PCG in the presence of a single error by a factor as high as 200 or more. Additionally, we show that the overhead of our technique is proportional to the cost of a few $O(n)$ vector operations, a value that is relatively low compared to the total cost of a PCG iteration with an SpMV operation and two triangular solutions. Our experimental evaluations indicate that the overheads of using our fault tolerance are 11.3% on average in the event of no error, and 3.2% in the event of a single error within PCG when compared to having no fault tolerance for the PCG method.

In this thesis, we highlight the challenges in tuning sparse scientific applications for current and future HPC systems, and seek to understand the interactions between application and hardware. We develop several novel algorithmic approaches to improve the performance and resilience of sparse scientific computations on HPC systems.

1.1 Background and Motivation

In this section, we set the context for our research, and motivate our thesis.

The quest for scientific discovery through advanced computing has led to a rapid growth of large-scale scientific applications and complex computer hardware. A majority of these applications involve sparse computations, primarily because the computational and storage costs of such computations scale linearly with problem size. Thus, sparse computations facilitate the solution of large-scale models of physical phenomena which would otherwise be impractical with dense computations that exhibit quadratic to cubic growth of computational costs with problem size.

However, unlike dense computations that operate close to the theoretical peak performance of the hardware in terms of floating point operations per second (FLOP/s), the sustained performance on sparse scientific workloads is a fraction of the theoretical peak. For example, Figure 1.1 illustrates the disparity in performance of the dense and
sparse benchmarks on 8 cores of the Intel Nehalem processor. We observe that the performance of the SpMV operation, in GFLOP/s, is less than 7.0% of the performance of the double-precision General Matrix Multiply (DGEMM) operation. The work by Oliker et al. [11] also demonstrates this disparity in performance on a set of large-scale scientific applications on a variety of HPC systems. The results from their study indicate a wide range of sustained application performance, from as low as 1.0% to as high as 65.0% of peak performance, across different applications and HPC systems. Such a disparity in the actual and peak performance can be attributed many factors including irregular memory accesses, insufficient bandwidth, scalability, and load balancing of sparse scientific applications.

Another consequence of the above mentioned factors, in particular, application scalability, is that sparse scientific applications tend to under-utilize the electrical power hungry HPC hardware leading to an increase in the system energy consumption. For workloads comprising such applications that execute on HPC systems, the issue with respect to system energy consumption aggravates even further. For example, consider Figure 1.2 that illustrates the normalized system energy consumption for running a workload comprising four scientific applications on a cluster of 128 cores. Each point in the figure represents a different schedule for the workload, where a schedule is defined as an order in which the applications get executed within the workload. We observe that in the worst case, the energy consumption of a schedule can be as high as a factor of 5.8 compared to the schedule with lowest energy consumption.

In addition to scalability and energy concerns, long running sparse scientific applications...
that simulate PDE-based models face resiliency issues as we move toward peta-to-exascale systems. Such applications are expected to encounter multiple hardware faults within a day [12], with faults ranging from fail-stop failures [13] to silent errors [8, 14]. Such errors can have devastating impact on the execution time and accuracy of the solution of scientific simulations [8]. Figure 1.3 illustrates the maximum observed performance degradation due to a single silent error within sparse linear system solution through the PCG method. Such errors can potentially increase the execution time of PCG by a factor of 200 or more.
We expect to see an interesting spectrum of vastly different impacts of silent errors if we consider a broader array of sparse computational methods.

The work in this thesis is in the area of sparse scientific computations with a particular focus on addressing the concerns discussed in this section, namely, execution time (performance in flops), scalability, energy consumption, and resilience of such computations. In the following section, we provide a brief overview of our main contributions.

1.2 Contributions

In this thesis, we present several novel algorithmic approaches to adapt sparse scientific computing to HPC systems that aim to improve a range of application performance measures. We present our main contributions in this section. Within each chapter, we briefly discuss background and related work, and present our research.

Part I: Enhancing Performance Measures

The first part of this thesis concerns developing algorithmic approaches to enhance the performance of sparse scientific computations defined by metrics such as execution time, throughput and energy costs.

Sparse scientific computations include computations arising from sparse scientific operations such as sparse matrix vector multiplication (SpMV), scientific applications that comprise multiple sparse scientific operations, and scientific workloads that consist of multiple such applications. In this part, we focus on improving the performance of sparse scientific computations resulting from (i) SpMV, which is at the heart of many scientific applications involving sparse linear system solution and (ii) sparse scientific workload, that is commonly executed on high performance computing systems.

- **Sparse Matrix Vector Multiplication.** The execution time of many scientific computing applications is dominated by the time spent in performing sparse matrix vector multiplication (SpMV, \( y \leftarrow A \cdot x \)). In Chapter 2, we consider improving the performance of SpMV on multicores by exploiting the dense substructures that are inherently present in many sparse matrices derived from partial differential equation models. First, we identify indistinguishable vertices, i.e., vertices with the same adjacency structure, in a graph representation of the sparse matrix (\( A \)) and group them into a supernode. Next, we identify effectively dense blocks within the matrix by grouping rows and columns in each supernode. Finally, by using a suitable data structure for this representation of the matrix, we reduce the number of load operations during SpMV while exactly preserving the original sparsity structure of \( A \).
current multicore processors, our scheme improves the execution time of SpMV by as much as 88.07% compared to the traditional SpMV implementation.

- **Sparse Scientific Workloads.** Today’s high performance computing (HPC) systems have immense parallel processing capacity, yet many high performance applications designed for such systems suffer from Amdahl’s Law. As a result, we observe sub-linear fixed-problem-speedup on increasing processor counts. For a workload comprising multiple such applications, it is desirable to reduce workload completion time (i.e. makespan). This will result in both increased performance and decreased energy consumption. In Chapter 3, we propose speedup-aware processor partitioning (SAPP) schemes to select application co-schedules that reduce system energy and improve workload throughput. These methods reduce total workload time by exploiting individual application scaling features and optimizing processor allocations per application. Our results indicate that SAPP schemes predict schedules that reduce workload completion time by as much as 49.3% and decrease the total energy consumption by as much as 57.0% on 128 cores configuration of the Intel Nehalem processors, compared to first come first serve (FCFS) schedules. Further, our SAPP schemes reduce the average waiting time across all workloads by 89.6% on 128 cores compared to the FCFS schedules.

### Part II: Enhancing Resilience

In the second part, we analyze the challenges posed by transient errors for sparse scientific computations. A transient error can be caused by cosmic radiation or voltage fluctuation and results in bit flips in memory and errors in logic circuit output, leaving the computing system state corrupt. The current high performance computing systems are increasingly susceptible to transient errors as the underlying hardware technology for such systems uses small feature sizes. Consequently, sparse scientific computations that comprise the majority of long running workloads on large scale high performance computing systems are at a greater risk of being hit by a transient error.

In this part, we analyze the impact of silent data corruption due to a single transient error on the resiliency of sparse linear system solution, and enhance the resiliency of sparse linear system solution through an algorithm based fault tolerance technique.

- **Characterizing the Impact of Transient Errors on Iterative Methods.** In Chapter 4, we characterize the challenges posed by transient or soft errors for large scale applications representative of workloads on supercomputing systems. Such applications are typically based on the computational solution of partial differential
equation models using either explicit or implicit methods. In both cases, the execution time of such applications is typically dominated by the time spent in their underlying sparse matrix vector multiplication operation (SpMV, \( t \leftarrow A \cdot y \)). We provide a theoretical analysis of the impact of a single soft error through its propagation by a sequence of sparse matrix vector multiplication operations. Our results indicate that explicit schemes will suffer from transient error induced numerical instabilities, thus exacerbating intrinsic stability issues for such methods, that impose constraints on relative time and space step sizes. For implicit schemes, linear solver performance through widely the used PCG scheme degrades by a factor as high as 200x, whereas, a stationary scheme such as SOR is inherently soft error resilient.

- Algorithm Based Fault Tolerance for Sparse Linear System Solution. In Chapter 5, we develop a new sparse checksum encoded algorithm-based fault tolerant PCG, S-ABFT-PCG. Our checksum based approach can be applied to all the key operations in PCG, including sparse matrix-vector multiplication (SpMV), vector operations and the application of a preconditioner through sparse triangular solution. We prove that our approach detects a single error in the matrix and vector elements and in the metadata representing the sparse matrix row or column indices, when the linear system has a coefficient matrix that is symmetric positive definite and strictly diagonally dominant. Our experimental results indicate that in the event of no errors, compared to a PCG with no fault tolerance, the overheads of S-ABFT-PCG are 11.3%. Furthermore, in the event of a single error, the overheads of S-ABFT-PCG are 3.2% on average.

In summary, this thesis will propose and evaluate several algorithmic approaches towards adapting sparse scientific computations onto HPC systems. We will characterize the interactions between applications and hardware, and use hints from these characterizations to develop new algorithms to enhance speedup, throughput, energy consumption and resiliency measures for sparse scientific computations.
Chapter 2

Enhancing Performance Measures: Sparse Matrix Vector Multiplication

2.1 Introduction

The performance of many scientific simulations of partial differential equation (PDE) models depends on execution rates that can be achieved in their underlying sparse matrix vector multiplication (SMV) kernel, which multiplies a sparse matrix, $A$, with a vector, $x$, and stores the result in another vector, $y$. The performance of SMV on modern multicores can be enhanced by exploiting both matrix and hardware properties. SMV performance improvement techniques proposed earlier concern: (i) schemes to enhance locality in accesses to $x$ for improved cache performance using vertex reordering schemes [15, 16, 17], (ii) schemes to reduce overheads incurred due to loading indices of $A$, including, matrix blocking [18, 19, 20] and compression techniques [21, 22, 23, 24, 25], and, (iii) data restructuring techniques to exploit specific hardware attributes, such as, prefetchers [26].

In this chapter, we propose an SMV kernel that seeks higher performance by exploiting effectively dense substructures that often occur within sparse matrices from PDE models. This chapter has previously appeared in proceedings of the International Journal of High Performance Computing Applications [7].

Sparse matrices arising from many PDE-based applications, for example, computational fluid dynamics (CFD) and structural mechanics, typically contain subsets of columns and rows that have identical sparsity patterns. This property is revealed in the graph rep-
presentation of the matrix as indistinguishable vertices, i.e., vertices that have the same adjacency structure [27, 28]. We seek to harness this property through efficient identification of indistinguishable vertices toward an SMV kernel with reduced indexing overheads that maintains the original sparsity structure. In contrast to our approach, earlier schemes for SMV optimizations that seek to reduce indexing overheads often incur additional fill-in, i.e., original zeroes that treated as nonzeros [18, 19, 20]. We thus offer a new option for effectively exploiting tradeoffs between decreases in the number of loads and increases in the number of nonzeros that can potentially also accommodate recent data restructuring schemes to exploit hardware attributes [26].

In this chapter, we propose a Sparsity-Exploiting Effectively Dense SMV (SpEED-SMV) scheme to enhance application performance on multicore architectures. First, our scheme identifies indistinguishable vertices in a graph representation of \( A \), which are grouped into a supernode; such supernodes represent effectively dense substructures in \( A \). Next, we represent \( A \) with a modified data structure that captures the effectively dense substructures and implement a corresponding SMV kernel that operates on this representation. Additionally, we can use a supernodal representation of the graph of \( A \) to introduce reorderings that can enhance locality in accesses to \( x \). We thus formulate an SMV kernel that seeks to improve performance by reducing the indexing overheads and promoting reuse in \( x \).

We provide an extensive comparative evaluation of the performance of SpEED-SMV and blocked compressed sparse row (BCSR) SMV, on the Intel Nehalem and AMD Shanghai processors. Our tests on an 8-core configuration of Intel Nehalem processors indicate that our scheme improves the performance of SMV by as much as 59.35% and 50.32% compared to traditional compressed sparse row and BCSR schemes, respectively. On a similar configuration of the AMD Shanghai processors, our scheme improves the performance of SMV by as much as 88.07% and 53.04% compared to traditional compressed sparse row and BCSR schemes. However, our results show that there is no single SMV optimization method that consistently performs well on all the matrices. In an effort to gain consistent higher performance across all matrices, we propose effective density (\( \eta \)) as an architecture independent measure to characterize the relative merits of SMV optimization schemes that consider tradeoffs between reduced loads and increased fill-in. Our tests indicate that \( \eta \) can be a useful measure in differentiating among the optimization alternatives. Additionally, by correlating \( \eta \) with observed relative performances of these methods on multicores, we can potentially extract a classification scheme that can accurately predict the optimization scheme which will lead to best performance. We thus add to the set of schemes that can
yield high performance through optimizations that match matrix and hardware properties within an auto-tuned SMV framework such as OSKI [29].

The rest of this chapter is organized as follows. Section 4.3 presents related work addressing SMV performance and basic terminologies and methods required for understanding SMV performance. Section 2.3 presents the Sparsity-Exploiting Effectively Dense SMV scheme. Section 2.4 presents a detailed discussion of our experimental setup and results. We conclude our discussion in Section 5.5 with a summary of our findings.

2.2 Background and Related Work

In this section, we define basic terminology and discuss earlier related work on improving the SMV performance. We first discuss an SMV algorithm with CSR storage format [30] before proceeding with overviews of techniques related to blocking, ordering and other optimizations to improve SMV performance.

**Compressed Sparse Row (CSR) storage.** An important aspect of SMV is the storage scheme used for sparse matrices. The number of nonzero elements in a sparse matrix is significantly less than the number of zero valued elements. Hence, only nonzero elements and their column offsets are stored. One of the common storage schemes used for sparse matrices is the CSR scheme. In this scheme, three arrays \texttt{row}, \texttt{col} and \texttt{val} are used to store the sparse matrix. The \texttt{row} array stores a pointer (or index) into the \texttt{col} array. The \texttt{col} array stores column indices of the nonzero elements and the \texttt{val} array stores nonzero values. Figure 2.1(a) shows an example of the CSR storage for a sample sparse matrix.

**Algorithm 2.1 SMV Algorithm**

```plaintext
procedure SMV(A,x,y)
1: for i ∈ (1, N) do
2: sum = 0.0
3: for j ∈ (A.row[i], A.row[i + 1)) do
4: sum = sum + A.val[j] * x[A.col[j]]
5: end for
6: y[i] = sum
7: end for
```

**Sparse Matrix Vector Multiplication with CSR storage.** The SMV kernel is an integral part of most scientific applications such as the Conjugate Gradient Solver [31], data mining within the k-means clustering [32] and many others. Algorithm 2.1 gives a pseudocode of a traditional SMV implementation using CSR storage representation.
Typically, sparse symmetric matrices resulting from finite difference and finite element modeling problems are widely studied as their sparsity and structure can be utilized to improve performance [6]. In finite difference and finite element methods, a sparse matrix arises as a result of discretization using a mesh (or grid) [33]. Such matrices have structures that can be exploited by matrix blocking and reordering schemes to improve the performance of SMV.

**Blocking technique.** The SPARSITY [18] package reduces the integer index overhead using various *matrix blocking* techniques and increases the temporal and spatial locality using different *cache blocking* techniques. Im et al. [19] use register blocking to improve performance of the SPARSITY package. They use Blocked Compressed Sparse Row (BCSR) storage format to store sparse matrices. Figure 2.1(b) shows an example of 2×2 blocked storage scheme. Vuduc et al. [20] present a variation of BCSR called Unaligned Block Compressed Sparse Row (UBCSR) that improves performance of SMV over BCSR. Toledo [34] describes prefetching, reordering and blocking techniques to improve performance of the SMV kernel. Goumas et al. [35] discuss the effect of multilevel caches and hardware prefetchers on SMV performance. Pinar et al. [17] provide a one-dimensional blocking scheme that solves a traveling salesman problem (TSP) to find optimal matrix blocks to reduce indirect references and increase locality.

**Ordering technique.** The Reverse-Cuthill-McKee (RCM) ordering [36] is a matrix bandwidth reduction technique, wherein, the bandwidth of a matrix is computed as the maximum bandwidth of each row of the matrix. The bandwidth of a row of the matrix
is the number of matrix elements between the first and last nonzero elements in the row. RCM uses renumbering of the nodes in a graph representation of the matrix to produce a new matrix with a much smaller bandwidth.

Das et al. [15] use matrix bandwidth reducing reordering techniques, like Cuthill-McKee (CM) and Reverse-Cuthill-McKee (RCM), to improve cache utilization of SMV. Yzelman et al. [16] propose and develop a matrix reordering technique to improve cache efficiency of SMV. They show performance improvements using their matrix reordering technique over traditional SMV implementations.

In addition to blocking and reordering techniques, SMV optimization schemes also include index and value compression. Willcock et al. [21] compress column indices using a delta encoding scheme. Kornilios et al. [22] use two compression techniques, CSR-DU, a delta encoding scheme for column indices, and CSR-VI, a value compression scheme for storing only unique values. Keyes [23], Langou et al. [24], and Lee et al. [25] use different kinds of value compression techniques ranging from exploiting the symmetry to representing the data using lower precision. Belgin et al. [37] use a pattern based representation (PBR) to divide a sparse matrix into blocks that share distinct patterns and propose a custom multiplication scheme to exploit these recurring block patterns.

Williams et al. [38] present several optimization techniques to improve the performance of parallel SMV. They use a combination of one or more of the following techniques: thread blocking, cache blocking, TLB blocking, register blocking, loop optimizations, software prefetching and an auto-tuning framework. Guo et al. [26] propose new sparse data structures for improving performance of SMV on the IBM POWER architecture. The sparse data structures are designed to effectively utilize the prefetch engine present in the IBM POWER architecture. On this architecture, they show that the SMV kernel optimized using the new data structures significantly outperforms traditional schemes. Their schemes also show benefits on the Intel and AMD architectures.

Our scheme, SpEED-SMV, exploits the sparsity structure of sparse matrices. Among the SMV optimization schemes discussed, blocking techniques used in the SPARSITY [18] and PBR [37] are close to our proposed approach. Unlike our scheme, blocking techniques typically introduce fill-in and selection of an appropriate block size is not trivial and architecture dependent. PBR is similar to our scheme in that it exploits recurring block patterns within a sparse matrix. A key difference is that it is limited to finding structures that are explicit in the original (natural) order whereas our scheme can identify latent dense supernodal structures independent of original orderings in sparse symmetric matrices from finite difference and finite element methods. Additionally, by using a reordering in the
supernodal structure, our scheme increases reuse in the vector, \( x \), in addition to reducing index load overheads.

### 2.3 Sparsity-Exploiting Effectively Dense Sparse Matrix Vector Multiplication (SpEED-SMV)

In this section, we propose SpEED-SMV, a method to improve the performance of SMV through a Sparsity-Exploiting, Effectively Dense (SpEED) representation that can be useful for scientific applications involving partial differential equation models. First, we identify indistinguishable vertices in a graph representation of the sparse matrix \( A \) and group them into a supernode. Next, we identify effectively dense blocks within the matrix by grouping rows and columns in each supernode. Finally, by using a suitable data structure for this representation of the matrix, we reduce the number of load operations during SMV while exactly preserving the original sparsity structure of \( A \). Additionally, we use ordering techniques to enhance locality in accesses to the vector, \( x \), to yield an SMV kernel that exploits the effectively dense substructures in the matrix.

We use the same definition of indistinguishable vertices as stated by George and Liu in their earlier work on minimum degree ordering [39]. Formally, two nodes \( u \) and \( v \) are *indistinguishable* in graph \( G \) if

\[
\text{Adj}_G(u) \cup \{u\} = \text{Adj}_G(v) \cup \{v\}, \tag{2.1}
\]

where \( \text{Adj}_G(u) \) and \( \text{Adj}_G(v) \) are the adjacency lists of vertices \( u \) and \( v \) in \( G \). Assume \( G \) represents sparse matrix \( A \). Row \( i \) in \( A \) represents vertex \( i \) in \( G \) and a nonzero \( a_{ij} \) at row \( i \) column \( j \) indicates the presence of an edge between vertex \( i \) and vertex \( j \) with edge-weight \( a_{ij} \). Therefore, a row \( i \) in \( A \) represents the adjacency list of vertex \( i \) in \( G \). Groups of rows or columns in \( A \) satisfying Equation 2.1 are considered *indistinguishable*. We use a variant of the hashing technique proposed by Ashcraft [27] to identify indistinguishable rows and columns in matrix \( A \).

Typically, a group of indistinguishable vertices is referred to as a *supernode* [39]. In a supernodal ordering of a matrix, we first identify indistinguishable rows and columns in the matrix corresponding to the indistinguishable vertices and group them. The supernodal ordered matrix has fewer numbers of rows and columns. Each element in this matrix corresponds to a supernode in the graph representation of the supernodal matrix.
Determining supernodal matrix $B$ from matrix $A$ based on adjacency similarity. Consider Figure 2.2. $A$ is an $N \times N$ sparse symmetric matrix. We transform matrix $A$ into a supernodal matrix $B$ using the transformation, $F_c(A)$. This transformation comprises identifying sets of rows and columns that are indistinguishable using the hashing technique in [27] and grouping these sets into a single coarse element in $B$. Thus, $B$ is a coarse representation of the original matrix $A$. We use this step to form effectively dense sub-blocks, represented by each element of $B$, without any fill-in. Our reduced CSR storage format and density-aware SMV implementation exploit these dense sub-blocks to decrease overheads due to loading indices and increase reuse in $x$.

Figure 2.2: SpEED sparse matrix vector multiplication.

Figure 2.3: An example of reduced CSR representation.

Representing matrix $C$ using reduced CSR storage. Figures 2.2 shows the permuted supernodal matrix $B$ expanded into the SpEED ordered matrix $C$. We refer to this as refinement, as it expands the coarse matrix $B$ to form $C$. Observe that $C$ has many effectively dense sub-blocks when compared to $A$. We utilize these dense sub-blocks to represent
<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>Index of the ( \text{val} ) vector</td>
</tr>
<tr>
<td>yind</td>
<td>Index of the output vector ( y )</td>
</tr>
<tr>
<td>maxrow</td>
<td>Maximum number of rows in the matrix</td>
</tr>
<tr>
<td>num</td>
<td>Number of overlapping rows</td>
</tr>
<tr>
<td>MAXNUM</td>
<td>Maximum number of overlapping rows (maximum size of a supernode)</td>
</tr>
<tr>
<td>rows</td>
<td>An array for vector ( \text{c_rowid} )</td>
</tr>
<tr>
<td>( \delta )</td>
<td>An array for vector ( \delta )</td>
</tr>
<tr>
<td>( x )</td>
<td>An array for a vector to be multiplied</td>
</tr>
<tr>
<td>( y )</td>
<td>Represents the output of SMV</td>
</tr>
</tbody>
</table>

Table 2.1: Description of important variables used in our algorithm.

\( C \) using a reduced CSR representation. To exploit the repeating rows (dense sub-blocks) in \( C \), we define a vector \( \delta \), wherein each element of \( \delta \) indicates the number of repeated similar rows. Our modified version of CSR storage format uses \( < \text{c\_rowid}, \text{c\_colid}, \text{val}, \delta > \) vectors as compared to traditional CSR which uses only \( < \text{rowid}, \text{colid}, \text{val} > \) to store \( C \). The vectors \( \text{c\_rowid} \) and \( \text{c\_colid} \) represent compressed row and compressed column vectors, respectively. These vectors store only unique row and column indices, i.e., they do not store indices of similar rows and columns. The sizes of \( \text{c\_rowid} \) and \( \text{c\_colid} \) are dependent on the amount of compression (more compression gives smaller vector sizes). In this modified format, the \( \text{val} \) vector remains unchanged and is the same as traditional CSR. The size of \( \text{val} \) is equal to the number of nonzeroes and the size of \( \delta \) is equal to the number of rows in \( C \). The value of elements in \( \delta \) lies in the interval \([-1, \text{MAXNUM}]\). An element of \( \delta \) with value of \(-1\) indicates that the row mapped to this element is similar to the row mapped to the previous non-negative element of \( \delta \). Any value between 0 to \( \text{MAXNUM} \) indicates the number of repeated similar rows. If the number of repeated similar rows is greater than \( \text{MAXNUM} \), in the interest of simplicity, we split them into two or more sets. For example, if the initial \( \delta \) is \((8,-1,-1,-1,-1,-1,-1,-1)\) and \( \text{MAXNUM} \) is 5, then, \( \delta \) is converted to \((5,-1,-1,-1,-1,2,-1,-1)\). Hence, we implement our SMV algorithm using \( \text{MAXNUM} \) conditional blocks (see Algorithm 2.2). Figure 2.3 shows an example of reduced CSR format. From this figure we see that the first two rows are indistinguishable, as nonzero elements of these rows have same column indices. Observe that the \( \delta \) value for the second row is \(-1\) indicating its similarity to the first row. Also observe that \( \text{c\_colid} \) does not store the indices of nonzero elements in the second row.

**Effectively dense SMV.** We store matrix \( C \) using our reduced CSR storage representation. We obtain a mapping \( m \) of reordered coarse elements in \( C \) to set of rows in \( A \).
Algorithm 2.2 Modified SMV Algorithm

procedure Modified SMV(A,x,y)
1: \(\text{valid} \leftarrow \text{validtrace}[\text{myid}]\)
2: \(yind \leftarrow 0\)
3: \(\text{while } \text{row} < \text{maxrow} \text{ do}\)
4: \(\text{num} \leftarrow \delta(\text{row})\)
5: \(\text{if } \text{num} == 0 \text{ then}\)
6: \(\text{for } i = \text{rows}[\text{row}] \text{ to } \text{rows}[\text{row} + 1] - 1 \text{ do}\)
7: \(y[yind] \leftarrow y[yind] + \text{val}[\text{valid} + i - \text{rows}[\text{row}]] \times x[\text{colid}[i]]\)
8: \(\text{end for}\)
9: \(yind \leftarrow yind + 1\)
10: \(\text{else if } \text{num} == 1 \text{ then}\)
11: \(t \leftarrow \text{rows}[\text{row} + 1] - \text{rows}[\text{row}]\)
12: \(\text{for } i = \text{rows}[\text{row}] \text{ to } \text{rows}[\text{row} + 1] - 1 \text{ do}\)
13: \(y[yind] \leftarrow y[yind] + \text{val}[\text{valid} + i - \text{rows}[\text{row}]] \times x[\text{colid}[i]]\)
14: \(y[yind + 1] \leftarrow y[yind + 1] + \text{val}[\text{valid} + i - \text{rows}[\text{row}] + t] \times x[\text{colid}[i]]\)
15: \(\text{end for}\)
16: \(yind \leftarrow yind + 2\)
17: \(\text{else if } \text{num} == 2 \text{ then}\)
18: \(t \leftarrow \text{rows}[\text{row} + 1] - \text{rows}[\text{row}]\)
19: \(\text{for } i = \text{rows}[\text{row}] \text{ to } \text{rows}[\text{row} + 1] - 1 \text{ do}\)
20: \(\text{....}\)
21: \(\text{end for}\)
22: \(yind \leftarrow yind + 3\)
23: \(\text{..}\)
24: \(\text{..}\)
25: \(\text{..}\)
26: \(\text{else if } \text{num} == \text{MAXNUM} \text{ then}\)
27: \(t \leftarrow \text{rows}[\text{row} + 1] - \text{rows}[\text{row}]\)
28: \(\text{for } i = \text{rows}[\text{row}] \text{ to } \text{rows}[\text{row} + 1] - 1 \text{ do}\)
29: \(\text{....}\)
30: \(\text{end for}\)
31: \(yind \leftarrow yind + \text{MAXNUM} + 1\)
32: \(\text{end if}\)
33: \(\text{valid} \leftarrow \text{valid} + (\text{num} + 1) \times (\text{rows}[\text{row} + 1] - \text{rows}[\text{row}])\)
34: \(\text{row} \leftarrow \text{row} + 1\)
35: \(\text{end while}\)

The last part of this phase performs an effectively dense matrix vector multiplication using \(C\), \(m\) and \(\tilde{x}\), where \(\tilde{x}\) is a permutation of \(x\) corresponding to the permutation required to form \(C\) from \(A\). Algorithm 2.2 gives the pseudocode of our density-aware SMV algorithm and Table 2.1 describes the variables used in the algorithm. We see from Algorithm 2.2 that depending on the value of \(\text{num}\), one of the conditional blocks is executed, and, each conditional block is representative of the size of a supernode (i.e., number of rows contained in the supernode). Note that \(\text{MAXNUM}\) represents the size of the largest supernode; our
analysis shows that the value of MAXNUM is typically a small constant. Observe that \textit{colid} and \textit{x} are being reused within the loop, thereby reducing the total number of load operations. We see that the higher value of \textit{num} leads to higher reuse and fewer load operations.

**Locality enhancing ordering.** As noted in Section 4.3, ordering techniques can be used to enhance locality in accesses to \textit{x}. Figure 2.4 illustrates RCM ordering used in conjunction with our technique.

![SpEED-SMV with RCM Ordering](image)

**Figure 2.4:** SpEED sparse matrix vector multiplication with RCM ordering.

### 2.3.1 Overheads of SpEED-SMV

In this subsection, we discuss the overheads of using our methodology. We incur overheads related to: (1) detection of supernodal structure and corresponding dense row representations (compression), (2) envelope ordering for locality of reuse in \textit{x}, and, 3) expanding the compressed graph.

Our supernodal schemes use a natural adaptation of techniques proposed by Ashcraft [27], which has three steps, First, vertices are assigned a checksum generated at a cost of \(|E|\), the number of edges or equivalently the number of nonzero elements in \(A\). Next, vertices are compared after sorting them by checksum. In our scheme, we check only a limited number of vertices for the “indistinguishable vertex” property without sorting them by checksum, to limit the cost of this step to \(|V|\) instead of the original \(|V|\log|V|\), where \(|V|\) is the number of vertices or equivalently, the matrix dimension. A third step concerns finding the structure of supernodes, which is typically \(O(|E|)\). The cost of envelope orderings is \(O(|E_c|)\) where, \(|E_c|\) is the number of edges in the compressed graph (\(|E_c| < |E|\)). Finally, the cost of expanding the compressed supernodal structure is \(O(|E|)\). Hence the total
overhead of our method is of the form which is, \(O(2|E| + |V|)\), which compares well to the cost of a single SMV. In practice, SMV is more efficient than graph traversals for a given sparse matrix; hence, we expect the set-up time of SpEED-SMV to be commensurate with the time for a few SMV operations.

### 2.4 Experimental Evaluation

In this section, we present our experimental setup and report performance results obtained using SpEED-SMV and BCSR schemes.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>N</th>
<th>NNZ</th>
<th>(\alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcsstk15</td>
<td>3,948</td>
<td>117,816</td>
<td>3,948</td>
</tr>
<tr>
<td>bcsstk24</td>
<td>3,562</td>
<td>159,910</td>
<td>892</td>
</tr>
<tr>
<td>bcsstk28</td>
<td>4,410</td>
<td>219,024</td>
<td>774</td>
</tr>
<tr>
<td>bcsstk16</td>
<td>4,884</td>
<td>290,378</td>
<td>1,778</td>
</tr>
<tr>
<td>crystk01</td>
<td>4,875</td>
<td>315,891</td>
<td>1,717</td>
</tr>
<tr>
<td>bcsstk38</td>
<td>8,032</td>
<td>355,460</td>
<td>3,456</td>
</tr>
<tr>
<td>bcsstk17</td>
<td>10,974</td>
<td>428,650</td>
<td>5,219</td>
</tr>
<tr>
<td>bcsstk33</td>
<td>8,738</td>
<td>591,904</td>
<td>4,344</td>
</tr>
<tr>
<td>bcsstk29</td>
<td>13,992</td>
<td>619,488</td>
<td>10,202</td>
</tr>
<tr>
<td>pkustk02</td>
<td>10,800</td>
<td>810,000</td>
<td>1,251</td>
</tr>
<tr>
<td>crplat2</td>
<td>18,010</td>
<td>960,946</td>
<td>3,004</td>
</tr>
<tr>
<td>crystk02</td>
<td>13,965</td>
<td>968,583</td>
<td>4,765</td>
</tr>
<tr>
<td>pcrystk02</td>
<td>13,965</td>
<td>968,583</td>
<td>4,765</td>
</tr>
<tr>
<td>pkustk01</td>
<td>22,044</td>
<td>979,380</td>
<td>3,740</td>
</tr>
<tr>
<td>bcsstk36</td>
<td>23,052</td>
<td>1,143,140</td>
<td>4,351</td>
</tr>
<tr>
<td>bcsstk31</td>
<td>35,588</td>
<td>1,181,416</td>
<td>17,403</td>
</tr>
<tr>
<td>bcsstk35</td>
<td>30,237</td>
<td>1,450,163</td>
<td>6,611</td>
</tr>
<tr>
<td>pkustk09</td>
<td>33,960</td>
<td>1,583,640</td>
<td>5,660</td>
</tr>
<tr>
<td>crystk03</td>
<td>24,696</td>
<td>1,751,178</td>
<td>8,339</td>
</tr>
<tr>
<td>pcrystk03</td>
<td>24,696</td>
<td>1,751,178</td>
<td>8,339</td>
</tr>
</tbody>
</table>

Table 2.2: A suite of smaller test matrices (Suite 1).

#### 2.4.1 Experimental Setup

Our setup includes benchmark matrices to evaluate SMV techniques, a multicore configuration with PAPI [40, 41, 42] to access hardware performance counters and SMV optimization methods (R×C-SMV and SpEED-SMV).

**Benchmark matrices.** We use a suite of 40 symmetric matrices from the *University of Florida Sparse Matrix Collection* [43]. Tables 2.2 and 2.3 present an overview of
the matrix properties, including, dimension (N), number of nonzero elements (NNZ), and number of supernodes (α). As described in Section 2.3, supernodes are formed using the indistinguishable vertices that are commonly found in matrices used in finite element formulations. This property is highlighted by α in the tables, which shows that the number of unique rows are a small fraction of the total number of rows. Note that the 40 matrices are split into two test suites based NNZ with the matrices in Table 2.3 being larger than those in Table 2.2. We use both the suites for an initial comparison of the different SMV optimization techniques, followed by a detailed analysis for the 20 largest matrices from Table 2.3.

Tables 2.4 and 2.5 illustrate the distribution of supernode sizes, i.e., the number of rows contained in a supernode. In the tables, we define set $S_i$ as the set of elements that have $i$ similar rows in the matrix ($i \in [1,6]$). The set $S^*$ consists of all elements that have more than six similar nodes. Observe that the sets $S_1 - S_6$ contain most of the elements in the matrix.

**Multicore setup.** We perform our experiments on the two-socket 8-core Intel Nehalem [44] and AMD Shanghai [45] processors. As indicated in Figure 2.5, the Intel Ne-
Table 2.4: Distribution of supernode sizes: Suite 1.

| ID | \[\|S_1\|,|S_2|,|S_3|,|S_4|,|S_5|,|S_6|,|S^*|\] | \[\sum_{i=1}^{6} |S_i| / \sum_{i=1}^{6} |S_i| \] |
|----|----------------|------------------|
| 1  | (3948,0,0,0,0,0) | 1                |
| 2  | (186,158,76,0,0,472) | 1                |
| 3  | (0,37,59,1,0,669,8) | 0.999            |
| 4  | (239,91,1404,6,1,37,0) | 1                |
| 5  | (125,26,1566,0,0,0,0) | 1                |
| 6  | (1821,564,315,6,396,348,6) | 0.998 |
| 7  | (2329,1301,1163,1,0,425,0) | 1                |
| 8  | (1801,692,1851,0,0,0,0) | 1                |
| 9  | (8626,713,299,141,59,364,0) | 0.729            |
| 10 | (0,0,0,0,912,339) | 1                |
| 11 | (0,0,0,6,2,2996,0) | 1                |
| 12 | (139,52,4574,0,0,0,0) | 1                |
| 13 | (139,52,4574,0,0,0,0) | 1                |
| 14 | (0,0,133,0,0,3606,1) | 0.999            |
| 15 | (188,209,194,4,690,3064,2) | 0.999 |
| 16 | (6824,5266,4346,9,590,368,0) | 1                |
| 17 | (981,627,653,4,55,4290,1) | 0.999            |
| 18 | (0,0,0,0,5660,0) | 1                |
| 19 | (126,69,8144,0,0,0,0) | 1                |
| 20 | (126,69,8144,0,0,0,0) | 1                |

Figure 2.5: Illustration of the Intel Nehalem (a), and the AMD Shanghai (b) architecture.

halem cache configuration consists of a 64KB L1 and a 256KB L2 cache per core and an 8MB L3 cache shared among all the cores, while that of AMD Shanghai consists of 128KB L1 and 512KB L2 cache per core, and an 6MB shared L3 cache.

We use PAPI to measure the performance related statistics like cycles and cache misses. We use Linux `sched_setaffinity` to map threads onto cores in our parallel SMV implementation.

**Optimization methods.** Table 2.6 lists the methods used in our experimental study.
Table 2.5: Distribution of supernode sizes: Suite 2.

| ID | (|S_1|,|S_2|,|S_3|,|S_4|,|S_5|,|S_6|,|S_*|) | \( \sum_{i=1}^{6} |S_i| \sum_{i=1}^{6} |S_i| \) | \( \frac{\sum_{i=1}^{6} |S_i|}{\sum_{i=1}^{6} |S_i|} \) |
|----|---------------------------------|-------------------------------------------------|----------------|
| 1  | (0,0,0,0,0.2,146,722)          | 0.748                                           |               |
| 2  | (4852,3091,2274,69,749,3770,16) | 0.999                                           |               |
| 3  | (982,0,0,9158,0,0)             | 1                                               |               |
| 4  | (0,0,4438,0,0,473,76)          | 0.985                                           |               |
| 5  | (5441,3451,4333,42,156,4226,75) | 0.996                                           |               |
| 6  | (5441,3451,4333,42,156,4226,75) | 0.996                                           |               |
| 7  | (2,0,349,0,0,14900,0)          | 1                                               |               |
| 8  | (0,0,0,0,0.1,3446,0)           | 1                                               |               |
| 9  | (7045,389,320,16,12,0,2)       | 0.989                                           |               |
| 10 | (0,0,7860,0,0,884,94)          | 0.986                                           |               |
| 11 | (0,0,143,1,1,5580,80)          | 0.986                                           |               |
| 12 | (0,0,47857,0,0,0,0)            | 1                                               |               |
| 13 | (450,0,31385,0,0,20,17)        | 1                                               |               |
| 14 | (0,0,757,0,2,18773,0)          | 1                                               |               |
| 15 | (0,0,1,1,1,20258,14)           | 1                                               |               |
| 16 | (13131,3550,34213,269,272,4740,0) | 1                                          |               |
| 17 | (4,0,902,0,0,29525,0)          | 1                                               |               |
| 18 | (4443,1289,1226,27,173,34369,4) | 1                                          |               |
| 19 | (0,0,25719,0,0,22716,0)        | 1                                               |               |
| 20 | (962,231,15865,7,0,16914,155)  | 0.995                                           |               |

Table 2.6: SMV optimization techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR-RCM</td>
<td>SMV on RCM ordered matrices</td>
</tr>
<tr>
<td>R×C-SMV</td>
<td>A blocking scheme where R, C are obtained using OSKI</td>
</tr>
<tr>
<td>SpEED-SMV</td>
<td>SpEED with envelope ordering</td>
</tr>
</tbody>
</table>

Our method, SpEED-SMV, uses an envelope ordering (RCM) scheme to enhance locality in x accesses. We compare the performance of our method with the blocking schemes proposed in [18, 19]. We use OSKI [29] to find R×C blocking for each matrix. OSKI uses several measures and heuristics to search for a blocking scheme with highest estimated performance. Note that each matrix can have varying block sizes on different processors as this parameter in R×C-SMV is hardware dependent.
Figure 2.6: Performance improvement of SMV optimization techniques relative to the base case on 8-cores of the Intel Nehalem processor for the test (a) Suite 1 matrices and (b) Suite 2 matrices.

### 2.4.2 Evaluation and Discussion

We present experimental results comparing R×C-SMV and SpEED-SMV to the traditional CSR SMV on 8-cores of the Intel Nehalem and AMD Shanghai processors. We refer to a traditional SMV implementation using CSR storage without any optimization as our base scheme. In addition, we define and discuss an architecture independent measure, effective density (η), that can accurately predict the SMV optimization scheme that will lead to best performance based on matrix properties, for a specific processor.

Figure 2.6 shows the relative performance (in terms of observed execution time) of three
different optimization techniques, CSR-RCM, RxC-SMV and SpEED-SMV, with respect to the base scheme on the Intel Nehalem processor. The percentage relative performance of scheme X is defined as $\frac{T(\text{base}) - T(X)}{T(\text{base})} * 100$, where $T(\text{base})$ and $T(X)$, are the execution time in cycles, of the base case and an SMV optimization scheme, respectively. Several observations related to Figure 2.6 prompt key questions that motivate a more in-depth comparative evaluation. First, observe that the performance benefits of CSR-RCM are
significantly lower than for the other schemes for most matrices. Consequently, we do not provide additional results for CSR-RCM but instead focus on R×C-SMV and SpEED-SMV. We observe that on average, both, R×C-SMV and SpEED-SMV, improve performance by 28.00% for matrices in Suite 1. However, R×C-SMV improves performance by 30.27% while SpEED-SMV improves performance by 37.35% for matrices in Suite 2. More importantly, if we pick for each matrix the best among these two methods, the average performance improvements increase to 33.30% in Suite 1 and 40.85% in Suite 2 (as indicated by the line in Figure 2.6).

Figure 2.7 illustrates the relative energy consumption of CSR-RCM, R×C-SMV and SpEED-SMV, with respect to the base scheme on the Intel Nehalem processor. We observe that on average, both, R×C-SMV and SpEED-SMV, reduce energy consumption by 28.00% for matrices in Suite 1. However for matrices in Suite 2, R×C-SMV reduces energy consumption by 29.22% and SpEED-SMV by 39.02%. From Figures 2.6 and 2.7, we observe that there is a strong correlation between performance and energy consumption that indicates performance improvement leads to reduction in the energy consumption. There, we focus only on performance analysis of the optimization schemes in remainder of this section.

Our evaluations suggest that there is no clear winner between R×C-SMV and SpEED-SMV and there is significant variability in their performance across matrices. We therefore focus on test suite of larger matrices (Suite 2) and perform additional experiments to seek answers to the following questions.

- What is the major contributing factor for the performance of SpEED-SMV? Is it reuse in $x$ through RCM ordering or its density-aware SMV implementation (Algorithm 2.2)? Analogously, can R×C-SMV benefit from RCM ordering?

- Can we expect further variations in the choice of the best performing method when we consider a different processor architecture, such as the AMD Shanghai processor? Given such variations, are there simple measures that can be fit to observed performance data to predict an SMV optimization scheme that is best for a given matrix and processor?

Figures 2.8(a) and 2.8(b) provide relative performance data to assess the impacts of index load overhead reduction and enhanced reuse in $x$ through RCM orderings for SpEED-SMV and R×C-SMV. Figure 2.8(a) indicates the impacts of using Algorithm 2.1 (base SMV algorithm) and Algorithm 2.2 on RCM-ordered supernodal matrices (see Figure 2.4). The difference indicates incremental improvements from exploiting dense substructures
in SpEED-SMV that go beyond the benefits of reuse in $x$ through RCM. Observe that although base SMV improves the performance marginally, using Algorithm 2.2 improves the performance significantly. Hence, the density-aware SMV algorithm is the main contributor towards improved performance and RCM ordering complements the scheme to utilize reuse in $x$. To consider the effect of RCM on $R \times C$-SMV, consider Figure 2.8(b). Observe that, on average, the performance of $R \times C$-SMV is the same with or without RCM ordering. Although, certain matrices like, Matrix 20, benefit from RCM ordering, Matrices 7 and 18 do not. In general, matrices arising out of finite-element and finite-difference modeling have inherent blocked structures, for which RCM ordering might not be useful. In fact, it may distort the blocked structures and thus detracting from performance. Consequently, in subsequent evaluations, we use $R \times C$-SMV with the original ordering of matrices.

We next consider further differences in the performance of SpEED-SMV and $R \times C$ SMV due to the impacts of hardware features. We use the percentage relative performance of SpEED-SMV relative to $R \times C$-SMV (defined as $\frac{T(R \times C - SMV) - T(\text{SpEED-SMV})}{T(R \times C - SMV)} \times 100$) for each of the two multicore architectures. Figures 2.9(a) and 2.9(b) indicate this measure on 8-cores of the Intel Nehalem and AMD Shanghai, respectively; negative values indicate degradation of our method relative to $R \times C$-SMV. Observe that SpEED-SMV shows improvements for two different subsets of 13 and 11 matrices, on the Intel Nehalem and AMD Shanghai. Observe that SpEED-SMV performs significantly better than $R \times C$-SMV on matrices like Matrices 3 and 20 (Figure 2.9(a)), and Matrices 11 and 14 (Figure 2.9(b)). Likewise, $R \times C$-SMV outperforms SpEED-SMV on matrices like Matrices 12 and 15 (Figure 2.9(a)), and Matrices 10 and 12 (Figure 2.9(b)). It is clear that neither methods perform consistently well on all the matrices across both types of processors. However, there can be consistent benefits across all matrices and processors if we can develop an effective scheme for method selection that can be applied within an auto-tuning framework such as OSKI.

We now consider matrix-specific measures related to the effectiveness of optimization mechanisms in SpEED-SMV and $R \times C$-SMV that could be potentially used toward method selection by deriving models for 2-way classification by analyzing performance data on different processors. The proposed approach is preliminary at best and far from a final method selection scheme with statistically validated prediction accuracies. Instead, it seeks to indicate how such a method could be developed based on extensive performance data gathering and predictive model-driven classification, toward a fully dynamic, auto-tuning framework for SMV.

Observe that both SpEED-SMV and $R \times C$-SMV, improve the performance of SMV by optimizing the number of load operations per useful floating point operation. We
therefore seek a measure that reflects this tradeoff. Recall that the fill-in ratio \( F \), defined as the ratio of nonzeros in the transformed matrix to original number of nonzeros), is a measure that captures the useful floating point operations. Additionally, we define the load improvement ratio, \( L \), the ratio of original number of loads to the load operations in the optimized scheme. A higher value of \( L \) reflects load reduction that could improve performance while a higher value of \( F \) indicates an increase in floating point operations that operate on fill-in and thus a potential degradation in performance. Figure 2.10(a) shows \( F \) for all the matrices. Observe that SpEED-SMV does not incur fill-in (\( F = 1 \)), whereas the
Figure 2.9: Performance of SpEED-SMV with respect to the RxC-SMV on (a) Intel Nehalem, and (b) AMD Shanghai, two socket quad-core processors.

fill-in ratio for RxC-SMV varies based on the matrix and the processor. Some matrices have perfect blocking structure (like Matrix 8), hence no fill-in, while others have high fill-in, like Matrix 11. Figure 2.10(b) shows the reduction in load improvement ratio \( L \) for both SpEED-SMV and RxC-SMV on the Intel Nehalem and AMD Shanghai processors. Note that for a given matrix, method and processor, a higher value of \( L \) and a lower value of \( F \) are desirable indicators of higher performance. To capture this tradeoff, we define, \( \text{transfer density} \), as \( \tau = \frac{L}{F} \). Observe that \( \tau \) depends on the matrix and the type of processor.
for \( R \times C \)-SMV while it depends only on the matrix for SpEED-SMV.

A second property that affects SMV performance is the \textit{row density} \((\rho)\), defined as \(\rho = \frac{nnz}{n}\), where, \(nnz\) indicates the total number of nonzeros (including any fill-in) and \(n\) indicates the total number of rows (for a specific SMV scheme, matrix and processor type). The value of \(\rho\) is a coarse measure of how often a branch instruction is encountered on average within the inner loop of SMV; this measure along with branch overheads of a specific processor determines the impact of specific optimizations. In general, a higher value of \(\rho\) should be correlated with higher performance.

Next, we seek a combined measure in terms of \(\tau\) and \(\rho\) as a predictor for selecting either SpEED-SMV or \(R \times C\)-SMV for a given matrix on a specific processor configuration. We call the combined measure, \textit{effective density}, \(\eta\) and define it as a simple linear combination of the form \(\eta = \alpha \tau + \beta \rho\). Now \(\alpha\) and \(\beta\) are constants that depend only on the SMV optimization scheme and the processor architecture. However, \(\tau\) and \(\rho\) are parameters that depend on the matrix and SMV optimization scheme and additionally, the processor architecture for \(R \times C\)-SMV. We fit observed performance data to these parameters to obtain equations for \(\eta\). For \(R \times C\)-SMV, we obtained \(\eta = \tau + 0.5 \rho\) for both processor types because the OSKI-based parameter selection changes the blocking level to take into account variations due to processor attributes. For SpEED-SMV, we obtained \(\eta = \tau + 1.2 \rho\) for Intel Nehalem and \(\eta = \tau + \rho\) for AMD Shanghai; the difference in the weightings arise from the fact that architecture-specific impacts are not explicitly taken into account in SpEED-SMV. Figure 2.10(c) shows the effective densities for all the matrices on both Intel Nehalem and AMD Shanghai. The red circles indicate the matrices on which there was a mismatch between prediction and observation, i.e., a higher value of \(\eta\) resulted in lower performance. These results thus indicate the potential to deliver consistently higher performing SMV through an auto-tuning framework, with model-driven selection of kernels tailored to matrix and hardware attributes.

### 2.5 Conclusions and Future Work

In this chapter, we improve performance of the SMV kernel by exploiting the dense substructures that are inherently present in many sparse matrices derived from partial differential equation models. We observe that there are several indistinguishable rows and columns in sparse matrices arising from finite difference or finite element based models. We form effectively dense blocks within the matrix by grouping indistinguishable rows and columns. Such a representation reduces the number of load operations for accessing indices
of each nonzero element of sparse matrix, $A$, during SMV without incurring additional fill-in, in contrast of the class of $R \times C$ blocking SMV. Further, we enhance locality in accesses to the vector, $x$, using matrix ordering techniques. Finally, we use a density-aware SMV algorithm to exploit the effectively dense structure of the reordered matrix.

We evaluated the performance of our method, SpEED-SMV and best $R \times C$-SMV in terms of their execution times, on two popular multicore systems, namely, the Intel Nehalem and AMD Shanghai processors. We observed that there are significant benefits (40.6%–48.1%) across all matrices on both processors by selecting the best among these two choices. However, there is no clear winner, as neither methods perform consistently well across all the matrices. We therefore propose an effective density measure that could

---

**Figure 2.10:** (a) fill-in ratio, (b) decrease in load operations, and (c) effective density on Intel Nehalem and AMD Shanghai processors.
be used as a predictor to choose an SMV optimization scheme that matches well to matrix and hardware attributes to yield high performance.

Our initial results are encouraging and we see SpEED-SMV as filling a gap in SMV optimization techniques for the relatively large class of sparse matrices with inherent dense substructures. We expect it to be a useful addition within an auto-tuning framework, to cater to matrices arising from higher order finite difference and finite element methods for partial differential equation models [33]. In such applications, many variables are associated with a single mesh or grid point, and the higher order operators span multiple neighboring elements to give rise to larger number of supernodes containing relatively large number of indistinguishable vertices. Additionally, Demmel et al. [46] have recently proposed a framework to avoid communication within sparse matrix computations in Krylov subspace iterative solvers. We conjecture that SpEED-SMV can be particularly useful in this context where it can be viewed as operating on a supernodal mesh or graph representation to reduce loads and the pressure on memory bandwidth and interprocessor communication. SpEED-SMV can also be combined with hardware-specific optimizations, that have recently been shown by Guo and Gropp [26] to be very successful for performance enhancements on the IBM POWER series multicores.
Chapter 3

Enhancing Performance Measures: Sparse Scientific Workloads

3.1 Introduction

In Chapter 2, we developed an algorithm to improve the speedup and energy measures of the SpMV kernel on multicores. As a next step, we seek to improve throughput and energy consumption measures for scientific workloads that run on a cluster of multicores.

Using a supercomputing environment poses several traditional challenges ranging from application optimizations for effective hardware utilization to managing a set of optimized applications (workload) for performance and energy efficiencies. One of the current challenges is the effective use of enormous parallelism present in a supercomputing setup. In addition to parallelism across the supercomputing nodes, the advent of multicore nodes has augmented parallelism at the node level. Due to this increase in parallelism, many scientific applications hit Amdahl’s knee [47] at a lower node count than before. Hence, there is a need to effectively utilize high computing performance (HPC) systems in the context of scientific workloads.

Typically, a large number of applications (jobs), both serial and parallel, execute concurrently on a supercomputer. It is a challenge to manage and schedule these applications to achieve a set of objectives that help maintain system efficiency. In general, objectives like high throughput, low waiting time and low energy consumption are important. Workload management tools (WMT) [48] [49] strive to achieve some of these objectives. They not only use smart techniques to schedule applications, but also provide good resource management capabilities. As most of these tools are agnostic to the application behavior, scheduling and node allocation decisions are solely based on the job request and resource
A majority of supercomputing applications involve modeling and simulation of scientific phenomena. Such applications are executed repeatedly, by changing the model parameters - the main objective being the correctness and accuracy of the output. Although the output of each run may vary, application characteristics like speedup and average power consumption remain unaltered. For such a class of applications, we focus on proactive scheduling schemes that seek to improve overall system efficiency, including both energy and performance measures, by utilizing known application behavior.

Figure 3.1 illustrates the normalized energy consumption for different schedules for a set of four benchmark applications on 8 cores of Intel Nehalem. We define a schedule of a workload as a way in which all applications within the workload are executed. In Figure 3.1, the energy consumption of all schedules are normalized to the one with the lowest energy consumption. Each point in the figure represents a schedule. For this experiment, we consider all permutations of schedules consisting of running two application concurrently (co-schedule). The total number of threads of the co-scheduled applications is equal to the total number of cores (this constraint is purely to reduce experimental overheads). We observe that in the worst case, the energy consumption of a schedule can be as high as a factor of 5 compared to the schedule with lowest energy consumption. Thus, there is a need for workload management schemes that select schedules based on improving a set of
objectives such energy and throughput efficiencies.

In this chapter, we provide theoretical analysis for workload management and explore the performance and energy benefits possible for parallel scientific workloads. We propose application-aware mapping schemes to reduce the makespan of a workload through speedup-aware processor partitioning (SAPP). We first propose an optimal co-scheduling scheme for workloads using a two-way partition of the available cores and then develop a low overhead greedy variant. Our results indicate that SAPP-predicted schedules reduce makespan by as much as 31.5\% on an 8-core node and by as much as 49.3\% on 128 cores, compared to the FCFS schedules. Additionally, our SAPP scheme decreases the total energy consumption by as much as 57.0\% and reduces the average waiting time across all workloads by 89.6\% on 128 cores compared to the FCFS schedules.

The rest of the chapter is organized as follows: Section 3.2 first presents previous research related to scheduling applications at single and multiple node-levels. Next, it briefly discusses how our work is different. Section 4.4 gives notation and terminology used in this chapter and provides theoretical insights into making scheduling decisions related to the speedup profile of an application. Section 5.4 describes experimental setup and gives a detailed evaluation of the experimental results. Finally, Section 5.5 concludes this chapter with a brief summary of our findings.

### 3.2 Related Work

There are various ways to schedule multiple processes onto a given set of processors. The scheduling decisions are either made by an operating system or a WMT like PBS [48], depending on process type. Typically, batch jobs are scheduled using a WMT, whereas, interactive (online) jobs are managed by the operating system. In either case, a scheduling technique involves assigning processes to the available processors with the intention of improving metrics such as, system utilization, fairness, and throughput.

We broadly classify the current research related to scheduling techniques into two categories: 1) at a multicore-level or node-level, where most techniques improve on the operating system scheduler and 2) at a large-scale system level, where a WMT is used to schedule jobs across multiple nodes. We discuss node-level scheduling with respect to the operating system and focus on WMT schedulers for large-scale systems.

**Scheduling at node-level.** Chandra et al. [50] predict inter-thread cache contention on a multicore processor and use the prediction to improve the performance of co-scheduled threads. Lakshminarayana et al. [51] propose an age based scheduling policy for asymmet-
ric multiprocessors. This policy assigns a thread with a larger remaining execution time to a fast core. They use prediction and profile techniques to compute the remaining execution time of a thread. Fedorova et al. [52] present an operating system scheduling algorithm, *fair-cache*, that improves performance isolation on a chip multiprocessor (CMP), by modifying the threads’ CPU timeslices. Apart from multicores, a few studies have focused on scheduling applications on simultaneous multithreaded (SMT) processors [53, 54, 55].

Unlike most of the contention-centric techniques discussed above, our work is based on an in-depth theoretical analysis of exploiting the speedup characteristics of applications to increase system throughput. However, we conjecture that using some of the existing contention-aware scheduling techniques can improve the prediction mechanism of our work (as our analysis and prediction mechanism do not consider contention related issues). The work by Jiang et al. [56] that presents theoretical analysis of an optimal co-schedule is closely related to our work in terms of problem formulation. However, this work varies significantly in terms of the problem being solved. We propose techniques to reduce makespan by using application characteristics, whereas [56] predict co-schedules that reduce overall performance degradation.

**Scheduling at scale.** In a large system consisting of multiple computing nodes, a very basic scheduler is based the FCFS algorithm [57], where jobs are queued and processed based on their arrival time. As a naive FCFS algorithm can decrease system utilization and throughput, backfilling [58] or its variants [59, 60, 61] are used to supplement FCFS performance. Backfilling requires each job to specify certain parameters like its maximum execution time and processor requirements, and schedules the job based on the resource availability while maintaining the FCFS order. An alternative to backfilling is gang scheduling [62, 63], where jobs can be preempted and rescheduled to improve efficiency and fairness [64]. There are many commercial schedulers and WMTs that support the above mentioned techniques. The Maui scheduler [61] supports a variety of scheduling policies, extensive reservations, and fairness capabilities. The WMTs like IBM LoadLeveler [49], Platform’s Load Sharing Facility [65] and the Portable Batch System (PBS) [48, 66] support FCFS, FCFS with backfilling and its variants, fair-share, and many other scheduling strategies. Apart from these job schedulers, Li et al. [67] propose a framework to predict the energy and performance impacts of power-aware MPI task aggregation scheduling strategy for MPI programs.

Most of the techniques discussed above are agnostic to application behavior. Although [67] uses application-centric scheduling strategy, its primary focus is on scheduling multiple tasks within an MPI program. We focus on scheduling a workload based on the
known speedup behavior of individual applications. Instead of taking input parameters like processor requirements from an application, we proactively allocate processors to each application based on its speedup-profile to reduce the overall makespan of the workload.

3.3 Speedup Aware Processor Partitioning

In this section, we describe notation and terminology used in this chapter, illustrate an example to motivate the speedup aware processor partitioning techniques and provide theoretical insights into making scheduling decisions related to application speedup profiles.

**Notation and Terminology.** \( A_i \) represents Application \( i \), for \( i = 1..K \). \( r_i \) represents number of cores used by \( A_i \). Execution time of \( A_i \) on \( r_i \) cores is given by \( T(i, r_i) \). Compute power consumed by \( A_i \) is denoted by \( W_i \). For ease of notation, ‘\( \sum \)’ refers to ‘\( \sum_{i=1}^{K} \)’. We formally define the terms used in this chapter here.

- **Speedup** \( (\eta_{ij} = \frac{T(i,1)}{T(i,j)}) \) represents the fixed-problem speedup of Application \( i \) on \( j \) cores.
- **Speedup-profile** is a curve representing speedup of the application for varying core counts.
- **Workload** constitutes a set of applications.
- **Schedule** of a workload represents a way in which the applications are assigned to cores.
- **Base schedule** of a workload refers to a schedule where each application is run in parallel on all cores, one after the other.
- **Co-schedule** represents a two-way processor partitioning of applications \( A_i \) and \( A_j \) on \( r_i \) and \( r_j \) cores, respectively, such that \( r_i + r_j = P \) (total number of cores). \( A_i \) and \( A_j \) are executed concurrently.
- **Energy consumption** \( (E_{\text{sched}}) \) of a workload for a given schedule ‘sched’ is the total energy
consumed by the workload.

The efficiency of a parallel application measures how well the application scales with increasing core count. A way to measure efficiency is to compute the speedup of an application on different core counts. Linear speedup indicates that the application is scalable and with high efficiency. For example, Figure 3.2 illustrates the speedup of ten applications from a benchmark suite on two Intel quadcores (total number of cores, \( P = 8 \)). Speedup behavior varies per application. While, \( A_1, A_4, \) and, \( A_5 \) have an almost linear speedups, Applications \( A_3 \) and \( A_6 \) have poor speedups. Note that executing \( A_3 \) on 4 cores is more efficient than running it on 8 cores. Given the wide range of commonly observed speedup behavior, we ask the following questions: will co-scheduling applications improve overall system efficiency? when can we co-schedule applications? The following theorems provide useful theory to answer these questions.

**Theorem 3.1.** Consider \( K \) applications \( \{A_1, A_2, \ldots, A_K\} \) and \( P \) processing cores. Let us assume that all the applications have linear speedups on all core counts (1 through \( P \)). Suppose that the applications \( A_1, A_2 \ldots A_K \) execute simultaneously on \( r_1, r_2 \ldots r_K \) cores respectively, such that \( \sum_{i=1}^{K} r_i = P \). Let the makespan of this configuration be \( M_{co} \). Let \( M_{base} \) represent the overall makespan when each application is executed on \( P \) cores sequentially. Then, \( M_{co} \geq M_{base} \).

**Proof.** Since all the applications have linear speedups, \( T(i, r_i) = \frac{T(i, 1)}{r_i} \) for \( i \in \{1..K\} \). When all the applications are co-scheduled simultaneously, the makespan of this configuration is given by the execution time of the longest running application. Hence,

\[
M_{co} = \text{MAX} \left\{ \frac{T(1, 1)}{r_1}, \frac{T(2, 1)}{r_2}, \ldots \frac{T(K, 1)}{r_K} \right\} \tag{3.1}
\]

By definition,

\[
M_{base} = \frac{\sum T(i, 1)}{P} \tag{3.2}
\]

Let us assume without loss of generality that \( M_{co} = \frac{T(1, 1)}{r_1} \). This implies,

\[
\begin{align*}
\frac{T(1, 1)}{r_1} &\geq \frac{T(2, 1)}{r_2} \Rightarrow \frac{T(1, 1) r_2}{r_1} \geq T(2, 1) \\
\frac{T(1, 1)}{r_1} &\geq \frac{T(3, 1)}{r_3} \Rightarrow \frac{T(1, 1) r_3}{r_1} \geq T(3, 1) \\
&\vdots
\end{align*}
\]
\[
\frac{T(1,1)}{r_1} \geq \frac{T(K,1)}{r_K} \Rightarrow \frac{T(1,1) \ast r_K}{r_1} \geq T(K,1)
\]

Also, we have, \( \frac{T(1,1) \ast r_1}{r_1} \geq T(1,1) \). Adding these inequalities, we get

\[
\sum \frac{T(1,1) \ast r_i}{r_1} \geq \sum T(i,1)
\]

\[
\Rightarrow \frac{T(1,1)}{r_1} \sum r_i \geq \sum T(i,1)
\]

\[
\Rightarrow \frac{T(1,1) \ast P}{r_1} \geq \sum T(i,1)
\]

Hence,

\[
r_1 \leq \frac{T(1,1) \ast P}{\sum T(i,1)} \quad (3.3)
\]

Let us assume, \( M_{co} < M_{base} \). Hence, \( \frac{T(1,1)}{r_1} < \frac{\sum T(i,1)}{P} \). This implies,

\[
r_1 > \frac{T(1,1) \ast P}{\sum T(i,1)} \quad (3.4)
\]

Equations 3.3 and 3.4 contradict each other. Hence, \( M_{co} \geq M_{base} \) (proof by contradiction).

\[\square\]

Theorem 3.1 shows that a K-way processor partitioning of arbitrary size cannot perform better than running each application one at a time on all cores when each application in the workload has a linear speedup. However, due to Amdahl’s law effects, many scientific applications have sublinear speedups when the processor count increases. The following theorem considers such applications.

**Theorem 3.2.** Consider K applications \( \{A_1, A_2, ..., A_K\} \) and P processing cores. Let us assume that the speedup of an application depends on the number of cores it is executing on. Suppose that the applications \( A_1, A_2, ..., A_K \) execute simultaneously on \( r_1, r_2, ..., r_K \) cores respectively, such that \( \sum_{i=1}^{K} r_i = P \). Let the makespan of this configuration be \( M_{co} \). Let \( M_{base} \) represent the overall makespan when each application is executed on \( P \) cores sequentially. If \( M_{co} \leq M_{base} \), then, \( \frac{T(1,1) \ast C(1,r_1) \ast P}{\sum T(i,1) \ast C(i,P)} \leq r_1 \leq \frac{T(K,1) \ast C(K,r_K) \ast P}{\sum T(i,1) \ast C(i,r_i)} \), where, \( C(i,r_i) \) is a factor indicating the sub-linear speedup of \( A_i \) on \( r_i \) cores and is greater than 1.

**Proof.** It is given that the speedup of an application depends on the number of cores it uses. Let \( T(i,r_i) = \frac{T(i,1) \ast C(i,r_i)}{r_i} \) for \( i \in [1..K] \).

\[
M_{co} = \text{MAX}\left\{ \frac{T(1,1) \ast C(1,r_1)}{r_1}, ..., \frac{T(K,1) \ast C(K,r_K)}{r_K} \right\} \quad (3.5)
\]
Similarly,

\[ M_{\text{base}} = \frac{\sum T(i,1) \times C(i,P)}{P} \]  

(3.6)

Let us assume without loss of generality that

\[ M_{\text{co}} = \frac{T(1,1) \times C(1,r_1)}{r_1} \]. Therefore, from definition of \( M_{\text{co}} \),

\[
\frac{T(1,1) \times C(1,r_1)}{r_1} \geq \frac{T(2,1) \times C(2,r_2)}{r_2}
\]

\[
\Rightarrow \frac{T(1,1) \times C(1,r_1) \times r_2}{r_1} \geq T(2,1) \times C(2,r_2)
\]

\[
\frac{T(1,1) \times C(1,r_1)}{r_1} \geq \frac{T(3,1) \times C(3,r_3)}{r_3}
\]

\[
\Rightarrow \frac{T(1,1) \times C(1,r_1) \times r_3}{r_1} \geq T(3,1) \times C(3,r_3)
\]

\[ : \]

\[
\frac{T(1,1) \times C(1,r_1)}{r_1} \geq \frac{T(K,1) \times C(1,r_1)}{r_K}
\]

\[
\Rightarrow \frac{T(1,1) \times C(1,r_1) \times r_K}{r_1} \geq T(K,1) \times C(K,r_K)
\]

Also, we have,

\[
\frac{T(1,1) \times C(1,r_1)}{r_1} \geq T(1,1) \times C(1,r_1).
\]

Adding these inequalities, we get

\[
\sum \frac{T(1,1) \times C(1,r_1)}{r_1} \geq \sum T(i,1) \times C(i,r_i)
\]

\[
\Rightarrow \frac{T(1,1) \times C(1,r_1)}{r_1} \sum r_i \geq \sum T(i,1) \times C(i,r_i)
\]

\[
\Rightarrow \frac{T(1,1) \times C(1,r_1) \times P}{r_1} \geq \sum T(i,1) \times C(i,r_i)
\]

\[
\Rightarrow r_1 \leq \frac{T(1,1) \times C(1,r_1) \times P}{\sum T(i,1) \times C(i,r_i)}
\]  

(3.7)

As \( M_{co} \leq M_{base} \), we have,

\[
\frac{T(1,1) \times C(1,r_1)}{r_1} \leq \frac{\sum T(i,1) \times C(i,P)}{P}
\]

This implies,

\[
\Rightarrow r_1 \geq \frac{T(1,1) \times C(1,r_1) \times P}{\sum T(i,1) \times C(i,P)}
\]  

(3.8)

Hence, from Equations 3.7 and 3.8, we have,
\[
\frac{T(1,1)\cdot C(1,r_1)\cdot P}{\sum T(i,1)\cdot C(i,P)} \leq r_1 \leq \frac{T(1,1)\cdot C(1,r_1)\cdot P}{\sum T(i,1)\cdot C(i,r_i)}.
\]

**Significance of analysis.** The theorems show that application scaling can play an important role in making scheduling decisions. Theorem 3.2 proves that a sub-linear speedup regime provides an opportunity to improve system efficiency by exploiting speedup profiles. To corroborate Theorem 3.2, we conduct a simple test with two applications on a 8-core system. We exhaustively run each application on all core counts to obtain speedup-profiles (illustrated in Figure 3.3(a)). Observe that the applications show sub-linear speedups and the gap between the actual and ideal speedup increases with the core count. From Theorem 3.2, these applications can be co-scheduled to have a lower makespan than the base schedule. Figure 3.3(b) illustrates the presence of such co-schedules and they satisfy the following:

\[
\frac{T(1,1)\cdot C(1,r_1)\cdot P}{\sum T(i,1)\cdot C(i,P)} \leq r_1 \leq \frac{T(1,1)\cdot C(1,r_1)\cdot P}{\sum T(i,1)\cdot C(i,r_i)}.
\]

For our test, \( r_1 \in \{3, 4, 5, 6\} \) satisfies the inequalities. Note that some schedules are inefficient compared to the base case and these do not satisfy the above inequalities.

![Figure 3.3: (a) Speedup-profiles of two applications showing deviation from the linear speedup. Note that the gap between the ideal and actual speedup increases with core count. (b) Makespan of two applications with different schedules. The black line is the makespan when each application is run on all cores individually (T(1,8) + T(2,8)).](image)

The previous analysis considered the impact of co-scheduling on completion time (makespan) of a workload. The following theorem considers the energy impact of co-scheduling applications with sublinear speedups.
**Theorem 3.3.** Consider the same setup as 3.2. Additionally, let $W_i$ be the average electrical power consumed in watts by $A_i$ per core. If $\frac{C(i,r_i)}{C(i,P)} < 1 \forall i$, then $E_{co} < E_{base}$, where $E_{co}$ and $E_{base}$ represent energy consumed by the co-schedule and base configurations, respectively.

**Proof.** We know that $energy = time \times power$. The total energy consumed by the base configuration is given by $E_{base} = \sum \{ \frac{T(i,1)}{P} \times C(i,P) \times P \times W_i \}$, equivalently,

$$E_{base} = \sum T(i,1) \times C(i,P) \times W_i \quad (3.9)$$

Similarly, the total energy consumed by the co-scheduled configuration is

$$E_{co} = \sum T(i,1) \times C(i,r_i) \times W_i \quad (3.10)$$

Let us consider the following terms: $T(i,1) \times C(i,P) \times W_i$ and $T(i,1) \times C(i,r_i) \times W_i$. If $\frac{C(i,r_i)}{C(i,P)} < 1$, then $T(i,1) \times C(i,r_i) \times W_i < T(i,1) \times C(i,P) \times W_i$. This holds for each application. Hence, when we sum all the inequalities, we have

$$\sum T(i,1) \times C(i,r_i) \times W_i < \sum T(i,1) \times C(i,P) \times W_i \quad (3.11)$$

Thus, $E_{co} < E_{base}$. □

Theorems 3.1, 3.2 and 3.3 provide insights into the presence of potential schedules for improving the overall system efficiency by reducing makespan. Figure 3.3(b) illustrates such schedules. However, they raise three important concerns: 1) as we use exhaustive experimentation to obtain application speedup-profile, is it a practical approach for a system with a large number of applications and hundreds of processors? 2) how do we choose the number of applications ($K$) to be co-scheduled? and 3) how do we choose the applications that form a co-schedule? We address these concerns in the following sections.

**Role of sampling to approximate speedup profile.** The availability of the speedup-profile for an application means that the application was executed for all core counts. For large numbers of applications and processors ($P$), the brute force combinatorial approach becomes infeasible. We know that parallel applications have a near-linear speedup up to a certain number of processing cores, before the performance flattens out (Amdahl’s law [47]). Hence, the speedup plot for an application is usually a smooth increasing curve that flattens out as $P$ increases. Thus, it is possible to construct an approximate speedup plot from a sampling set consisting of application runs on a subset of core counts. For example, Figure 3.4 shows the actual speedup and curvefit-speedup plots for two applications.
In these plots, the blue curve represents the actual speedup, the red curve represents a curvefit-speedup plot given 3 sample points (1, 4 and 8 cores) and the green curve represents a curvefit-speedup plot given 4 sample points (1, 4, 7 and 8 cores). Observe that we require only 3 sample points to fit $App_1$ (Figure 3.4(a)), while 4 sample points fit $App_2$ (Figure 3.4(b)) better than 3 sample points. So, depending on the application characteristics, a variable number of sample points are required to get a close approximation to the actual speedup plots.

**Number of applications in a co-schedule.** In this chapter, we co-schedule two applications (two-way partitioning) at a time for the sake of simplicity in analysis and experimentation. We can extend our methodology for $K > 2$ with minimal changes using hierarchical mapping. For example, for a co-schedule of four applications on $P$ cores, we can split the applications into two sets of two applications each and co-schedule each set on $\frac{P}{2}$ cores. We would like to emphasize that $K = 2$ yields substantial performance benefits over the base case and the benefits could potentially increase with hierarchical mapping.

We address the last concern related to the choice of applications that form a co-schedule in the following section.

### 3.3.1 Construction of an optimal schedule for two-way processor partitioning

We present the construction of an optimal schedule for speedup-aware processor partitioning scheme in this section. For simplicity of analysis, we assume that the interference due to the co-scheduled applications is negligible. The optimal schedule for a two-way processor
A partitioning scheme consists of two main steps: 1) obtain a set of best pairwise schedule for each pair of applications and 2) use the “set” from step 1 to obtain the optimal schedule. Before delving into the construction, we define a few well-known graph related terms and that are used in our algorithm.

**Definition 3.1.** A matching on a graph is a set of edges of the graph such that no two of them have a common vertex.

**Definition 3.2.** A perfect matching on a graph is a matching that covers all vertices in the graph.

**Definition 3.3.** A minimum weight perfect matching (MWPM) on an edge-weighted graph is a perfect matching that has minimum edge weight sum.

Let \( S \) and \( S^* \) represent an arbitrary and an optimal schedule, respectively, for a two-way processor partitioning scheme. Let \( \omega(i, j) \) and \( \omega^*(i, j) \) represent makespans of an arbitrary co-schedule and an optimal co-schedule of applications \( A_i \) and \( A_j \), respectively. Then, \( \omega(i, j) = \max \{ T(i, r_i), T(j, r_j) \} \), such that \( r_i + r_j = P \) and \( r_i, r_j \geq 1 \) and \( \omega^*(i, j) = \min \{ \max \{ T(i, r_i), T(j, r_j) \} \} \), \( \forall r_i, r_j \), such that \( r_i + r_j = P \) and \( r_i, r_j \geq 1 \).

Consider a set of \( K \) applications, \( A = \{ A_1, A_2, \ldots, A_K \} \). Let \( G(V, E, W) \) represent a graph with \( V \) vertices, \( E \) edges and \( W \) edge weights, such that \( V = \{ v_i : i \in \{ 1..K \} \} \), \( E = \{ e_{ij} = (v_i, v_j) : v_i, v_j \in V, i \neq j \} \) and \( W = \{ \omega^*(i, j) : e_{ij} \in E \} \). Now, consider in \( G \) a perfect matching \( G \). It defines a schedule \( (S_G) \) wherein applications are co-scheduled (application pair at a time). Then, \( M_{S_G} = \sum \omega^*(i, j), \forall (i, j) \in G \), is the makespan of \( S_G \).

**Theorem 3.4.** The optimal schedule for a two-way processor partitioning scheme, \( S^* \), is defined by a perfect matching of minimal weight \( (G^*) \), such that \( M_{S^*} = \min_{G^*} \{ \sum \omega^*(i, j), \forall (i, j) \in G \} \).

**Proof.** Let us assume that there is a schedule, \( S \), that has a makespan, \( M_S \), such that, \( M_S < M_{S^*} \). As each schedule is defined by a matching and \( M_{S^*} \) is the minimum makespan over all matchings, the schedule, \( S \), is possible if there exists (\( i, j \)) such that \( \omega(i, j) < \omega^*(i, j) \). However, by definition, \( \omega^*(i, j) \) is the minimum makespan for application pair \( (A_i, A_j) \) co-schedule. Hence, \( \omega(i, j) \geq \omega^*(i, j), \forall (i, j) \). Thus, we have a contradiction. This implies that \( S^* \) is an optimal schedule for a two-way processor partitioning scheme.

Theorem 3.4 proves that we can obtain a schedule with minimum makespan by formulating the schedule problem as a minimum weight perfect matching problem. However, it does not provide details of how applications are co-scheduled, i.e., if applications \( A_i \) and
Figure 3.5: An example of the optimal schedule for a two-way processor partitioning scheme for set a 6 applications on 8 cores. (a) a set of best pairwise schedule for each pair of applications, (b) graph formulation of the set from (a) and (c) minimum weight perfect matching co-schedule. Note that A(i,j) represents Application ‘i’ running with ‘j’ threads.

When A_j are co-scheduled on P cores, how many threads are allocated to each application? This is achieved by performing extra book-keeping in step 1. We augment each edge in G with the thread configuration of each application.

Figure 3.5 shows an example of obtaining an optimal schedule for a two-way processor partitioning scheme on a set of 6 randomly chosen benchmark applications. Note that A(i,j) represents Application ‘i’ running with ‘j’ threads. A configuration represents a co-schedule and there are 105 ways to co-schedule given a set of 6 applications and 8 cores.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sProf</td>
<td>Speedup profile of all the applications</td>
</tr>
<tr>
<td>K</td>
<td>Total number of applications</td>
</tr>
<tr>
<td>P</td>
<td>Total number of cores</td>
</tr>
<tr>
<td>minSet ((\omega))</td>
<td>Contains a set of pairwise predicted best schedule across all applications in the given set. Each tuple in the set consists of: Makespan, application pair, thread counts</td>
</tr>
<tr>
<td>temp, minVar, maxVar</td>
<td>Temporary variables</td>
</tr>
<tr>
<td>schedule</td>
<td>Contains the predicted best schedule with the corresponding threads per application</td>
</tr>
<tr>
<td>appSet</td>
<td>Contains a set of application IDs</td>
</tr>
</tbody>
</table>

Table 3.1: A list of variables used in Algorithm 3.1 and their descriptions.
Algorithm 3.1 SAPP-G Algorithm

procedure SAPP-G(sProf, K, P)
1: minSet = ∅
2: temp =<>
3: minVar = Inf
4: maxVar = −Inf
5: schedule = ∅
6: appSet = {1..K}
7: for i ∈ (1,K) do
8: sum = 0.0
9: for j ∈ (i+1,K) do  
10: maxVar = max(sProf(i,l),sProf(j,P−l))  
11: if maxVar ≤ minVar then
12: minVar = maxVar
13: temp =< minVar, i, j, l, P−l >
14: end if
15: end for
16: end for
17: minSet = minSet ∪ temp
18: end for
19: minSet = sort(minSet,1)
20: while appSet ̸= ∅ do
21: schedule = schedule ∪ minSet(1)
22: appSet = appSet − {minSet(1)(2),minSet(1)(3)}
23: minSet = minSet − minSet(1)
24: end while

3.3.2 Heuristics

We define the scheduling technique using minimum weight perfect matching as Speedup-Aware Processor Partitioning-Optimal (SAPP-O). Although, SAPP-O gives an ideal schedule for pairwise application combination, the computational overhead is high. The minimum weight perfect matching algorithm (blossom) by Edmonds [68] has time complexity of O(mn²), where m and n are the total numbers of edges and nodes in the graph. In our formulation, n = K and m = \( \binom{K}{2} \). Thus, the complexity of blossom is O(K⁴). Hence, we use a simple sorting based greedy heuristic to predict a schedule for pairwise application combination with reduced makespan, we call this, Speedup-Aware Processor Partitioning-Greedy (SAPP-G). Algorithm 3.1 gives the pseudo-code of SAPP-G and Table 3.1 describes the variables used in the algorithm. In Algorithm 3.1, the code segment between lines 7-19 is used to obtain the set of edge weights, \( \omega \), called minSet in the algorithm. This takes
O(PK²) time. The code segments between lines 20-25 represents the sorting heuristics used to find a schedule. The time complexity of this heuristics is O(K²) assuming makespans are rounded to the nearest integer values. Observe that the variable schedule contains the final predicted schedule. This includes, makespan, pairwise application combinations and their corresponding processor (thread) counts. To obtain the overall makespan, we sum all the pairwise makespans in schedule. Although one can use SAPP-O to obtain ideal schedule, we show in Section 5.4 that SAPP-G and SAPP-O give similar results. Note that for a given workload and the speedup-profiles of applications in the workload, we need to run SAPP-O or SAPP-G only once to get the right schedule. Hence, the overhead of using these algorithms is minimal for long running applications.

Algorithm 3.2 EPP-G Algorithm

```plaintext
procedure EPP-G(sProf, K, P)
1: gSet = ∅
2: schedule = ∅
3: appSet = {1..K}
4: for i ∈ (1, K) do
5:    gSet = gSet ∪ < sProf(i, P/2), i >
6: end for
7: gSet = sort(gSet, 1)
8: while appSet ≠ ∅ do
9:    schedule = schedule ∪ < gSet(1), gSet(2) >
10:   appSet = appSet − {gSet(1)(2), gSet(2)(2)}
11:   gSet = gSet − {gSet(1), gSet(2)}
12: end while
```

Algorithm 3.2 gives the pseudo-code of equal processor partitioning-greedy (EPP-G) algorithm. In this algorithm, we divide the total number of processor cores (P) equally among the application pair. For a given set of applications, we first sort the P/2-core execution times of the applications in a non-increasing order. Next, we co-schedule the adjacent applications in the sorted list. This is a fast way of choosing a schedule. We show that even this simple algorithm reduces the overall makespan when compared to a random schedule.

3.4 Experimental Evaluation

We present experimental setup in Section 3.4.1 and provide a detailed evaluation of our schemes on a suite of parallel benchmark applications in Section 3.4.2. We assess the practical significance of the schemes using commercial parallel scientific applications on upto 128 cores in Section 3.4.3.
3.4.1 Experimental Setup

We use a two-step experimental evaluation process. In the first step, we perform experiments on a node consisting of a two-socket, 8-core Intel Nehalem multicore [44] using ten benchmark applications from Parsec benchmark suite [69, 70]. The Parsec suite comes with different input sets. As our experiments are on a real hardware, we use the largest input set (native). We do not use the benchmarks, fluidanimate, facesim and freqmine. The fluidanimate benchmark requires the number of threads to be a power of two. The benchmarks facesim and freqmine do not run for certain thread counts. In the second step of the evaluation process, we scale upto 128 cores (16 compute nodes) and use commercial parallel scientific applications such as VASP [71, 72], ABAQUS [73], and LAMMPS [74]. These applications are used in the study of molecular dynamics (VASP and LAMMPS) and finite element analysis (ABAQUS).

<table>
<thead>
<tr>
<th>Scheduling Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPP-Random</td>
<td>Schedules two applications at a time Co-schedule pair selection is random Applications get equal number of cores</td>
</tr>
<tr>
<td>EPP-G</td>
<td>Schedules two applications at a time Greedy co-schedule pair selection Applications get equal number of cores</td>
</tr>
<tr>
<td>SAPP-G</td>
<td>Schedules two applications at a time Greedy SAPP co-schedule pair selection Applications get cores based on the curvefit speedup-profile</td>
</tr>
<tr>
<td>SAPP-O</td>
<td>Schedules two applications at a time MWPM co-schedule pair selection Applications get cores based on the curvefit speedup-profile</td>
</tr>
</tbody>
</table>

Table 3.2: Different scheduling techniques and their descriptions.

Experimental Methodology. For a given set of applications, our base configuration \((B\_Sched)\) involves running each application one at a time on all the available cores. Thus, the makespan of \(B\_Sched\) is the time elapsed between start time of the first application and end time of the last application. We evaluate four scheduling techniques, equal processor partitioning - random (EPP-Random), equal processor partitioning - greedy (EPP-G), SAPP-G and SAPP-O. Table 3.2 gives a brief description of these techniques. We obtain energy related statistics using the WattsUp Pro power meter connected to a compute node.
We sample the data from the meter every second. For experiments spanning multiple nodes, we compute the energy consumed at a node and estimate the total energy consumption across all nodes.

**Impact of incomplete data.** How does curvefit speedup-profile based schedules compare to those obtained using the actual speedup-profile? To answer this, we conduct experiments using curvefit as well as actual speedup-profile based schedules. Our results show that the maximum gains across all workloads using the actual speedup-profile based schedules is a mere 1.6% over the curvefit speedup-profile based schedules, indicating the similarity between schedules predicted using the actual and curvefit speedup-profiles. Hence, using partial data for speedup-profile construction can potentially give accurate makespan estimate and the experimental evaluation in Sections 3.4.2 and 3.4.3 are based on curvefit speedup-profile data.

### 3.4.2 Node-Level Evaluation

In this section, we present a detailed analysis of the node-level experimental results. We evaluate our techniques on application sets of sizes 4 and 6, wherein, the applications within each set are selected at random. To have a manageable experimentation time, we choose 10 different application sets such that all 10 benchmark applications are covered for each application set size (4 and 6).

**Impact on performance and waiting time.** Figures 3.6(a) and 3.6(b) show the makespan reduction due to the predicted schedules for different application set sizes when compared to the base case. Note that application schedules for all schemes are predicted using the curvefit speedup-profile. Observe that the performance trend across the scheduling techniques is very similar. While EPP-Random is the least effective method, SAPP-G performs the best. On average, SAPP-G reduces makespan by 24.9% and by as much as 31.6% compared to the base case. EPP-G shows lower benefits than SAPP-G (on average, 11.2%) and EPP-Random increases makespan marginally compared to the base case (-1.8%). Additionally, our schemes, EPP-G and SAPP-G, reduce the average waiting time compared to the base case. We define average waiting time as $T_{\text{avgw}} = \frac{\sum_{i=1}^{K} T_w(A_i)}{K}$, where $T_w(A_i)$ is the waiting time of Application $A_i$ and ‘$K$’ is the total number of applications. Our results indicate that EPP-G and SAPP-G reduce average waiting time by 32.5% and 55.6% in the case of workloads with 4 applications, and by 25.9% and 37.3% in the case of workloads with 6 applications, respectively.

**Impact on energy consumption.** Figures 3.6(c) and 3.6(d) show that EPP-G and
SAPP-G reduce energy consumption of the workloads compared to the base case, on average by 6.5% and 17.4%, respectively, across all workloads. Observe that the decrease in energy consumption of our schemes is directly correlated to the corresponding reduction in makespan (see Figures 3.6(a) and 3.6(b)).

**SAPP-G vs SAPP-O.** From the previous evaluation, we observe that SAPP-G finds better schedules (i.e., schedules that reduce makespan) compared to other schemes. Now we evaluate SAPP-G with respect to SAPP-O. From Section 3.3.1, we know that SAPP-O finds an ideal pairwise schedule for a given set of applications. Figure 3.7 shows a comparison between the SAPP-G and SAPP-O. Note that, although there are 10 workloads in each set size, the plot shows only a subset, as both schemes, SAPP-G and SAPP-O, give the same schedule for the remaining workloads. Figure 3.7 shows that schedules predicted by the two schemes have close makespans, varying within 4% of each other. Also note that SAPP-G predicted schedules perform better in some cases (combination 1 in 4-set...
and combinations 2, 4 and 7 in 6-set). Does it mean that SAPP-O predicts sub-optimal schedules? The answer is no; the predicted schedules are based on a curvefit speedup-profile, hence there could be some discrepancies. Note that in the cases where SAPP-G does well, the percentage reduction in makespan is a mere 2%-4% and only 4 out of 20 combinations show such discrepancy.

3.4.3 Large-Scale Evaluation

We evaluate the efficacy of our schemes on large-scale systems of up to 128 cores that consists of 16 computing nodes, each with 8 cores. We use well-known commercial scientific applications, VASP, ABAQUS and LAMMPS - each with two different inputs, effectively providing 6 applications for our study, and each workload consists of 4 applications chosen at random. We utilize the sampling described earlier (4 sample points: 8, 32, 64 and 128 cores) to construct speedup-profiles of the applications and use these speedup-profiles to predict schedules. Also note that for large-scale evaluation, we do not partition the cores within a node, i.e., applications are scheduled on core count of multiples of 8.

**Impact on performance and waiting time.** Figures 3.8(a) and 3.8(b) show the performance benefits of using our schemes on 64 and 128 cores systems compared to the base case. Note that we do not plot the SAPP-O scheme in these figures as reduction in makespan using SAPP-G is the same as that of SAPP-O. On 64 cores, SAPP-G consistently performs better than the base case, whereas, EPP-Random and EPP-G perform poorly. As
Figure 3.8: Benefits of our schemes for large-scale systems. (a) and (b) show reduction in makespan for 10 different workloads of 4 applications each. (c) and (d) show the energy benefits of using our schemes on the same set of workloads.

The equal processor partitioning schemes do not consider application speedup characteristics across different core counts, they tend to have poor performance when the pairwise co-scheduled applications have very different execution times or have not hit Amdahl’s knee. On 128 cores, 4 out of 6 applications hit Amdahl’s knee, thus as expected, all schemes perform better than the base case across workloads (see Figure 3.8(b)) and the reduction in makespan can be as high as 49.3%. It is interesting to note that even a naive scheme like EPP-Random outperforms the base case, indicating the importance and need for processor partitioning schemes when applications with sublinear speedups are scheduled at large core counts. Further, our schemes reduce the average waiting time of the applications significantly compared to the base case. On 128 cores, EPP-Random, EPP-G and SAPP-G reduce the average waiting time by 58.2%, 87.6% and 89.6%, respectively.

**Impact on energy consumption.** Figures 3.8(c) and 3.8(d) show processor partitioning schemes significantly reduce total energy consumption for large core counts compared...
to the base case. On 128 cores, EPP-Random, EPP-G and SAPP-G reduce energy consumption by as much as 58.0%, 58.0% and 57.0%, respectively. Observe that, although EPP-Random and EPP-G can lead to different makespans for a workload, their total energy consumptions are the same. The reason for this counter-intuitive result is the fact that the total energy consumption of a workload is purely based on three factors: 1) the number of cores used by each application, 2) energy profile of each application and 3) execution time of each application, within the workload. These three factors remain the same for EPP-Random and EPP-G, so the energy consumptions are the same. However, depending on how the applications are co-scheduled within the workload, makespan can change. We observe that energy consumptions of the EPP schemes are lower than SAPP-G. This can be attributed to 1) the variable sized processor partitioning scheme used by SAPP-G that could potentially allocate more processors to the longer running application within a co-schedule and 2) reduction in the idle processor time within a co-schedule. As we use pairwise co-schedule, it is possible that one of the applications completes its execution much before the other, resulting in idle processor time (for processors allocated to the faster application). We observe that EPP schemes have more idle processor time than SAPP-G. Since we assume energy consumption of such idle processors is minuscule, EPP schemes consume lower energy than SAPP-G.

### 3.5 Conclusion

In current and future HPC installations, typical workloads will likely comprise individual applications that do not speedup linearly with increasing numbers of cores due to the effects of Amdahl’s law. For such a situation we develop speedup-aware processor partitioning schemes that can be utilized within a workload management framework such as PBS [48] and LoadLeveler [49] to significantly enhance system performance and energy efficiencies, and reduce average waiting time.

We propose optimal and greedy formulations of SAPP and EPP schemes for two way partitions of the available number of cores (extensible to multiway partitions). We evaluate our schemes from a single compute node with 8 cores to 16 compute nodes with 128 cores. Our results indicate that the SAPP schedules reduce makespan by as much as 31.5% on a 8-core node. Additionally, on 128 cores, makespan is reduced by as much as 49.3%. Furthermore, our evaluations show significant reductions in terms of average waiting time per application and total energy consumption per workload. Our work shows performance and energy benefits of using proactive application-aware workload management techniques.
and is a step toward maintaining large-scale system efficiency.
Chapter 4

Enhancing Resilience: Characterizing the Impact of Transient Errors on Iterative Methods

4.1 Introduction

In the previous chapters, we discussed algorithmic approaches to improve speedup, throughput, and energy consumption measures for sparse scientific computing. In this chapter, we analyze the impact of transient errors on the resiliency of sparse scientific computing, especially, the resiliency of sparse linear system solution. This chapter has previously appeared in proceedings of the 25-th International Conference on Supercomputing.

A current trend among chip manufacturers is to increase transistor densities on the chip using small feature sizes. This facilitates achieving higher performance, at the expense of greater soft error susceptibility. Increasing soft errors will increase intermittent error rate by almost two orders of magnitude. Consequently, the scientific simulation applications that comprise the majority of long running workloads on large scale high performance computing installations are at a greater risk of being hit by a soft error. In this chapter, we seek to assess the vulnerability of such scientific applications that are often based on the solution of partial differential equation (PDE) models through explicit and implicit schemes.

Soft errors can be seen as transient errors that result in bit flips in memory and errors
in logic circuit output, leaving the computing system state corrupt. The errors can be caused by cosmic radiations [81], radiation from packaging materials [82], as well as voltage fluctuation [83]. Although the increase in transistor density facilitates higher performance, it also leads to a higher soft error susceptibility. In fact, with increases in the transistor densities the soft error rates have been growing exponentially, with typical values ranging between 1k and 10k FIT/Mb [82] (FIT is failure per billion hours of operation). The 106,496 dual-processor compute node BlueGene/L, for example [84], experiences one soft error in its L1 cache every 4-6 hours induced by the radioactive decay in lead solders. Michalak et al. [85] report that the ASCI Q experienced 27.7 CPU failures per week due to radiation. These high soft error rates are a cause of concern for long running scientific applications as it could potentially degrade the performance or produce erroneous solutions.

A majority of large-scale scientific applications including those that run on multicores and their multiprocessor clusters (supercomputers) involve solving PDE based systems, such as those found in heat diffusion, computational fluid dynamics (CFD) and structural mechanics [6]. Such applications typically use software tool kits such as PETSc [79], for solutions using explicit and implicit schemes; with the latter requiring the use of solvers packages such as hypre [86] and Trinilos [87, 88]. At the lower levels, the basic underlying computation in such methods is a sequence of SpMV operations of the form \( t \leftarrow A \cdot y \) that are performed iteratively. The explicit schemes may directly use SpMV operations to find the solution across different time steps, while in the implicit methods, SpMV operation is often within an iterative sparse linear solver such as conjugate gradient (CG) and its preconditioned forms. Across all cases, reliable and accurate fast simulations can be viewed as depending on the performance of the SpMV kernel and the propagation of numerical attributes through relevant sequence of SpMV operations [80]. In this chapter, we seek to characterize the impact of soft errors on such PDE-based applications by an in-depth analysis of soft error propagation through series of underlying SpMV operations.

The earlier studies related to soft errors in scientific applications focus on error detection and mitigation of their effect using algorithmic techniques [84], and, compiler and hardware based soft error prevention techniques [89]. Unlike these studies, we focus on theoretical analysis and characterization of the impact of soft errors on scientific simulations. We show that a single soft error is capable of causing multiple errors, and this propagation has a pattern corresponding to the sparsity structure of the coefficient matrix. In the case of an explicit method, we show that the growing magnitude of the error makes the final solution inviable. For the implicit methods, we show that, although, it may converge to a viable solution, the total iterations required for the convergence may increase significantly leading
to performance degradation. Additionally, we prove that the magnitude of the error grows non-linearly with the 2-norm of the matrix rows. Further, we provide empirical evaluation of performance degradation in terms of the relative increase in iteration count due to the presence of a soft error. Our results indicate that for practical problems, a single soft error could potentially degrade the preconditioned conjugate gradient (PCG) performance by as much as 200x compared to soft error free PCG performance.

The chapter is organized as follows. In Section 5.2, we discuss our work in the context of recent results related to the detection and mitigation of soft errors in sparse linear solvers, [84, 89]. In Section 4.3, we use the heat equation as an example to indicate the pivotal role of SpMV operations in the explicit and implicit solution of PDE-based models. In Section 4.4, we provide an analysis on the propagation and growth of a single soft error through a sequence of SpMV operations. In Section 4.5, we present our experimental methodology and the results of empirical tests in support our theoretical analysis. Additionally, we provide an in-depth evaluation of the practical impact of soft errors on implicit schemes using popular solvers such as PCG and successive over relaxation (SOR). In Section 5.5, we provide brief concluding remarks and motivate potential directions for future work.

4.2 Related Work

Prior work considers dealing with soft errors in silicon as well as at the application level. We can categorize the earlier results into three directions: soft error detection [76, 77, 90, 91], soft error protection [92, 93, 94, 95], and soft error characterization in microarchitecture [96],[97],[98]. Smolens et al. [77] propose an error detection technique, called fingerprinting, which detects differences in execution across a dual modular redundant processor pair. This technique requires special hardware modifications to the processor pipeline. On the other hand, to avoid any hardware overhead, software based techniques have been developed to detect soft errors [90], [91]. For example, Rebaudengo et al. [90] propose a technique to automatically transform programs written in a high-level language so that they can detect most of the errors affecting data and code. The scheme proposed by Hu et al.[91] enlists the compiler’s help in duplicating instructions for error detection in VLIW datapaths. It is further supported by a hardware enhancement for efficient result verification, which also avoids the need of additional comparison instructions. In their proposed approach, the tradeoff between performance, reliability and energy consumption is achieved using the compiler by determining degree of instruction duplication.

Recently there has been interest in understanding the effect of soft errors on sparse
linear solvers. Bronevetsky et al. [84] report observations on the effect of soft errors on iterative solvers like, CG, preconditioned Richardson, and Chebyshev methods in SparseLib [99]. Based on the experimental observations, the paper groups the impact of soft errors as (i) no effect, (ii) silent error, (iii) application hangs, and (iv) application crashes. The paper also proposes and evaluates several soft error detection and tolerance schemes, like, residual tracking, checkpointing and data structure encoding that could potentially lead to significant improvements in the reliability of these libraries. Malkowski et al. [89] consider PCG and GMRES and they focus on utilizing the manner in which the coefficient matrix $A$ is resident in L1 and L2 caches in these methods. They use the concept of vulnerable time to propose and evaluate energy and reliability tradeoffs for two schemes, for adaptively turn-off the Error Correction Code (ECC) for L1 cache and L2 caches. They assume little or no cache reuse for the vector $v$ and therefore is not protected. The data structures having higher resident time in the caches than their corresponding vulnerable time, are protected.

Our proposed work differs from the earlier work as follows: first, we focus on a sequence of SpMV operations as the common core of PDE-based simulations using explicit or implicit schemes. Second, we provide an in-depth analysis of the propagation of a single soft error through a sequence of SpMV operations in terms of the sparsity structure of the coefficient matrix $A$ and growth of this error in terms of the numerical properties of $A$. Third, we relate this analysis to the stability of explicit schemes and the convergence of iterative solvers in an implicit scheme. We thus characterize algorithmic aspects of computational schemes for such large scale applications, to motivate the need for soft error resilient approaches in software libraries, such as, PETSc [79], Trilinos [87, 88], hypre [86], SparseLib [99].

4.3 Background and the role of SpMV

In this section, we use the heat equation [100] to provide a brief overview of explicit and implicit methods with a focus on the central role of a sequence of SpMV operations. PDE-based simulations play a vital role in many problems, ranging from astrophysics to material science and there are many complex underlying issues related to the type of the PDE (like elliptic, parabolic), discretization methods, the choice of explicit or implicit schemes, the non-linearity of the underlying operator among other factors. The material in the section
is highly simplified for illustrative purposes and the reader is referred to the book by Heath[100] for an accessible overview including references to classical text on this very broad topic.

Figure 4.1: One dimensional heat flow problem.

Figure 4.1 shows an example of one dimensional heat flow problem. The x-axis represents discretization of space (a physical object) and the y-axis represents time steps over which the PDE solution evolves. In general, a PDE can be solved using either an explicit or implicit method as reviewed below.

**Explicit Methods.** An explicit method calculates the state of a system ($u$), as a function of time ($t$) and space ($x$), at a later time from the state of the system at the current time. For example, a one dimensional heat equation for an explicit method can be written as

$$u_i^{m+1} = u_i^m + \mu \{u_{i+1}^m - 2u_i^m + u_{i-1}^m\}$$

(4.1)

where the temperature at point $x_i$ at time $t_m$ is denoted by $u_i^m$ and $\mu$ is a constant related to the discretization sizes and the material property. Equation 4.1 can be rewritten as:

$$u^{m+1} = u^m + \mu Au^m$$

(4.2)

where $A$ is a sparse coefficient matrix. Note that solution to every time step involves a SpMV operation of the form $Au_m$.

**Implicit Methods.** Unlike an explicit method, an implicit method finds the solution of a PDE by solving the system of equations involving both the current state of the system and the state at a later time. The following equation shows a one dimensional heat equation
obtained using an implicit method.

\[ u_i^m = (1 + 2\mu)u_{i+1}^{m+1} - \mu \{u_{i+1}^{m+1} + u_{i-1}^{m+1}\} \]  
(4.3)

where the temperature at point \(x_i\) at time \(t_m\) is denoted by \(u_i^m\) and \(\mu\) is a constant related to the discretization sizes and the material property. Equation 4.3 can be reformulated as:

\[ Au^{m+1} = u^m \]  
(4.4)

where \(A\) is a tridiagonal coefficient matrix.

The above methods can be similarly extended to 2D and 3D heat flow problems [80]. The method selection to solve a given PDE depends on the computational overheads and numerical stability tradeoffs. The explicit methods are simple to use and computationally less expensive per time step compared to implicit methods. However, the explicit methods are prone to numerical instabilities if the discretization intervals are not chosen accurately.

**Role of SpMV.** For easy of analysis, we henceforth focus on systems where the matrix \(A\) is sparse, symmetric and positive definite. Such systems represent a large class of widely used PDE-based simulations on large scale high performance computers [6]. Additionally, methods developed for such systems often form the basis for extensions relating to non-symmetric \(A\) [6, 100, 80].

*Explicit Method:* Observe from Equation 4.2 for the explicit method that SpMV propagates \(u^m\) into \(u^{m+1}\) by the factor of \(\mu A\). Consequently, \(u^{m+1}\) directly holds the effects of a sequence of \(m\) SpMV operations.

*Implicit Method:* Equation 4.4 for the implicit method indicates that \(u^{m+1}\) is obtained by a linear system solution involving \(A\) and \(u^m\). Now, a sequence of SpMV operations occur in each linear system solution using popular iterative methods such as Conjugate Gradient (CG) and its preconditioned forms (PCG) or stationary iterative methods such as successive over relaxation (SOR). To illustrate the role of SpMV in such sparse linear solutions consider Algorithm 1. Line 9 in the algorithm represents the SpMV kernel which is executed in every iteration until termination. Observe that an occurrence of soft error in \(v\) leads to an error in the other vectors like \(t\) and \(r\). Since these vectors govern the convergence time of CG, the number of iterations executed by CG may change due to soft errors in \(v\). In most practical settings, the convergence of CG is accelerated through preconditioning; such preconditioned CG methods involve the use of SpMV as in Algorithm 1.

To illustrate the role of SpMV in the SOR method, consider a linear system \(Ax = b\),
Algorithm 4.1 Conjugate Gradient

procedure CG(A, b, MAXIT, TOL)
1: \( x = 0 \)
2: \( iters = 0 \)
3: \( nb = \text{norm}(b) \)
4: \( r = b \)
5: \( rsq = r' \ast r \)
6: \( v = r \)
7: \( \textbf{while} \ \text{iters} < \text{MAXIT} \ \&\& \ \text{sqrt}(rsq)/nb > \text{TOL} \ \textbf{do} \)
8: \( \text{iters} = \text{iters} + 1 \)
9: \( t = \text{SpMV}(A, v) \)
10: \( alpha = \text{rsq} / (v' \ast t) \)
11: \( x = x + alpha \ast v \)
12: \( r = r - alpha \ast t \)
13: \( rsqprev = rsq \)
14: \( rsq = r' \ast r \)
15: \( beta = rsq/rsqprev \)
16: \( v = r + beta \ast v \)
17: \( \textbf{end while} \)

where, \( A \) is an \( N \times N \) coefficient matrix, \( b \) is a known vector and \( x \) is an unknown vector. To solve this system, \( A \) is decomposed into a diagonal (\( D \)), strict lower and upper triangular (\( U, L \)) matrices such that \( A = L + D + U \). The linear system is reformulated as Equation 4.5, that represents SOR iteration.

\[ Mx^{k+1} = Nx^{k} + \omega b \quad (4.5) \]

where, \( M = D + \omega L, N = (1 - \omega)D - \omega U \), \( x^k \) and \( x^{k+1} \) are vectors that represent the solution of the system at time \( t_k \) and \( t_{k+1} \), and \( \omega \) is a fixed search parameter. As indicated in the equation, the SpMV kernel is executed to solve for \( x^{k+1} \).

4.4 Analysis of soft error propagation

In this section, we focus on analytically characterizing the propagation and growth of a single soft error in the vector \( y^0 \) for a sequence of SpMV operations given by \( \{y^0, y^1 \leftarrow Ay^0, \ldots, y^k \leftarrow Ay^{k-1}\} \). Observe that this sequence is equivalent to \( \{y^0, Ay^0, \ldots, A^k y^0\} \). We use a graph formulation that corresponds to the SpMV sequence to capture effects related to sparsity structure of \( A \) (zero-nonzero pattern) and its numeric attributes.

\textbf{Notation.} In the rest of this chapter, matrices are represented by bold uppercase
letters such as $\mathbf{A}$, and vectors are represented by bold lowercase letters such as $\mathbf{y}$. The matrices are assumed to be of size $N \times N$ and vectors are correspondingly $N \times 1$. Additionally, superscripts, such as $\mathbf{y}^j$ are used to represent a vector within the sequence $\{\mathbf{y}^0, \mathbf{y}^1, ..., \mathbf{y}^p\}$ and $\mathbf{y}^k_i$ represent the $i^{th}$ component in the vector $\mathbf{y}^k$. $\mathbf{A}_{i*}$ represents the elements in the $i^{th}$ row of $\mathbf{A}$. $\text{NNZIndex}(\mathbf{x})$ represents the set of indices that correspond to the non-zeroes in the vector $\mathbf{x}$. E.g., if $\mathbf{x} = [0, 0, -, -, 0, -]$ where '−' indicates non-zeroes and $\text{NNZIndex}(\mathbf{x}) = [3, 4, 6]$.

**Graph Representation.** For our analysis, we assume that the coefficient matrix $\mathbf{A}$ has a single connected component. In general, it is observed that many matrices arising from practical problems have a single connected component.

We consider the traditional graph representation $\mathbf{G}(\mathbf{A}) = (\mathbf{V}, \mathbf{E})$ for an $N \times N$ sparse symmetric coefficient matrix $\mathbf{A}$. Now the vertex set $\mathbf{V} = \{v_1, v_N\}$, where $v_i$ represents $i^{th}$ row or column of $\mathbf{A}$; the edge set $\mathbf{E} = \{(v_i v_j) : i \neq j, 1 \leq i, j \leq N, A_{ij} \neq 0\}$, where $A_{ij}$ is an element of $\mathbf{A}$.

Define a level set partitioning of the vertex set $\mathbf{V} = \bigcup_{i=0}^p S_i$, $S_r \cap S_q = \phi, r \neq q, 0 \leq r, q \leq p$, where $S_i$ is the set of vertices in the $i^{th}$ partition. Let $\text{Reach}[S_i]$ represent the set of all the immediate neighbors of the vertices in the set $S_i$. It can be defined as, $\text{Reach}[S_i] = \bar{S}_i = \{v_j | v_j \in V, (v_i v_j) \in \mathbf{E} \& v_i \in S_i\}$. Additionally, define $\text{NewReach}(S_{j-1}) = S_j = \bar{S}_j \setminus \bar{S}_{j-1}$.

For a given $\mathbf{G}(\mathbf{A})$ and $S_0 = \{v_i\}$, define $\text{Depth}(\mathbf{G}(\mathbf{A}), v_i) = p$, such that $\text{NewReach}(S_p) = S_{p+1} = \phi$. Observe that such a level set partitioning can be obtained starting with $S_0 = v_i$ and a breadth first search (BFS) in $\mathbf{G}(\mathbf{A})$ from $v_i$, where $S_1$ represents vertices in the next level of the search tree and so on.

**Analysis.** Using the notation and definitions above, we now state and prove theorems related to soft error propagation in a sequence of SpMV operations with respect to the sparsity structure and numerical properties of matrix $\mathbf{A}$.

**Theorem 4.1.** Consider an $N \times N$ sparse symmetric coefficient matrix $\mathbf{A}$, an initial $N \times 1$ vector $\mathbf{y}^0$, and a sequence of SpMV operations that generate the vector set $\{\mathbf{y}^0, \mathbf{y}^1, ..., \mathbf{y}^p\}$ where $\mathbf{y}^i = \mathbf{A}^i \mathbf{y}^0$. Consider a single soft error inserted in $\mathbf{y}^0$ in the $i^{th}$ position ($\mathbf{y}_i^0$). At the end of the $p^{th}$ SpMV operation, the vector $\mathbf{y}^p$ will have errors in all its components, where $p = \text{Depth}(\mathbf{G}(\mathbf{A}), v_i)$, i.e., the maximum depth of the tree starting at vertex $v_i$ in $\mathbf{G}(\mathbf{A})$.

**Proof.** Let us assume without loss of generality that a soft occurs in $\mathbf{y}_1^0$, i.e., $i = 1$. For example, we can use symmetric reordering of $\mathbf{A}$ and induced reordering of $\mathbf{y}^0$ that per-
mutes the $i^{th}$ position to the $1^{st}$ position. Let the vector with the change due to error be $\bar{y}^0 = y^0 + e^0$, where $e^0 = [\delta, 0, 0, ..., 0]^T$. Observe that $NNZIndex(e^0) = [1]$.

At the end of the $1^{st}$ SpMV operation we have $\bar{y}^1 = A\bar{y}^0 = y^1 + e^1$, where $y^1 = Ay^0$ and $e^1 = Ae^0$. Here $e^1_j \neq 0$ iff $A_{1j} \neq 0$ (or equivalently, from symmetry $A_{j1} \neq 0$). Next, consider in $G(A)$, $S_0 = \{v_1\}$ and $\tilde{S}_0 = Reach\{S_0\}$. Observe that $NNZIndex(e^1)$ corresponds to $\{j : v_j \in \tilde{S}_0\}$. Additionally, indices in $e^1$ that now have error propagated from $e^0$ are given by $NNZIndex(e^1) - NNZIndex(e^0) = \{j : v_j \in NewReach(S_0)\}$.

By applying this inductively, we can show that $NNZIndex(e^p) = \{1..n\}$, where $p = Depth(G(A), v_1)$.

The output vector $\bar{y}^p$ can be defined as $\bar{y}^p = y^p + e^p$, where $e^p = A^pe_0$. Thus, the output vector $\bar{y}^p$ will have errors in all its components at the end of the $p^{th}$ SpMV operation, where $p$ is the depth of the BFS tree starting at vertex $v_1$ in $G(A)$.

We illustrate Theorem 1 with a small example matrix and its graph counterpart shown in Figures 4.2-4.4. Figure 4.2 shows a sparse symmetric coefficient matrix $A$, where ‘X’ represents a non-zero value; an initial vector $y^0$, where ‘x’ represents a non-erroneous value and ‘$\delta$’ represents erroneous value; and the resultant vectors of a sequence of SpMV operations ($\bar{y}^1, \bar{y}^2, \bar{y}^3, \bar{y}^4$). Figure 4.3 and Figure 4.4 show the growth of the BFS tree in which elements of $y^i$ are corrupted. In this figure, we assume that the $1^{st}$ element of $y^0$ is affected by a soft error initially. Figure 4.4 shows the order and positions (1, 6, 7, 2, 5, 3, 4) in which the elements of $y^1$ are affected. We observe that the resultant vector at the end of the $4^{th}$ iteration has an error in all of its components and the depth of the BFS tree is 4.

![Figure 4.2: Sparse matrix A, vector y, and the sequence of SpMV operations on A and y.](image-url)
Theorem 4.2. Consider an $N \times N$ sparse symmetric coefficient matrix $A$, an initial $N \times 1$ vector $y^0$, and a sequence of SpMV operations that generate the vector set $\{y^0, y^1, ..., y^p\}$ where $y^i = A^i y^0$. Consider a single soft error inserted in $y^0$ in the $i$th position ($y^0_i$). The magnitude of error in the element $y^k_i$ (at the end of the $k$th iteration) grows non-linearly as $(\|A_i\|_2)^k$, i.e., the $k$th power of the 2-norm of $i$th row of $A$.

Proof. Assume that a soft error occurs in $y^0_1$, i.e., $i = 1$. Let the vector with the change due to error be $\tilde{y}^0 = y^0 + e^0$. Consider the set of vectors $\{\tilde{y}^0, \tilde{y}^1, ..., \tilde{y}^k\}$ corresponding to a sequence of SpMV operations $\tilde{y}^i = A \tilde{y}^{i-1}$ or equivalently, $\tilde{y}^i = A^i \tilde{y}^0$; now, $\tilde{y}^i = y^i + e^i$, where, $y^i = Ay^{i-1}$ and $e^i = Ae^{i-1}$. 
By Theorem 1, $NNZIndex(e^1) = \{j : v_j \in \bar{S}_0\}$, where $\bar{S}_0 = Reach(S_0) = \{v_1\}$.

At the end of the 1st iteration we have

$$\bar{y}^1 = A\bar{y}^0 = Ay^0 + Ae^0 = y^1 + e^1$$

Additionally, for each $j \in NNZIndex(e^1)$, $e^1_j = A_{j1}\delta$

Next consider $\bar{y}^2 = y + e^2$. Here

$$e^2 = \sum A_{1j}e^1_j$$

$$= \sum A_{1j}A_{j1}\delta$$

$$= \delta \sum A^2_{1j}$$

By the symmetric property of the matrix $A$

$$= (||A_{1*}||_2)^2 \delta$$

By repeating this analysis for $\bar{y}^3$ and $\bar{y}^4$, consequently we see that

$$e^4_1 = (||A_{1*}||_2)^4 \delta + \beta$$

where $\beta$ represents additional error from non-zero components in $e^1$, $e^2$ and $e^3$ and their propagation.

Now writing the resultant vector $\bar{y}^4$ we have

$$\bar{y}^4 = y^4 + e^4$$

Using Equation 4.6 in Equation 4.7 we see that the error is the resultant vector $\bar{y}^4$ grows non-linearly as $(||A_{1*}||_2)^4$

By applying this inductively, we can show that

$$e^k = (||A_{i*}||_2)^k \delta + \beta \text{ (for } k \mod 2 = 0)$$

Therefore we prove that the magnitude of error in the element $y^k_i$ (at the end of the $k^{th}$ iteration) grows non-linearly as $(||A_{i*}||_2)^k$, i.e., the $k^{th}$ power of the 2-norm of $i^{th}$ row of $A$.

Theorems 1 and 2 indicate that absolute errors in $\bar{y}^k$ can be much higher if the initial soft error occurred in a component $\bar{y}^0_i$ corresponding to a row $i$ of $A$ with the highest 2-norm. Additionally, a single error would propagate the fastest to all components if it were to initially occur in a component $i$ such that $Depth(G(A), v_i) = min\{\forall j, Depth(G(A), v_i), v_j \in V\}$. Sparse matrices from many PDE applications tend to have graphs where the depth measure starting from any vertex does not vary dramatically. Consequently, for such systems, we expect the highest norm position to be more critical in error growth.
4.5 Experimental Methodology and Results

We present our experimental setup and methodology in Section 5.4.1. Subsequently, in Section 4.5.2, we report empirical observations in support of the theoretical analysis in Section 4.4 and the impacts for both explicit and implicit methods. In Section 4.5.3, we assess in-depth how soft errors can dramatically change the space of tradeoffs in the relative performance of PCG vs SOR for the sparse linear system solution within implicit methods. We thus highlight how soft errors add to the complexity of managing performance and reliability tradeoffs of sparse solvers, a key challenge in enabling fast simulations of large scale PDE-based models.

4.5.1 Experimental Methodology

We use the model heat equation (Equation 4.2) using a 5-point finite stencil to analyze the effects of soft errors on the explicit method. For the implicit methods, we use PCG with incomplete Cholesky preconditioner \[101\] with threshold as a representative iterative linear solver. We evaluate PCG with two different threshold values, 0.01 (PCG1) and 0.001 (PCG2). We use 28 symmetric positive definite matrices\[1\] from The University of Florida Sparse Matrix Collection \[102\]. Table 5.1 gives the properties of these matrices. The first column represents matrix ids and the second column gives matrix names. The third and fourth columns represent, respectively, the matrix dimension and number of non-zero elements in the matrix.

**Soft error insertion:** As we intend to study the impact of soft errors with respect to the SpMV kernel, we insert an error within the SpMV kernel. For the explicit method, we insert a soft error in an element of \(u^{th}\) that is involved in the SpMV operation (see Equation 4.2). For the CG and PCG methods, we insert a soft error in the vector \(v\) within the SpMV kernel of Algorithm 4.1. For SOR, we insert a soft error in the vector \(x\), that is involved in the SpMV operation (see Equation 4.5). We use small and large perturbations to simulate bit flips near the least significant bit and most significant bit of the exponent, respectively. To study the in-depth behavior of the performance impact of a soft error, we insert soft errors in each component of the vector iteratively and record the required data. The fault injection schemes proposed in [84] and [89] insert an error at a random location of a random data structure. However, to analyze the varying performance impacts due to a soft error, we take a more systematic error insertion scheme as described above.

\[1\] A symmetric matrix \(A\) is positive definite if \(x^TAx > 0\) for all non-zero vectors \(x\), where \(x^T\) denotes the transpose of \(x\).
<table>
<thead>
<tr>
<th>Matrix ID</th>
<th>Matrix</th>
<th>N</th>
<th>NNZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bcsstk01</td>
<td>48</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>mesh1e1</td>
<td>48</td>
<td>306</td>
</tr>
<tr>
<td>3</td>
<td>lund_b</td>
<td>147</td>
<td>2,441</td>
</tr>
<tr>
<td>4</td>
<td>mesh2e1</td>
<td>306</td>
<td>2,018</td>
</tr>
<tr>
<td>5</td>
<td>mesh2em5</td>
<td>306</td>
<td>2,018</td>
</tr>
<tr>
<td>6</td>
<td>bcsstk07</td>
<td>416</td>
<td>2,312</td>
</tr>
<tr>
<td>7</td>
<td>nos5</td>
<td>468</td>
<td>5,172</td>
</tr>
<tr>
<td>8</td>
<td>nos6</td>
<td>675</td>
<td>3,255</td>
</tr>
<tr>
<td>9</td>
<td>msc00726</td>
<td>726</td>
<td>34,518</td>
</tr>
<tr>
<td>10</td>
<td>bcsstk09</td>
<td>1,083</td>
<td>18,437</td>
</tr>
<tr>
<td>11</td>
<td>plbuckle</td>
<td>1,282</td>
<td>30,644</td>
</tr>
<tr>
<td>12</td>
<td>bcsstk15</td>
<td>3,948</td>
<td>117,816</td>
</tr>
<tr>
<td>13</td>
<td>mhd4800b</td>
<td>4,800</td>
<td>27,520</td>
</tr>
<tr>
<td>14</td>
<td>crystm01</td>
<td>4,875</td>
<td>105,339</td>
</tr>
<tr>
<td>15</td>
<td>bcsstk16</td>
<td>4,884</td>
<td>290,378</td>
</tr>
<tr>
<td>16</td>
<td>Kuu</td>
<td>7,102</td>
<td>340,200</td>
</tr>
<tr>
<td>17</td>
<td>Muu</td>
<td>7,102</td>
<td>170,134</td>
</tr>
<tr>
<td>18</td>
<td>fv1</td>
<td>9,604</td>
<td>85,264</td>
</tr>
<tr>
<td>19</td>
<td>fv2</td>
<td>9,801</td>
<td>87,025</td>
</tr>
<tr>
<td>20</td>
<td>fv3</td>
<td>9,801</td>
<td>87,025</td>
</tr>
<tr>
<td>21</td>
<td>tedB</td>
<td>10,065</td>
<td>144,579</td>
</tr>
<tr>
<td>22</td>
<td>t2dahe</td>
<td>11,445</td>
<td>176,117</td>
</tr>
<tr>
<td>23</td>
<td>bcsstk18</td>
<td>11,948</td>
<td>149,090</td>
</tr>
<tr>
<td>24</td>
<td>cbuckle</td>
<td>13,681</td>
<td>676,515</td>
</tr>
<tr>
<td>25</td>
<td>crystm02</td>
<td>13,965</td>
<td>322,905</td>
</tr>
<tr>
<td>26</td>
<td>gyrom</td>
<td>17,361</td>
<td>340,431</td>
</tr>
<tr>
<td>27</td>
<td>bodyy4</td>
<td>17,546</td>
<td>121,550</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The UFL benchmark matrices with their dimension (N) and number of nonzeroes (NNZ).

### 4.5.2 Empirical Corroboration of Analysis

We support our theoretical analysis (in Section 4.4) with experimental observations in this section. Consider the explicit method to find the solution of the heat equation in Equation 4.2. We insert an error in $u^0$ to give $\bar{u}^0$ (here, $u^0$ represents the solution in the first time step of the explicit method). We track $u^m$ (original solution without soft error) and $\bar{u}^m$ (solution with a soft error). Figure 4.5 shows the relative norm of the solutions, $u^m$ and $\bar{u}^m$, for 100 iterations for a relatively small 40x40 grid based PDE using an explicit method. The y-axis represents the relative norm of the solutions, defined as, $\frac{\text{norm}(\bar{u}^m - u^m)}{\text{norm}(u^m)}$. It
illustrates relative norms for three different error insertion positions corresponding to the highest (red), median (black), and lowest (blue) row norms of $A$. For the problem in this figure, the depth measure (see Theorem 1) is nearly the same for all positions of $u^m$, implying that similar iterations are required to corrupt all the elements in $u^m$. Hence, row norm of $A$ plays a major role in the growth of a soft error (see Theorem 2). The plots in Figure 4.5 support this theorem. A key point to observe is that the magnitude of relative error growth easily surpass 200, making the solutions physically meaningless. Further, propagation of a soft error could potentially manifest as a change in the value of $\mu$ in Equation 4.2 leading to numerical instabilities.

Now, consider solving an implicit scheme. It typically uses an iterative linear solver such as, CG, to find the solution. CG works on the vectors represented by a SpMV sequence, $span\{r, Ar, ..., A^n r\}$ to find a solution for the linear system, where, $A$ is the coefficient matrix and $r$ is the initial residual vector. This span of vectors is known as Krylov subspace [80]. Let us assume that $span\{\bar{r}, A\bar{r}, ..., A^n \bar{r}\}$ and $span\{r, Ar, ..., A^n r\}$ represent Krylov subspaces of CG with and without soft error inserted in the residual vector. Figure 4.7(a) tracks the relative norm of the Krylov vectors for three sample matrices. Observe that an error in the highest row norm position (red) of the vector propagates faster in terms of relative norm compared to an error in either the median (black) or the lowest (blue) row
Figure 4.6: Correlation between row 2-norm values of a matrix and iteration count when a soft error is inserted in the corresponding position of the vector.

norm positions. Unlike the explicit method, the SpMV sequence of CG (Krylov subspace) alone, does not provide enough insights about the solution, but manifests on the measure, A-norm of errors. We define A-norm of errors, as \( \sqrt{(x - \hat{x})' A (x - \hat{x})} \), where, \( x \) is the true solution and \( \hat{x} \) is an intermediate solution during a CG iteration. It plays an important role in the convergence of CG\[103\]. Figure 4.7(b) tracks relative A-norm errors, defined as, \( \frac{A\text{NormErr} - A\text{NormErr}}{A\text{NormErr}} \), where, \( A\text{NormErr} \) and \( A\text{NormErr} \) are the A-norm errors for CG with and without soft errors, respectively. The behavior of relative A-norm error is similar to that of the relative norm of Krylov vectors with respect to the position of a soft error. Observe that unlike plots in Figure 4.7(a), the plots in Figure 4.7(b) are not smooth but oscillating, but both follow similar trends. As A-norm of errors plays an important role in the convergence of CG, we conjecture that a soft error in an element of the vector corresponding to the highest 2-norm row of the matrix could potentially lead to significant performance degradation in terms of iteration count. Figure 4.6 shows the correlation between the row 2-norm values of the matrix and the iteration count of CG. Observe that these two are highly correlated, with correlation coefficient greater than 0.8. Hence, our conjecture about the performance degradation is correct.

4.5.3 Discussion

The correlation between a soft error (propagation and magnitude), numerical property (row-norm) and sparsity structure of a sparse matrix is evident from the previous section. We have shown that explicit methods are highly vulnerable to soft errors and could po-
Figure 4.7: Relative errors in (a) Krylov subspace, (b) A-norm error during CG iterations. Note that red, black and blue colors indicate that a soft error was inserted in a vector position corresponding to the highest, median and lowest row norm positions, respectively.
tentially produce erroneous results, and the convergence of implicit methods are affected (as indicated by the relative A-norm in Figure 4.7(b)). Now, we discuss the impact of soft errors on two of the implicit methods. For ease of analysis, we provided a theoretical study of the error propagation pattern and the growth of error with respect to the Krylov subspace and A-norm of errors for CG. However, for most practical purposes, CG has slow convergence rate, making it prohibitively expensive in terms of computational time. Hence, a majority of implicit schemes usually employ faster solvers like PCG. However, the choice of an effective solver that accounts for both performance and reliability is not clear [104]. We use PCG and SOR as representative fast solvers, to analyze the impact of soft errors on practical problem.

Figure 4.8 shows the sensitivity of SOR (red) and PCG (black) to error insertion position. We observe an increase in the number of iterations in the case of PCG for certain error insertions positions, while, SOR is not affected. Here, it is important to note that we obtain all the relevant data by exhaustive experimentation, which involves inserting a soft error in each position of the vector and recording the corresponding iteration count. Figure 4.9 shows the impact of a soft error on the convergence of PCG and SOR. Figure 4.9(a) illustrates the maximum observed performance degradation due to an error. Observe that performance of PCG on matrices like, Matrices 4, 5, and, 10, could potentially degrade by more than 150 times. However, observe that SOR has no degradation due to a soft error. We conjecture that these results for SOR robustness is due to the inherent solution
correction property of the SOR algorithm. Figure 4.9(b) shows the potential average performance degradation is much less when compared to maximum performance degradation for PCG. This implies that a small fraction of the vector elements are more impacted by a soft error than the remaining elements.

Figure 4.10 illustrates the tradeoffs of using PCG and SOR in the presence of soft errors for a sample set of matrices. A soft error was inserted in the same vector position for both PCG and SOR for both Average Error and Maximum Error cases. The y-axis represents relative performance, defined as, the ratio of execution times of SOR and PCG, in log2 scale. A positive value indicates that PCG performs better than SOR and a negative value
indicates SOR is better. A key observation from this figure is the variable behavior of PCG in the presence of a soft error. We observe that, in the absence of a soft error, PCG performs significantly better than SOR. However, in the presence of an error, SOR tends to perform substantially better than PCG.

![Relative performance: SOR vs PCG](image)

Figure 4.10: Performance tradeoffs of PCG and SOR in the presence of soft errors. Average and maximum error corresponds to the average increase and highest iteration count observed in PCG and SOR on soft error insertion.

### 4.6 Conclusion and Future Work

The efficient and reliable simulation of PDE-based models using explicit and implicit methods on high performance multicore multiprocessors is an area of active research. Such simulations are needed for enabling advances in multiple areas of science, ranging from biology, materials, energy and environmental sciences. Many recent results concern new formulations reflecting effective tradeoffs between multiple metric including the scalability, performance, reliability and accuracy of solution methods. In this context, our results indicate that the stability of explicit methods can be greatly degraded by the occurrence of a single soft error. Additionally, for the inherently numerically stable implicit methods, a single soft error can have a serious impact on performance, with potential degradation by factors of 200x when using PCG for linear system solution. On the other hand we observed that there are essentially no performance degradation for the stationary SOR scheme. Consequently, there are interesting crossover instances, i.e., systems where PCG has vastly superior performance relative to SOR by a factor of 21x when there are no soft
errors while SOR outperforms PCG by a factor of 12x given a single soft error. We expect to see an interesting spectrum of vastly different impacts of soft errors if we consider a broader array of linear solvers including methods from direct [105], domain decomposition [106] and multigrid approaches [107]. These results motivate and indicate the need for developing cost effective soft error detection and mitigation strategies. Additionally, they also motivate the development of new techniques for the selection of sparse linear solution methods that take into account distortions in the tradeoffs between performance and reliability due to the differential impacts of soft errors.
Chapter 5

Enhancing Resilience: Algorithm Based Fault Tolerance for Sparse Linear System Solution

5.1 Introduction

In Chapter 4, we characterized the performance and energy impacts of a single transient error on sparse linear system solution. To tolerate against such errors and enhance the resiliency of sparse linear system solution, we develop a new sparse checksum encoded algorithm-based fault tolerance technique for the preconditioned conjugate gradients method. This chapter has previously appeared in proceedings of the 26-th International Conference on Supercomputing[9].

Current large-scale high performance computing (HPC) installations have over 100,000 processor cores [2] and some may soon have over a million cores [1]. Increasing transistor densities per unit area and smaller feature sizes have resulted in increased core performance, and thus overall improved performance of the HPC installations. However, the performance benefit comes at the expense of higher susceptibility to hardware errors including, especially, transient or soft errors. For example, the BlueGene/L experiences one transient error in its L1 cache every 4-6 hours [14]. Consequently, long running scientific applications on large-scale HPC installations are more susceptible to transient errors.

Several algorithm-based fault tolerance (ABFT) techniques have been proposed for scientific applications with numeric computing involving dense matrix operations [108, 109, 110, 111]. Typically, these techniques are based on checksum encoding to detect
and recover from data errors for various dense matrix operations, such as, matrix-matrix multiplication and matrix factorization. However, such checksum encoding based ABFT techniques have not been developed when sparse matrices are involved, for example, in sparse linear system solution through methods such as preconditioned conjugate gradients (PCG). These are often used to solve PDE models in a variety of application such as those found in structural mechanics and computational fluid dynamics [6].

Recent work [112, 14, 113] show the differential impact of transient errors on sparse linear system solution. Bronevetsky et al. [14] show that a sparse linear solver affected by a transient error can have a wide range of behavior, from having no effect to application crash. More specially for PCG, Shantharam et al. [112] show that performance degradation can be as high as a factor of 250 relative to the case with no error, for a single transient error, i.e., there is a very large increase in the number of iterations to convergence. Thus, there is a strong need for protection mechanisms against such errors.

In this chapter, we develop a new checksum encoded sparse ABFT PCG, S-ABFT-PCG, with checksum encodings for all its key operations, including, sparse matrix-vector multiplication (SpMV), vector operations and in particular, sparse triangular solution. Our formulation differs from the traditional encodings for dense matrices in the following ways. First, our method allows only for the detection and not the correction of a single error; however, such detection extends to a single error in metadata specific to sparse systems, including row and column subscript information. Second, we construct a formulation that enables single error detection in sparse triangular solution. Third, we seek to manage the tradeoffs between fault tolerance and the overheads of ABFT in a way that benefits PCG instances that may be more susceptible to soft errors, i.e., on systems of large dimension with hundreds to thousands of iterations to convergence. We have made the overhead of our scheme per iteration of PCG lower at the expense of having to recompute a single iteration if an error is detected.

The remainder of this chapter is organized as follows. In Section 5.2, we first provide classification of faults. Next, we present recent work related to the ABFT techniques and discuss the relevance of our work. Section 5.3 contains our main contributions, namely, the development of our sparse checksum encoded algorithm-based fault tolerant PCG, S-ABFT-PCG, and analytic proofs indicating that the underlying checksum properties will enable the detection of a single error in each key operation within PCG. In Section 5.4, we provide empirical evaluation to illustrate the tradeoffs between the overheads of ABFT versus the benefits of fault tolerance, including comparisons with the traditional ABFT technique. In Section 5.5, we conclude this chapter with a brief summary of our findings.
and directions for future work.

## 5.2 Related Work and Background

Faults or errors can be broadly classified as permanent and transient. Henceforth, we use faults and errors interchangeably. The permanent faults that are often caused by aging hardware, such as, non functional memory cells, can only be overcome by avoiding the use of faulty hardware. In contrast, a transient fault is a temporary error that can occur due to several reasons such as cosmic radiation and voltage fluctuation. Both types of faults can leave the computing system state corrupt by either causing a fail-stop failure [13] or a silent error [8, 14]. In a fail-stop failure, the process that encounters a fault stops working. However, silent errors do not interrupt the process, but can eventually lead to incorrect computation. It is very difficult to find the actual faulty elements (permanent or transient) within a HPC system with billions of hardware components. Thus, it is important to develop fault tolerance techniques [114, 115, 116, 117, 111, 58, 109, 118, 108, 119, 120] to improve the resilience of HPC systems.

Many ABFT techniques have been used for application specific fault tolerance [108, 117, 111, 110]. Although, developing such techniques require a thorough understanding of the application, the low overhead costs and scalability of ABFT techniques make them an attractive alternative to the fault tolerance techniques based on checkpoint/restart [114, 115]. One of the common ABFT methods is to encode the input data such as matrices using checksums and verify that the output has a valid encoding. Huang et al. [108] propose an ABFT technique to handle errors in numerical computation, in particular, to detect and correct errors in dense matrix computations using checksum-encoded matrices. These methods use the linear transformation property of matrix computations to detect and correct a single error. Subsequently, a number of ABFT schemes have been proposed for error detection in different computations such as Fourier transforms [118] and dense matrix factorizations solvers[109]. Many build upon the ABFT proposed by Huang et al [108]. Chen et al. [110] show that the ScaLAPACK matrix-matrix multiplication kernel can tolerate fail-stop process failures without checkpointing or message logging. Yeh et al.[111] propose the use of ABFT techniques within matrix inversion with maximum pivoting.

Most of the prior work on ABFT has concerned dense linear algebra as indicated above. However, many scientific applications typically involve sparse linear algebra operations such as a sparse linear system solution. Hence, it is very important to develop and analyze the efficacy of ABFT techniques on such sparse computations. To the best of our knowledge,
there has been very little development of such sparse ABFT techniques. The earliest work that we know of is that of Roy-Chowdhury et al. [117] that proposes an ABFT successive over relaxation (SOR) scheme for solving the Laplace equation on a model 5-point finite difference grid. More recently, Bronevetsky et al. [14] consider checksum-based ABFT in the context of sparse solvers as a relatively minor component of a complex set of fault tolerant approaches, including checkpointing and prediction of errors. In their work, they primarily apply the traditional checksum encoding of Huang et al. ([108]) to the SpMV kernel and vector operations. ABFT of sparse triangular solution, an operation that is required to apply popular incomplete factorization preconditioners is not considered. In contrast to their work, the focus of our work is primarily on developing a sparse checksum encoding for all the key operations of PCG, including triangular solves. Additionally, unlike the checksum encoding of Huang et al. [108] that is also used in [14], we propose a lower overhead encoding tailored to PCG instances where convergence often depends on many hundreds of iterations, as explained later in Section 3.

Background: Classical ABFT. We now provide a brief overview of the algorithm-based fault tolerance technique of Huang et al. [108], as it forms the building block of many ABFT techniques. We call this technique classical ABFT or C-ABFT in this chapter. Figure 5.1 illustrates C-ABFT for a dense matrix-matrix multiplication operation. In this technique, the input matrices are augmented with row or column checksums. In Figure 5.1, Matrix A is augmented with a row vector whose elements are the checksum of the corresponding columns (called $A^c$), whereas, Matrix B is augmented with a column vector whose elements are the checksum of the corresponding rows (called $B^r$), i.e.,

$$A^c_{n+1,j} = \sum_{i=1}^{n} A_{ij} \text{ and } B^r_{i,n+1} = \sum_{j=1}^{n} B_{ij}$$

Given $C = A \times B$ and $C^f = A^c \times B^r$, the output matrix, $C^f$, has the following properties:
\[ i) \quad C_{n+1,j}^f = \sum_{i=1}^{n} C_{ij}^f, \quad ii) \quad C_{i,n+1}^f = \sum_{j=1}^{n} C_{ij}^f. \]

These properties ensure that a single erroneous element in any of the matrices will be detected and corrected. Huang et al. [108] give a formal proof that indicates the ability of the checksum encoded dense matrix-matrix multiplication operation to detect and recover from a single error in one of the matrix elements.

5.3 A Sparse Checksum Encoding for ABFT-PCG

We present the main contributions of our work in this section and it is organized as follows. In Section 5.3.1, we provide an overview of a PCG algorithm and explain key kernel operations. In Section 5.3.2, we discuss the issues with the C-ABFT technique with respect to sparse operations. In Section 5.3.3, we describe in detail our sparse checksum encoded ABFT technique with respect to the sparse matrix vector multiplication (SpMV) and triangular solve operations and prove the correctness of our technique. Theorem 5.1 proves that our technique detects a single error in an SpMV operation. Theorems 5.3-5.5 prove the error detection property of our technique for a sparse triangular solution. As the vector operations: addition and dot product are performed on dense vectors, we do not consider these operations when analyzing our technique. Note that we do not explicitly provide the pseudocode for our sparse checksum encoded ABFT-PCG (S-ABFT-PCG) as it can be obtained by combining our fault tolerance technique and the PCG algorithm presented in Section 5.3.1.

5.3.1 Overview of PCG and its primary operations

In this section, first, we give a brief overview of a PCG algorithm. Next, we explain the basic matrix and vector operations that comprise the PCG algorithm.

Algorithm 5.1 illustrates the pseudocode of a typical PCG algorithm used to solve a large sparse linear system. The algorithm takes several input parameters; a coefficient matrix \( A \) (which is sparse and symmetric positive definite), a known vector \( b \), the solution vector \( x \) initialized to random values, a preconditioner matrix \( M \), the maximum number of iterations allowed \( maxit \) and a tolerance \( TOL \), to detect the convergence of the algorithm. The primary operations in Algorithm 5.1 are: SpMV (line 9), triangular solve (line 15, note that computing matrix inverse is expensive, hence a triangular solve is use in practice) and vector operations like vector additions and dot products (lines 10, 11, ...).

**Sparse matrix-vector multiplication (SpMV).** SpMV is the most compute inten-
Algorithm 5.1 Preconditioned Conjugate Gradient

```plaintext
procedure PCG(A,b,x,M,maxit,TOL)

1: r = b - A * x
2: z = M^-1 * r {solve using a triangular solve}
3: iters = 0
4: nb = norm(b)
5: rsq = r' * r
6: p = z
7: while iters < maxit && sqrt(rsq)/nb > TOL do
8:   iters = iters + 1
9:   t = SpMV(A, p)
10:  v = r' * z
11:  alpha = v / (p' * t)
12:  x = x + alpha * p
13:  r = r - alpha * t
14:  rsq = r' * r
15:  z = M^-1 * r
16:  beta = (r' * z) / v
17:  p = z + beta * p
18: end while
```

sive operation in the PCG algorithm. It multiplies a sparse matrix (A) in a compressed storage format (CSR)[121] with a dense vector (p). It is important to note here that, unlike the dense matrix-vector multiplication, SpMV loads the matrix indices to access corresponding data. The computational complexity of an SpMV operation is $O(nnz)$, where, $nnz$ represents the number of nonzero elements in A.

**Preconditioning.** Preconditioning is an essential part of PCG as it ensures fast convergence of the conjugate gradient method. In Algorithm 5.1, line 15 represents the preconditioning operation. This operation could be an SpMV operation or a triangular solve depending on the preconditioner matrix. For a diagonal preconditioner, it is an SpMV operation as computing $M^{-1}$ is simple, however, for a more complex preconditioner, computing $M^{-1}$ is expensive. Typically, $z$ is obtained as follows

$$z = M^{-1} * r \implies M * z = r$$

$$L * L^T * z = r$$

$$L * y = r$$

$$L^T * z = y$$

Equation 5.1 represents the incomplete Cholesky factorization [122] of M into L and $L^T$. 
Figure 5.2: C-ABFT technique and its overheads for (a) dense matrix-matrix multiplication, (b) sparse matrix-vector multiplication.

Note that this incomplete factorization is performed only once per linear system following which sparse triangular solves are performed in Equations 5.2 and 5.3 to obtain $z$. The complexity of a sparse triangular solution (Equations 5.2, 5.3) is proportional to the number of non-zeroes in $L$.

Vector operations. There are two types of vector operations within an iterative linear solver. First, an inner product which involves multiplying the corresponding elements of two vector to yield a scalar. For example, if $a$ and $b$ are $n \times 1$ vectors, then the inner product $a^T \cdot b = \sum_{i=1}^{n} a_i \cdot b_i$. Second, a vector addition which involves adding the corresponding elements of two or more vectors to yield a vector. Note that these operations take $O(n)$ operations.

5.3.2 Issues with application of C-ABFT to PCG

Recall that most earlier related work on ABFT have used C-ABFT or its variants to protect dense matrix-matrix operations. However, there are several issues related to the applicability of such techniques in the context of a sparse PCG. First, overheads of C-
Figure 5.3: An illustration of the ABFT technique when there is an error in the metadata (a) column index, (b) row index.

ABFT for the SpMV kernel are relatively high. Second, C-ABFT does not consider fault tolerance of the metadata that is a part of the sparse matrix representation. Third, C-ABFT does not include the sparse triangular solution kernel. In the remainder of this section, we explain the first two issues. We address the last issue is Section 5.3.3.2.

The dense matrix-matrix operations like matrix multiplication have computational complexity of $O(n^3)$ for an $n \times n$ matrix. The overhead of C-ABFT on such operations is $O(n^2)$ due to an additional matrix-vector multiplication. In Figure 5.2(a), this additional multiplication corresponds to $A \times \text{checksum of } B$. Therefore, asymptotically, the overhead of C-ABFT on dense matrix-matrix operations is relatively small. In contrast, the overhead
of C-ABFT on an SpMV operation is directly proportional to the number of non-zero elements in the sparse matrix \((kn)\), i.e., \(O(n)\). However, an SpMV operation takes \(O(n)\) time, making the C-ABFT overhead significantly high\(^1\). This is illustrated in Figure 5.2(b). Consequently, especially for long running PCG instances on large sparse systems with hundreds to thousands of iterations to convergence, we focus on developing a new sparse ABFT that has lower overheads. Our scheme (developed in Section 5.3.3.1) trades off a lower overhead per SpMV operation and PCG iteration to provide detection of a single error in all PCG operations, at the expense of having to recompute the PCG iteration in the event an error does occur.

The sparse matrices used within an iterative solver are stored in a compressed representation and requires the use of metadata to access the actual matrix elements. For example, an \(n \times n\) sparse matrix with \(nnz\) non-zeroes requires two arrays of sizes \(nnz\) and \(n\) representing column and row pointers respectively, in addition to the \(nnz\) data elements. As the size of the metadata is almost half the size of the actual data, there is need to detect an error in the metadata. C-ABFT is vulnerable to an error in the metadata during an operation such as SpMV. For example, consider Figures 5.3(a) and 5.3(b). Figure 5.3(a) shows an instance of an error in the column index. Observe that such an error is equivalent to changing the non-zero structure of the matrix. As we can see from the figure, C-ABFT detects this error; however, it may not detect an error in the row index (see Figure 5.3(b)). Note that a single error in the row index could manifest as multiple errors in the output, hence C-ABFT may not be able to detect it.

In the following section, we address these issues in C-ABFT by developing sparse checksum encoding schemes proven to detect a single error in PCG operations and the sparse matrix metadata.

### 5.3.3 Sparse-ABFT for PCG operations

In this section, we describe our ABFT technique for the SpMV and sparse triangular solve operations. We provide theoretical analysis and prove that S-ABFT works for SpMV and sparse triangular solve.

#### 5.3.3.1 SpMV Operation

Typically, an SpMV operation is of the form, \(t \leftarrow A \cdot p\), where, \(t\) and \(p\) are output and input vectors, respectively, and \(A\) is a sparse matrix. Thus, a fault tolerance technique

\(^1\)Although analytically this overhead is at 100% the computational cost of a single SpMV, in practice, an efficient implementation with locality of data accesses can reduce this to a small fraction.
should take into account an error in any of these three data structures. Additionally, as the elements in A (data) are accessed through metadata (column and row indices), it is important to protect the metadata. First, we discuss fault tolerance considerations for the metadata during an SpMV operation. Next, we present our ABFT technique for the SpMV operation that focuses on the actual matrix data.

**Fault tolerance for the metadata.** As illustrated earlier in Figure 5.3(a), an error in an element of Colid results in exactly one error in the output vector, hence it is detected by C-ABFT. Later in this section, we prove that our technique will also detect such an error. Now, let us consider an error in Rowid that may not be detected by C-ABFT. To protect against this case, our technique does the following: (i) computes and stores the checksum of Rowid before an SpMV operation in a variable (say Rowcksm) and (ii) during the SpMV operation, whenever an element of Rowid is accessed, subtracts the value of the accessed element from Rowcksm. After the SpMV operation, a non-zero value of Rowcksm indicates an error. It is important to note here that an error in either vector (Rowid or Colid) can change the value of the affected element to an arbitrary value. However, we consider only the cases wherein the changed value is in the range \([1, n]\). On the other hand, an out-of-range value can potentially cause fail-stop failures such as segmentation faults. As the focus of this chapter is on silent errors, we ignore the cases that could lead to fail-stop failures.

**Low overhead fault tolerance for SpMV.** Figure 5.4 shows the S-ABFT technique for an SpMV operation. Unlike C-ABFT that has two SpMV operations, S-ABFT has one SpMV operation and a vector dot product. This reduces the fault tolerance overhead significantly. The following theorems show that our technique detects a single error (in the matrix elements or the vectors) during an SpMV operation when the matrix is symmetric positive definite \(^2\) (SPD) and strictly diagonally dominant.

**Theorem 5.1.** Consider an encoded SpMV operation, \(t^e = A^c \ast p\), where \(A^c\) is the \((n+1) \times n\) checksum encoded matrix of a sparse SPD and diagonally dominant matrix \(A\), \(p\) and \(t^e\) are input and output vectors, respectively. By definition of a checksum matrix,

\[
A^c_{(n+1)j} = \sum_{i=1, A_{ij} \neq 0}^{n} A_{ij} \quad \text{for} \quad 1 \leq j \leq n.
\]

Let \(p^r = p\) and \(dotp = \sum_{i=1}^{n} A^c_{(n+1)i} \ast p^r_i\). A single error in the encoded SpMV operation causes one of the following conditions to fail:

\[
(i) \quad t^e_{n+1} = \sum_{i=1}^{n} t^e_i, \quad (ii) \quad dotp = \sum_{i=1}^{n} t^e_i
\]

\(^2\)A symmetric matrix \(A\) is positive definite if \(x^T A x > 0\) for all non-zero vectors \(x\), where \(x^T\) denotes the transpose of \(x\).
Figure 5.4: S-ABFT technique and its corresponding overhead for an SpMV operation.

**Proof.** We consider the following three error scenarios that can affect an SpMV operation, a single error in (a) \( t^c \), (b) \( A^c \), and (c) \( p \).

**Case (a):** Without loss of generality, let us assume an error in the \( j^{th} \) element of \( t^c \) and denote it by \( \hat{t}_j^c \). Let the error prone output vector be \( \hat{t}^c \). Therefore, \( \hat{t}_i^c = t_i^c \) for \( 1 \leq i \leq n+1 \) and \( i \neq j \). \( \hat{t}_j^c = t_j^c + \delta \), where, \( \delta \) is change in the value of \( t_j^c \) due to an error. Now, adding the elements of \( \hat{t}^c \), we have,

\[
\sum_{i=1}^{n} \hat{t}_i^c = \sum_{i=1, i \neq j}^{n} t_i^c + \hat{t}_j^c = \sum_{i=1}^{n} t_i^c + \delta = t_{n+1}^c + \delta
\]

As, \( \hat{t}_{n+1}^c = t_{n+1}^c \), \( \sum_{i=1}^{n} \hat{t}_i^c \neq \hat{t}_{n+1}^c \). Hence condition (i) fails and the error is detected.

**Case (b):** Without loss of generality, let us assume an error in the element \( A^c_{jk} \). By definition of the SpMV operation, this translates to an error in the \( j^{th} \) element of output vector. Hence, the proof follows from **Case (a)**.

**Case (c):** Without loss of generality, let us assume an error in the \( j^{th} \) element of \( p \) and denote it by \( \hat{p}_j \). Let the error prone input vector be \( \hat{p} \). Therefore, \( \hat{p}_i^r = \hat{p}_i \) for \( 1 \leq i \leq n \) and \( i \neq j \). \( \hat{p}_j = p_j + \delta = p_j^r + \delta \), where, \( \delta \) is change in the value of \( p_j \) due to an error.

Given the encoded SpMV operation, \( t^c = A^c \ast \hat{p} \), consider \( \sum_{i=1}^{n} t_i^c \).

\[
\sum_{i=1}^{n} t_i^c = \sum_{i=1}^{n} \sum_{k=1, A_{ik} \neq 0} \begin{pmatrix} c \\ i \\ k \end{pmatrix} \hat{p}_k
\]

\[
= \sum_{i=1}^{n} \sum_{k=1, k \neq j, A_{ik} \neq 0} \begin{pmatrix} c \\ i \\ k \end{pmatrix} \hat{p}_k + \sum_{i=1}^{n} \begin{pmatrix} c \\ i \\ j \end{pmatrix} \hat{p}_j
\]

\[
= \sum_{i=1}^{n} \sum_{k=1, k \neq j, A_{ik} \neq 0} \begin{pmatrix} c \\ i \\ k \end{pmatrix} p_k^r + \sum_{i=1, A_{ij} \neq 0}^{n} \begin{pmatrix} c \\ i \\ j \end{pmatrix} (p_j^r + \delta)
\]
\[
\sum_{i=1}^{n} \sum_{k=1, A_{ik}^{c} \neq 0}^{n} A_{ik}^{c} p_{k}^{r} + \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \delta \\
= \left( \sum_{k=1}^{n} \sum_{i=1, A_{ik}^{c} \neq 0}^{n} A_{ik}^{c} p_{k}^{r} \right) + \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \\
= \left( \sum_{k=1}^{n} \left( p_{k}^{r} \sum_{i=1, A_{ik}^{c} \neq 0}^{n} A_{ik}^{c} \right) \right) + \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \\
= \left( \sum_{k=1}^{n} p_{k}^{r} A_{(n+1)k}^{c} \right) + \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \\
= dotp + \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c}
\]

Therefore,
\[
\sum_{i=1}^{n} t_{i}^{c} = dotp + \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c}
\]

Condition (ii) fails if, \( \delta \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \neq 0 \). As \( A \) is strictly diagonally dominant, we have,
\[
|A_{jj}^{c}| > \sum_{i=1, A_{ij}^{c} \neq 0, i \neq j}^{n} |A_{ij}^{c}|
\]

From this inequality, it follows that \( \sum_{i=1, A_{ij}^{c} \neq 0}^{n} A_{ij}^{c} \neq 0 \), thus condition (ii) fails and the error is detected.

5.3.3.2 Sparse triangular solution

The sparse triangular solve operation involves solving a linear system of the form \( L \ast y = r \), where \( L \) is an \( n \times n \) lower triangular matrix, \( y \) is an unknown vector \( n \times 1 \) and \( r \) is an \( n \times 1 \) known vector. Figure 5.5 gives an example of a lower triangular solve system. The graph on the right represents the dependence among the solution elements during the triangular solve process using forward substitution. For example, the solution element \( y_{5} \) depends on the already solved values \( y_{1} \) and \( y_{3} \). We present a graph-based formulation of a sparse lower triangular solve process, the impact of an error on the solution vector, and theoretical analysis of our ABFT technique for triangular solve, in the following subsections.

**Graph Representation.** Consider a linear system \( L \ast y = r \). Define a graph representation of \( L \), \( G(L) = (V, E) \), where, \( V = \{v_{1}..v_{n}\} \) is the vertex set; \( E = \{(v_{i}v_{j}) : i > j, 1 \leq i, j \leq n, L_{ij} \neq 0\} \) is the edge set. Note that \( G \) is a directed graph. Let \( reach[S] \) represent the set of all the immediate neighbors of a set of vertices \( S \), i.e.,
reach\{S\} = \{v_j | v_j \in V, (v_iv_j) \in E, v_i \in S\}.

Algorithm 5.2 Fault set construction

function BuildFaultSet (G(V,E),k,\hat{y})
1: if \( k > |V| \) then
2: \( CS = V \)
3: return
4: end if
5: \( CS = \phi \) \{set of correctly solved elements in \( \hat{y} \}\}
6: \( VS = V \)
7: \( i = 1 \)
8: while \( i < k \) do
9: \( CS = CS \cup \{\hat{y}_i\} \)
10: \( VS = VS \setminus \{v_i\} \)
11: end while
12: \( FS = \{\hat{y}_i\} \) \{set of corrupted elements in \( \hat{y} \}\}
13: \( VS = VS \setminus \{v_i\} \)
14: \( RS = reach(\{v_i\}) \)
15: while \( RS \neq \phi \) do
16: \( FS = FS \cup \{\hat{y}_j : v_j \in RS\} \)
17: \( VS = VS \setminus RS \)
18: \( RS = reach(RS) \)
19: end while
20: \( CS = CS \cup \{\hat{y}_j : v_j \in VS\} \)
21: return

Algorithm 5.2 illustrates the partitioning the elements in \( \hat{y} \) into fault (FS) and correct (CS) sets due a fault in \( L \). The input to the algorithm are: \( G(V,E) \) - directed graph representing \( L \), \( k \) - implying \( \hat{y}_k \) is corrupt due a fault in one of elements in \( L_{k*} \), and \( \hat{y} \) - the corrupt vector. Figures 5.6(a), 5.6(b), 5.6(c) illustrate the differential impact of an error on the solution vector based on the error occurrence position. Note that the number of corrupt elements vary based on the error position as indicated by the size of FS. The following theorems use the sets FS and CS to illustrate the protection mechanism that can
Figure 5.6: The effect of the error occurrence position on the solution vector. The grey elements are corrupt. (a) FS = \{y_3, y_4, y_5, y_6\}, (b) FS = \{y_5\}, (c) FS = \{y_2, y_6, y_7\}. detect a single transient error in any of the data structures.

Fault tolerance for sparse triangular solve. Figure 5.7 shows the S-ABFT technique for a sparse triangular solve operation. In this technique, we first encode the triangular matrix, L by augmented it with a row representing the column checksum of L. Next, we compute the solution, y and perform a dot product operation between the augmented row (checksum) and y. We compare the value of this dot product with S \((S = \sum_{i=1}^{n} r_i)\). In the case of a single error, this value will not match and we detect an error. We prove the correctness of this fault tolerance technique in the following theorems.

In an attempt to improve the readability of the following theorems, we use a vector, p, to represent the checksum augmented row in L, i.e., \(p_j = \sum_{i=1, L_{ij} \neq 0}^{n} L_{ij}\). Although we prove our scheme for a lower triangular solution \((L \ast y = r)\), similar results hold for solving an upper triangular system \((L^T \ast z = y)\).

**Theorem 5.2.** Consider a linear system \(L \ast y = r\), where, L is an \(n \times n\) sparse lower triangular matrix and r is an \(n \times 1\) known vector. Assume that y is solved correctly without
any errors. Then, the value of the inner product of $p$ and $y$ is equal to the sum of the elements of $r$, i.e., $p^T y = \sum_{i=1}^{n} r_i$.

**Proof.** Let us expand the linear system $Ly = r$ into a set of $n$ linear equations:

$$\sum_{\forall j: L_{1j} \neq 0} L_{1j} y_j = r_1,$$

$$\sum_{\forall j: L_{2j} \neq 0} L_{2j} y_j = r_2,$$

$$\vdots$$

$$\sum_{\forall j: L_{nj} \neq 0} L_{nj} y_j = r_n.$$

Adding the equations, we get,

$$\sum_{i=1}^{n} \sum_{\forall j: L_{ij} \neq 0} L_{ij} y_j = \sum_{i=1}^{n} r_i.$$

Consider LHS,

$$\sum_{i=1}^{n} \sum_{\forall j: L_{ij} \neq 0} L_{ij} y_j = \sum_{j=1}^{n} \sum_{\forall i: L_{ij} \neq 0} L_{ij} y_j \quad \text{(row summation is same as the column summation)}$$

$$= \sum_{j=1}^{n} \{y_j \sum_{\forall i: L_{ij} \neq 0} L_{ij}\}$$

$$= \sum_{j=1}^{n} y_j p_j = p^T y.$$

Therefore, $p^T y = \sum_{i=1}^{n} r_i$ (theorem proved). ☐

**Theorem 5.3.** Consider a linear system $\hat{L} \hat{y} = \hat{r}$, where, $\hat{L}$ is an error prone $n \times n$ sparse lower triangular matrix, i.e., $L$ affected by exactly one error, and $\hat{r}$ is an $n \times 1$ known vector. Let $\hat{y}$ be the incorrect solution to the linear system. Then, $p^T \hat{y} \neq \sum_{i=1}^{n} r_i$.

**Proof.** Consider a single error in $L$ during the triangular solve. Let us assume without loss of generality that the error occurs in $L_{ij}$ and $L_{ij} \neq 0$. Essentially, we are solving a modified system $\hat{L} \hat{y} = \hat{r}$. Let the error prone solution be $\hat{y}$. We know that some of the elements in $\hat{y}$ will be error prone. Let FS and CS represent sets of elements of $\hat{y}$ that are affected and unaffected by the error, respectively (as in Algorithm 5.2). From Theorem 5.2 we have,

$$p^T \hat{y} = \sum_{i=1}^{n} r_i.$$
\[ \sum_{\forall k: y_k \in CS} p_k \hat{y}_k + \sum_{\forall k: y_k \in FS} p_k \hat{y}_k = \sum_{i=1}^{n} r_i. \]

This is equivalent to,

\[ \sum_{\forall k: y_k \in CS \; k \neq j} p_k \hat{y}_k + \sum_{\forall k: y_k \in FS \; k \neq j} p_k \hat{y}_k + p_j \hat{y}_j = \sum_{i=1}^{n} r_i. \]  \hspace{1cm} (5.4)

As \( \hat{y} \) is the solution for \( \hat{L}y = r \), we have \( \hat{L}\hat{y} = r \). Expanding this similar to Theorem 5.2, we have:

\[ \sum_{j=1}^{n} \{ \hat{y}_j \sum_{\forall i: L_{ij} \neq 0} \hat{L}_{ij} \} = \sum_{i=1}^{n} r_i. \]

Hence,

\[ \sum_{\forall k: y_k \in CS \; k \neq j} p_k \hat{y}_k + \sum_{\forall k: y_k \in FS \; k \neq j} p_k \hat{y}_k + \hat{p}_j \hat{y}_j = \sum_{i=1}^{n} r_i. \]  \hspace{1cm} (5.5)

Note that \( \hat{p}_j \) in the above equation is due an error in \( L_{ij} \). Equations 5.4 and 5.5 are not equal unless \( p_j = \hat{p}_j \), hence, an error is detected as Equation 5.4 does not hold. This test can be used to detect an error in matrix \( L \).

**Theorem 5.4.** Consider a linear system \( L \ast \hat{y} = r \), where, \( L \) is an \( n \times n \) sparse lower triangular matrix, \( r \) is an \( n \times 1 \) known vector and \( \hat{y} \) is the error prone solution. Then, \( p^T \hat{y} \neq \sum_{i=1}^{n} r_i. \)

**Proof.** Consider a single error in \( y \), say, in \( y_j \). Let the solution be \( \hat{y} = [\hat{y}_1, \hat{y}_2, ..., \hat{y}_j, ..., \hat{y}_n] \).

Let FS and CS represent sets of elements of \( \hat{y} \) that are affected and unaffected by the error, respectively. From Theorem 5.2 we have,

\[ p^T \hat{y} = \sum_{i=1}^{n} r_i. \]  \hspace{1cm} (5.6)

Expanding the LHS of Equation 5.6, we have,

\[ p^T \hat{y} = \sum_{\forall i: \hat{y}_i \in CS} p_i \hat{y}_i + \sum_{\forall i: \hat{y}_i \in FS} p_i \hat{y}_i. \]

\[ = \sum_{\forall i: \hat{y}_i \in CS} \sum_{\forall k: \hat{y}_k \in CS \; L_{ki} \neq 0} L_{ki} * \hat{y}_i + \sum_{\forall i: \hat{y}_i \in FS} \sum_{\forall k: \hat{y}_k \in FS \; L_{ki} \neq 0} L_{ki} * \hat{y}_i \]
Since the first \( j - 1 \) elements of \( \hat{y} \) are not affected by the error, we have

\[
= \sum_{k=1}^{j-1} r_k + \sum_{\forall i: y_i \in CS} \sum_{\forall k: L_{ki} \neq 0} L_{ki} \hat{y}_i + \sum_{\forall k: y_i \in FS} L_{ki} \hat{y}_i
\]

Although, we know that some elements \( \hat{y}_i \) for \( i > j \) could be corrupt due an error in \( y_j \), the equations represented by \( r_i \) for \( i > j \), still hold, i.e., \( \sum_{\forall l: L_{li} \neq 0} L_{li} \hat{y}_l \), hence

\[
= \sum_{k=1,k \neq j}^{n} r_k + \sum_{\forall i: \hat{y}_i \in CS} L_{ji} \hat{y}_i + \sum_{\forall i: \hat{y}_i \in FS} L_{ji} \hat{y}_i
\]

Equations 5.6 and 5.7 are equivalent if

\[
r_j = \sum_{\forall i: \hat{y}_i \in CS \atop L_{ji} \neq 0} L_{ji} \hat{y}_i + L_{jj} \hat{y}_j,
\]

However,

\[
r_j = \sum_{\forall i: \hat{y}_i \in CS \atop L_{ji} \neq 0} L_{ji} \hat{y}_i + L_{jj} y_j.
\]

Hence, if \( \hat{y}_i \neq y_i \), \( p^T \hat{y} \neq \sum_{k=1}^{n} r_k \). Consequently, the error will be detected. This test can be used to detect an error in the vector \( y \).

Theorem 5.5. Consider a linear system \( L \hat{y} = \hat{r} \), where, \( L \) is an \( n \times n \) lower triangular matrix, \( \hat{r} \) is an error prone \( n \times 1 \) known vector. Let \( \hat{y} \) be the incorrect solution to the linear system. Define an \( n \times 1 \) vector \( p \), such that, \( p_j = \sum_{k=1}^{n} L_{kj} \). Then, \( p^T \hat{y} \neq \sum_{k=1}^{n} r_k \).

Proof. Consider a single error in \( r_j \) leading to an error in \( y_j \). The proof follows from Theorem 5.4.
5.3.4 Overheads: C-ABFT-PCG vs S-ABFT-PCG

We analyze the fault tolerance overheads of C-ABFT-PCG and S-ABFT-PCG and develop analytical models that capture these overheads. First, we define the following notation and terms that are used in our models. Subscripts \( b \), \( s \) and \( c \) are used to denote base, S-ABFT-PCG, and C-ABFT-PCG methods respectively. For example, \( T_b \) represents the time to convergence of the base method. Similarly, superscripts \( spmv \), \( vec \) and \( trisol \) are used to denote the SpMV, vector operations and sparse triangular solution kernels. For example, \( t_{vec}^b \), represents the time spent in the vector operations for the base method. Let \( T_o^x \) represent the absolute overhead of method ‘x’ compared to the base method.

Consider the total time per iteration of PCG for each method:

\[
T_b = t_{spmv}^b + t_{vec}^b + t_{trisol}^b,
\]

\[
T_c = t_{spmv}^c + t_{vec}^c + t_{trisol}^c \quad \text{and} \quad T_s = t_{spmv}^s + t_{vec}^s + t_{trisol}^s.
\]

Note that the execution time of each kernel could vary based on the fault tolerance method. We know that:

\[
t_{spmv}^x = \alpha_x nnz,
\]

\[
t_{vec}^x = \beta_x n
\]

and

\[
t_{trisol}^x = \gamma_x nnz,
\]

for a method ‘x’ and \( \alpha_x, \beta_x \) and \( \gamma_x \) are method specific constants. Thus, we have:

\[
T_b = \alpha_b nnz + \beta_b n + \gamma_b nnz,
\]

\[
T_c = \alpha_c nnz + \beta_c n + \gamma_c nnz
\]

and

\[
T_b = \alpha_s nnz + \beta_s n + \gamma_s nnz.
\]

Now, we compute the overheads of C-ABFT-PCG as,

\[
T_c^o = T_c - T_b = (\alpha_c - \alpha_b) nnz + (\beta_c - \beta_b)n + (\gamma_c - \gamma_b) nnz.
\]

As C-ABFT-PCG does not protect the triangular solve operation, \( \gamma_c = \gamma_b \). Thus,

\[
T_c^o = (\alpha_c - \alpha_b) nnz + (\beta_c - \beta_b)n. \quad (5.8)
\]

For S-ABFT-PCG, \( T_s^o = T_s - T_b = (\alpha_s - \alpha_b) nnz + (\beta_s - \beta_b)n + (\gamma_s - \gamma_b) nnz \). From the previous sections, we know that the encoded SpMV and sparse triangular solution operations in S-ABFT-PCG can be considered equivalent to the SpMV and sparse triangular solution operations of the base method plus some extra vector operations. Using this information, we have,

\[
T_s^o = (\beta_s - \beta_b)n + \zeta n, \quad \text{where,} \quad \zeta \quad \text{denotes the extra vector operations. Thus,}
\]

\[
T_s^o = (\zeta + \beta_s - \beta_b)n. \quad (5.9)
\]

Comparing Equations 5.8 and 5.9, for a large average non-zero density per row of a matrix, i.e., for large \( \frac{nnz}{n} \), we expect that C-ABFT-PCG will have higher overheads than S-ABFT-PCG. This analysis considers the overheads in the event of no error. In the event of an error, C-ABFT-PCG can detect or correct a single error in SpMV without additional overheads; however, it cannot detect and correct a single error in the application of the preconditioner through sparse triangular solution. In contrast, S-ABFT-PCG will require an additional PCG iteration to correct an error that it detects, but, it guarantees detection.
and correction of an error in all kernels of PCG including triangular solution.

5.4 Experimental Evaluation

We present our experimental setup in Section 5.4.1. In Sections 5.4.2 and 5.4.3, we present the experimental results on overheads for error-free and a single error prone instances of PCG. We compare overheads for S-ABFT-PCG relative to PCG implementation with no ABFT and C-ABFT-PCG.

![Overheads for an error free run](image)

Figure 5.8: Performance degradation due to overheads of ABFT, in terms of the percentage increase in execution time to convergence.

5.4.1 Experimental Setup

We use PCG with incomplete Cholesky preconditioner with thresholding [122] as a representative iterative linear solver. We evaluate the runtime of PCG using C-ABFT-PCG and S-ABFT-PCG techniques. We use 20 SPD matrices from The University of Florida Sparse Matrix Collection [102]. Table 5.1 lists the properties of the these matrices. The first column represents matrix ids and the second column gives matrix names. The third and fourth columns represent, respectively, the matrix dimension and number of non-zero elements in the matrix.
<table>
<thead>
<tr>
<th>Matrix ID</th>
<th>Matrix</th>
<th>N</th>
<th>NNZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cvxbqp1</td>
<td>50,000</td>
<td>349,968</td>
</tr>
<tr>
<td>2</td>
<td>crankseg_1</td>
<td>52,804</td>
<td>10,614,210</td>
</tr>
<tr>
<td>3</td>
<td>nasasrb</td>
<td>54,870</td>
<td>2,677,324</td>
</tr>
<tr>
<td>4</td>
<td>Andrews</td>
<td>60,000</td>
<td>760,154</td>
</tr>
<tr>
<td>5</td>
<td>qa8fm</td>
<td>66,127</td>
<td>1,660,579</td>
</tr>
<tr>
<td>6</td>
<td>cfd1</td>
<td>70,656</td>
<td>1,828,364</td>
</tr>
<tr>
<td>7</td>
<td>finan512</td>
<td>74,752</td>
<td>596,992</td>
</tr>
<tr>
<td>8</td>
<td>thermal1</td>
<td>82,654</td>
<td>574,458</td>
</tr>
<tr>
<td>9</td>
<td>consph</td>
<td>83,334</td>
<td>6,010,480</td>
</tr>
<tr>
<td>10</td>
<td>s3dkq4m2</td>
<td>90,449</td>
<td>4,820,891</td>
</tr>
<tr>
<td>11</td>
<td>shipsec8</td>
<td>114,919</td>
<td>6,653,399</td>
</tr>
<tr>
<td>12</td>
<td>ship_003</td>
<td>121,728</td>
<td>8,086,034</td>
</tr>
<tr>
<td>13</td>
<td>cfd2</td>
<td>123,440</td>
<td>3,087,898</td>
</tr>
<tr>
<td>14</td>
<td>boneS01</td>
<td>127,224</td>
<td>6,715,152</td>
</tr>
<tr>
<td>15</td>
<td>shipsec1</td>
<td>140,874</td>
<td>7,813,404</td>
</tr>
<tr>
<td>16</td>
<td>Dubcova3</td>
<td>146,689</td>
<td>3,636,649</td>
</tr>
<tr>
<td>17</td>
<td>shipsec5</td>
<td>179,860</td>
<td>10,113,096</td>
</tr>
<tr>
<td>18</td>
<td>thermomech_dM</td>
<td>204,316</td>
<td>1,423,116</td>
</tr>
<tr>
<td>19</td>
<td>hood</td>
<td>220,542</td>
<td>10,768,436</td>
</tr>
<tr>
<td>20</td>
<td>offshore</td>
<td>259,789</td>
<td>4,242,673</td>
</tr>
</tbody>
</table>

Table 5.1: The UFL benchmark matrices with their dimension (N) and number of non-zeroes (NNZ).

### 5.4.2 Overheads for Error-Free PCG instances

We present experimental results comparing C-ABFT-PCG and S-ABFT-PCG to traditional PCG without fault tolerance. We refer to the PCG implementation without fault tolerance as our *base scheme*. Figure 5.8 shows the relative performance degradation of three different fault tolerant techniques with respect to the base scheme. The technique, S-ABFT-SpMV-PCG, refers to our S-ABFT-PCG with no fault tolerance for the sparse triangular solution operation. This technique gives a fair comparison between S-ABFT and C-ABFT as both protect the same kernels (SpMV and vector operations) within PCG, whereas, S-ABFT-PCG protects all kernels within PCG, including sparse triangular solution. Observe that fault tolerance overheads for an error free run can be as high as 37.4%, 17.8% and 24.1% for C-ABFT-PCG, S-ABFT-SpMV-PCG and S-ABFT-PCG techniques, respectively. The average fault tolerance overheads are 23.1%, 7.5% and 11.3%, for C-ABFT-PCG, S-ABFT-SpMV-PCG and S-ABFT-PCG techniques, respectively, as indicated by the horizontal lines.
5.4.3 Overheads for Single Error PCG instances

We now evaluate the benefits to the performance of PCG in the event of a single error. Recall that differential impacts are expected only for errors in the triangular solve kernel because both C-ABFT-PCG and S-ABFT-PCG can detect and recover from an error in SpMV. Results in [112] indicate that the component in which error occur impacts convergence, with greatest degradations corresponding to rows with high norm values. In Figure 5.9, we report the maximum observed performance degradation of C-ABFT-PCG and S-ABFT-PCG with an error in the component corresponding to a median value of the row norm during sparse triangular solution. Observe that C-ABFT-PCG can have significantly high performance degradation of over 250% increase in iteration counts in the worst case and 63.2% on average. Overheads of S-ABFT-PCG are 3.2% on average for recovering from the error by recomputing one iteration of PCG.

5.5 Conclusion and Future Work

Resiliency is a key issue as we move toward peta-to-exascale HPC systems that are expected to encounter multiple faults within a day [12], with faults ranging from fail-stop failures to silent errors. A natural concern is the vulnerability of long running scientific applications on such HPC systems, often involving computations with very large sparse matrices [6].
Such applications can benefit from algorithm-based fault tolerance (ABFT) techniques that offer detection and recovery from errors at the expense of low overheads. Although ABFT techniques have been developed for dense matrix operations, they have been largely ignored for sparse matrix operations.

We consider sparse linear system solution using preconditioned conjugate gradients, a popular choice when the coefficient matrix is sparse and symmetric positive definite. We develop a new sparse checksum encoded algorithm-based fault tolerant PCG, S-ABFT-PCG, that can provably detect a single error in the data for PCG when the matrix is strictly diagonally dominant. Our approach trades-off the ability to correct a single error in the data in favor of lower overheads per PCG iteration at the expense of recomputing a single PCG iteration when an error occurs. Our experimental evaluations indicate that in the event of no errors, compared to a PCG with no ABFT, the overheads of S-ABFT-PCG are 11.3% on average. For comparison, the overheads of applying the classical ABFT to the SpMV kernel of PCG, C-ABFT-PCG, are on average higher at 23.1%. Furthermore, in the event of a single error in the application of the preconditioner through triangular solution, C-ABFT-PCG suffers from significant increases in iteration counts to convergence, with overheads of 63.2% on average compared to 3.2% on average for S-ABFT-PCG. Our evaluations motivate the need for further study of algorithm-based fault tolerance approaches for widely used sparse computations in large-scale computational modeling and simulation software.
Chapter 6

Conclusions

High performance computing enables advances in diverse areas of science ranging from biology, astrophysics, materials, and environmental sciences through many scientific simulations that are based on PDE models [6]. Sparse scientific computing is at the heart of such simulations. In this thesis, we have shown the existence of an opportunity space to optimize sparse scientific applications for performance defined by measures, such as speedup, turnaround time and energy costs, and resilience, on HPC systems. To effectively use this opportunity space, we have developed new algorithmic formulations that enhanced performance and resiliency of sparse scientific computations.

In the first part of this thesis, we developed algorithmic approaches to enhance the performance of sparse scientific computations defined by metrics such as execution time, throughput and energy costs. In particular, we focused on sparse scientific computations resulting from (i) sparse matrix vector multiplication (SpMV) (in Chapter 2) and (ii) sparse scientific workload (in Chapter 3).

In Chapter 2, we presented a new SpMV algorithm that exploited the dense substructures that are inherently present in many sparse matrices derived from PDE models [7]. We developed a new compressed sparse matrix representation that grouped indistinguishable rows and columns within sparse matrices to form effectively dense blocks without incurring fill-in. Furthermore, we implemented a new SpMV algorithm to take advantage of the new sparse matrix format that reduced the number of load operations and enhanced locality in accesses to the vector. Our evaluations showed that our technique improved the SpMV performance on average by over a third compared to the traditional compressed sparse row scheme on the Intel Nehalem processor.

In Chapter 3, we considered improving the performance of scientific workloads comprising sparse applications having sub-linear fixed-problem-speedup on increasing processor...
counts. We developed speedup-aware processor assignment algorithms that exploit individual application scaling features to reduce workload completion time and decrease system energy consumption. Our results indicated that the speedup-aware algorithms reduced workload completion and total energy consumption of workloads compared to the traditional first-come-first-serve algorithm on 128 cores of the Intel Nehalem processor.

In the second part of this thesis, we examined the impact of transient errors on execution time and accuracy of the sparse linear system solution in Chapter 4 and presented a new algorithm based fault tolerance technique to enhance resilience of sparse linear system solution in Chapter 5.

In Chapter 4, our theoretical analysis illustrates that a single transient error during an SpMV operation can corrupt the entire output vector in a relatively short sequence of SpMV operations [8]. Additionally, our evaluations indicated that execution time of sparse linear system solution through the PCG method could increase by a factor as high as 200 or more in the event of a single transient error. Avoiding such slowdowns can be particularly important from the perspective of efficient use of large-scale facilities. In Chapter 5, we developed a new sparse checksum encoded algorithm-based fault tolerance technique for the PCG method that detects a single error in all the key operations within the method [9]. Our technique trades-off the ability to correct a single error in the data in favor of lower overheads per PCG iteration at the expense of recomputing a single PCG iteration when an error occurs. The overheads of using our fault tolerance technique were low, eleven percent on average in the event of no error, and three percent in the event of a single error within PCG, when compared to having no fault tolerance for the PCG method.

In summary, our work in this thesis is an important step toward the development of algorithmic approaches to improve performance and resilience of sparse scientific computations. Looking forward, we expect our analyses and characterization of the interactions between sparse scientific applications and hardware to yield predictive performance models. Such models can inform the development of optimizations that can improve a range of performance measures. Additionally, our algorithmic formulations motivate the development of method-specific multiple-objective optimizations that trade-off different aspects of performance including speedup, scalability, execution times, application resiliency, system energy and throughput. We foresee such efforts to play a vital role in auto-tuning frameworks developed for efficient and reliable sparse scientific computations on current and next-generation advanced systems.
Bibliography


URL http://portal.acm.org/citation.cfm?id=838237.838463


URL http://dx.doi.org/10.1109/TPDS.2003.1206505

URL http://portal.acm.org/citation.cfm?id=173497.173499

URL http://portal.acm.org/citation.cfm?id=646376.689372


URL http://lammps.sandia.gov


Vita

Manu Shantharam

Education

Ph.D.  Computer Science & Engineering
The Pennsylvania State University
University Park, PA. 2012

B.E.  Computer Science & Engineering
University of Mysore
Mysore, India. 2001

Publications


Prestigious Awards

[1] Best poster award, SC 2010, USA

[2] 8th Rank in Regional Mathematics Olympiad Karnataka 1996, India