Höchstleistungsrechenzentrum HLRS
Universität Stuttgart
Prof. Dr.-Ing. M. Resch
Nobelstrasse 19
70569 Stuttgart

Efficient Solution of Sparse Linear Systems Arising in Engineering Applications on Vector Hardware

Von der Fakultät Energie-, Verfahrens- und
Biotechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Abhandlung

vorgelegt von

MSc. Sunil Reddy Tiyyagura

aus Tenali/Andhra Pradesh/Indien

Hauptberichter:	Prof. Dr.-Ing. Michael Resch
Mitberichter:	Prof. Dr.-Ing. Eberhard Göde
Tag der Einreichung:	25.09.2008
Tag der mündlichen Prüfung:	08.07.2010

July 2010

Höchstleistungsrechenzentrum Stuttgart (HLRS)
Universität Stuttgart
Nobelstrasse 19
D-70569 Stuttgart

Zusammenfassung

Block-Based-Linear-Iterative-Solver (BLIS) ist eine skalierbare numerische Software-Bibliothek zur Lösung großer schwachbesetzter linearer Gleichungssysteme, wie sie besonders in Ingenieursanwendungen auftreten. BLIS wurde entwickelt mit Rücksicht auf die Besonderheiten von parallelen Vektorsystemen. Erreichen hoher Gleitkommaleistung war unser zentrales Ziel. Einige fehlgeschlagene Versuche auf dem Vektorrechner NEC SX-8 mit üblichen Public Domain Solvern machten deutlich, dass die Entwickler keine Rücksicht auf die Eigenschaften von Vektorsystemen genommen haben. Dies gilt ebenso für andere Softwareprojekte auf Grund der Tatsache, dass Cluster von Skalarprozessoren das Höchstleistungsrechnen der letzten Jahrzehnte dominiert haben. Inzwischen haben aber Vektorrecheneinheiten in skalare Prozessoren Einzug gehalten. Vektorisierbare Algorithmen und deren Implementierungen erhalten größere Bedeutung. Um die Unterschiede der verschiedenen Architekturen bewerten zu können, haben wir verschiedene Benchmark-Studien durchgeführt. Diese Studien haben nachgewiesen, dass Vektorsysteme besser balanciert sind als die üblichen Skalarsysteme, wenn Aspekte betrachtet werden, die die sustained Performance vieler realer Anwendungen bestimmen.

Zwei wesentliche Probleme werden in dieser Arbeit adressiert, die die Leistungsfähigkeit von Public Domain Solvern begrenzen. Das Problem der ungeeigneten kurzen Vektorlänge konnte durch die Verwendung eines vektorgeeigneten Datenformates für schwach besetzte Matrizen gelöst werden. Dieses Format ist gleichermaßen auch für moderne Skalarprozessoren geeignet. Das zweite Problem der schädlichen Auswirkung der hohen Memory-Latenzzeiten konnte durch die Verwendung von Blöcken in der Matrixstruktur gelöst werden. Die meisten Ingenieursanwendungen weisen mehrfache Unbekannte (Freiheitsgrade) an jedem Knotenpunkt des zu lösenden Problems auf. Die typischen Public Domain Löser nutzen diese Besonderheit nicht. Stattdessen assemblieren und lösen sie jede Unbekannte separat und vergrößern damit den Druck auf das Memorysystem insbesondere beim indirekten Zugriff. Unser Ansatz veringert diesen Druck durch das Blocking der Knotenpunkt-Variablen und der Lösung des daraus resultierenden globalen Systems von Block-Gleichungen. Dies ist ein natürlicher Ansatz für Simulationen aus dem Ingenieursumfeld und ermöglicht Leistungsverbesserungen auf Skalarsystemen durch Cache-Blocking und auf Vektorsystemen durch Reduk-

tion des Memory-Verkehrs.

Präkonditionierung ist eines der Gebiete der Lösungsverfahren linearer Gleichungssysteme, auf dem noch immer aktiv geforscht wird. Ein präkonditioniertes Gleichungssystem hat bessere spektrale Eigenschaften. Deshalb konvergieren die Lösungsmethoden schneller als beim originalen System. Schlüssel zu einem erfolgreichen Präkonditionierungsverfahrens ist, die Konditionszahl der veränderten Matrix möglichst klein zu halten bei geringem zusätzlichem Zeitaufwand für die Präkonditionierung. Blockbasierte Splitting-Methoden und blockbasierte Skalierung sind numerisch effektivere Präkonditionierer als ihr punktbasiertes Gegenstück und benutzen die Hardware effizienter. Blockbasierte unvollständige Zerlegung (ILU), wie sie in BLIS implementiert wurde, ist ebenfalls effizienter als das entsprechende punktbasierte Verfahren. Robuste skalierbare Präkonditionierer wie die Algebraische Multigridmethode sind ebenfalls in BLIS verfügbar.

Leistungsmessungen auf der NEC SX-8 für drei Anwendungsprogramme unter Benutzung von BLIS werden dargestellt. Darüberhinaus werden Bandbreitenbegrenzungen neuer Hardware-Architekturen wie der STI Cell Broadband Engine analysiert. Effizienz und Skalierung von BLIS auf Multi-Core Systemen wird getestet. Die Leistungsfähigkeit des Matrix mal Vektor Kerns für schwachbesetzte Matrizen mit Blöcken wird getestet und beschrieben.

Abstract

Block-based Linear Iterative Solver (BLIS) is a scalable software library for solving large sparse linear systems, especially arising from engineering applications. BLIS has been developed during this work to particularly take care of the performance issues related to Krylov iterative methods on vector systems. After several failed attempts to port some general public domain linear solvers onto the NEC SX-8, it is clear that the developers of most solver libraries do not focus on performance issues related to vector systems. This is also true for other software projects due to the fact that clusters of scalar processors were the dominant high performance computing installations in the past few decades. With the advent of vector processing units on most commodity scalar processors, vectorization is again becoming an important software design consideration. In order to understand the strengths and weaknesses of various hardware architectures, benchmarking studies have been done in this work. These studies show that the vector systems are well balanced than most scalar systems with respect to many aspects that determine the sustained performance of many real world applications.

The two main performance problems with the public domain solvers are addressed in this work. The first problem of short vector length is solved by introducing a vector specific sparse storage format. The second and the more important problem of high memory latencies is addressed by blocking the sparse matrix. Most engineering problems have multiple unknowns (degrees of freedom) per mesh point to be solved. Typically, public domain solvers do not block all the unknowns to be solved at each mesh point. Instead, they assemble and solve each unknown separately which requires a huge amount of memory traffic. The approach adopted in this work reduces the load on the memory subsystem by blocking all the unknowns at each mesh point and then solving the resulting blocked global system of equations. This is a natural approach for engineering simulations and results in performance improvement on scalar systems due to cache blocking and on vector systems due to reduced memory traffic.

Preconditioning is one of the areas in linear solvers that is still actively researched. A preconditioned system of equations has better spectral properties and hence the solution methods will converge faster than with the original system. The key consideration is to keep the time needed for the additional work of

preparing the preconditioner as low as possible while at the same time improving the condition number of the resulting system as much as possible. Block based splitting methods and scaling are effective preconditioners than their point based counterparts and at the same time are also efficient. Block based incomplete factorization implemented in BLIS is also more efficient than the corresponding point based method. Robust scalable preconditioners such as the algebraic multi-grid method are also available in BLIS. The performance measurements of three application codes running on the NEC SX-8 and using BLIS to solve the linear systems are presented.

Lastly, memory bandwidth limitations of new hardware architectures such as the multi-core systems and the STI CELL Broadband Engine are studied. The efficiency and scaling of BLIS is tested on the multi-core systems. Also, the performance of blocked sparse matrix vector product kernel is studied on the STI CELL processor.

Acknowledgements

I would like to thank Prof. Dr.-Ing Michael Resch for giving me an opportunity to work at HLRS. I would also like to thank Prof. Dr.-Ing Eberhard Göde for accepting to be on my examination committee.

I am greatly indebted to Mr. Uwe Küster for all his valuable advice during my work at HLRS. I really admire his personality, vision and an in-depth understanding of real problems. I would also like to point out his spending time with young students to help them make understand the real issues related to scientific computing, which at the time of my Masters study was of immense help. It has been my pleasure to have worked under his supervision.

I would like to express my deep sense of gratitude to Peter Lammers and Katharina Benkert, my Teraflop Workbench project colleagues. Their help and support was very useful for my stay at HLRS. I would like to thank Sabine Roller for making Research also as a priority in the Teraflop Workbench project. Her involvement in the project gave it a right direction and was specially very helpful to me. She is the best *academic* manager i have ever worked with and i am also thankful to her for making my extended stay at HLRS possible.

I would like to thank Malte von Scheven for all his help and support with running the applications and coupling the solver to CCARAT. I would like to thank Ralf Schneider for preparing the input files required for the cancellous bone simulation. I have learned greatly from my interactions with Rainer Keller, who has immense experience with MPI libraries and HPC tools. I deeply admire his passion and unlimited enthusiasm for what he does. I would like to thank Harald Klimach for his help and discussions. I would also like to thank Stefan Borowski who worked for NEC in the Teraflop Workbench project. I learned greatly from his approach towards hard to crack issues. His attention to detail is really commendable.

I would like to thank Thomas Beisel and Panagiotis Adamidis, die-hard VFB Stuttgart fans, for having introduced me to football. I really loved our occasional visits to the stadium for VFB Stuttgart matches. I would like to thank Bernd Krischok for his advice on how to go about learning paragliding. I would like to thank Dieter Raith for sharing his expert opinions on everything that happened and is happening in Germany. I as a foreigner could get a better perspective on local issues from such discussions. I would like to thank Natalia Currle-Linde for

her help and support. I am thankful to Ina Arnold, who always has a smiling face and infinite patience, for her help with administrative matters during my stay at HLRS.

I would like to thank all the NEC colleagues working at HLRS (Stefan Haberhauer, Holger Berger, Danny Sternkopf and Martin Ostertag) for their excellent support and friendly discussions. I would like to thank Rudolf Fischer for his support and help. I am thankful to NEC for its overall support within the Teraflop Workbench project.

I am thankful to Rolf Rabenseifner for all the discussions we had. I would like to thank Frank Pister for his help with programming on the STI CELL processor. I would also like to thank both my earlier project managers, Matthias Müller and Thomas Bönisch, for their support. I am thankful to Michael Schliephake and Tünde Erdei for their friendly words. I am really thankful to all my colleagues at HLRS for making my stay in Germany a pleasant experience.

I would finally like to thank my Mother and both my Sisters for encouraging me to follow my passion. My stay in Germany would not have been possible without their support and understanding.

Contents

1 Introduction **1**
- 1.1 Complexity of solving linear systems . . . 1
- 1.2 Iterative methods . . . 2
  - 1.2.1 Stationary methods . . . 3
  - 1.2.2 Non-stationary methods . . . 6
- 1.3 Hardware architectures . . . 11
  - 1.3.1 NEC SX-8 . . . 11
  - 1.3.2 Intel Xeon cluster . . . 15
  - 1.3.3 Commodity multicore processors . . . 15
  - 1.3.4 STI CELL BE . . . 15
- 1.4 Motivation and outline . . . 16

2 Analysis of modern hardware architectures **17**
- 2.1 HPC challenge suite . . . 18
  - 2.1.1 Comparing the uploaded results . . . 18
  - 2.1.2 Scaling on different architectures . . . 20
- 2.2 Intel MPI benchmark suite . . . 25
  - 2.2.1 Single transfer benchmarks . . . 25
  - 2.2.2 Parallel transfer benchmarks . . . 25
  - 2.2.3 Collective benchmarks . . . 26
  - 2.2.4 IMB benchmark results . . . 26
- 2.3 Conclusions and consequences for sparse iterative methods . . . 32
  - 2.3.1 HPCC benchmark suite . . . 32
  - 2.3.2 IMB benchmark suite . . . 33

3 Implementation of a vectorized sparse solver **35**
- 3.1 Sparse storage formats . . . 36
  - 3.1.1 Block storage formats . . . 38
  - 3.1.2 Recent work on vector specific storage formats . . . 39
- 3.2 Improvements to the JAD sparse MVP algorithm . . . 40
  - 3.2.1 Original JAD MVP algorithm . . . 41
  - 3.2.2 Working on groups of diagonals . . . 42
  - 3.2.3 Use of vector registers . . . 42
  - 3.2.4 Performance of AZTEC on NEC SX-8 . . . 43
- 3.3 Block-based approach . . . 45
  - 3.3.1 Avoiding bank conflicts on the vector system . . . 46
  - 3.3.2 Performance in sparse MVP with increasing block size . . . 49

3.3.3 Related work on blocking sparse matrices . . . 51
3.4 Improvements to incomplete factorization algorithm . . . 54
3.5 Scheduling communication . . . 57
3.5.1 Implementation in BLIS . . . 57
3.5.2 Different MPI implementations and performance . . . 58
3.6 Coloring and vectorization . . . 58
3.6.1 Communication coloring . . . 63
3.7 Preconditioning . . . 65
3.7.1 Block vs point based preconditioning . . . 68
3.7.2 Algebraic multigrid . . . 69
3.8 Available features in the solver . . . 75

4 Usage of BLIS in applications **77**
4.1 CCARAT . . . 77
4.1.1 Vectorization of element integration . . . 78
4.1.2 BLIS performance . . . 80
4.1.3 Simulations using BLIS . . . 87

5 Innovative hardware architectures and challenges **93**
5.1 Memory bandwidth challenge . . . 94
5.2 Multi-core performance with BLIS . . . 94
5.3 STI CELL BE performance with BLIS . . . 98

6 Conclusions and future work **103**

Bibliography **105**

A BLIS Interface **115**
A.1 Implemented algorithms . . . 115
A.2 BLIS user options . . . 115
A.3 List of C functions . . . 116
A.4 Fortran interface . . . 123
A.4.1 Implementation details . . . 123
A.4.2 List of functions . . . 123

List of Figures

1.1 Installation at HLRS. 11
1.2 CPU architecture of SX-8. 12
1.3 Concept of Bank Caching in SX-8 node. 13

2.1 Ratio based analysis for global (left) and embarrassingly parallel (right) benchmarks in HPCC suite. 19
2.2 STREAM Copy bandwidth Vs. HPL (black), RandomRing bandwidth Vs. HPL (gray). 20
2.3 Accumulated Random Ring Bandwidth versus HPL performance . 22
2.4 Accumulated Random Ring Bandwidth ratio versus HPL performance 22
2.5 Accumulated EP Stream Copy versus HPL performance 23
2.6 Accumulated EP Stream Copy ratio versus HPL performance . . 23
2.7 Comparison of all the benchmarks normalized with HPL value . . 24
2.8 Execution time of Barrier benchmark (the smaller the better) . . 27
2.9 Execution time of Allreduce benchmark for a message size of 1MB (the smaller the better) 27
2.10 Execution time of Reduction benchmark for a message size of 1MB (the smaller the better) 28
2.11 Execution time of Allgather benchmark for a message size of 1MB (the smaller the better) 29
2.12 Execution time of Allgatherv benchmark for a message size of 1MB (the smaller the better) 29
2.13 Execution time of AlltoAll benchmark for a message size of 1MB (the smaller the better) 30
2.14 Bandwidth of Sendrecv benchmark on varying number of processors, using a message size of 1MB 31
2.15 Bandwidth of Exchange benchmark on varying number of processors, using a message size of 1MB 31
2.16 Execution time of Broadcast benchmark for a message size of 1MB (the smaller the better) 32

3.1 Sparse storage formats. 37
3.2 Block sparse storage formats. 38
3.3 Original JAD MVP algorithm. 41
3.4 Operation flow diagram for original JAD MVP algorithm. 41
3.5 JAD MVP algorithm grouping at most 2 diagonals. 42
3.6 JAD MVP algorithm using vector registers. 43

3.7 Single CPU performance of modified AZTEC kernels on NEC SX-8. 44
3.8 Block based JAD MVP algorithm (for 2×2 blocks). 45
3.9 Operation flow diagram for block based JAD MVP algorithm. . . 45
3.10 Influence of bank conflicts on DAXPY performance on the NEC SX-8. The highest performance level of about 4.5 GFlop/s corresponds to odd array strides and stride 2. The next performance level of about 2.5 GFlop/s corresponds to array stride 4. Each performance level that proceeds corresponds to an even stride that is double that of the previous level. 46
3.11 Influence of bank conflicts on DAXPY performance on the NEC SX-6+. The highest performance level of about 2.7 GFlop/s corresponds to odd array strides. The next performance level of about 1.7 GFlop/s corresponds to array stride 2. Each performance level that proceeds corresponds to an even stride that is double that of the previous level. 47
3.12 Array placement into memory banks on the NEC SX-8. 47
3.13 Influence of bank conflicts on blocked sparse MVP performance. . 49
3.14 Performance in JAD sparse MVP with different block sizes. The x-axis is a logarithmic scale. 50
3.15 Performance in JAD sparse MVP for small problem sizes with different block sizes. The x-axis is a linear scale. 50
3.16 Performance in JAD sparse MVP with different pseudo diagonals. 51
3.17 Point or block IKJ variant of the ILU factorization algorithm. . . 55
3.18 Scatter-Gather approach to the point or block IKJ LU factorization algorithm. 56
3.19 Greedy coloring algorithm. 59
3.20 Reordering nodes on each processor (global problem size of 10065 nodes) using greedy coloring algorithm. 60
3.21 Reordering nodes on each processor (global problem size of 10065 nodes) using greedy coloring algorithm with a cutoff color size of 10. 60
3.22 Reordering the nodes on each processor (global problem size of 37760 nodes) using greedy coloring algorithm. 61
3.23 Reordering nodes on each processor (global problem size of 37760 nodes) using greedy coloring algorithm with a cutoff color size of 100. 61
3.24 Reordering nodes on each processor (global problem size of 677894 nodes using greedy coloring algorithm. 62
3.25 Reordering nodes on each processor (global problem size of 677894 nodes) using greedy coloring algorithm with a cutoff color size of 200. 62
3.26 Edge Coloring point-to-point communication for a 32 CPU run. . 64
3.27 Vampir trace output of one BiCGSTAB iteration on 8 SX-8 CPUs (left) and 32 SX-8 CPUS (right) for the same fluid problem. . . . 66
3.28 Vampir trace output of time (sec) needed in one BiCGSTAB iteration on 8 SX-8 CPUs (left) and 32 SX-8 CPUS (right) for the same fluid problem. Green corresponds to time in calculation and red to time in communication. 66

3.29 Convergence in BiCGSTAB with Block Jacobi as preconditioner (left) and with AMG as preconditioner (right) for solving a fluid problem with 8K unknowns. 74
3.30 Convergence in BiCGSTAB with Block Jacobi as preconditioner (left) and with AMG as preconditioner (right) for solving a fluid problem with 40K unknowns. 75
3.31 Single CPU performance of Sparse MVP on NEC SX-8. 76

4.1 Original (left) and modified (right) flow of instructions in element integration. 79
4.2 Performance improvement (factor) with the modified element implementation on NEC SX-6 and HP Itanium 2 systems. 80
4.3 The geometry and boundary conditions of the flow around a cylinder with a square cross-section. 81
4.4 Velocity in the midplane of the channel. 81
4.5 Scaling of BLIS wrt. problem size on NEC SX-8 (left) Computation to communication ratio in BLIS on NEC SX-8 (right). 82
4.6 Elapsed time (seconds) in linear solver. 83
4.7 Memory degradation on the NEC SX-8 with direct (STREAM benchmark - left) and indirect memory addressing (right). 85
4.8 Memory performance on a single node NEC SX-8 with STREAM Triad for small problem sizes (left) and large problem sizes (right). 86
4.9 Performance degradation in BLIS with increasing number of CPUs per node. 86
4.10 Rigid membrane roof: geometry and material parameters 88
4.11 Sectional view of a x-velocity field over the rigid roof 89
4.12 Discretized bone geometry. 90
4.13 Mean Displacement contour fill for a 300 N/mm^2 compressive loading. 91
4.14 Stresses in Z direction for a 300 N/mm^2 compressive loading. . . . 91

5.1 Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 151k equations. 95
5.2 Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 151k equations. 95
5.3 Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 151k equations. 96
5.4 Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 151k equations. 96
5.5 Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 40k equations. 97
5.6 Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 40k equations. 97
5.7 Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 40k equations. 98
5.8 Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 40k equations. 98

5.9 Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 8k equations. 99
5.10 Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 8k equations. 99
5.11 Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 8k equations. 100
5.12 Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 8k equations. 100

List of Tables

2.1 System characteristics of the computing platforms. 21
2.2 Ratio values corresponding to 1 in Figure 2.7. 24

3.1 Performance comparison of Block ILU implementations in BLIS. . 57
3.2 Performance comparison of different point-to-point MPI implementations in BLIS. 58
3.3 Performance of colored Symmetric Gauss Seidel preconditioner including setup time for coloring and rearranging. 63
3.4 Performance comparison of colored point-to-point MPI implementations in BLIS. 64
3.5 Comparison of different methods in HYPRE used for solving a 2d 500x500 Laplace example on the NEC SX-8. 68
3.6 Performance comparison for AMG kernels between NEC SX-8 and Itanium 2. 75

4.1 Different discretizations of the introduced example. 82
4.2 Performance comparison in solver between SX-8 and Xeon for a fluid structure interaction example with 25168 fluid equations and 26352 structural equations. 84
4.3 Performance in linear solvers for the membrane roof example. . . 88
4.4 Performance in linear solvers for the bone simulation. 91

5.1 Performance comparison for scalar, VMX and SPU implementations on the STI CELL BE 101

Chapter 1

Introduction

Sparse linear system of equations arise in a wide range of scientific and engineering applications that discretize PDEs using Finite Element, Finite Volume or Finite Difference methods. As computer simulation establishes itself as a supplement or in some cases even as an alternative to experiments, there has been an ever growing demand for more accurate computer models and very fine meshes which in turn leads to a demand for larger computing resources.

The present trend in high performance computing is to connect more and more commodity processors using innovative interconnects. But, this approach to increase the compute capacity places a huge demand on scientists and engineers to scale their applications to hundreds of thousands of processors. Algorithms that need global synchronization at regular intervals do not scale well on such architectures. Even small load imbalances due to partitioning can seriously limit strong scaling of applications on to a large processor count. But imbalances play a comparatively lesser role in weak scaling where the problem size is also proportionally increased, resulting in each processor spending more time in computation rather than synchronization. The most important consideration while selecting an algorithm is its linear scalability with increasing problem size. This is of course not always possible, which means that for most applications the increase in computing resources does not mean a proportional increase in problem size. Algorithmic scalability in the context of solving sparse linear systems is discussed in the following section.

1.1 Complexity of solving linear systems

The widely used method to solve a dense linear system is matrix factorization followed by forward and backward substitutions. The complexity to factorize a matrix into lower and upper triangular factors is $O(N^3)$ and the substitutions require $O(N^2)$ operations. The main drawback of this method when used for sparse systems is that it normally leads to fill-in which may result in nearly

dense triangular factors even for sparse matrices. This cannot be acceptable if the size of the application is already limited by the available memory because dense matrices would need prohibitively large memory when compared to sparse matrices for large problems. Another disadvantage for large systems (both dense and sparse) is the time complexity of factorization algorithms.

Iterative methods on the other hand preserve the sparsity pattern of the linear system. Hence, they are a better choice than direct methods for solving large sparse linear systems as no additional memory is needed. They are further subdivided on the basis of whether or not the computations in each iteration involve changing information into stationary and non-stationary methods. The main drawback of stationary methods is their slow convergence. Spectral analysis of the matrix reveals that stationary methods remove the high frequency error components quickly but are very slow at resolving the lower frequency error components. Non-stationary methods are more robust than the stationary methods and for Symmetric Positive Definite (SPD) matrices the method of Conjugate Gradients (CG) is widely used. But, one drawback of the CG method is the worst case convergence which can take N iterations (without floating point rounding errors), where N is the size of the Krylov subspace, which is also the total number of unknowns to be solved. The most time consuming part in each CG iteration is the matrix vector product (MVP) which has a time complexity of $O(nnz)$ for sparse matrices where nnz is the total number of matrix non-zero entries. Hence, the total time complexity of the CG method is utmost $O(N \times nnz)$. Though, in most cases the method needs far less than N iterations to converge.

Although iterative methods have a lower complexity of $O(N \times nnz)$ for solving sparse systems when compared to direct methods, they are still not optimal. Multigrid methods are asymptotically linear in complexity and thereby qualify as truly scalable solvers. They eliminate the drawback of stationary methods by accelerating the elimination of also the low frequency error components. The main idea is to evaluate the error on a coarser grid by using a restricted residual originally evaluated on the fine grid. The coarse grid operator eliminates the low frequency error components that could not be quickly eliminated on the finer grid as they become relatively higher frequency error components with respect to the coarse grid. The new coarse grid correction is interpolated back to the fine grid to update the fine grid solution. A true multigrid method generates a hierarchy of grids till a point where it is possible to directly solve the coarsest grid problem using a direct method.

1.2 Iterative methods

In this section, a brief review is provided on the iterative methods for solving linear systems. The origin of these methods can be traced back to the early nineteenth century by Carl Friedrich Gauss, a German mathematician. Iterative

methods have traditionally been used for solving large sparse linear systems. For such systems the methods of Gauss-Jacobi and Gauss-Seidel were used due to their memory benefit when compared to Gaussian Elimination. The Gauss-Seidel method was developed in the 19th century, originally by Gauss in the mid 1820s and then later by Seidel in 1874 (see references in [44]). This was followed by the method by Jacobi which forces stronger diagonal dominance by applying plane rotations to the matrix resulting in the loss of sparsity. The acceleration of the slightly more difficult to analyze Gauss-Seidel method led to point successive overrelaxation techniques introduced simultaneously by Frankel [32] and by Young [103]. In the PHD work of David Young a robust convergence theory was introduced with the help of important notions such as consistent ordering and property A.

Another important branch in iterative methods started with the development of conjugate gradient method by Lanczos [53] and Hestenes and Stiefel [42]. The ideas behind the gradient methods are based on some kind of global minimization. For instance, for positive definite symmetric matrix A, the CG method minimizes the so-called A-norm of the iterate that are in the Krylov subspace $K_i(A, r^0) \equiv \{r^0, Ar^0, ..., A^{(i-1)}r^0\}$. For some PDE problems this norm is known as Energy norm, which has physical relevance. Another interpretation of the gradient methods is that the residual is orthogonal to the space of previously generated residuals, or some related space. Both these interpretations help in the formulation and analysis of these methods. For a detailed description of the iterative methods see [11, 14, 35, 74, 78]. For historical developments in the field of iterative methods refer [15, 94] and specifically for conjugate gradient method see [36, 105]. For efficient parallel implementation of iterative methods see [14, 22]. A brief description of the iterative algorithms implemented in the sparse solver BLIS follows.

1.2.1 Stationary methods

Given an $n \times n$ real matrix A and a real vector b, the problem considered is to find x such that

$$Ax = b \tag{1.1}$$

Stationary methods can be written in the form:

$$x^{(k+1)} = Bx^{(k)} + c \tag{1.2}$$

where B and c are constants and k is the iteration count. The solution x^* of the linear system is called a stationary point of Equation 1.2, because if $x^{(k)} = x^*$, then $x^{(k+1)}$ will also equal x^*. To rewrite Equation 1.2 in terms of error

$$x^{(k+1)} \quad = \quad Bx^{(k)} + c$$

$$\begin{aligned} &= B(x^* + e^{(k)}) + c \\ &= Bx^* + c + Be^{(k)} \\ &= x^* + Be^{(k)} \\ e^{(k+1)} &= Be^{(k)} \end{aligned} \tag{1.3}$$

Each iteration does not affect the correct part of $x^{(k)}$ (because x is a stationary point) but rather only the error term. It is apparent from Equation 1.3 that if $\rho(B) < 1$, then the error term $e^{(k)}$ will converge to zero as k approaches infinity. Where the spectral radius $\rho(B)$ is defined as

$$\rho(B) = max|\lambda_i|, \qquad \lambda_i \text{ is the } i\text{th eigen value of B}$$

Hence, the initial vector $x^{(0)}$ has no effect on the inevitable outcome but rather only affects the number of iterations required to converge to x^* within a given tolerance. Hence, the rate of convergence of all the stationary methods depends largely on the spectral radius of B, which depends on A.

The main methods that fall under this category are Jacobi, Gauss-Seidel, Successive Overrelaxation (SOR) and Symmetric Successive Overrelaxation (SSOR). All these methods differ in the choice of B, depending on the way of splitting matrix A. The Jacobi method can be easily derived by examining each of the N equations in the linear system 1.1 in isolation. If in the ith equation

$$\sum_{j=1}^{N} a_{i,j}x_j = b_i$$

we solve for the value of x_i while assuming the other entries of x remain fixed, we obtain

$$x_i = (b_i - \sum_{j \neq i} a_{i,j}x_j)/a_{i,i}$$

This suggests an iterative method defined by

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j}x_j^{(k-1)})/a_{i,i} \tag{1.4}$$

which is the Jacobi method, where k represents the iteration count. Note that the order in which the equations are examined is irrelevant, since the Jacobi method treats them independently. For this reason, the Jacobi method is also known as the method of simultaneous displacements, since the updates could in principle be done simultaneously.

In matrix terms, the definition of the Jacobi method in 1.4 can be expressed as

$$x^{(k)} = D^{-1}(L+U)x^{(k-1)} + D^{-1}b \tag{1.5}$$

where the matrices $D, -L$ and $-U$ represent the diagonal, the strictly lower-triangular, and the strictly upper-triangular parts of A, respectively.

Similarly, the Gauss-Seidel iteration corrects the ith component of the current approximate solution, in the order $i = 1, 2, ..., N$ again to annihilate the ith component of the residual. However, this time the approximate solution is updated immediately after the new component is determined. The result at the ith step is

$$\sum_{j=1}^{i-1} a_{i,j}x_j^{(k)} + a_{i,i}x_i^{(k)} + \sum_{j=i+1}^{N} a_{i,j}x_j^{(k-1)} = b_i$$

which leads to the iteration

$$x_i^{(k)} = \left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k)} - \sum_{j=i+1}^{N} a_{i,j}x_j^{(k-1)} \right) / a_{i,i} \tag{1.6}$$

This in the matrix notation is

$$x^{(k)} = (D-L)^{-1}Ux^{(k-1)} + (D-L)^{-1}b \tag{1.7}$$

A backward Gauss-Seidel iteration can be defined as

$$(D-U)x^{(k)} = Lx^{(k-1)} + b \tag{1.8}$$

which is equivalent to making the coordinate corrections in the order $N, N-1, ..., 1$. A symmetric Gauss-Seidel iteration consists of a forward sweep followed by a backward sweep.

The successive overrelaxation method, or SOR, is devised by applying extrapolation to the Gauss-Seidel method. This extrapolation is a weighted average between the current approximate solution of the ith component of the Gauss-Seidel method and the ith component of the approximate solution from the previous iteration $k-1$ given as

$$x_i^{(k)} = \omega\bar{x}_i^{(k)} + (1-\omega)x_i^{(k)} \tag{1.9}$$

where $\bar{x}$ denotes a Gauss-Seidel iterate, and ω is the extrapolation factor. The idea is to choose a value for ω that will accelerate the rate of convergence to the solution. In matrix notation, the SOR algorithm can be written as follows:

$$x^{(k)} = (D-\omega L)^{-1}(\omega U + (1-\omega)D)x^{(k-1)} + (D-\omega L)^{-1}b \tag{1.10}$$

If $\omega = 1$, the SOR method simplifies to the Gauss-Seidel method. A theorem due to Kahan [49] shows that SOR fails to converge if ω is outside the interval $(0, 2)$. It is generally not possible to compute the optimal value of ω. Frequently, some heuristic estimate is used, such as $\omega = 2-O(h)$, where h is the mesh spacing of the discretization of the underlying physical domain. For SPD matrices, the SOR method is guaranteed to converge for any value of ω between 0 and 2, though the choice of ω can significantly affect the rate of convergence. For coefficient matrices of a special class called consistently ordered with property A [104], which includes certain orderings of the matrices arising from the discretization of elliptic PDEs, there is a direct relationship between the spectra of the Jacobi and SOR iteration matrices. In principle, given the spectral radius ρ of the Jacobi iteration matrix, one can determine a priori the theoretically optimal value of ω for SOR:

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2}} \tag{1.11}$$

Finding the optimal relaxation parameter is not practical as it requires a large amount of computation. In symmetric successive overrelaxation (SSOR) method the forward SOR sweep is followed by a backward SOR by reversing update of the unknowns.

$$x^{(k)} = B_1 B_2 x^{(k-1)} + \omega(2 - \omega)(D - \omega U)^{-1} D (D - \omega L)^{-1} b \tag{1.12}$$

where

$$B_1 = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)$$
$$B_2 = (D - \omega U)^{-1}(\omega L + (1 - \omega)D)$$

Note that B_2 is simply the iteration matrix for the SOR from 1.10, and that B_1 is the same, but with the roles of L and U reversed.

1.2.2 Non-stationary methods

Nonstationary methods differ from the stationary methods in that the computations involve information that changes at each iteration. The conjugate gradient (CG) method is the oldest and the most widely used method for symmetric positive definite matrices. CG method is a special kind in the broader class of steepest descent methods, where a functional is constructed which when extremized will deliver the solution of the problem. The functional will have convexity properties, so that the vector which extremizes the functional is the solution of the algebraic problem in question. This means that we search for the vector for which the gradient of the functional is zero, and this can be done in an iterative fashion.

The quadratic functional of Equation 1.1 is simply a special, quadratic function of a vector with the form

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{1.13}$$

where A is a matrix, x and b are vectors, c is a scalar constant. If A is symmetric and positive-definite, $f(x)$ is minimized by the solution Equation 1.1. The gradient of the quadratic functional in Equation 1.13 is

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b \tag{1.14}$$

If A is symmetric, this equation reduces to

$$f'(x) = Ax - b \tag{1.15}$$

Setting the gradient to zero, we obtain Equation 1.1, the linear system we wish to solve. Therefore, the solution to Equation 1.1 is a critical point of $f(x)$. If A is positive-definite as well as symmetric, then this solution is a minimum of $f(x)$, so Equation 1.1 can be solved by finding an x that minimizes $f(x)$.

In the method of steepest descent, starting form a arbitrary point $x^{(0)}$, each step attempts to move toward the solution. The step is chosen in the direction of negative gradient of the function f as this is the direction in which the functional decreases most rapidly.

$$x^{(k)} = x^{(k-1)} + \alpha^{(k-1)}(-f'(x^{(k-1)})) \tag{1.16}$$

It is to be noted from Equation 1.15 that this gradient is the negative of the residual. i.e. $f'(x^{(k)}) = -r^{(k)}$, where $r^{(k)}$ is the residual after k iterations. This results in the iteration

$$x^{(k)} = x^{(k-1)} + \alpha^{(k-1)}r^{(k-1)} \tag{1.17}$$

Here α represents the length of the step in this particular direction and $k = 1, 2, ...$ The choice of α is done so as to minimize f along a line, i.e. $\frac{d}{d\alpha}(f(x^{(k)}))$ is equal to zero. By applying the chain rule this leads to $f'(x^{(k)})^T r^{(k-1)}$. Setting this expression to zero, we find that α should be chosen so that $r^{(k-1)}$ and $f'(x^{(k)})$ are orthogonal. This results in

$$\alpha^{(k-1)} = \frac{(r^{(k-1)})^T r^{(k-1)}}{(r^{(k-1)})^T Ar^{(k-1)}} \tag{1.18}$$

Putting all of this together, the Steepest Descent is:

$$r^{(0)} = b - Ax^{(0)}$$

$$\begin{aligned} \alpha^{(k-1)} &= \frac{(r^{(k-1)})^T r^{(k-1)}}{(r^{(k-1)})^T A r^{(k-1)}} \\ x^{(k)} &= x^{(k-1)} + \alpha^{(k-1)} r^{(k-1)} \\ r^{(k)} &= r^{(k-1)} - \alpha^{(k-1)} A r^{(k-1)} \end{aligned} \tag{1.19}$$

Steepest Descent often finds itself taking steps in the same direction as earlier steps. The method of conjugate directions avoids this by picking a set of A-orthogonal search directions. By doing this, it can be ensured that a correction of the right length is done only once in each search direction. This way it will take exactly N steps to reach to the solution. Equation 1.17 can be re-written as

$$x^{(k)} = x^{(k-1)} + \alpha^{(k-1)} d^{(k-1)} \tag{1.20}$$

where $d^{(k-1)}$ is a search direction that is A-orthogonal to the rest. The value of α is determined from the condition that $e^{(k)}$ should be A-orthogonal to $d^{(k-1)}$

$$\alpha^{(k-1)} = \frac{(d^{(k-1)})^T r^{(k-1)}}{(d^{(k-1)})^T A d^{(k-1)}} \tag{1.21}$$

Note that if the search vector were the residual, Equation 1.21 would be identical to Equation 1.18 used by steepest descent. A-orthogonal search directions are generated with the conjugate Gram-Schmidt process [74].

The method of conjugate gradients (CG) is simply the method of conjugate directions where the search directions are constructed by conjugation of the residuals. The main advantage to do this is that the residual has the nice property that it is orthogonal to the previous search directions. This guarantees always to produce a new linearly independent search direction unless the residual is zero, in which case the problem is already solved. This additional condition results in the following algorithm:

$$\begin{aligned} d^{(0)} = r^{(0)} &= b - A x^{(0)} \\ \\ \alpha^{(k-1)} &= \frac{(r^{(k-1)})^T r^{(k-1)}}{(d^{(k-1)})^T A d^{(k-1)}} \\ x^{(k)} &= x^{(k-1)} + \alpha^{(k-1)} d^{(k-1)} \\ r^{(k)} &= r^{(k-1)} - \alpha^{(k-1)} A d^{(k-1)} \\ \beta^{(k)} &= \frac{(r^{(k)})^T r^{(k)}}{(r^{(k-1)})^T r^{(k-1)}} \\ d^{(k)} &= r^{(k)} + \beta^{(k)} d^{(k-1)} \end{aligned} \tag{1.22}$$

For the CG method, the error can be bounded in terms of the spectral condition number c of the matrix A defined as $c(A) = \lambda_{max}(A)/\lambda_{min}(A)$. If x^* is the

exact solution, then for CG with symmetric positive definite matrix A, it can be shown that

$$||x^{(k)} - x^*||_A \leq 2\left[\frac{\sqrt{c}-1}{\sqrt{c}+1}\right]^k ||x^{(0)} - x^*||_A \tag{1.23}$$

From this relation we see that the number of iterations to reach a relative reduction of ϵ in the error is proportional to $\sqrt{c}$. In some cases, practical application of the above error bound is straightforward. For example, elliptic second order partial differential equations typically give rise to coefficient matrices A with $c(A) = O(h^{-2})$ (where h is the discretization mesh width), independent of the order of the finite elements or differences used, and of the number of spatial dimensions of the problem. Thus, without preconditioning, we expect a number of iterations proportional to h^{-1} for the conjugate gradient method.

The success of the Krylov methods for symmetric matrices has inspired the construction of similar methods for unsymmetric matrices. One such popular method is the generalized minimal residual method (GMRES) introduced in [75]. The search direction vector $d^{(m)}$ in GMRES is A-orthogonal and the residual vector $r^{(m)}$ is constructed so as to minimize the L_2 norm $||r^{(m)}||_2$. The approximate solution is updated in the form $x^{(m)} = x^{(0)} + Q_m y^m$ where y^m minimizes the function $J(y) = ||\beta e_1 - \tilde{H}_m y||_2$, i.e.,

$$y^m = \min_y ||\beta e_1 - \tilde{H}_m y||_2 \tag{1.24}$$

Here Q_m is the $n \times m$ matrix with column vectors $p^l, ..., p^m$ and $\tilde{H}_m$ is the $(m + l) \times m$ Hessenberg matrix, $\beta = ||r^{(0)}||_2$, and $e_1 = (10...0)^T$. Minimization of the residual in $AK_m(A, r^{(0)})$ requires the solution of the above least squares problem. This algorithm utilizes a modified Gram-Schmidt orthogonalization method based on the Arnoldi algorithm to construct an orthogonal basis. For details of this orthogonalization process and the minimization process, refer [74].

Note that the convergence of GMRES is strictly monotonic in most cases. The major drawback to GMRES is that the amount of work and storage required per iteration rises linearly with the iteration count. The usual way to overcome this limitation is by restarting the iteration. After a chosen number of m iterations, the accumulated data are cleared and the intermediate results are used as the initial data for the next m iterations. This procedure is repeated until convergence is achieved.

Another widely used method for unsymmetric matrices is BiCGSTAB [90] which is a hybrid of the GMRES(m) and BiConjugate Gradient [31] methods. The BiConjugate Gradient (BiCG) method replaces one orthogonal sequence of residuals by two mutually orthogonal sequences and no longer ensures minimization as in GMRES. The update relations for residuals in the Conjugate Gradient method are augmented in the BiCG method by relations that are similar but based on A^T instead of A. Thus we update two sequences of residuals

$$r^{(k)} = r^{(k-1)} - \alpha^{(k-1)} A d^{(k-1)}, \qquad \bar{r}^{(k)} = \bar{r}^{(k-1)} - \alpha^{(k-1)} A^T \bar{d}^{(k-1)}$$

and two sequences of search directions

$$d^{(k)} = r^{(k)} + \beta^{(k)} d^{(k-1)}, \qquad \bar{d}^{(k)} = \bar{r}^{(k)} + \beta^{(k)} \bar{d}^{(k-1)}$$

The choices

$$\alpha^{(k-1)} = \frac{(\bar{r}^{(k-1)})^T r^{(k-1)}}{(\bar{d}^{(k-1)})^T A d^{(k-1)}}, \qquad \beta^{(k)} = \frac{(\bar{r}^{(k)})^T r^{(k)}}{(\bar{r}^{(k-1)})^T r^{(k-1)}}$$

ensure the bi-orthogonality relations

$$(\bar{r}^{(n)})^T r^{(m)} = (\bar{d}^{(n)})^T A d^{(m)} = 0, \qquad for\ m \neq n$$

The convergence behavior of BiCG is in practice observed to be quite irregular and the method may even break down. The main breakdown scenario arises when the bi-orthogonalization process leads to a zero inner product of the two new vectors in the Krylov subspace and its adjoint space (i.e. the Krylov subspace generated with A^T). Likewise, a near breakdown should also be avoided since it may lead to inaccuracies in the approximated solution. This has lead to a further research to handle the case of unsymmetric matrices.

In BiCG method, the residual can also be regarded as the product of r^0 and an kth degree polynomial in A as

$$r^{(k)} \quad = \quad P_k(A) r^{(0)}$$

Likewise, the residual in the adjoint space $\bar{r}^{(k)}$ can be expressed as a product of $\bar{r}^0$ and an kth degree polynomial in A^T. This results in the method of Conjugate Gradient Squared (CGS) also known as BiCGS which converges about twice as fast but is also twice as erratic as BiCG method.

BiCGSTAB combines the BiCG method with minimum residual steps of degree one. In this method the operations with A^T are transformed to other polynomials in A that describe a steepest descent update. This can be interpreted as a product of BiCG and repeatedly applied GMRES(1). At least locally, a residual vector is minimized, which leads to a considerably smoother convergence behavior.

1.3 Hardware architectures

Hardware details of a classical vector architecture (NEC SX-8) and a typical scalar architecture (Intel Xeon) are outlined below. Most performance studies included in the thesis use these architectures.

1.3.1 NEC SX-8

Fig. 1.1 illustrates the NEC installation at HLRS. The main building block is the 72 nodes of the NEC SX-8. The SX-8 nodes are connected to the Internode Crossbar Switch (IXS). A Global File System enables the entire multi-node complex to view a single coherent file system. Additionally, a NEC TX-7 Itanium server equipped with 32 Itanium II 1.5 GHz CPUs and a large main memory of 256 GB acts as a frontend.

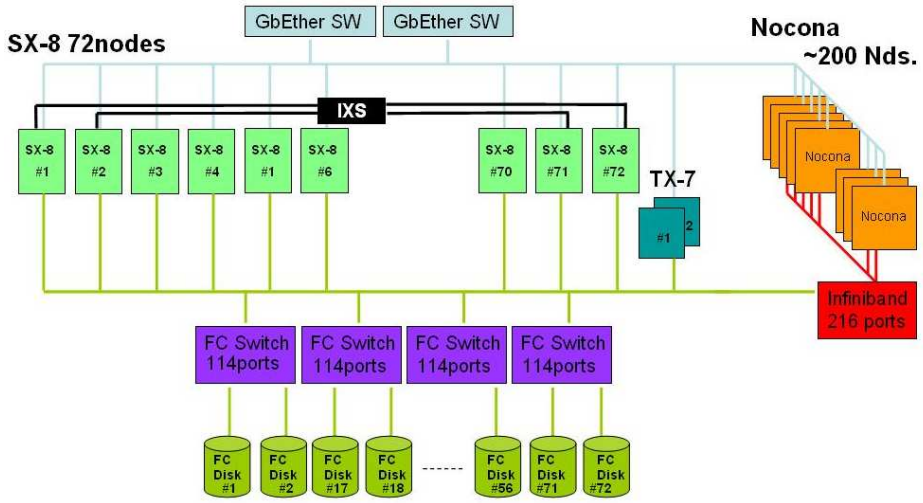

Figure 1.1: Installation at HLRS.

Description of the SX-8 node

The SX-8 architecture combines the traditional shared memory parallel vector design in single node systems with the scalability of distributed memory architecture in multi node systems. Each shared memory type single-node system contains 8 CPUs which share a large main memory of 128 GB.

Central Processor Unit The central processing unit (CPU) is a single chip implementation which consists of a vector and a scalar processor. Fig. 1.2 gives an overview of the functional units of the CPU.

Vector unit A vector unit is equipped with four floating point add/shift and four floating point multiply vector pipelines working in parallel on one single

instruction. Taking into account only these vector pipelines and neglecting the fully independent hardware divide/square-root pipe and the scalar units, this means that every clock cycle 8 results are produced. The major clock cycle of the SX-8 is 0.5 nsec, thus the vector floating point peak performance of each processor is 16 GFLOP/s. Additionally to the above mentioned vector add/shift and vector multiply pipelines the vector processor also contains four vector logical and four vector divide pipelines. One vector divide pipeline, which also supports vector square root, generates 2 results every second clock cycle, leading to additional 4 GFLOP/s.

The vector processor contains 16 KB of vector arithmetic registers which feed the vector pipes as well as 128 KB of vector data registers which serve as a high performance programmable vector buffer that significantly reduces memory traffic in most cases. They are used to store intermediate results and thus avoid memory bottlenecks. The maximum bandwidth between each SX-8 CPU and the shared memory is 64 GB/s. In addition, the CPU is equipped with registers for scalar arithmetic operations and base-address calculations so that scalar arithmetic operations can be performed efficiently.

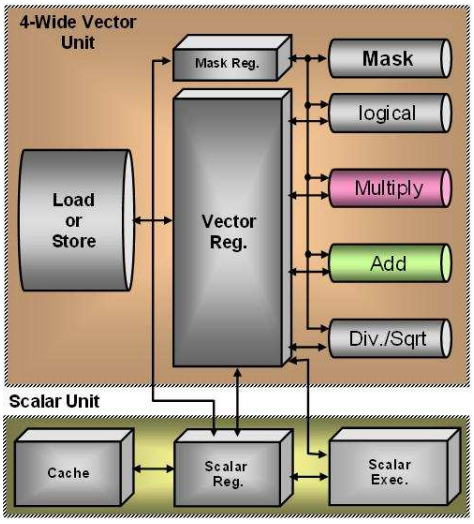

Figure 1.2: CPU architecture of SX-8.

Scalar unit Each CPU contains a 4-way super-scalar unit with 64-kilobyte operand and 64-kilobyte instruction caches. The scalar unit controls the operation of the vector processor as well as executes scalar instructions. It has 128 x 64 bit general-purpose registers and operates at a 1 GHz clock speed. Advanced features such as branch prediction, data prefetching and out-of-order instruction execution are employed to maximize the throughput. The scalar processor supports one load/store path and one load path between the scalar registers and scalar data cache. Each of the scalar floating point pipelines supports floating point add, floating point multiply and floating point divide operations. The

scalar unit executes 2 floating point operations per clock cycle, thus runs at 2 GFLOP/s.

Memory subsystem The processor to memory port is classified as a single port per processor. Either load or store can occur during any transfer cycle. Each SX processor automatically reorders main memory requests in two important ways. Memory references look-ahead and pre-issue are performed to maximize throughput and minimize memory waits. The issue unit reorders load and store operations to maximize memory path efficiency.

A new feature of the SX-8 Series is the memory bank cache which is depicted in Fig. 1.3. Each vector CPU has 32 KB of memory bank cache exclusively supporting 8 bytes for each of the 4096 memory banks. It is a direct-mapped write-through cache decreasing the bank busy time from multiple vector access to the same address. On specific applications this unique feature has proven to reduce performance bottlenecks caused by memory bank conflicts.

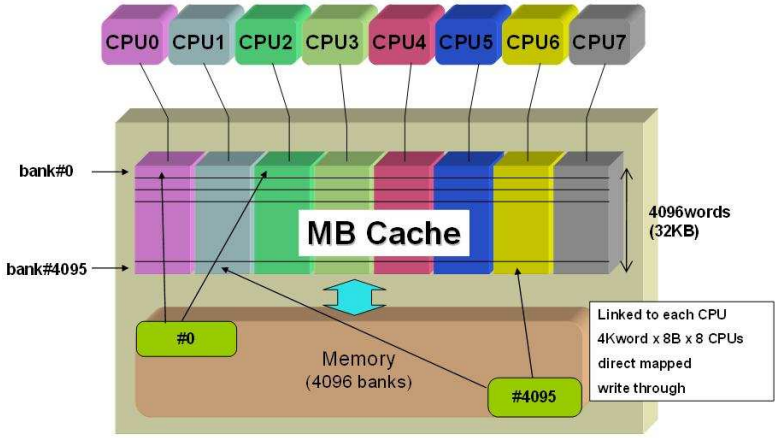

Figure 1.3: Concept of Bank Caching in SX-8 node.

Main memory unit To achieve efficient vector processing a large main memory and high memory throughput that match the processor performance are required. 128 GB DDR2-SDRAM are installed in every node. The bandwidth between each CPU and the main memory is 64 GB/s thus realizing an aggregated memory throughput of 512 GB/s within a single node.

The memory architecture within each single-node frame is a non-blocking crossbar that provides uniform high-speed access to the main memory. This constitutes a symmetric multiprocessor shared memory system (SMP) also known as a parallel vector processor (PVP).

SX-8 Series systems are real memory mode machines but utilize page mapped addressing. Demand paging is not supported. The page mapped architecture

allows load modules to be non-contiguously loaded, eliminating the need for periodic memory compaction procedures by the operating system and enabling the most efficient operational management techniques possible. Another advantage of the page mapped architecture is that in case of swapping only that number of pages needs to be swapped out which are needed to swap another job in thus reducing I/O wait time considerably.

SX-8/M series multi node models

Multi node systems of the SX-8 are constructed using the NEC proprietary high speed single-stage crossbar (IXS) linking multiple single node chassis together. The IXS provides very tight coupling between nodes, thus, virtually enabling a single system image both from a hardware and a software point of view.

The IXS is a full crossbar providing a high speed single stage non-blocking interconnect. The provided IXS facilities include inter-node addressing and page mapping, remote unit control, inter-node data movement, and remote processor instruction support (e.g. interrupt of a remote CPU). It also contains system global communication registers to enable efficient software synchronization of events occurring across multiple nodes. There are 8 x 64 bit global communication registers available for each node.

Both synchronous and asynchronous transfers are supported. Synchronous transfers are limited to 2 KB, and asynchronous transfers to 32 MB. This is transparent to the user as it is entirely controlled by the NEC MPI library.

The interface technology is based on 3 Gb/s optical interfaces providing approximately 2.7μs (microsecond) node-to-node hardware latency (with 20 m cable length) and 16 GB/s of node-to-node bi-directional bandwidth per RCU (Remote Control Unit). Each SX-8 node is equipped with two RCUs. Utilizing the two RCUs allow for connecting the 72 nodes to a single IXS with a bidirectional bandwidth of 32 GB/s per node.

Compilers

For the SX-8 series FORTRAN, C and C++ compilers are available. The compilers offer advanced automatic optimization features including automatic vectorization and automatic parallelization, partial and conditional vectorization, index migration, loop collapsing, nested loop vectorization, conversions, common expression elimination, code motion, exponentiation optimization, optimization of masked operations, loop unrolling, loop fusion, in-line subroutine expansion, conversion of division to multiplication and instruction scheduling.

SX-6+ series

The NEC SX-6+ is in all aspects similar to NEC SX-8 as it is a previous generation system. Before the installation of NEC SX-8 at HLRS, four nodes of NEC

SX-6+ were installed as a test system. Each CPU is clocked at 565 MHz and the peak vector performance is 9 GFlop/s (8 add and 8 multiply pipelines). The scalar unit has a peak performance of 1.13 GFlop/s. Each shared memory node has 8 CPUs with a main memory bandwidth of 36 GByte/s per CPU and the memory is divided into 4096 independent modules, also called as banks. One main difference with SX-8 is that this system does not have bank cache.

1.3.2 Intel Xeon cluster

The cluster has 200 PC cluster nodes each with two 3.2GHz Intel Xeon EM64T (Nocona) processors which are connected with a 1 GB/s Bandwith Voltaire Infiniband High-Speed Switch. each processor has a 1MB of integrated L2 cache and the Front Side Bus (FSB) of the memory subsystem operates at 800MHz.

1.3.3 Commodity multicore processors

All the major commodity microprocessor manufacturers are moving in the direction of adding more cores onto the processor. Both AMD and Intel have two to four cores available per chip at the moment and are moving in the direction of adding tens or even hundreds of cores in the near future. Two systems that are used for performance measurements in this thesis are detailed in this section. The first system tested was a dual-socket quad-core 2.13 GHz Intel Xeon with 1MB L2 cache per core. The other system being a quad-socket dual-core 2.61 GHz AMD Opteron with 512KB L2 cache per core. AMD Opteron dual-core sockets are connected using the proprietary HyperTransport network.

1.3.4 STI CELL BE

The CELL Broadband Engine Architecture (CBEA) was jointly developed by Sony Computer Entertainment, Toshiba, and IBM (STI). The CELL processor is a heterogenous multicore architecture combining general purpose PowerPC architecture with Synergistic Processing Elements (SPE) which greatly accelerate multimedia and vector processing applications. The architecture emphasizes efficiency/watt, prioritizes bandwidth over latency, and favors peak computational throughput over simplicity of program code. For these reasons, Cell is widely regarded as a challenging environment for software development.

The CELL BE installed at HLRS consists of eight CELL processors. Each processor can be split into four components – external input and output structures, the main processor called the Power Processing Element (PPE) (a two-way simultaneous multithreaded Power core), eight fully functional SPEs, and a specialized high bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus (EIB). The peak accumulated single precision FP performance on CBE is about 204 GF/s (four

single precision FP units × 2 opers. × 3.2 GHz × 8 SPEs). The peak double precision FP on 8 SPE cores adds to 14.6 GF/s. The memory bandwidth between the SPEs via the EIB is about 307 GB/s (four 16 byte rings × 3 transfers × 1.6 GHz). The Memory Interface Controller (MIC) read & write peak to XDRAM (main memory) is 25.6 GB/s.

1.4 Motivation and outline

The main motivation to develop a new iterative solver originated from the failed attempts to improve the performance of general public domain iterative solvers like AZTEC, PETSc and Trilinos on the NEC SX-8 vector system. For a flow problem tested, the performance of AZTEC on SX-6+ was 450 MFlop/s and on SX-8 was 680 MFlop/s. Over 75% of total time was spent in AZTEC on both the machines for this example. The sustained performance of a public domain solver on vector systems is typically under 5%.

There have also been attempts to port PETSc on CRAY X1 architecture, which combines the advantages of vector pipelining with memory caching. Sparse MVP with diagonal sparse storage format on this machine performs at about 14% peak [9]. An upper bound of around 30% peak is estimated for most sparse matrix computations on the CRAY X1. The author knows of no MPI based general public domain iterative sparse solver library that runs efficiently on vector systems such as NEC SX-8 or Cray X1.

The remainder of the thesis is organized as follows. In Chapter 2 an analysis of modern hardware architectures is presented based on the results of HPCC and Intel MPI Benchmark suites. The implementation issues related to BLIS are detailed in Chapter 3. The central idea of blocking sparse matrices is presented here along with vector specific optimizations needed in sparse linear solvers. Matrix preconditioning algorithms and their performance is also explained in this chapter. The performance results of applications using BLIS on the NEC SX-8 is presented in Chapter 4. Finally, some experiences on the suitability of new hardware architectures, such as the multi-core and STI CELL processor, for solving sparse linear systems are stated in Chapter 5. The last chapter concludes the thesis by highlighting the results achieved with the new solver on vector systems and gives some ideas on future directions to this research.

Chapter 2

Analysis of modern hardware architectures

This chapter is dedicated to the analysis of the present hardware architectures. Solving sparse linear systems requires a very high sustained memory bandwidth and low memory latencies. There is also a considerable demand on the interconnect performance as each iteration of a Krylov subspace method needs at least one global reduction operation. Appropriate benchmarks are chosen in order to evaluate the memory subsystem and the interconnect of various contemporary High Performance Computing platforms. Though High Performance Linpack (HPL) [24] has long been used as a yardstick to rank the most powerful computers in the world, it measures only a subset of the hardware characteristics. There are various other Benchmark suites that try to measure the capabilities of the high performance computing architectures more extensively than HPL. These benchmark suites can be broadly classified as a collection of application benchmarks like the SPEC MPI2007 [1] or a collection of kernel benchmarks like the HPC Challenge [54] or a mixture of both like the NAS Parallel Benchmarks [12].

In order to compare both major architectural designs, i.e. scalar and vector hardware, two benchmark suites are chosen. One is the HPCC suite developed at the Innovative Computing Laboratory, University of Tennessee. In addition to HPCC suite, Intel MPI Benchmark (IMB) [2] results are also presented in order to compare the interconnect performance of different hardware architectures for various MPI calls. The performance of the interconnect is becoming increasingly important as the design of High Performance systems is in the direction of connecting more and more off-the-shelf processors. For this approach of agglomeration of weak processing units to be scalable, the interconnect performance plays a crucial role.

2.1 HPC challenge suite

All the HPC Challenge [54, 3] Benchmarks are used which are multi-faceted and provide comprehensive insight into the performance of modern high-end computing systems. They are intended to test various attributes that can contribute significantly to understanding the performance of high-end computing systems. These benchmarks stress not only the processors, but also the memory subsystem and system interconnects. They provide a better understanding of an application's performance on the computing systems and are better indicators of how high-end computing systems will perform across a wide spectrum of real-world applications. There are three different modes possible to run the kernels. In single and embarrassingly parallel modes, there is no explicit communication between the processes (DGEMM [26, 25], STREAM [55], RandomAccess and FFTE [82]). In global mode, the kernels are run on all the selected processes using MPI to communicate (HPL, PTRANS [4], RandomAccess, FFTE and b_eff [51, 69]).

Two different sets of results are provided below. Firstly, base-run uploads to the HPCC web page as on 02/05/06 [5] are analyzed. Secondly, run results are presented for increasing processor count on a few selected hardware architectures that were accessible.

2.1.1 Comparing the uploaded results

The largest 15 submissions have been selected of which several machines appear more than once. In this case, only the largest submission of any particular architecture is considered though there can be small differences between installations. This finally leads to the comparison of six different high performance computing architectures based on HPCC suite. The systems in the figures that follow are sorted according to the HPL value, i.e., in order of the machine size.

Ratio based analysis of all benchmarks

Figure 2.1 compares the systems based on several kernels in the HPCC suite. This analysis is similar to the current Kiviat diagram analysis on the HPCC web page, with the exception that parallel (Global) or embarrassingly parallel (EP) benchmark results and plotted instead of single process run results when available. A detailed ratio based analysis of the HPCC suite can be found in [70]. In order to compare systems with different total system performance, all the global benchmark results (except RandomRing latency) are normalized with the Global HPL (G-HPL) value and all the EP benchmark results are normalized with Per Process HPL (PP-HPL, i.e., G-HPL/#processes) value. As the reported RandomRing bandwidth is an average per process value, global value is calculated by multiplying it with the number of processes. The normalized values are further

divided by the largest value among all the systems to have a relative comparison between different systems.

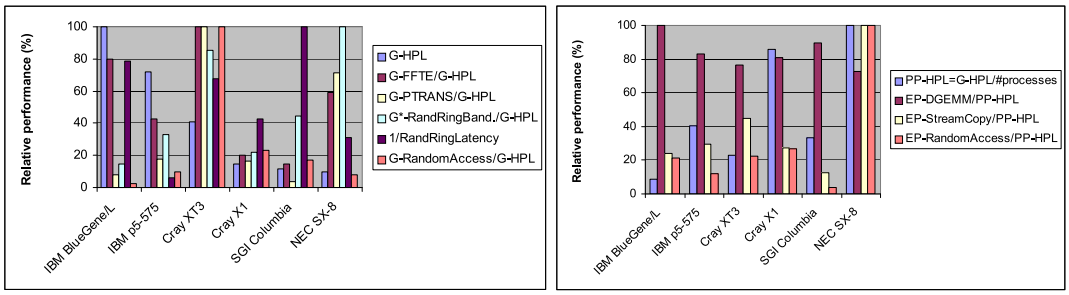

Figure 2.1: Ratio based analysis for global (left) and embarrassingly parallel (right) benchmarks in HPCC suite.

The left plot of Figure 2.1 presents the relative normalized performance for global benchmarks that need MPI communication, i.e., depend on the network performance. Whereas the right plot presents similar data for embarrassingly parallel benchmarks that do not use the network interconnect. The left most bars in both the plots compare the absolute system performance and the power of a single CPU correspondingly. It can be noted from this figure that the relative performance of NEC SX-8 is above 50% for seven benchmarks plotted (G-FFTE, G-PTRANS, G-RandomRing bandwidth, PP-HPL, EP-DGEMM, EP-STREAM and EP-RandomAccess benchmarks). Followed by Cray XT3 whose relative performance is above 50% for six benchmarks plotted (G-FFTE, G-PTRANS, G-RandomRing bandwidth, G-RandomRing latency, G-RandomAccess and EP-DGEMM benchmarks). NEC SX-8 has a relatively poor network latency when compared to other systems. The RandomAccess benchmark intends to measure the performance of random memory access (non-strided access) which is typical in many engineering applications. This benchmark is run in all the three modes (Single, EP and Global). The performance in Global and EP modes are shown in Figure 2.1. The huge difference in the relative performance between different modes (Global and EP) comes from the fact that the implementation of the Global RandomAccess is mainly network latency bound. The performance of EP-RandomAccess on the NEC SX-8 is much better than on the others (right plot in Figure 2.1) and is the main reason for the superior performance of sparse codes, which involve a huge amount of indirect memory addressing [65].

Balance of network/memory bandwidth to computation

A good balance of different aspects of a high performance computing platform is crucial to sustain high performance for a wide spectrum of applications. Most important among them are the network bandwidth to computational speed and the memory bandwidth to computational speed. Figure 2.2 plots this balance for

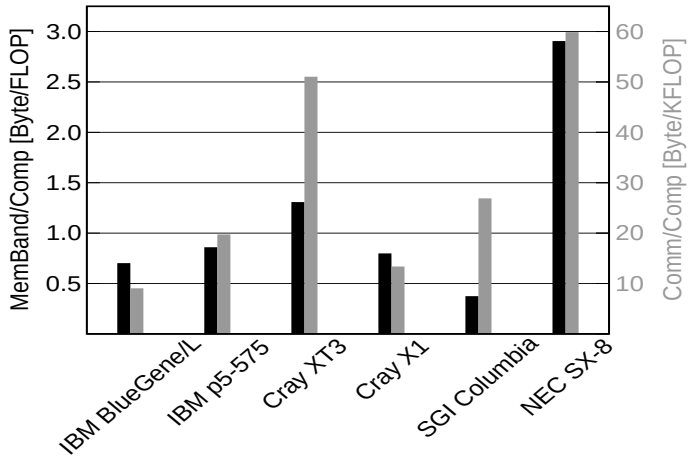

Figure 2.2: STREAM Copy bandwidth Vs. HPL (black), RandomRing bandwidth Vs. HPL (gray).

the six leading computing platforms considered. The vector system installed at HLRS, NEC SX-8, is far superior in both the ratios when compared to the rest. It is in fact this aspect along with the superior performance on Random Access benchmarks that puts the vector systems well suited for sparse linear algebra kernels which are mostly memory performance bound. With minor modifications to such kernels it is possible to achieve a floating point performance of upto 50% peak on the NEC SX-8.

2.1.2 Scaling on different architectures

Full HPC Challenge Benchmarks were run with different processor counts on SGI Altix BX2, Cray X1, Cray Opteron Cluster, Dell Cluster and NEC SX-8. These five systems use different networks (SGI NUMALINK4, Cray network, Myrinet, InfiniBand, and NEC IXS) and their system characteristics are listed in Table 2.1. The results on increasing processor counts are described below.

Balance of communication to computation

For multi-purpose HPC systems, the balance of processor speed, along with memory, communication, and I/O bandwidth is important. In this section, the ratio of inter-node communication bandwidth to the computational speed is analyzed. To characterize the communication bandwidth between SMP nodes, the random ring bandwidth is used, because for a large number of SMP nodes, most MPI processes will communicate with MPI processes on other SMP nodes. This means, with 8 or more SMP nodes, the random ring bandwidth reports the available inter-node communication bandwidth per MPI process. Although the balance is calculated based on MPI processes, its value should be in principle independent

Platform/ (Type)	**Location**	**CPUs/ node**	**Clock (GHz)**	**Peak/ node (GFlop/s)**	**Network/ (Topology)**
SGI Altix BX2 (Scalar)	NASA Ames (USA)	2	1.6	12.8	NUMALINK 4 (Fat-tree)
Cray Opteron Cluster (Scalar)	NASA Ames (USA)	2	2.0	8.0	Myrinet (Fat-tree)
Cray X1 (Vector)	NASA Ames (USA)	4	0.8	12.8	Proprietary (4D-Hypercube)
Dell Xeon Cluster (Scalar)	NCSA (USA)	2	3.6	14.4	InfiniBand (Fat-tree)
NEC SX-8 (Vector)	HLRS (Germany)	8	2.0	128.0	IXS (Multi-stage Crossbar)

Table 2.1: System characteristics of the computing platforms.

of the programming model, i.e., whether each SMP node is used with several single-threaded MPI processes, or some (or one) multi-threaded MPI processes, as long as the number of MPI processes on each SMP node is large enough that they altogether are able to saturate the inter-node network [68, 71]. Fig. 2.3 shows the scaling of the accumulated random ring performance with the computational speed. To compare measurements with different number of CPUs and on different architectures, all data is presented based on the computational performance expressed by the Linpack HPL value. For this, the HPCC random ring bandwidth was multiplied with the number of MPI processes. The computational speed is benchmarked with HPL.

The diagram in Fig. 2.3 shows absolute communication bandwidth, whereas the diagram in Fig. 2.4 plots the ratio of communication to computation speed. Better scaling with the size of the system is expressed by less decreasing of the ratio plotted in Fig.2.4. A strong decrease can be observed in the case of Cray Opteron, especially between 32 CPUs and 64 CPUs runs. NEC SX-8 system scales well which can be noted from only a slight inclination of the curve. In case of SGI Altix, it is worth noting the difference in the ratio between NUMALINK3 and NUMALINK4 interconnects within the same box (512 CPUs). Though the theoretical peak bandwidth between NUMALINK3 and NUMALINK4 has only doubled, Random Ring performance improves by a factor of four for runs up to 256 CPUs. A steep decrease in the B/Kflop value for SGI Altix with NUMALINK4 is observed above 512 CPUs runs (203.12 B/Kflop for 506 CPUs run to

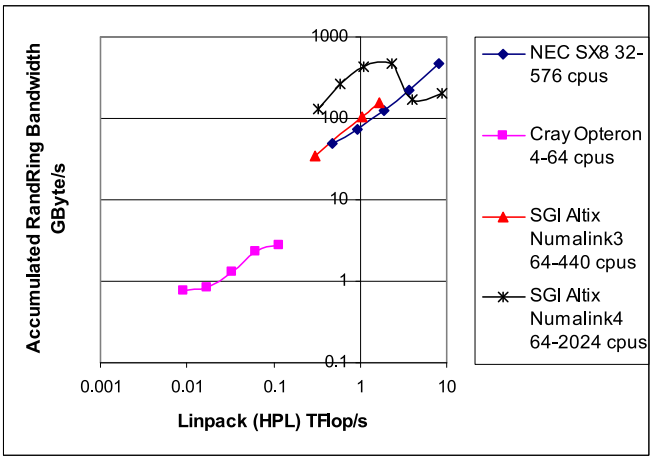

Figure 2.3: Accumulated Random Ring Bandwidth versus HPL performance

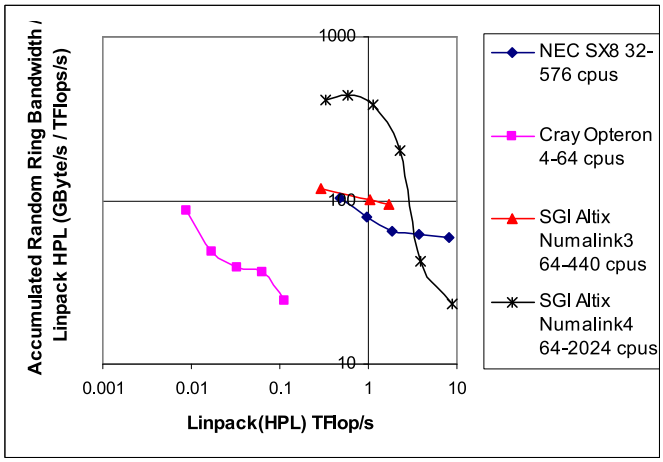

Figure 2.4: Accumulated Random Ring Bandwidth ratio versus HPL performance

23.18 B/Kflop for 2024 CPUs run). This can also be noticed from the cross over of the ratio curves between Altix and the NEC SX-8. Where as with NUMALINK3 it is 93.81 (440 CPUs run) when run within the same box. For the NEC SX-8, B/Kflop is 59.64 (576 CPUs run), which is consistent between 128 and 576 CPUs runs. For the Cray Opteron it is 24.41 (64 CPUs run).

Fig. 2.5 and Fig. 2.6 compare the memory bandwidth with the computational speed analog to Fig. 2.3 and Fig. 2.4. Fig. 2.5 shows absolute values whereas Fig. 2.6 plots the ratio of STREAM Copy to HPL on the vertical axis. The accumulated memory bandwidth is calculated as the product of the number of MPI processes with the embarrassingly parallel STREAM Copy result. In Fig. 2.6, as the number of processors increase, the slight improvement in the ratio curves is due to the fact that the HPL efficiency decreases. In the case of CRAY Opteron HPL efficiency drops down around 20% between 4CPU and 64CPU runs. The high memory bandwidth available on the NEC SX-8 can clearly be seen with

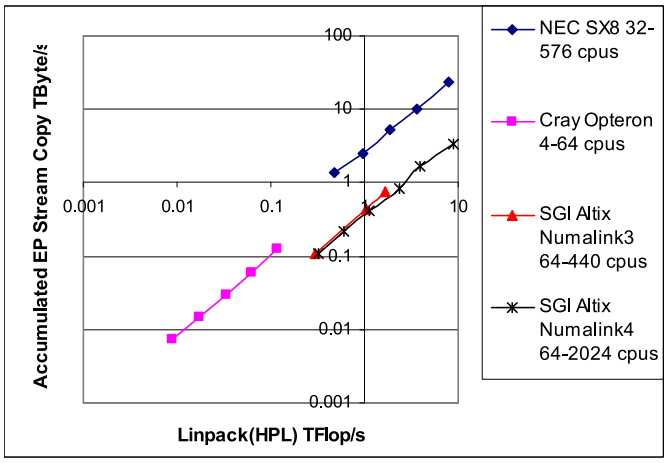

Figure 2.5: Accumulated EP Stream Copy versus HPL performance

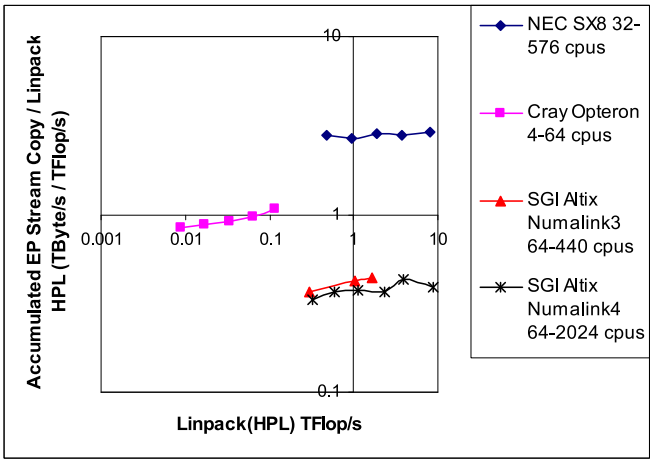

Figure 2.6: Accumulated EP Stream Copy ratio versus HPL performance

the stream benchmark. The Byte/Flop for NEC SX-8 is consistently above 2.67 Byte/Flop, for SGI Altix (NUMALINK3 and NUMALINK4) it is above 0.36 and for the Cray Opteron is between 0.84 and 1.07. The performance of Memory intensive applications heavily depends on this value.

Ratio based analysis of all benchmark

It should be noted that RandomAccess benchmark between HPCC versions 0.8 and 1.0 has been significantly modified. Only values based on HPCC version1.0 are shown in Fig. 2.7 for fair comparison.

Figure 2.7 compares the systems based on several HPCC benchmarks. This analysis is similar to the current Kiviat diagram analysis on the HPCC web page [5], but it uses always parallel or embarrassingly parallel benchmark results instead of single process results, and it uses only accumulated global system values

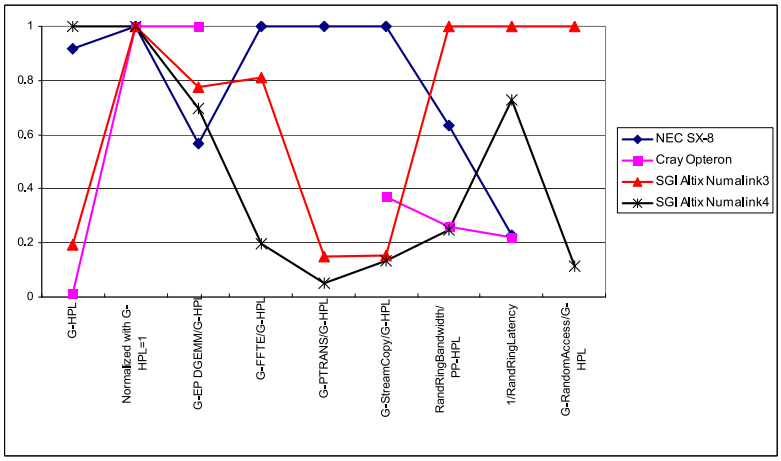

Figure 2.7: Comparison of all the benchmarks normalized with HPL value

Ratio	Maximum value
G-HPL	8.729 TF/s
G-EP DGEMM/G-HPL	1.925
G-FFTE/G-HPL	0.020
G-Ptrans/G-HPL	0.039 B/F
G-StreamCopy/G-HPL	2.893 B/F
RandRingBW/PP-HPL	0.094 B/F
1/RandRingLatency	0.197 1/micro-sec
G-RandomAccess/G-HPL	4.9e-5 Update/F

Table 2.2: Ratio values corresponding to 1 in Figure 2.7.

instead of per process values. If one wants to compare the balance of systems with quite different total system performance, this comparison can be done hardly on the basis of absolute performance numbers. Therefore all benchmark results are normalized with the HPL system performance, i.e., divided by the HPL value. Only the left column can be used to compare the absolute performance of the systems. This normalization is also indicated by normalized HPL value in the second column that is per definition always 1. For latency, the reciprocal value is shown. Each column itself is additionally divided by the largest value in the column, i.e., the best value is always 1. The corresponding absolute ratio values for 1 in Fig. 2.7 are provided in Table 2.2.

One can see from Fig. 2.7 that the Cray Opteron performs best in EP DGEMM because of its lower HPL efficiency when compared to the other systems. When looking at the global measurement based ratio values such as FFTE, Ptrans and RandomAccess, it is to be noted that small systems have an undue advantage over the larger ones because of better scaling. For this reason, the global ratios of systems with over 1 TFlop/s HPL performance are plotted. The

NEC SX-8 wins in the benchmarks where high memory bandwidth coupled with network performance is needed (Ptrans, FFTE and EP Stream Copy). On the other hand the NEC SX-8 has relatively high Random Ring latency compared to the other systems. SGI Altix with NUMALINK3 wins in Random Ring bandwidth and latency benchmarks (NUMALINK4 performs much better than NUMALINK3 within the same box). This shows the strength of its network within a box. Despite this fact it loses to Cray Opteron in RandomAccess which is heavily dependent on the network performance.

2.2 Intel MPI benchmark suite

Intel MPI Benchmark (IMB) suite is a successor of Pallas MPI Benchmark (PMB) from Pallas GmbH [2]. This benchmark suite is very popular among high performance computing community to measure the performance of important MPI functions. The IMB 2.0 version has three parts (a) IMB for MPI-1, (b) MPI-2 one sided communication, and (c) MPI-2 I/O. In standard mode, different sizes of messages upto 4,194,304 bytes is possible. There are three classes of benchmarks:

2.2.1 Single transfer benchmarks

Single Transfer Benchmarks (STB) focus on a single message transferred between two processes. There are two benchmarks in this category namely Ping-Pong and Ping-Ping.

- Ping-Pong: In a Ping-Pong, a single message is sent between two processes. Process 1 sends a message of size "x" to process 2 and process 2 sends "x" back to process 1.
- Ping-Ping: Ping-Ping is same as Ping-Pong except that messages are obstructed by oncoming messages.

2.2.2 Parallel transfer benchmarks

Parallel Transfer Benchmarks (PTB) focus on patterns and activity at a certain process in concurrency with other processes.

- Sendrecv: The processes form a periodic communication. Here each process sends a message to the right and received from the left in the chain.
- Exchange: Here process exchanges data with both left and right in the chain. This communication pattern is used in applications such as unstructured adaptive mesh refinement computational fluid dynamics codes involving boundary exchanges.

2.2.3 Collective benchmarks

The Collective Benchmarks (CB) are collective in the sense that all the processes within a defined communicator group take part collectively. They test not only the message passing power of the computing system but also the algorithms used underneath, e.g., reduction benchmarks measure the message passing tests as well as efficiency of the algorithms used in implementing them. Collective communications are mostly built around point-to-point communications. Several features distinguish collective communications from point-to-point communications.

- A collective operation requires that all processes within the communicator group call the same collective communication function with matching arguments.
- The size of data sent must exactly match the size of data received. In point-to-point communications, a sender buffer may be smaller than the receiver buffer. In collective communications they must be the same.
- Except for explicit synchronization routines such as MPI_Barrier, MPI collective communication functions are not synchronizing functions.
- Collective communications exist in blocking mode only, i.e., a process will block until its role in the collective communication is complete, no matter what the completion status is of the others participating in the communication.
- Collective operations do not use the tag field. They are matched according to the order they are executed.

Collective communications can also be divided into three further categories according to function: synchronization, data movement, and global reduction operations. Collective MPI calls important in the context of a linear iterative solver as all the Krylov methods need global reductions to evaluate the inner products of vectors.

2.2.4 IMB benchmark results

When running the IMB Benchmarks on the NEC SX-8 system, memory allocation was done with MPI_Alloc_mem, which allocates global memory. The MPI library on the NEC SX-8 is optimized for global memory.

Fig. 2.8 shows the performance of the Barrier benchmark from the IMB suite of benchmarks. The plots show the time (in microseconds per call) for various number of processors ranging from 2 to 512 (and 568 in case of NEC SX-8).

A barrier function is used to synchronize all processes. A process calling this function blocks until all the processes in the communicator group have called

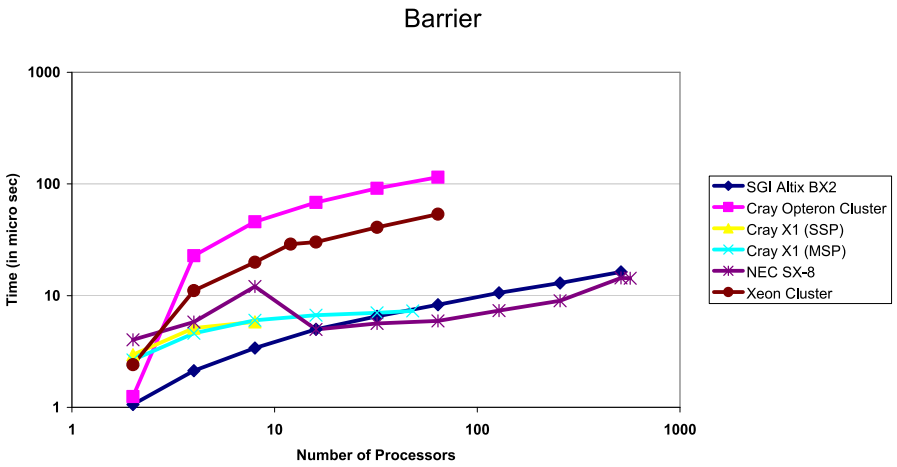

Figure 2.8: Execution time of Barrier benchmark (the smaller the better)

this function. This ensures that each process waits till all the other processes reach this point before proceeding further. Here, all the five computing platforms exhibit the same behavior up to 64 processors i.e. barrier time increases gradually with the increase of number of processors, except for the Cray X1 in MSP mode where barrier time increases very slowly. On NEC SX-8, the barrier time is measured for each processor count with their own full (MPI_COMM_WORLD) communicator and not with subset-communicators. With this modification, NEC SX-8 has the best barrier time compared to other systems for large CPU counts. Although, for less than 16 processor runs, SGI Altix BX2 is the fastest.

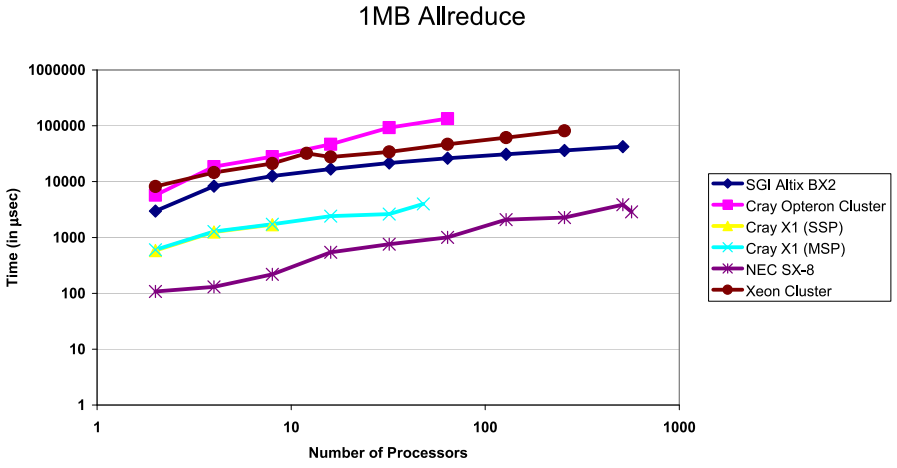

Figure 2.9: Execution time of Allreduce benchmark for a message size of 1MB (the smaller the better)

The execution time of the Allreduce benchmark for 1 MB message size is shown in Fig. 2.9. All the five systems scale similarly when compared to their performance at 2 processors. There is more than one order of magnitude dif-

ference between the fastest and slowest platforms. Both the vector systems are clearly the winner, NEC SX-8 better than Cray X1 both in MSP and SSP mode. Up to 16 processors, both Cray Opteron cluster and Dell Xeon cluster follow the same trend as well with almost identical performance. Here, best performance is on NEC SX-8 and worst performance is on Cray Opteron cluster (uses Myrinet network).

Execution time of IMB Reduction benchmark for 1 MB message size on all five computing platforms is shown in Fig. 2.10. A clear performance clustering of architectures can be noticed here – vector systems (NEC SX-8 and Cray X1) and cache based scalar systems (SGI Altix BX2, Dell Xeon Cluster, and Cray Opteron Cluster). Performance of vector systems is an order of magnitude better than scalar systems. Between vector systems, performance of NEC SX-8 is better than that of Cray X1. Among scalar systems, performance of SGI Altix BX2 and Dell Xeon Cluster is almost the same and is better than Cray Opteron Cluster.

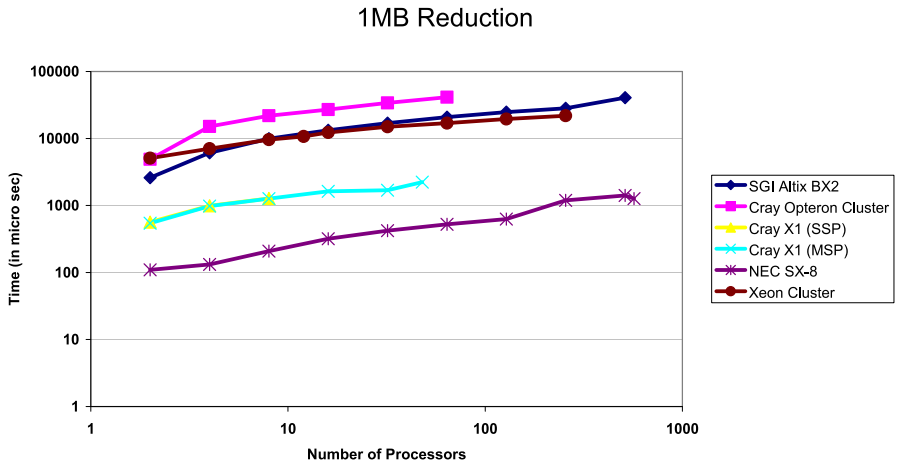

Figure 2.10: Execution time of Reduction benchmark for a message size of 1MB (the smaller the better)

Fig. 2.11 shows the execution time of IMB Allgather benchmark for 1 MB message size on five computing platforms. Performance of vector system NEC SX-8 is much better than that of scalar systems (Altix BX2, Xeon Cluster and Cray Opteron Cluster). Cray X1 (both SSP and MSP modes) is slightly better than the scalar systems. Between two vector systems, performance of NEC SX-8 is an order of magnitude better than Cray X1. Among the three scalar systems, performance of Altix BX2 and Dell Xeon Cluster is almost the same and is better than Cray Opteron Cluster.

Results presented in Fig. 2.12 are the same as in Fig. 2.11 but for Allgatherv which is a version of Allgather with variable message sizes. The performance results are similar to the results of the (symmetric) Allgather. On the NEC SX-8, the performance increase between 8 and 16 processors is based on the changeover from a single shared memory node to a multi SMP node execution.

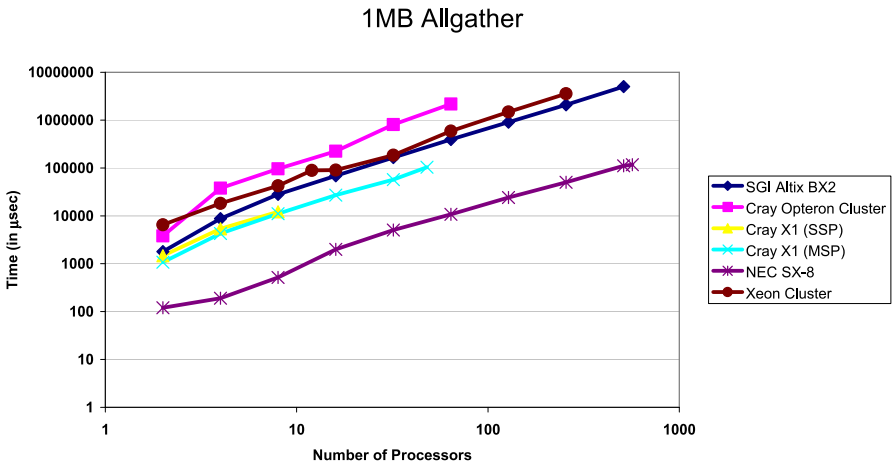

Figure 2.11: Execution time of Allgather benchmark for a message size of 1MB (the smaller the better)

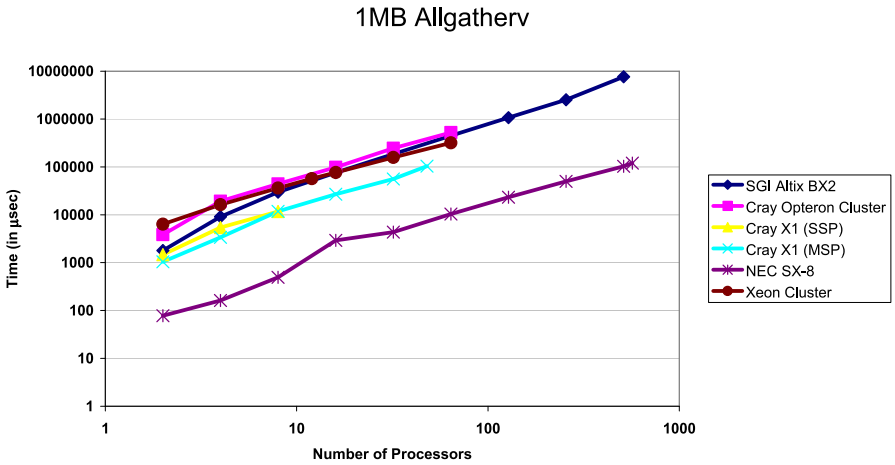

Figure 2.12: Execution time of Allgatherv benchmark for a message size of 1MB (the smaller the better)

Performance of scalar systems is almost same. Between two vector systems, the performance of NEC SX-8 is almost an order of magnitude better than Cray X1.

Fig. 2.13 shows the execution time of AlltoAll benchmark for a message size of 1 MB on five computing architectures. This benchmark stresses the global network bandwidth of the computing system. Performance of this benchmark is very close to the performance of global FFT and randomly ordered ring bandwidth benchmarks in the HPCC suite. Performance of Cray X1 (both SSP and MSP modes) and SGI Altix BX2 is very close. However, the performance of SGI Altix BX2 up to eight processors is better than Cray X1 as the SGI Altix BX2 (uses NUMALINK4 network) has eight Intel Itanium 2 processors in a C-Brick. Performance of Dell Xeon Cluster (uses IB network) and Cray Opteron Cluster (uses Myrinet PCI-X network) is almost same up to 8 processors and

then performance of Dell Xeon cluster is better than Cray Opteron Cluster. The performance of NEC SX-8 (IXS) is clearly superior to all other systems tested and it is also interesting to note that performance is directly proportional to the randomly ordered ring bandwidth, which is related with the cost of the global network.

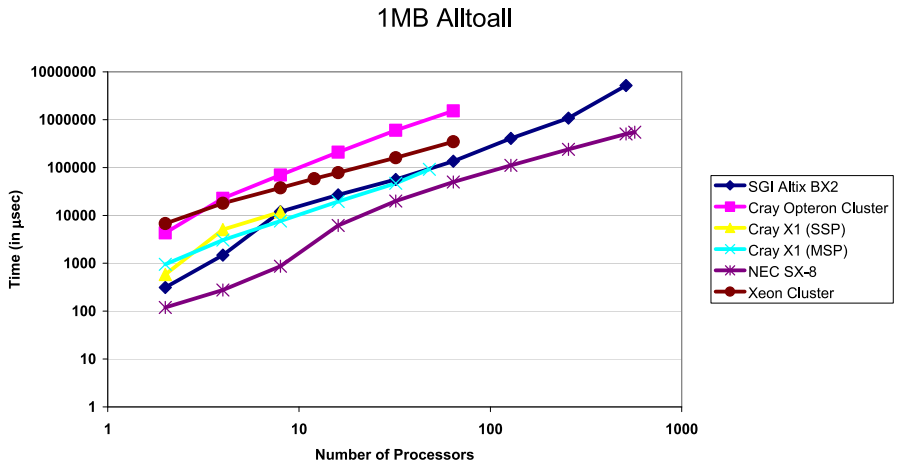

Figure 2.13: Execution time of AlltoAll benchmark for a message size of 1MB (the smaller the better)

Fig. 2.14 presents the bandwidth of IMB Sendrecv benchmark using 1 MB message. Clearly, performance of NEC SX-8 is the best followed by SGI Altix BX2. Performance of Xeon cluster and Cray Opteron is almost the same. After 16 processors, the performance of all the computing system becomes almost constant. For all platforms, the system performance is best for 2 processors. It is expected for BX2, Opteron and Xeon because all of them are dual processor nodes and also for NEC SX-8 with its 8-way SMP nodes. Therefore this Sendrecv is done over the memory and not over the network. Here, it would be interesting to note that on the NEC SX-8 with 64 GB/s peak memory bandwidth per processor, the IMB Sendreceive bandwidth for 2 processors is 47.4 GB/s. Whereas for the Cray X1 (SSP), IMB Sendreceive bandwidth is only 7.6 GB/s.

Fig. 2.15 shows the performance of the IMB Exchange benchmark for 1 MB message size. The NEC SX-8 is the winner but its lead margin over the Xeon cluster is decreased compared to the Sendrecv benchmark. The second best system is the Xeon Cluster and its performance is almost constant from 2 to 512 processors, i.e., compared to Sendrecv, the shared memory gain on 2 CPUs is lost. For a number of processors greater than or equal to 4, the performance of the Cray X1 (both SSP and MSP modes) and the Altix BX2 is almost same. For two processors, the performance of the Cray Opteron cluster is close to the BX2, and the performance of Cray Opteron cluster is the lowest.

In Fig. 2.16, the time (in micro seconds) is plotted against varying numbers of processors for 1 MB broadcast on five computing platforms. Up to 64 proces-

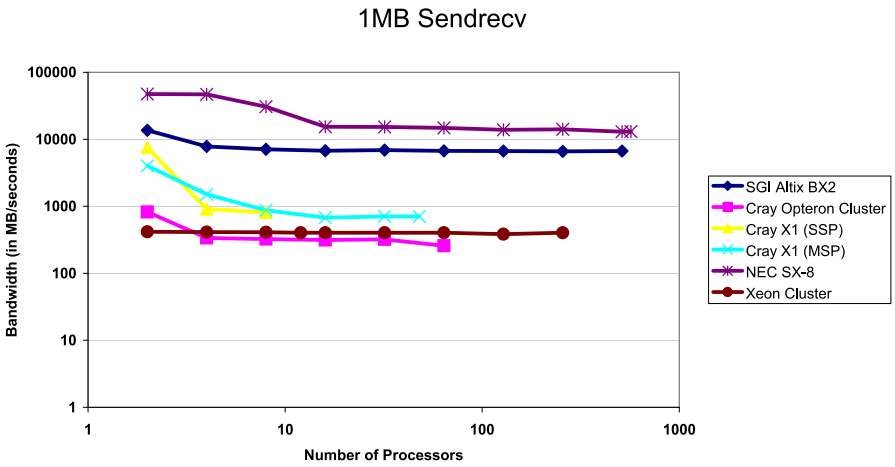

Figure 2.14: Bandwidth of Sendrecv benchmark on varying number of processors, using a message size of 1MB

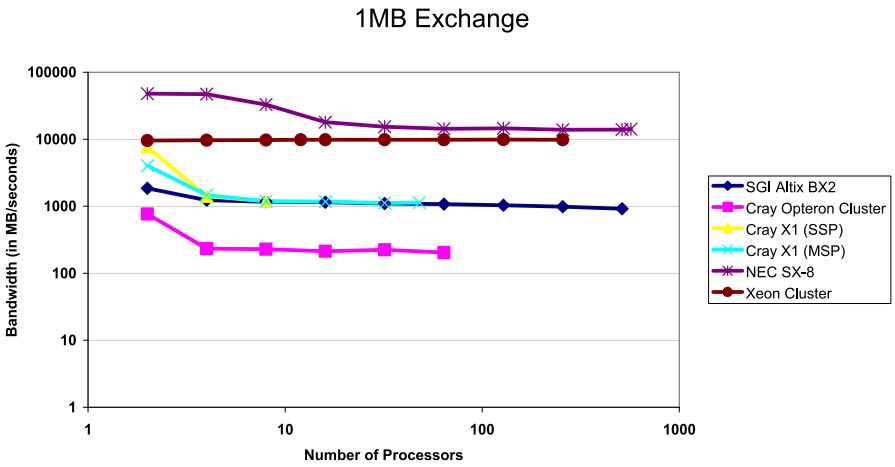

Figure 2.15: Bandwidth of Exchange benchmark on varying number of processors, using a message size of 1MB

sors, the broadcast time increases gradually and this trend is exhibited up to 64 processors by all the computing platforms. Only 512 processor results are available for SGI Altix BX2 and NEC SX-8. For the BX2, broadcast time suddenly increases for 256 processors and then again decreases at 512 processors, which cannot be well interpreted. A similar but quite smaller behavior is seen for NEC SX-8 – increase on broadcast time at 512 and then it decreases at 576 processors. The best systems with respect to broadcast time in decreasing order are NEC SX-8, SGI Altix BX2, Cray X1, Xeon cluster and Cray Opteron cluster. The broadcast bandwidth of NEC SX-8 is more than an order of magnitude higher than that of all other presented systems.

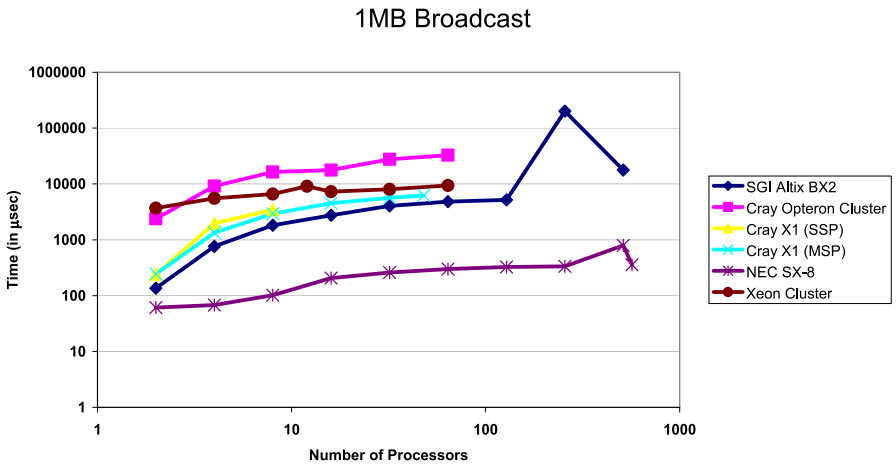

Figure 2.16: Execution time of Broadcast benchmark for a message size of 1MB (the smaller the better)

2.3 Conclusions and consequences for sparse iterative methods

2.3.1 HPCC benchmark suite

The HPCC benchmark suite highlights the importance of memory bandwidth and network performance along with HPL performance. The growing difference between the peak and sustained performance underlines the importance of such benchmark suites. A good balance of all the above quantities should make a system perform well on a variety of application codes. Here, the benchmark analysis is used to see the strengths and weaknesses of the architectures considered. The ratio based analysis presented in Sections 2.1.1 and 2.1.2 provides a good base to compare different systems and their interconnects.

It is clear from the analysis that NEC SX-8 performs extremely well on benchmarks that stress the memory and network capabilities, like Global PTRANS and Global FFTs. It is worth mentioning that Global FFT benchmark in HPCC suite is not completely vectorized, hence on vector machines (like Cray X1 and NEC SX-8) the performance of FFTs using vendor provided optimized libraries would be much higher. The interconnect latency of SGI Altix BX2 is the best among all the platforms tested. However, a strong decrease in the sustained interconnect bandwidth is noticed when using multiple SGI Altix BX2 boxes. On SGI Altix BX2, global FFT does not perform well beyond one box (512 CPUs) and this degradation in performance is also reflected by decrease in the random order bandwidth benchmark of HPCC suite. G-FFT involve all-to-all communication and therefore for G-FFT to perform well, the system should also score high on the IMB All-to-All benchmark. The scalability and performance of small machines (Cray Opteron and Cray X1) cannot be compared to that of larger machines as

the complexity and cost of the interconnect grows more than linearly with the size of the machine.

The benchmarking studies done using a wide range of application codes on various high performance computing platforms [65, 66, 67] also conclude that the vector systems (Cray X1 and NEC SX-6) are promising for many codes. These results also support our analysis of various architectures using the HPCC benchmark suite. In light of these findings there has also been an interesting study [34] suggesting the expansion of the present SIMD multimedia instruction set (ISA) extensions on 80x86 and PowerPC scalar architectures to a full vector ISA. Another major change suggested by the authors is the increase of the SIMD register size from the current 128 bits to 512 bits or even more.

With respect to sparse solution methods, vector systems cleary show an advantage over scalar systems because of their superior performance with memory intensive benchmarks (STREAM, PTRANS and FFTE). The HPL normalized performance in Fig. 2.7 with G-PTRANS, G-FFTE and EP-STREAM Copy benchmarks is far better on NEC SX-8 when compared to other scalar systems tested. The EP-RandomAccess benchmark which measures the memory latency is also superior on NEC SX-8 when compared to other systems (Fig. 2.1). Both memory bandwidth and latency play a crucial role to attain high sustained performance in sparse matrix methods (please refer Sect. 3.2). Since the vector systems are better balanced than most scalar systems (Fig. 2.2) with respect to memory and network performance, they are more suitable for sparse iterative solvers.

2.3.2 IMB benchmark suite

Performance of both the vector systems (NEC SX-8 and Cray X1) is consistently better than all the scalar systems (SGI Altix BX2, Cray Opteron Cluster and Dell Xeon Cluster). Between the two vector systems, performance of NEC SX-8 is consistently better than Cray X1. Among scalar systems, the performance of SGI Altix BX2 is better than both Dell Xeon Cluster and Cray Opteron Cluster. The interconnects with decreasing order of Performance are IXS (NEC SX-8), Cray X1network, SGI Altix BX2 (NUMALINK4), Dell Xeon Cluster (InfiniBand) and lastly Cray Opteron Cluster (Myrinet).

Krylov subspace methods which are implemented in most sparse iterative solvers need a global collective call (MPI_Allreduce) for evaluating inner products in each iteration. Almost all iterative methods need point-to-point neighbour communication. For both collective and point-to-point calls, the network on both the tested vector systems (NEC SX-8 and Cray X1) consistently out perform the network on the scalar systems. The performance with MPI_Allreduce on the NEC SX-8 is over an order of magnitude better than on the tested scalar systems (Fig. 2.9). Also, the performance with MPI_Sendrecv is far superior on NEC SX-8 than most scalar systems tested (Fig. 2.14). This clearly shows that the

interconnect on vector systems is more suited for iterative algorithms than on scalar systems.

Chapter 3

Implementation of a vectorized sparse solver

There are mainly two popular approaches to solving linear systems - direct and iterative. This work is mainly focused on the implementation issues relating to the iterative methods used for solving sparse linear systems. Such linear systems arise in numerous engineering applications that need to solve PDEs over a given domain. The domain is first spatially discretized using any one or a combination of the various discretization methods like the Finite Element, Finite Difference, etc. After the space of the domain is discretized into small control regions, the local equilibrium equations of each control block are assembled to get a global system of equations. This systems is typically sparse as the local equations in each control region typically involve interactions only with the local neighboring regions in the discretized space. Some applications where large sparse linear systems have to be solved are electronic circuit design, electrical power systems, heat flow problems, fluid dynamics, structural mechanics, image processing, data mining applications, geoscience, weather, etc.

Direct methods rely on the principle of Gaussian Elimination to find the solution. Iterative methods on the other hand slowly proceed towards the solution using successive approximations. The success of the iterative method hence strongly depends on the properties of the coefficient matrix. In most cases a preconditioner is necessary, that improves the spectral properties of the coefficient matrix, in order to get to the solution. Iterative methods can be broadly classified into stationary and non-stationary methods. Stationary methods like the Jacobi and the Gauss-Seidel methods solve each variable of the system locally with respect to all other connected variables. So, each iteration in these methods correspond to solving for one variable in the system. Non-stationary methods like the Krylov subspace methods generate vectors that form a conjugate basis. These vectors which are the residuals of the iterates are also the gradients of a quadratic functional formed using the coefficient matrix. The minimization of this functional is the solution to the linear system.

For solving Symmetric Positive Definite (SPD) linear systems, CG is the best known Krylov subspace method with a guaranteed convergence and a monotonically decreasing error. Given A is a SPD matrix, to solve the system of equations $A\mathbf{x}$=$\mathbf{b}$ an equivalent quadratic functional is defined as

$$\phi(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$$

Now, the minimization of this functional gives the solution of the original linear system. The minimization is done with in a Krylov subspace of dimension n defined as

$$K_n = span\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, ..., A^{n-1}\mathbf{r}_0\}$$

where $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$, is the initial residual vector given by and $\mathbf{x}_0$ the start vector. The vectors which define the span form an orthogonal basis and are the directions pointed by the residual vectors of the iterates. Because of the nestedness of the spaces, the error decreases monotonically and the exact solution is reached in utmost n iterations (without FP rounding errors). For solving unsymmetric linear systems various Krylov subspace methods like the GMRES, BiCG, QMR and BiCGSTAB have been proposed which are based on similar principles as the CG method.

The main kernels in any Krylov subspace method is the parallel Matrix Vector Product (MVP) and inner products. Once a parallel sparse MVP is programmed efficiently [87], all the Krylov subspace methods can be easily implemented. For unstructured meshes, the neighbours have to be determined after the mesh has been partitioned. Domain decomposition of unstructured meshes is usually done using partitioning tools such as Metis [50] and Jostle[99]. The choice of storage format to store a sparse matrix plays an important role in the overall efficiency of the sparse linear solver (parallel sparse MVP is the most time consuming part of any sparse iterative solver). The issues related to sparse storage formats on vector systems are detailed in the following section.

3.1 Sparse storage formats

There are different possibilities to store the non-zero elements of a sparse matrix. Following is only a brief overview on sparse storage formats and a detailed description can be found in [73, 74] The storage formats can be broadly classified into Compressed Row Storage (CRS), Compressed Column Storage (CCS) and pseudo-diagonal storage (Jagged Diagonal - JAD). The row format stores the compressed rows of non-zeros consecutively in memory. The pseudo diagonal format stores the columns consecutively in memory of the matrix obtained by permuting the compressed rows according to the descending order of their length, i.e. the number of non-zeros in each row. This is better illustrated in

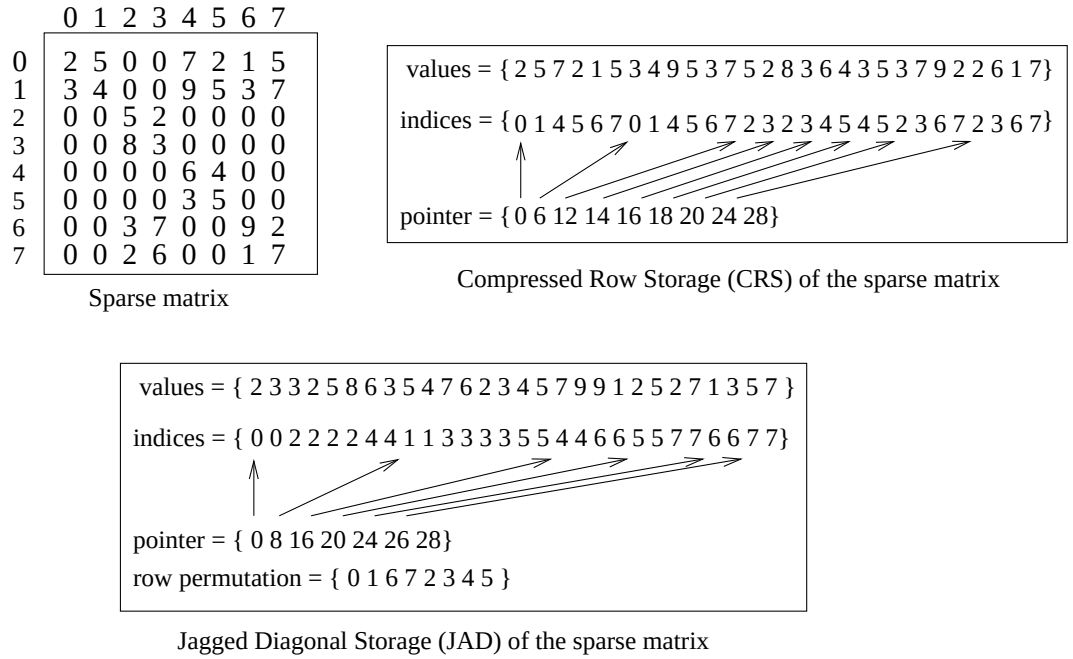

Figure 3.1: Sparse storage formats.

Fig 3.1. The CRS format consists of three vectors which hold the values of the matrix, their column indices and a row pointer holding the starting position of each row in the values/index vectors. The JAD format in addition to these three vectors consists of one more vector that holds the row permutation information. The column format (CCS), which is not shown in the figure, is a transpose of the row format, i.e. it stores the compressed non-zeros in each column consecutively in memory. It is to be noted that the values of the sparse matrix, which are integers in the figure, are usually real numbers for engineering simulations. The performance of the matrix vector product kernel is directly connected to the sparse storage format as the algorithm depends on the selected storage scheme. One particular algorithm may perform well on one architecture and not on the other depending on the features of the underlying hardware.

Average vector length which represents the extent of vectorization of a kernel on a typical vector system is an important metric that has a huge effect on the performance. In sparse linear algebra, the matrix object is sparsely filled where as the vectors are dense. So, any operations involving only vectors, like the dot product, result in a good average vector length as the innermost vectorized loop runs over long vectors. The problem is with operations involving the sparse matrix object, like the matrix vector product (MVP). Naturally resulting row based data structures adopted in most public domain solvers result in low average vector length. This leads to partially empty pipeline processing and hence hinders

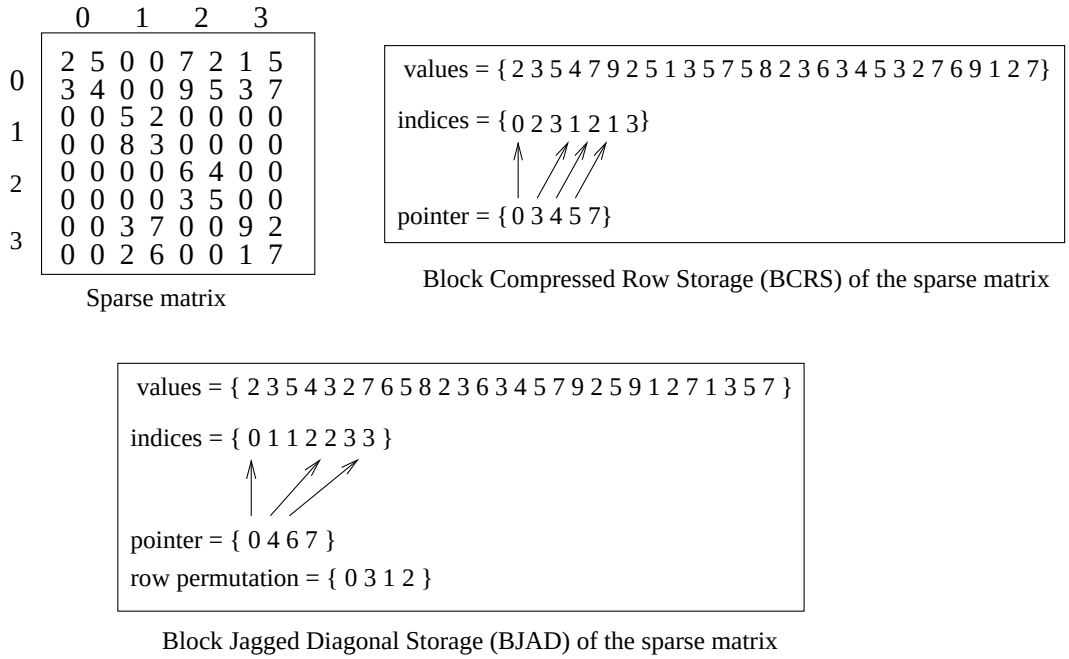

Figure 3.2: Block sparse storage formats.

performance in a key solver kernel on vector systems.

A well known solution to this problem is to use pseudo diagonal data structure to store the sparse matrix [74]. Pseudo diagonal formats like the jagged diagonal format (JAD) are commonly used on vector machines as they fill-up the vector pipelines and result in long average vector length. The length of the first few pseudo diagonals is equal to the size of the matrix and it slowly decreases depending on the nature of the problem. This helps in filling up the vector pipelines and results in a superior performance over other formats. But one of the disadvantages associated with such formats is that they are not as natural as the row/column formats which makes their initial setup difficult.

3.1.1 Block storage formats

Block sparse storage formats are presented in this section without going into the details of performance of kernels that use such formats. The performance of blocked kernels is detailed in Sect. 3.3. All the formats described in the previous section can be extended to store small dense blocks instead of single values. Figure 3.2 shows how the same matrix shown in Fig. 3.1 can be stored in a block CRS and block JAD formats. Though, the matrix in the figure has regular blocks of size two, such block storage formats are general and can also store irregular blocks. It is to be noted that the developed solver is intended to be used which engineering applications with multiple degrees of freedom per mesh point. For

such problems, regular blocks are common.

The number of indices in a block format reduces by a factor of square of the block size and the number of row/diagonal pointers reduces by a factor of block size when compared to a point based storage. Each dense block can be stored in row or column order. In the figure, the columns of each dense block are stored successively. The main advantage with the block formats is not the reduction of memory needed to store the indices (as mentioned in literature) but the reduction in indirect memory access in kernels that use the sparse matrix.

3.1.2 Recent work on vector specific storage formats

There has been some recent studies over optimizing pseudo diagonal storage formats and improving the MVP algorithm on vector systems. A brief overview on these studies is presented below.

Optimizing the storage

Transposed jagged diagonal format helps in optimizing the amount of storage needed for a sparse matrix by eliminating the need for storing the permutation vector [56]. It is to be noted that this format is based on compressed column storage. This forces a shift in indirect addressing from the multiplied vector to the result vector. As the result vector has to be both loaded and stored, this effectively doubles the amount of indirect addressing needed for the MVP algorithm. This is a matter of concern on conventional vector architecture, but not on cache based systems. Since this format is principally used on vector systems, this does not provide a good alternative. The performance penalty for the MVP algorithm would simple overweigh its advantage of saving memory.

Improving average diagonal length

Bi-JDS(Bi-Jagged diagonal storage) combines both the conventional JAD and Transposed JAD and improves the average length of the diagonals [43]. The main idea is to store all the full length diagonals in the JAD format and the rest of the matrix data is stored in the TJAD (Transposed JAD) format. TJAD has the disadvantage of increasing the indirect addressing as explained in the previous section. Setting up such a format in the context of the whole iterative solver could mean changing some other algorithms which use the matrix data, such as preconditioning. It is also to be noted that a lot of problems generate matrices with long average vector lengths in JAD format. A common example of this is matrices from structured finite element, finite difference and finite volume applications. For problems involving surface integrals, it is common that the number of non-zeros in only some of the rows are extremely high. In this case, it may be advantageous to use such a scheme. This has to be extensively tested to

measure its pros (long diagonals) against its cons (doubled indirect addressing). The results may depend on the kind of vector architecture (with or without cache memory).

CRSP data format

Compressed row storage with permutation (CRSP) was introduced in [20]. Results show that the performance of this algorithm on Cray X1 were an order of magnitude faster than the normal CRS format. The permutation introduced in the CRSP format adds an additional level of indirection to both the sparse matrix and the vector (than the normal CRS format), i.e. effectively two additional indirectly addressed memory loads per iteration. This is the overhead incurred because of permuting the rows in ascending order of their length. Although this results in tremendous improvement in performance on the Cray X1, due to caching, such algorithms would perform poorly on conventional vector systems due to the heavy cost of indirect addressing involved (absence of cache memory). The overhead of indirect addressing is studied in Section 3.3.

BBCS and HiSM storage formats

These formats were proposed to optimize the performance of sparse matrix vector product. The Block Based Compression Storage (BBCS) exploits the temporal locality of the multiplied vector by dividing the sparse matrix into column blocks and processing one block at a time [95]. For the efficient usage of the proposed format, two new vector instructions are proposed. This format cannot be used on the present vector processors such as the NEC SX-8 without these instructions. The second proposed format is the Hierarchical Sparse Matrix (HiSM) which also needs new vector instructions for its usage [79]. This format as the name suggests is a hierarchical representation of the sparse matrix data and needs considerably less memory to store the positional information of the non-zero entries. The efficiency of these formats is tested on a simulated vector processor and the results show that the HiSM based sparse MVP is over five times faster than CRS and about four times faster than the JAD based sparse MVP for the tested problems [80].

3.2 Improvements to the JAD sparse MVP algorithm

Here, some small changes to the sparse MVP algorithm are proposed which improve the performance on vector systems. The first change in the algorithm that could be effective is processing on more diagonals in the same loop instead of one at a time. Then the use of vector registers will be presented which could help in

```
//initialize res vector to zero
for (i=0; i<number_of_diagonals; i++) {
  offset = jad_ptr[i];
  diag_length = jad_ptr[i+1] - offset;
  for (j=0; j<diag_length; j++) {
    res[j] += mat[offset+j] * vec[index[offset+j]];
  }
}
```

Figure 3.3: Original JAD MVP algorithm.

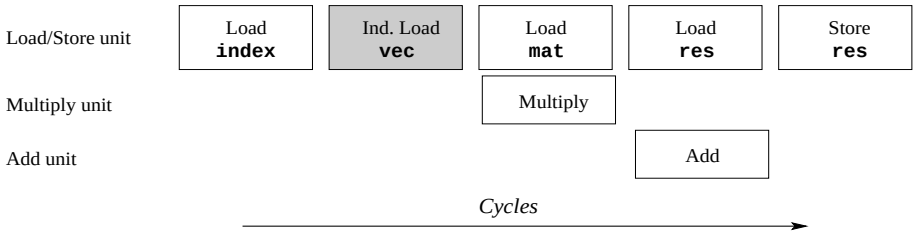

Figure 3.4: Operation flow diagram for original JAD MVP algorithm.

saving load/store operations for the result vector. Lastly, performance improvements in AZTEC, after implementing the proposed algorithmic modifications, on the NEC SX-8 are shown.

3.2.1 Original JAD MVP algorithm

The original JAD MVP algorithm is listed in Fig. 3.3. For simplicity, the permutation of the result vector is not shown. The performance limitations of this algorithm can be better understood in terms of an operation flow diagram shown in Fig. 3.4, which explains the execution of the vector instructions. It should be noted that there is only one load/store unit in the NEC SX-8 processor. In each clock cycle, two floating point (FP) operations per pipeline are possible : 1 add and 1 multiply. The main bottleneck of this algorithm is the indirect load of the multiplied vector (`vec`) which takes roughly five times longer than a directly addressed one on NEC SX-8. On super-scalar architectures, this factor is in general even greater and more complicated to predict.

In nine cycles (five for indirectly addressed vector), the possible number of FP operations is 18 (add and multiply). But the effective FP operations in the innermost loop of the algorithm are only two (one add and one multiply). Hence, the expected performance of this operation would be 2/18th of the vector peak. The actual measured performance as shown in Figs. 3.7 and 3.14 is about 1.8 GFlop/s on the NEC SX-8.

```
//initialize res vector to zero
for (i=0; i<number_of_diagonals; i++) {
  offset = jad_ptr[i];
  dia_length = jad_ptr[i+1] - offset;
  if (((i+1)<number_of_diagonals) &&
      (dia_length==(jad_ptr[i+2]-jad_ptr[i+1]))) {
    offset1 = jad_ptr[i+1];
    for (j=0; j<diag_length; j++) {
      res[j] +=   mat[offset+j] * vec[index[offset+j]]
                + mat[offset1+j] * vec[index[offset1+j]];
    }
  i++;
  }
  else
    for (j=0; j<diag_length; j++) {
      res[j] += mat[offset+j] * vec[index[offset+j]];
    }
}
```

Figure 3.5: JAD MVP algorithm grouping at most 2 diagonals.

3.2.2 Working on groups of diagonals

Matrices originating from structured meshes have groups of pseudo diagonals with equal length (stored in JAD format). One way to improve the performance is to operate on groups of equal length diagonals in the innermost loop instead of a single diagonal. This considerably saves load/store operations for the result vector (`res`) and additionally results in a better scheduling of indirect memory load operations. The accordingly modified algorithm is listed in Fig. 3.5. For simplicity, the algorithm works on utmost 2 diagonals of equal length. Extending this procedure to more diagonals improves the performance notably. This modification has been implemented in the AZTEC solver and the performance improvement can be seen in Fig. 3.7.

3.2.3 Use of vector registers

Most vector systems provide a programmer interface to vector registers in order to temporarily store data, like the result vector (`res`). Using vector registers would need the user to strip mine the innermost loop. The resulting algorithm is listed in Fig. 3.6. This procedure does not depend on the type of mesh (structured/unstructured) and hence equally reduces the memory access for the result vector.

```
//Size of the hardware vector register
  strip = 256
  for j0=0, number_of_rows, strip
//Initialising the vector register
    for j=0, strip-1
      vregister[j]=0.0
    end for
//Performing the multiplication
    for i=0, number_of_diagonals
      offset = jad_ptr[i]
      diag_length = jad_ptr[i+1] - offset
      for j=j0, min(diag_length, j0+strip-1)
        vregister(j-j0) += mat[offset+j] * vec[index[offset+j]]
      end for
    end for
//Write results to memory
    for j=j0, min(number_of_rows,j0+strip-1)
      res[j] = vregister[j-j0]
    end for
  end for
```

Figure 3.6: JAD MVP algorithm using vector registers.

3.2.4 Performance of AZTEC on NEC SX-8

All the above described algorithmic modifications have been implemented in AZTEC [86], which is a general purpose public domain sparse iterative solver developed at the Sandia National Labs. This package is used by many FEM codes to solve linear system of equations. So, some work was done to improve the performance of AZTEC on the NEC SX-8. All the algorithmic modifications described above have been implemented in AZTEC and their performance was assessed on the NEC SX-8. Fig. 3.7 plots the single CPU performance in Sparse MVP on the NEC SX-8 with the native row based (Modified Sparse Row – MSR) and the introduced pseudo diagonal based (JAgged Diagonal – JAD) matrix storage format with various modifications. The case tested was a structured finite element problem with 8500 hexahedral elements, 10065 nodes and 40260 unknowns. The original algorithm using the native row format performs at 890 MFlop/s per CPU. The original JAD based sparse MVP reaches a performance of 1780 MFlop/s per CPU which is about 2X faster than the native format. Using vector registers improves the performance by about 25% in JAD sparse MVP. By processing pseudo diagonals of equal length in groups, performance improves remarkably to 3180 MFlop/s for groups of size nine. The main reason for this is better scheduling of indirect memory load instructions in the innermost loop

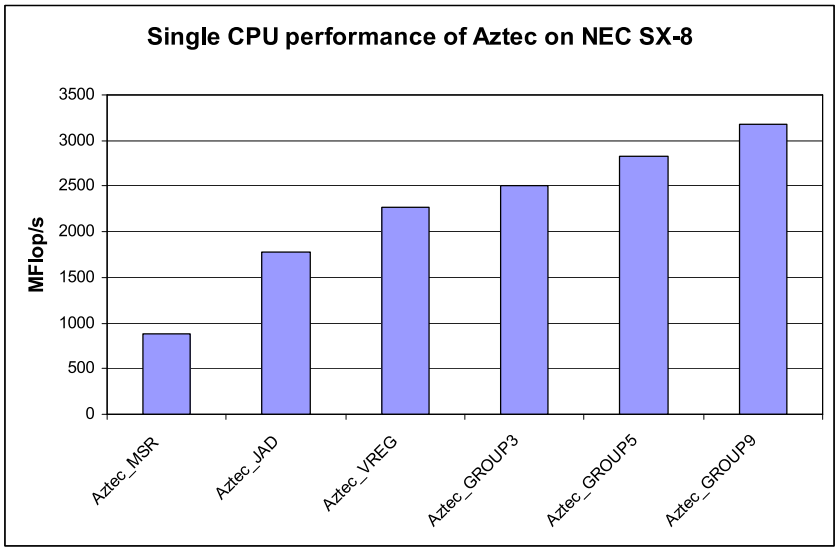

Figure 3.7: Single CPU performance of modified AZTEC kernels on NEC SX-8.

along with the reduced load/store instructions for the result vector. Performance studies of different sparse MVP algorithms on the NEC SX-8 are also detailed in [85]. JAD processing of sparse data is also very efficient for assembling element contributions into a global matrix on vector systems [16].

Memory bandwidth and latency limitations

The performance of sparse MVP on vector as well as on superscalar architectures is not only limited by memory bandwidth as can mostly be found in literature [9, 23], but rather by memory latencies. Due to the sparse storage, the vector to be multiplied in a sparse MVP is accessed randomly (non-strided access). Hence, the performance of this operation completely depends on the implementation and performance of the "vector gather" assembly instruction on vector systems. Though the memory bandwidth and byte/flop ratios on a vector architecture are in general far more superior to any superscalar architecture, superscalar architectures have the advantage of cache re-usage for this operation. Cache re-usage for point based sparse kernels is not as good as for the blocked sparse kernels. This is mainly because of the lack of spatial locality in point based sparse kernels, which may also result in many TLB (Translation Lookaside Buffer) misses as the matrix size increases. The main problem on cache-based systems is the high cost of accessing the main memory. For sparse kernels, prefetching data can help improve the performance. Without special optimizations [47], sparse MVP performs at around 5% CPU peak on superscalar systems.

A normal point based JAD implementation of the sparse MVP as shown in Fig. 3.3 needs over 10 Bytes/Flop for the computation. The peak Byte/Flop ratio

```
//initialize res vector to zero
blksize = 2
for (i=0; i<number_of_diagonals; i++) {
  offset = jad_ptr[i];
  diag_length = jad_ptr[i+1] - offset;
  for (j=0; j<diag_length; j++) {
    temp_mat = (offset+j)*blksize*blksize;
    temp_vec = (index[offset+j])*blksize;
    vec1 = vec[temp_vec+0];
    vec2 = vec[temp_vec+1];
    res[j*blksize+0] +=   mat[temp_mat+0] * vec1
                        + mat[temp_mat+1] * vec2;
    res[j*blksize+1] +=   mat[temp_mat+2] * vec1
                        + mat[temp_mat+3] * vec2;
  }
}
```

Figure 3.8: Block based JAD MVP algorithm (for 2×2 blocks).

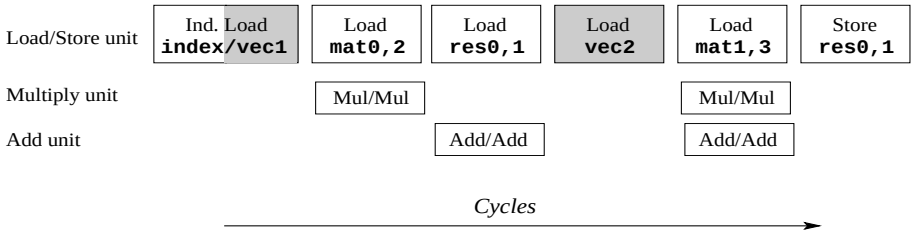

Figure 3.9: Operation flow diagram for block based JAD MVP algorithm.

on the NEC SX-8 is 4. So, the expected performance is $(4/10)^{th}$ peak, i.e. 40% peak. But the measured performance was only 2.2 GFlop/s, about 14% vector peak. The reason is the indirect memory access (non-strided) needed to load the vector to be multiplied. The memory latency for an indirect load or store is prohibitively high on the tested vector system and hence has a drastic effect on performance.

3.3 Block-based approach

Most engineering applications have more than one unknown to be solved per mesh point. For example, in Finite Element Structural analysis, a 3D solid Brick element would have to evaluate displacements in three directions at each of its node. This property can be used to improve performance by grouping the equations at each node into a single block. Unrolling the operations on

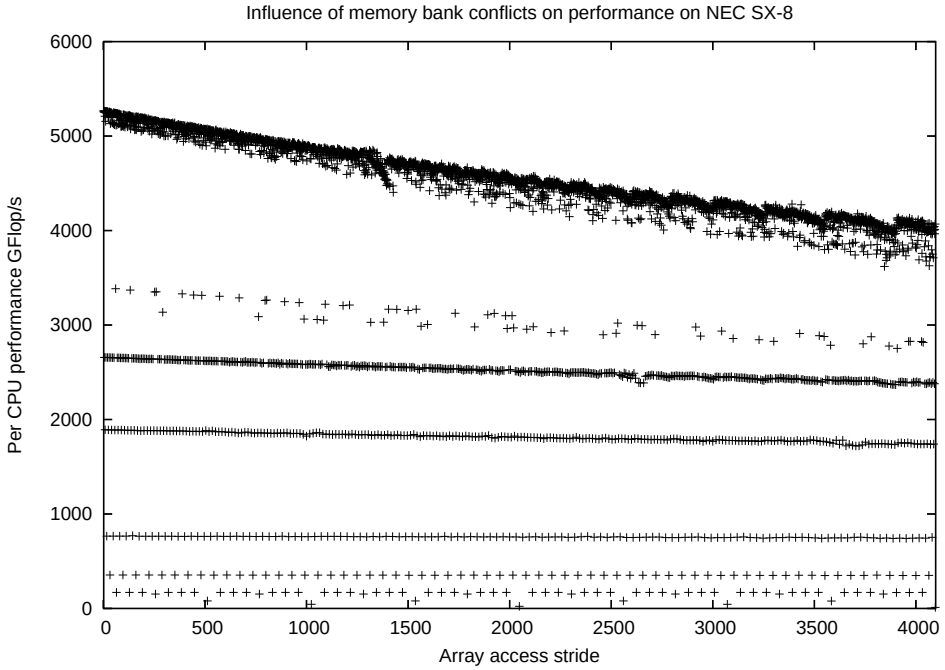

Figure 3.10: Influence of bank conflicts on DAXPY performance on the NEC SX-8. The highest performance level of about 4.5 GFlop/s corresponds to odd array strides and stride 2. The next performance level of about 2.5 GFlop/s corresponds to array stride 4. Each performance level that proceeds corresponds to an even stride that is double that of the previous level.

each such dense block considerably reduces the amount of indirect addressing required for Sparse MVP. This improves the performance dramatically on vector machines [57] and also remarkably on superscalar architectures [87]. The block based JAD sparse MVP algorithm is listed in Fig. 3.8 (block size 2 for simplicity). The reduction in indirect addressing can be clearly seen in the corresponding operation flow diagram shown in Fig. 3.9. The multiplied vector is indirectly addressed only twice (`vec1` and `vec2`) instead of four times as it would be the case for the original algorithm. The needed index vector `index` is only loaded once and then incremented. To generalize, indirect memory addressing is reduced by a factor of block size (2 in this case) [84]. As the decrease in indirect memory addressing is linearly proportional to the block size, the performance increases as the block size grows (see Section 3.3.2).

3.3.1 Avoiding bank conflicts on the vector system

The NEC SX-8, as any typical vector system, has a many-way interleaved bank memory with 4096 banks (Fig. 1.3). Figure 3.12 shows an array placement in

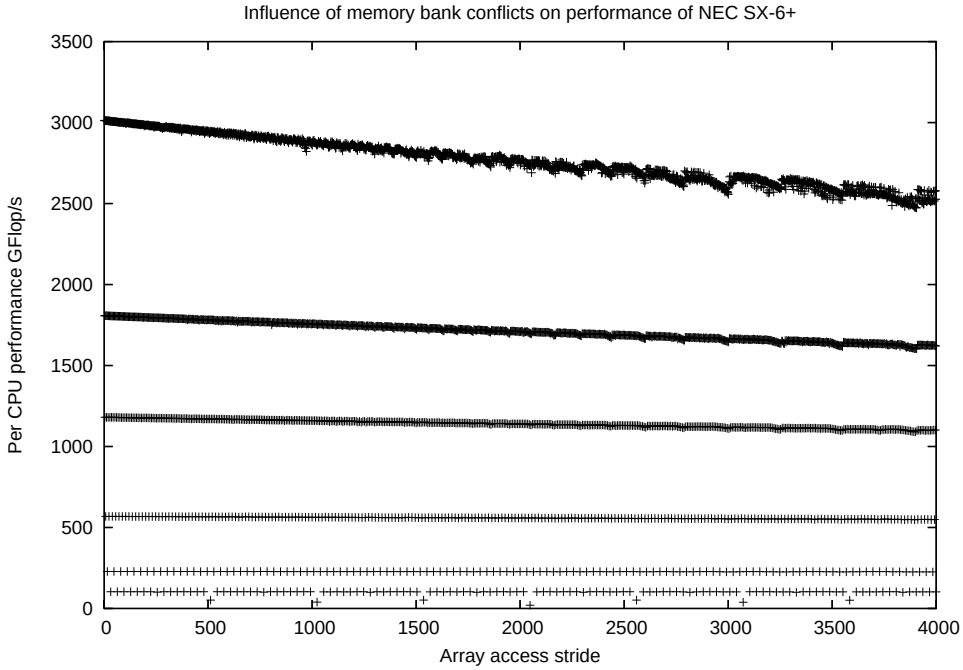

Figure 3.11: Influence of bank conflicts on DAXPY performance on the NEC SX-6+. The highest performance level of about 2.7 GFlop/s corresponds to odd array strides. The next performance level of about 1.7 GFlop/s corresponds to array stride 2. Each performance level that proceeds corresponds to an even stride that is double that of the previous level.

BANK1	BANK2	BANK3		BANK4094	BANK4095	BANK4096
a[0]	a[1]	a[2]	.....	a[4093]	a[4094]	a[4095]
a[4096]	a[4097]	a[4098]	.....	a[8189]	a[8190]	a[8191]
and so on..						

Figure 3.12: Array placement into memory banks on the NEC SX-8.

memory on the NEC SX-8. One drawback of such a design is the bank waiting time, which is the minimum time needed between two accesses to the same bank. So, if the time between two successive memory accesses is less than the bank waiting time, as can be the case with strided memory access, the processor has to wait to get the data from memory resulting in empty processing cycles. The worst possible case is to access the same bank repeatedly, i.e. a strided array access with an increment of 4096, which is the number of banks on the NEC SX-8. It is to be noted that only even increments that are multiples of four cause bank conflicts on the NEC SX-8 (On the NEC SX-6+, bank conflicts occur for strided array accesses with multiples of two and this has been improved on the NEC SX-

8). Figure 3.10 shows the performance of DAXPY (*y[i]=y[i]+x[i]*scalar*) with different array increments on the NEC SX-8 for arrays of size 10 million doubles. The maximum achievable performance for this kernel on the NEC SX-8 is 1/3 * peak as there are three memory accesses for two FP operations. On the NEC SX-8, two FP operations are theoretically possible for each memory access. So, the performance is limited by memory bandwidth for this kernel and the highest sustained performance with the best memory access is just a little over 5 GFlop/s per CPU (roughly 1/3 * 16 GFlop/s as expected). It can be noted from the figure that the performance falls into particular levels depending on the access pattern. For strided array accesses of multiples of four, the best sustained performance is below 3 GFlop/s. For accesses of multiples of 8, the performance is below 2 GFlop/s, for multiples of 16 below 1 GFlop/s, for multiples of 32 below 500 MFlop/s, and for multiples of 64 the maximum measured performance is below 250 MFlop/s on the NEC SX-8. The two worst measured performances occur for array access with increment 4096 where the measured performance is 11.1 MFlop/s and with increment 2048, the measured performance is 22.2 MFlop/s. This cycle repeats itself for increments larger than 4096 as this is the number of banks on the SX-8. Figure 3.11 plots the performance of DAXPY with different array increments on the NEC SX-6+. The main difference with the measurements on the SX-8 is that bank conflicts occurs for even strides starting with 4 instead of 2 (as on the SX-6+). This is also the reason that the top most bar is thicker in case of SX-8 when compared to SX-6+ as more array strides can sustain this performance. The performance sustained on the SX-6+ for this kernel without bank conflicts is 1/3 rd the peak (9 GFlop/s) which is about 3 GFlop/s.

This consideration is very important for block implementations of Krylov methods, particularly in kernels like the block JAD sparse MVP. Severe performance penalties are measured with even block JAD sparse MVP because of memory bank conflicts. The typical solution to this problem is array padding where a dummy variable is allocated between each block of the matrix and the memory access to the matrix is appropriately adjusted to simply neglect the dummy variables between matrix blocks. Figure 3.13 shows the performance with the original matrix access and after matrix array padding. The time spent in memory bank conflicts in the original block 4 sparse MVP kernel is about 78% and that with array padding is about 25% of the total execution time (array padding was enabled only for the sparse matrix array and not for the vectors) for matrix sizes of over 3000 rows. Hence, all the kernels with even block sizes implemented in BLIS have array padding for the matrix.

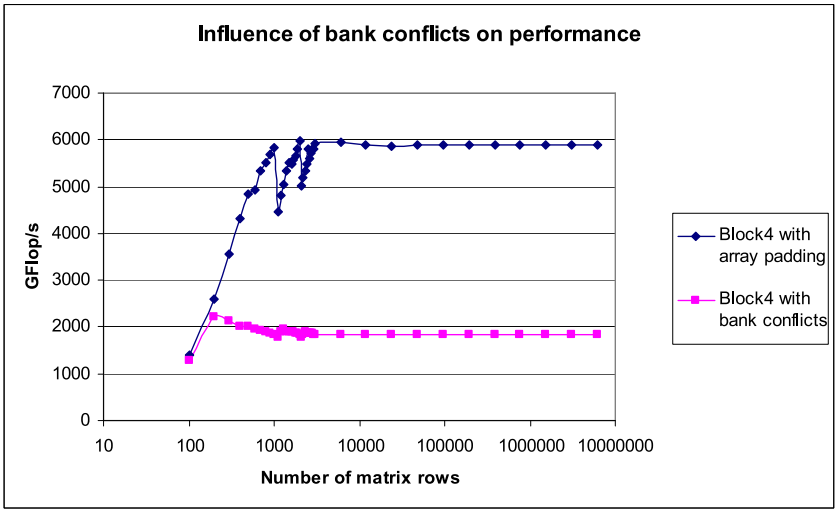

Figure 3.13: Influence of bank conflicts on blocked sparse MVP performance.

3.3.2 Performance in sparse MVP with increasing block size

Here, the performance in Block JAD sparse MVP kernel with increasing matrix block sizes is tested using a self implemented test suite on the NEC SX-8. Array padding is implemented for the matrix array in the test programs for even block sizes. The performance of odd block kernels may be slightly higher as the vectors are not padded in even block size kernels. But this difference is very small as the result vector is written only once at the end of the computation and the multiplied vector is indirectly accessed. Also, hardware vector registers are not used in these test programs, but are used in the final implementations to reduce the memory traffic. This can also slightly improve the performance as explained in Sect. 3.2.3.

The performance with increasing block sizes in block JAD sparse MVP is shown in Fig. 3.14, where the x-axis is in logarithmic scale. The matrix size is plotted on the x-axis and the number of non-zeros, which is the number of pseudo-diagonals in JAD storage format, is chosen to be a multiple of block size that is nearest to 100. It can be noted that the performance increases with increasing block size due to the reduction of amount of indirect memory addressing by a factor of block size as explained in Sect. 3.3. The highest performance achieved with the largest block size tested (block size 15) is a little over 11 GFlop/s, which is about 70% peak vector performance on the NEC SX-8. This is a considerable improvement when compared to the performance with point based implementation, which is about 1.8 GFlop/s (11% vector peak). This is the main reason for the general public domain sparse solvers like AZTEC to perform poorly on typical vector systems such as the NEC SX-8 (Fig. 3.7).

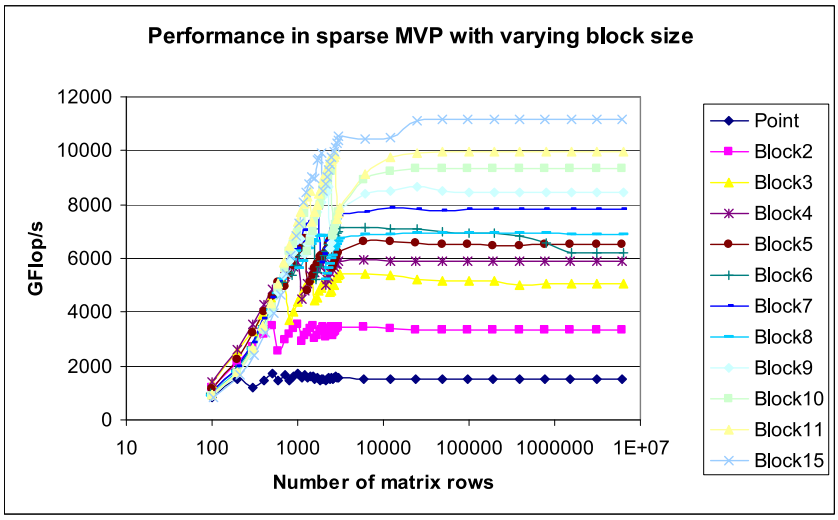

Figure 3.14: Performance in JAD sparse MVP with different block sizes. The x-axis is a logarithmic scale.

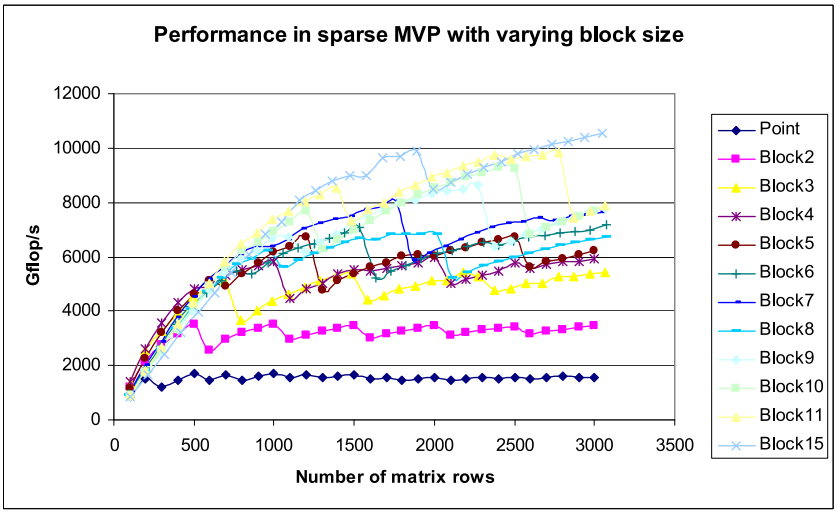

Figure 3.15: Performance in JAD sparse MVP for small problem sizes with different block sizes. The x-axis is a linear scale.

Figure 3.15 plots the same data up to a matrix size with about 3000 rows. The plot uses linear scale on the x-axis to look into the performance with smaller problem sizes. The sudden falls in performance for very small problem sizes can be attributed to the hardware register size, which is 256, on the SX-8. For problem sizes (i.e. innermost loop sizes) just slightly larger than the hardware register size, the performance drops as the FP pipelines are not completely filled with the second data chunk in the vectorized processing. This happens as the

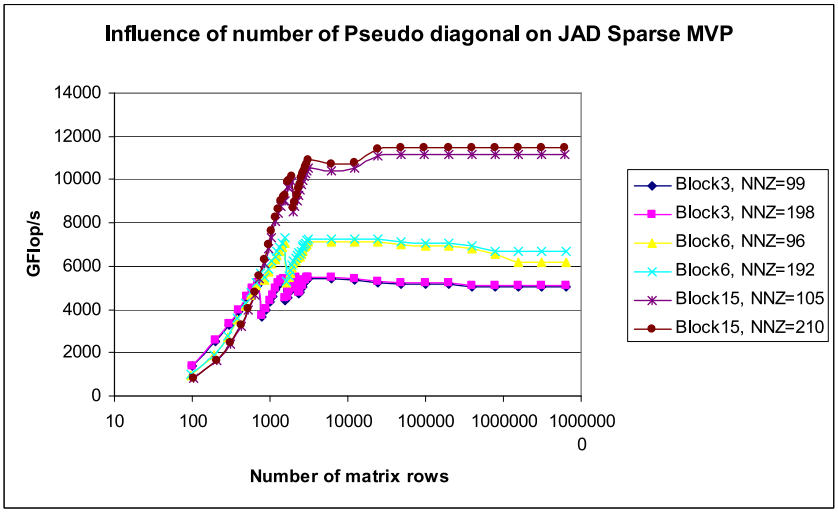

Figure 3.16: Performance in JAD sparse MVP with different pseudo diagonals.

data is processed in chunks of hardware register size. This performance drop due to partially empty pipelines corresponding to the processing of the last vector data chunk becomes insignificant with the increase in the problem size. The shift in the performance drops toward right in the plots with increasing block size is due to the fact that the actually processed problem size in the innermost loop of JAD block sparse MVP kernel is number of matrix rows / block size (due to unrolling the computation in each block).

Figure 3.16 plots the performance of block JAD sparse MVP with different block sizes after doubling the number of pseudo-diagonals. A performance improvement of less than 5% is measured for doubling the number of non-zeros/row, which are chosen to be of equal size in each matrix row for these tests.

3.3.3 Related work on blocking sparse matrices

There has been many studies in the past on the idea of blocking sparse matrices to improve performance on both the scalar and vector architectures. Some of the projects are reviewed here along with the reasons for their inapplicability on the NEC SX-8.

GeoFEM project

One successful project using the largest installation of NEC SX-6 (Earth Simulator project in Japan) was GeoFEM, which was active between 1997-2001, funded by The Science and Technology Agency of Japan [64]. The main aim of this project was a long-term prediction of the activities of the plate and mantle near the Japanese islands through the modeling and calculation of the solid earth

problems, including the dynamics and heat transfer inside the earth. A considerable effort went into optimizing the linear iterative solver used in the FEM software used for this purpose [58]. Block JAD format was used on the NEC SX-6 with Finite Elements having 3 unknowns/mesh point. General block sizes (for other Element types) are only implemented in the scalar version of the code using Block CRS format [6]. Also, robust preconditioning methods are missing in the GeoFEM solver. Two preconditioners that are available are local block ILU [60] which normally fails for highly ill-conditioned problems and the second is a problem specific preconditioning called Selective Blocking [57, 59]. In this method, the matrix blocks that belong to the contact elements (at the intersection of tectonic plates), which usually are the cause of ill-conditioning, are grouped into larger blocks and its inverse is used as a preconditioner in the Krylov methods. So, it is clear that this solver cannot be used on the NEC SX-8 for general FEM problems due to the above mentioned problem dependent constraints.

Scalable Software Infrastructure (SSI) project

The aim of the SSI project [63] is the development of basic libraries to be used in large scale scientific simulations and their integration into a scalable software infrastructure. The key component is a scalable iterative solver library (Lis), having a number of solvers, preconditioners, and matrix storage formats. In order to optimize the linear solver on the Earth Simulator (NEC SX-6), JAD format has also been made available in the solver. Block format is only available for row oriented storage and not for JAD storage [7] which makes its usage on the vector systems such as NEC SX-8 questionable.

There have been many projets adopting the block based approach to improve performance on the scalar systems. Some of the projects will be mentioned below. The main reason for not using them on the NEC SX-8 is that they use row oriented sparse storage formats that lead to short average vector lengths and result in poor sustained performance.

Trilinos project

In AZTEC, there is a block row format called the VBR (Variable Block Row) which blocks sparse matrix as small dense blocks of different sizes and uses Level 2 BLAS routines to exploit the data locality and thereby improve performance on the scalar systems [87]. In order to evaluate the performance of VBR format, Trilinos [40, 41], of which AztecOO [38] is a part, was run on the NEC SX-8. Trilinos is an object-oriented software framework that provides a unified access to various packages that solve linear and non-linear systems of equations, eigensystems and other related problems. One of the core packages in Trilinos is Epetra [39], which facilitates the construction and manipulation of distributed and serial graphs, sparse and dense matrices, vectors and multivectors. The point

(MSR) to block (VBR) conversion of sparse matrix objects is one such functionality available in this package. AztecOO provides an object-oriented interface to the AZTEC solver library and uses the distributed matrix and vector objects constructed via Epetra matrix and vector classes. Tests on the NEC SX-8 revealed that the performance of VBR format available in AztecOO was very poor. One of the reasons for this is due to the object oriented approach because of which the sparse MVP operation is realized as a series of function calls that multiply each row of the sparse matrix with the vector. The result is a performance of 100 MFlop/s per CPU (about 6% peak) on the NEC SX-8 in VBR sparse MVP.

Blocksolve95 project

Another interesting project on the scalar systems is Blocksolve95, an efficient sparse solver linear specially targeting FEM problems involving multiple degrees of freedom at each node [48]. This solver also uses block row formats and BLAS 2 calls on small dense matrices for achieving high sustained performance. Scalable incomplete factorization algorithms for symmetric and unsymmetric matrices can be used as preconditioners in the Krylov subspace methods. PETSc software package [13] contains an interface to Blocksolve95 preconditioners, i.e, the incomplete factorization algorithms.

SPARSITY system

This project is a recent effort to optimize the performance of sparse MVP on the scalar systems. SPARSITY [47] is a software system developed at the University of California at Berkeley to automatically build tuned sparse matrix vector product kernels for specific matrices and machines. The main idea behind the project is blocking sparse matrices in order to improve performance in the row-oriented sparse matrix vector product. The user has to provide the matrix structure resulting from his application as an input to this system. So, the SPARSITY system can be used by any iterative solver to generate the optimized kernels for the user data and then use these kernels in various algorithms implemented in the solver. This package has not been tested on the NEC SX systems as it works only for row oriented sparse storage formats. Blocking decisions (such as the maximum block size) are made depending upon two important optimizations. First is register blocking [46] where all the temporary results of the matrix vector multiplication of the small dense blocks are stored in hardware registers. Second is cache blocking [45] where the vector to be multiplied and the temporary results vector are chosen to fit in the top level (L1) cache of the particular processor. The size of these two vectors depends on the dimensions of the dense blocks in the sparse matrix.

OSKI library

The functionality in SARSITY was extended to accommodate blocking of sparse matrices with variable block sizes [98] and also generating optimized code for triangular solves [97]. The resulting work is distributed as Optimized Sparse Kernel Interface (OSKI) library [96] which is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices, for use in solver libraries and applications. OSKI provides basic tuned kernels like sparse matrix-vector multiply and sparse triangular solve. This package too only works for row oriented sparse storage formats. Generation of vectorized kernels is planned in the future.

3.4 Improvements to incomplete factorization algorithm

Factorization methods are commonly used to solve sub-domain problems in domain decomposition (DD) methods. The computational complexity in DD methods is a function of the problem size of each domain rather than the global problem size. Additive Schwarz domain decomposition methods based on incomplete sub domain factorizations are a popular choice of preconditioners for Krylov iterative methods. The main reasons being the quality of preconditioning and their suitability to parallel processing.

Factorization algorithms are heavily recursive and have data dependencies which makes them hard to vectorize and parallelize. The LU factorization of a matrix results in Lower and Upper triangular factors for general unsymmetric matrices which are then used for forward and backward substitution to reach the solution. These substitutions are sparse MVPs but have data dependencies such as in the Gauss Seidel method. Typically coloring [21, 74] or hyperplane ordering [10, 23, 33] is adopted to overcome the performance problems due to data dependencies in the forward and backward substitutions. Both these techniques have some drawbacks. Coloring changes the order of unknowns and generally leads to a different method than with the original order of the unknowns. Most reordering schemes at least double the number of iterations needed for convergence [27]. The performance improvement with hyperplane ordering on structured grids are elaborated in [88, 89]. It is to be noted that hyperplane ordering though retains the original ordering of the unknowns normally introduces indirect memory addressing to selectively process unknowns in order to resolve the data dependencies in the substitutions. Another problem with hyperplane ordering is memory bank conflicts which effect the performance on vector systems [33]. Lastly, the innermost loop length is generally shorter in case of hyperplane ordering when compared to ordering the unknowns by coloring.

The data dependency problem within the ILU factorization is nonetheless

```
for i = 1, n
  for k < i such that A_ik ≠ 0
    for j < k such that A_kj ≠ 0
      A_ij = A_ij − A_ik * A_kj
    end for
  end for
  for j > i such that A_ij ≠ 0
    A_ij = A_ii^-1 * A_ij
  end for
end for
```

Figure 3.17: Point or block IKJ variant of the ILU factorization algorithm.

harder to overcome. Due to the adoption of block based approach, the amount of data dependency is tremendously reduced. In addition a gather scatter algorithm [76] is used in BLIS that enhances performance over the original blocked algorithm due to the reduction of the amount of indirect addressing needed in the sparse factorization algorithm. The implemented algorithm does not allow for fill-in in incomplete factorization. The original and the vectorized algorithms are listed in Figs. 3.17 and 3.18 respectively. The algorithms listed in these figures can be applied on blocks of matrices, where A_{ij} will then be a small block matrix and not a single scalar value. In BLIS, block algorithms are implemented where matrix block inverses are necessary in place of a scalar division (as is the case with point based implementation). Also, scalar multiplication in the point based algorithm has to be replaced by matrix-matrix multiplication in case of the block based algorithm. The various one-dimensional arrays that are listed in Fig. 3.18 are:

- `A` – which upon entry is the original matrix with space for fill-ins and on return contains the ILU factors
- `ca` – which contains the column index of `A`
- `il` – which points to the beginning of row i of the matrix in the array `A`
- `iu` – which points to the beginning of the strictly upper triangular part of row i in `A`
- `trow` – which is a temporary array of size n

The matrix `A` is of size $n \times n$. In the vectorized algorithm, the present row to be factorized is first scattered to temporary array of size n. This array which is only sparsely filled would help eliminate the conditional statements needed in sparse LU factorization. Also, the indirect memory addressing in the factorization algorithm is now replaced by direct addressing of the sparse matrix rows. Indirect

```
for tr = 1, n
  // Scatter target row
  for it = il(tr), il(tr + 1) − 1
    trow(ca(it)) = A(it)
  end for

  for it = il(tr), iu(tr) − 2
    sr = ca(it)
    A(it) = trow(ca(it))
    for is = iu(sr), il(sr + 1) − 1
      trow(ca(is)) = trow(ca(is)) − A(it) ∗ A(is)
    end for
  end for
  A(iu(tr) − 1) = trow(tr)

  // Gather target row
  for it = iu(tr), il(tr + 1) − 1
    A(it) = (trow(tr))^{−1} ∗ trow(ca(it))
  end for
end for
```

Figure 3.18: Scatter-Gather approach to the point or block IKJ LU factorization algorithm.

memory addressing would be still needed while preparing this temporary matrix row array (Scattering) and while writing the factorized results back to the matrix array A (Gathering).

Performance results

Here, a performance comparison is provided between the ILU factorization algorithm in AZTEC, the original block ILU algorithm and the vectorized block ILU algorithm implemented in BLIS using the scatter-gather approach as described in [76]. Three problems were run on one node (8CPUs) of NEC SX-8. Table 3.1 lists the matrix sizes, the time taken in (block) ILU factorization and the sustained FP performance in the kernel. It can be noted from the table that the average vector length, which is the average length of the innermost loops, is better for the vectorized variant (Fig. 3.18) of the algorithm than the original one (Fig. 3.17) in BLIS. Also, the vector operation ratio, which is the percentage of the operations vectorized, is higher for the vectorized variant of the block ILU algorithm in BLIS. These improvements lead to a higher floating point performance in the vectorized block factorization when compared to the original algorithm. The improvement in time measured with the vectorized algorithm is over 20%

Matrix size rows/nnz	Algorithm	Time (msec)	MFlop/s per CPU	Vector operation ratio	Average vector length
8008/720481	AZTEC	71.18	44.1	6.77	119.9
”	BLIS-Orig.	18.92	156.3	20.66	12.2
”	BLIS-Vect.	13.03	315.4	31.59	26.4
40260/3891121	AZTEC	463.67	44.7	5.90	130.1
”	BLIS-Orig.	124.97	144.7	20.86	12.0
”	BLIS-Vect.	87.34	293.8	50.16	51.7
151040/15146881	AZTEC	1962.09	44.9	5.55	135.3
”	BLIS-Orig.	527.47	140.7	21.01	12.0
”	BLIS-Vect.	410.70	261.1	72.62	104.1

Table 3.1: Performance comparison of Block ILU implementations in BLIS.

for moderate problem sizes. The point based ILU algorithm available in AZTEC performs poorly with very low FP performance and vector operation ratio. The improvements seen in BLIS are due to the block implementation which heavily reduces the amount of recursion and increases the data locality via dense operations on small blocks. The vectorized block ILU factorization algorithm in BLIS is about 5 times faster in time than the point based algorithm in AZTEC.

3.5 Scheduling communication

Most public domain solvers provide functionality to identify the ghost variables that need to be communicated in some kernels like the sparse matrix-vector product. This is relatively straight forward for structured grids than for unstructured ones. Identifying ghost points is done only once (during the set-up phase) if the mesh does not change with time. With adaptive meshes, this may be needed every time the mesh is refined. This is especially true for unstructured grids where the application programmers do not normally make these ghost cell exchange lists. As the lists are ultimately used within the solver for exchanging boundary information, this functionality is typically provided within most solvers.

3.5.1 Implementation in BLIS

This functionality in BLIS is presently only implemented sequentially and its parallelization is straightforward. A new approach was adopted to build the exchange lists. In order to avoid heavy searching to determine the neighboring partitions, the global partitioned list is used. This list is typically the result from the domain decomposition tool which distributes the problem among the processors. The problem decomposition can also be provided by the user in case

Average message size per call	Type of MPI call	Wall time for neighbor exchange (msec)
3 Kbytes	Blocking	0.235
3 Kbytes	Non-blocking	0.129
3 Kbytes	Persistent	0.136

Table 3.2: Performance comparison of different point-to-point MPI implementations in BLIS.

of regular meshes. Most of the work in the process of communication scheduling goes in sorting which is of O(NLog(N)) complexity, where N is the global problem size. For moderate problem sizes this procedure has the advantage of overcoming the searching necessary to determine the neighbors. A parallel algorithm will be implemented to make this process scalable for large problem sizes.

3.5.2 Different MPI implementations and performance

There are mostly two kinds of calls used in a sparse iterative solver. MPI global reduction is needed to evaluate the inner products and point-to-point MPI send and receive calls are needed in the global sparse MVP calculations to exchange the neighbor information. Blocking and non-blocking MPI send and receive calls are implemented in BLIS for this purpose. Also, MPI persistent communication, which reduces communication overhead in programs which repeatedly call the same point-to-point message passing routines with the same arguments, is available in BLIS. The performance of these three different implementations is listed in Table 3.2. A pure fluid problem with 151K unknowns is run on 4 nodes (32 CPUs) of NEC SX-8. The wall time in MPI communication for one global sparse MVP is listed in the table along with the average message size for each point-to-point message transfer. Non-blocking communication is much faster when compared to blocking MPI send and receive calls. When using Persistent MPI calls for sending the messages, a small increase in wall time is measured on the NEC SX-8. This is usually the case when the MPI implementation is not optimized for persistent calls.

3.6 Coloring and vectorization

As mentioned in Sect. 3.4, coloring is absolutely necessary to resolve data dependencies in some algorithms and there by vectorize them. This is absolutely crucial on a vector system in order to obtain reasonable performance.

Optimal Graph Coloring in the general case is an NP-Complete problem. So, heuristic algorithms are used in practice for coloring graphs. The Gauss-Seidel al-

```
for i = 1, n
  Color(i) = 0
end for
for i = 1, n
  Color(i) = min {k > 0 | k≠Color(j), ∀ j ∈ Adj(i)}
end for
```

Figure 3.19: Greedy coloring algorithm.

gorithm described in Sect. 1.2.1 is often used as a preconditioner with the Krylov methods. In this algorithm, the results at each node are updated immediately and used in the evaluation of the unknowns at the next node. This leads to data dependencies that limit the vectorization to a single matrix row. This results in poor vector length as the number of non-zeros per row is typically small. Hence, independent rows are grouped into a single color and the calculations can then be vectorized using the block JAD storage format (Sect. 3.1.1) which results in a good vector length.

The greedy algorithm listed in Fig 3.19 is used in order to group nodes into different colors. It is observed that the number of colors with this algorithm normally does not exceed the maximum number of adjacent neighbors of a vertex in the graph plus one. The overhead associated with using the colored Gauss-Seidel method in BLIS is the memory needed to hold the colored sparse matrix using the block JAD storage format. Another approach that avoids this overhead stores the indices of the nodes associated with each color instead of actually storing a colored matrix in memory. The disadvantage then would be a severe performance penalty due to indirect memory access needed in the kernel.

One drawback of reordering the nodes via coloring is the degradation of convergence [27]. Additionally, there is a user option to rearrange the colors (for performance reasons) in the implementation available in BLIS. This rearrangement is necessary when there are colors with small number of nodes (e.g. < 100 nodes per color). In this case, the user can choose to reassign the nodes to other colors starting from the largest color so as to make the node count in the first few large colors to be a multiple of the hardware vector length (256 on the NEC SX-8). The number of colors before and after such a rearrangement for different problem sizes and cutoffs for the color size on 8 CPUs of NEC SX-8 is listed in Figs. 3.20 to 3.25. So, all the colors having less than the cutoff number of nodes are reassigned to other colors until their sizes become a multiple of the hardware vector length of that particular architecture. Since the matrix vector product in each color is vectorized using the Block JAD storage format, the performance is slightly better when the size of the colors is an exact multiple of hardware vector length. The cutoff as well as the hardware vector length have to be provided by the user. This rearrangement may also have an effect on the convergence of the method, which is generally a degrading one.

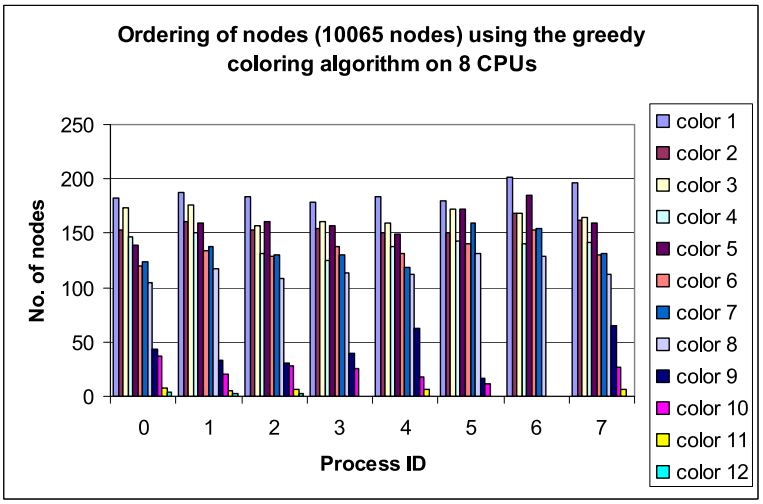

Figure 3.20: Reordering nodes on each processor (global problem size of 10065 nodes) using greedy coloring algorithm.

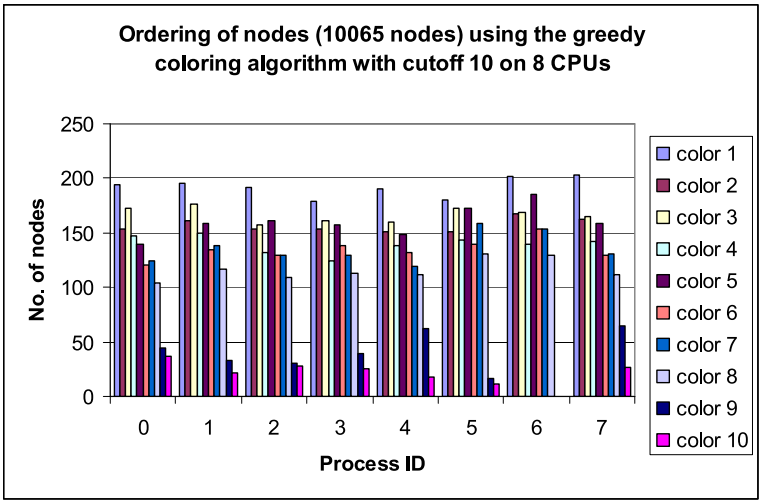

Figure 3.21: Reordering nodes on each processor (global problem size of 10065 nodes) using greedy coloring algorithm with a cutoff color size of 10.

Table 3.3 lists the performance of the BiCGSTAB method used along with different preconditioners to solve different FEM discretizations (problem sizes) of a laminar flow problem on 8 CPUs of NEC SX-8. The results in the table also list the iterations needed for convergence and time to solution with Block Jacobi (BJAC), Block Symmetric Gauss Seidel (BSGS), colored Block SGS (BSGS+C) and colored Block SGS with rearrangement (BSGS+C+R) preconditioners. The time listed in the table is the CPU time in seconds needed to solve three sparse linear systems while solving the discretized non-linear Navier-Stokes equation.

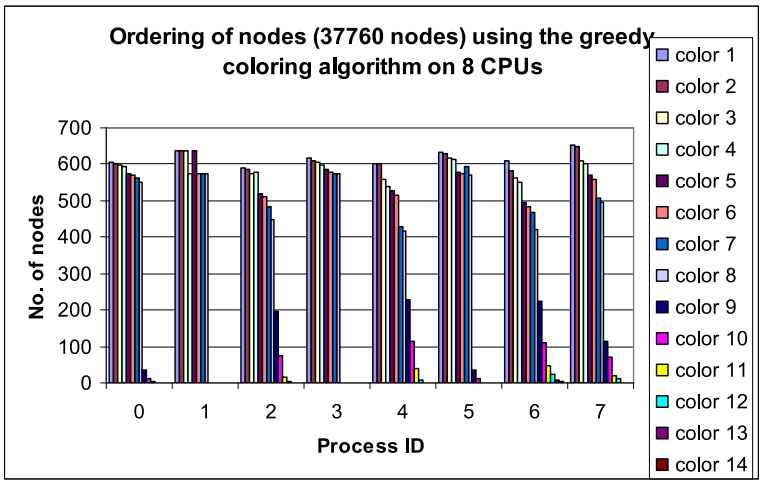

Figure 3.22: Reordering the nodes on each processor (global problem size of 37760 nodes) using greedy coloring algorithm.

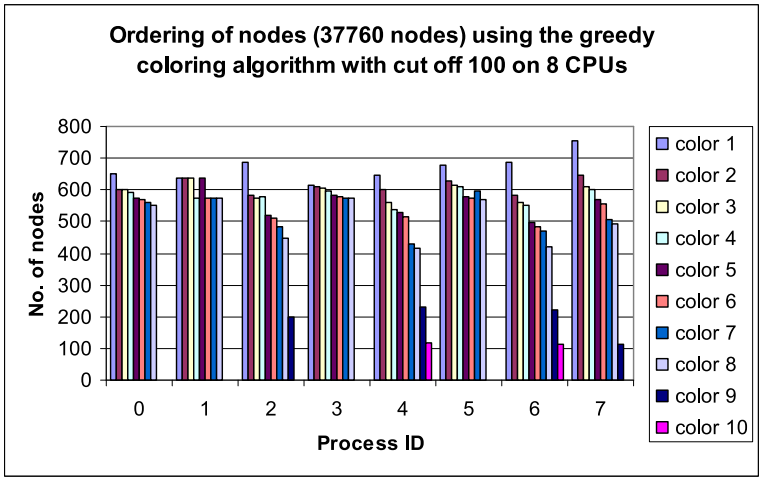

Figure 3.23: Reordering nodes on each processor (global problem size of 37760 nodes) using greedy coloring algorithm with a cutoff color size of 100.

This time also includes the time for coloring and rearranging the nodes in colors when using the BSGS+C and BSGS+C+R preconditioners respectively. The cutoff used for rearranging the colors is as mentioned in Figs. 3.20 to 3.25. The number of preconditioner iterations in the BiCGSTAB method are set to five for all the listed methods in the table.

The main reason for coloring can be clearly seen from the time to solution improvement between the original and the colored SGS algorithms. The improvement is at least an order of magnitude for the tested problems due to vectorization on the NEC SX-8. It has to be noted that the convergence with the BSGS+C

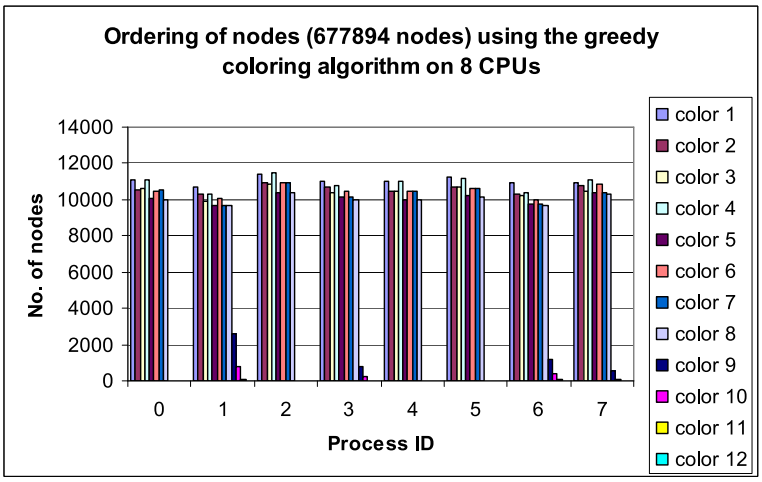

Figure 3.24: Reordering nodes on each processor (global problem size of 677894 nodes using greedy coloring algorithm.

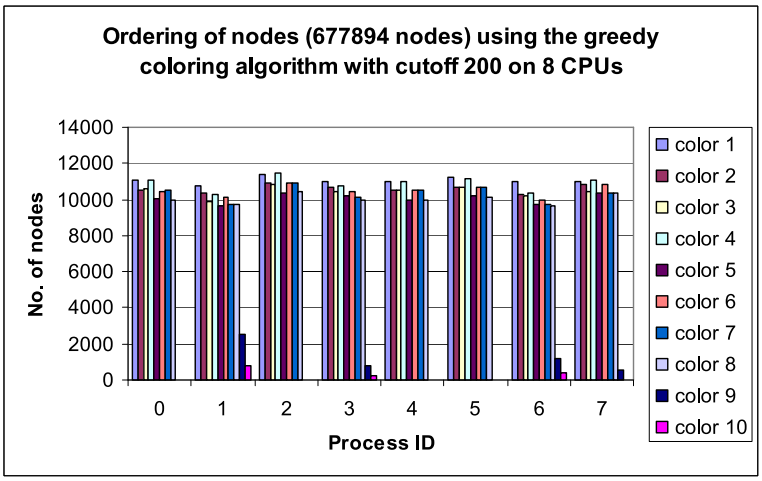

Figure 3.25: Reordering nodes on each processor (global problem size of 677894 nodes) using greedy coloring algorithm with a cutoff color size of 200.

preconditioner is slightly poor than with the original BSGS algorithm. Reordering the nodes in the colors (BSGS+C+R preconditioner) improves the average vector length and hence the FP performance. But, due to a slight degradation in convergence when compared to the original coloring algorithm, the performance advantage is usually lost and the time to solution is mostly measured to be the same as with the original coloring preconditioner (BSGS+C algorithm). The BJAC preconditioner is the best when considering the FP performance but the BiCGSTAB algorithm needs more iterations for convergence. Hence, for large problems, BSGS+C algorithm outperforms the BJAC algorithm with respect to

Problem size (# nodes)	Precond-itioner	CPU Time (sec)	BiCG-STAB iters	MFlop/s per CPU	Avg. vector length
10065	BJAC	5.4	164	3054	207
”	BSGS	267	69	48.1	1.3
”	BSGS+C	7.2	83	2139	105
”	BSGS+C+R	6.7	81	2276	115
37760	BJAC	33.3	354	4116	235
”	BSGS	1528	103	48.1	1.3
”	BSGS+C	30.7	129	2987	181
”	BSGS+C+R	32.7	143	3113	196
677894	BJAC	2727	1810	4789	250
”	BSGS	152605	553	47.9	1.3
”	BSGS+C	2233	633	3742	247
”	BSGS+C+R	2229	633	3750	247

Table 3.3: Performance of colored Symmetric Gauss Seidel preconditioner including setup time for coloring and rearranging.

time to solution.

3.6.1 Communication coloring

Efficient implementation of MPI communication is very important in order to achieve good scaling of the application onto large processor counts. In Sect. 3.5.2 different implementations of the MPI point-to-point calls have been described. In this section, the work done in order to organize the communication so as to reduce the wait times is described. It is clear from the measurements listed in Sect. 2.2.4 that the Blocking MPI_Sendrecv call uses the full memory bandwidth on the NEC SX-8 which is 64 GB/s for large messages. So, the communication is ordered using edge coloring in order to reduce the waiting time in Blocking calls. Edge coloring of a graph is same as vertex coloring of its line graph and the same greedy algorithm described in Fig. 3.19 is used. The number of colors with this algorithm normally does not exceed the maximum number of edges connected to a vertex in the graph plus one.

Figure 3.26 shows this ordering for a 32 CPU run on the NEC SX-8 where each color represents a communication step and processor numbers are also marked in the figure. Edge coloring ensures that the edges joining at each vertex (processor) are of a different color. So, in each color, every processor communicates with utmost one other processor. This ensures that there are no conflicts in any single colored communication step using the MPI_Sendrecv call. Coloring non-blocking point-to-point communication may sometimes improve performance where as it is absolutely necessary in the case of blocking MPI point-to-point calls in order

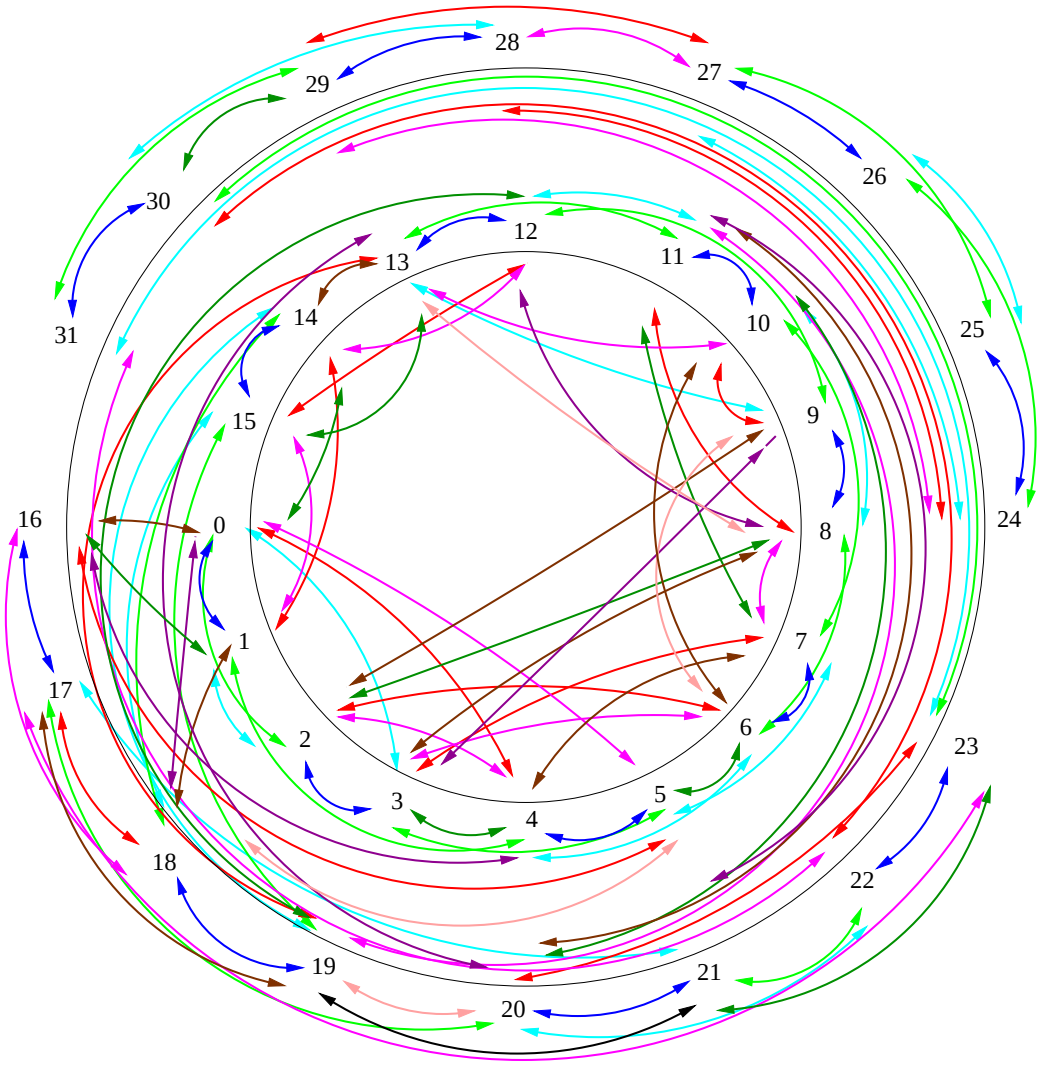

Figure 3.26: Edge Coloring point-to-point communication for a 32 CPU run.

Average message size per call	**Type of MPI call**	**Wall time for neighbor exchange (msec)**
3 Kbytes	Blocking	0.235
3 Kbytes	Colored Blocking	0.202
3 Kbytes	Non-blocking	0.129
3 Kbytes	Colored Non-blocking	0.134

Table 3.4: Performance comparison of colored point-to-point MPI implementations in BLIS.

to avoid cyclic dependencies resulting in a deadlock.

The results of coloring are provided in Table 3.4 for the same problem (a pure fluid problem with 151K unknowns) used to test the performance of different MPI implementations in Sect. 3.5.2 on 32 CPUs. For this example, the maximum number of neighbors are nine on processor nos. 8, 13 and 19 and hence the maximum colors is 10. The number of colors on processor no. 8 are nine and on processor nos. 13 and 19 are 10. So, in colored communication, processor nos. 13 and 19 both have a color iteration where they are idle and do not participate in communication. They have to simply wait till that color communication step is complete. One drawback of coloring is that there is an increase in the number of communication iterations by one due to the greedy coloring algorithm used. On the other hand, the advantage of coloring is of course the prevention of deadlock when using blocking communication. From the results it is apparent that performance improves when the MPI_Sendrecv point-to-point communication is colored. No performance improvement is measured in case of colored non-blocking point-to-point communication. From the results, it can be concluded that non-blocking point-to-point communication (without coloring) is found to be the most efficient when the application has an unstructured communication pattern as is the case here.

3.7 Preconditioning

The convergence of iterative methods depends on spectral properties of the coefficient matrix. In order to improve these properties one often transforms the linear system by a suitable linear transformation. A preconditioner is a matrix that effects such a transformation by nearly approximating the coefficient matrix A. The transformed system

$$M^{-1}Ax = M^{-1}b \tag{3.1}$$

has the same solution as the original system $Ax = b$, but the spectral properties of its coefficient matrix $M^{-1}A$ may be more favorable. In practice, instead of transforming the original linear system $A \rightarrow M^{-1}A$, the preconditioner is applied to the inner product used for orthogonalization of the residuals in Krylov methods. The key to a good preconditioning method is that it should be easy to setup and at the same time numerically robust. These conflicting needs keep the research active as most robust preconditioners such as incomplete factorization algorithms are computationally very expensive. An important consideration when selecting a preconditioner for a particular iterative method is the reduction in time to solution with the use of the preconditioner. This does not necessarily mean that the preconditioner is the best in terms of convergence of the method.

Another important reason for using a preconditioner is to reduce the number of iterations needed for convergence in the Krylov subspace methods which

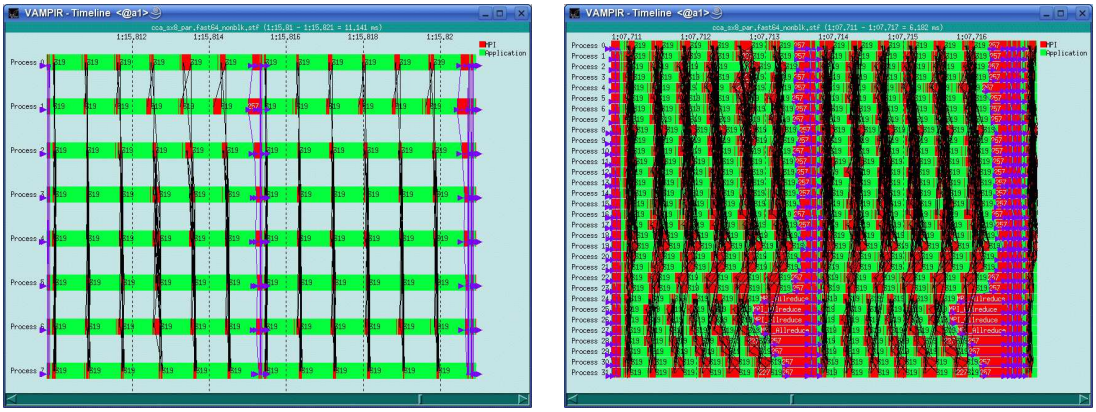

Figure 3.27: Vampir trace output of one BiCGSTAB iteration on 8 SX-8 CPUs (left) and 32 SX-8 CPUS (right) for the same fluid problem.

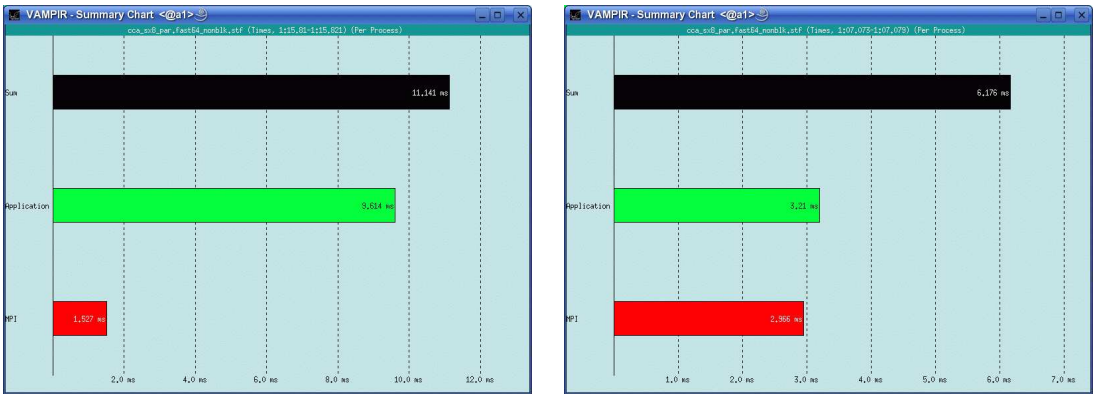

Figure 3.28: Vampir trace output of time (sec) needed in one BiCGSTAB iteration on 8 SX-8 CPUs (left) and 32 SX-8 CPUS (right) for the same fluid problem. Green corresponds to time in calculation and red to time in communication.

need MPI Allreduce calls to evaluate global inner products in each iteration. This becomes specially important if one considers the direction of modern high performance computing which connects more and more commodity processors (hundreds of thousands) using advanced interconnect technologies. The main concern with this development is the increase in sensitivity with respect to Operating System (OS) jitter [52] and performance of MPI collective calls. This is the main reason behind running lightweight OS kernels on the compute nodes and a complete OS only on some special nodes [8]. Applications (unstructured meshes, structured adaptive refinement, etc) with even a small load imbalance would waste a lot of time in synchronization as most processors need to wait for

the slowest processor to join the MPI collective call. So, with applications/kernels that frequently need global synchronization such as a sparse linear solver, preconditioning plays an important role to efficiently scale the applications to large processor counts.

In order to show the extent of the problem, Fig. 3.27 show the time line plots of a single iteration of BiCGSTAB method used to solve the linear systems arising from linearized Navier-Stokes equations when simulating an instationary laminar flow problem with 151K unknowns. The plot on the left of the figure correspond to a 8 CPU run and the plot on the right correspond to a 32 CPU run. The color green in the plots corresponds to the time spent in computation and red corresponds to the time in MPI communication. The average time spent in computation and communication on each of the processes for one BiCGSTAB iteration is shown in Fig. 3.28. The time per CPU spent in computation reduces from 9.6 sec on a 8 CPU run to 3.2 sec on a 32 CPU run due to the decrease of problem size/process. Though, it is interesting to note that the average time spent in communication increases from 1.5 sec to 2.9 sec per CPU with the increase in the processor count. This is mainly due to global synchronization needed for MPI collective operations. The small load imbalances, typical with many scientific applications, accumulate at each global synchronization point. As can be seen in Fig. 3.27, after six sparse MVPs (needed in BiCGSTAB and the block Jacobi preconditioner), there is always an MPI_Allreduce call where all the processors need to synchronize. A major part of the time spent in communication can be associated to this MPI operation.

Dynamic load balancing is a well known solution to the above problem. The PHD thesis by Christen [19] and the resulting software PAISS (Parallel Adaptive Iterative linear System Solver) tries to address this problem in the context of iterative solvers. So, in view of the above mentioned problems, it is important to keep the iteration count of methods that need global synchronization as low as possible. This becomes crucial when considering the scaling of such methods on to a very large processor count. Preconditioning the linear systems in order to improve the spectral properties (condition number) of the matrix, thereby reducing the number of iterations needed for convergence, is an elegant way to achieve this goal.

There are various possibilities for preconditioning the Krylov subspace methods. Both stationary and multigrid methods are a popular choice. One important consideration while selecting a preconditioner is the time needed to set up and use the preconditioner. HYPRE (High Performance Preconditioners) solver package [30] developed at the Lawrence Livermore National Labs was tested on the NEC SX-8 to measure the performance improvement using preconditioners. This solver package is only used for these tests and in general performs poorly on the NEC SX-8 because of the reasons listed in Sect 3.2.4. The time taken in the solution and the convergence behavior using different methods to solve a Laplace problem on a 2D structured grid of size 500×500 is listed in Table 3.5. A convergence tol-

Method	# v-cycles or iters.	operator complexity	total time to solution (sec)
AMG/Flagout coarsening	11	2.347	27.414
AMG/Ruge-Stüben coarsening	12	2.365	28.303
CG/AMG preconditioner	7	2.347	29.756
CG/Jacobi preconditioner	860	1	109.815

Table 3.5: Comparison of different methods in HYPRE used for solving a 2d 500x500 Laplace example on the NEC SX-8.

erance of 1e-8 was chosen (for relative residuals). Most of the time (about 70%) for solving the Laplace problem was spent in relaxation on each grid. About 10% of time was spent in the coarsening algorithm and another 10% of the time in making the coarse grid operators. Roughly 10% of time was in the interpolation and restriction of the coarse grid correction and fine grid residuum vectors respectively. The performance in the relaxation algorithm (based on sparse MVP) was around 600 MFlop/s due to the point-based implementation.

As can be noted from Table 3.5, time to solution for CG with Algebraic Multigrid (AMG) method as preconditioner (1 V cycle per CG iteration) is comparable to the time to solution using a purely AMG method. It is to be noted that the above comparison is only valid for point-based methods on the SX-8 where the relaxation operator (sparse MVP) performs rather poorly. The main reason for using AMG as a preconditioner or a complete linear solver is due to the reason that the convergence is independent of problem size. So, the number of iterations for convergence would nearly stay the same even for larger problems (elliptic PDEs) which makes the method truly scalable. It can be noted from the table that convergence is much slower in case of CG with Jacobi preconditioner and the number of iterations increases with the increase in the problem size. The AMG method is available as a preconditioner for Krylov methods in BLIS and the implementation details are elaborated in Sect. 3.7.2.

3.7.1 Block vs point based preconditioning

Jacobi method as described in Sect. 1.2.1 is the simplest form of preconditioner that can be used with Non-Stationary iterative methods. This is very easy to implement as only the diagonal of the matrix need to be inverted. No additional storage is necessary to implement the Jacobi preconditioner. Block variant of this method would combine different variables into small block matrices and operate using each such block as a single entity. So, any operation such as inverting a diagonal entry of a matrix would now mean inverting a block diagonal [14, 23]. The size of the blocks can vary depending on the way the problem is blocked. In problems with multiple physical variables per node, blocks can be formed by grouping the equations at each node. BLIS uses this approach both in

implementing the Non-Stationary methods and the preconditioning algorithms. Other ideas of blocking can be based on partitioning the physical domain on each processor where each partition will be handled as a block matrix. This approach is famous among some public domain solvers [40] in order to efficiently use caches as well as to improve the quality of the preconditioner. Typically, BLAS routines are used on the block matrices in order to improve the performance.

Incomplete factorization algorithms are also a popular choice of preconditioners for the Non-Stationary methods. Block incomplete factorizations would mean that the pivot blocks have to be inverted which in the case of point incomplete factorizations is just a scalar inversion. The choice of the block sizes can be made in a flexible way as described above. The incomplete LU factorization implementation in BLIS operates on blocks resulting from combining multiple physical variables per node. In case of large pivot blocks, only an approximation of block inverses are computed in order to preserve the sparsity pattern of the block. In BLIS, complete block inverses are computed as the block sizes are relatively small.

3.7.2 Algebraic multigrid

Multigrid methods fall under the category of truly scalable solvers because of their optimal complexity. The method was first popularized by Achi Brandt in the early seventies [17]. This method builds upon the stationary iterative methods by accelerating their convergence. The Krylov subspace methods can also be viewed as methods for accelerating the stationary/relaxation methods. The multigrid methods are based on the spectral analysis of the residual vector in relaxation methods such as the Jacobi or Gauss-Seidel. It is observed that the high frequency / oscillatory error components (i.e. components associated with larger eigenvalues of the matrix) are quickly canceled but the low frequency / smooth error components which are associated with the lower eigenvalues of the matrix are slowly eliminated by these methods. In order to accelerate the removal of low frequency error components, coarser grids are used. This process is called *coarse grid correction* and it compliments the *smoothing* properties of relaxation methods. Coarse grid correction involves transferring information to a coarser grid through *restriction*, solving a coarse-grid system of equations, and then transferring the solution back to the fine grid through *interpolation* which is also called prolongation. The main step in multigrid methods starts with smoothing on the fine grid Ω^h (pre-smoothing) using a relaxation method (typically some Gauss-Seidel iterations). This is followed by the restriction of the residual on to the coarser grid Ω^{2h}, where h is the mesh spacing of the discretization and I_h^{2h} is the restriction operator.

$$r^{2h} = I_h^{2h}\ r^h \tag{3.2}$$

Then, relaxation of the residual equation on the coarse grid Ω^{2h} to obtain an

approximation to the error e^{2h} (this contains the low frequency error components on the fine grid).

$$A^{2h}\ e^{2h} = r^{2h} \tag{3.3}$$

The error is again interpolated back to the fine grid using the interpolation operator I^h_{2h} and is added to the iterate.

$$x^h = x^h + I^h_{2h}\ e^{2h} \tag{3.4}$$

This transfer of error to the fine mesh by interpolation generally works well as the error is smooth (the oscillatory components of the error, if any, are already removed by pre-smoothing on the finer grid). This is typically followed by post-smoothing on the fine mesh using a relaxation method. If the residual equation on the coarser grid is solved using an even coarser grid Ω^{4h} and so on till the problem to be solved on the coarsest grid is small enough to be solved by a direct method, this results a true multigrid scheme. Various multigrid cycling schemes are possible. The V-cycle is the basic recursive scheme which is a special case of a μ-cycle. A Full Multigrid V-cycle (FMV) improves the initial guess for the first fine grid relaxation in the V-cycle.

Standard multigrid methods cannot be applied without the existence of an underlying mesh (discretized geometry). Algebraic Multigrid (AMG) method initiated by Ruge and Stüben [72] overcomes this limitation. For a good tutorial on AMG see [18] and for a more complete description of the method refer [37]. A quick overview on the method and some recent developments can be found in [29, 81]. For parallel implementations of AMG and available software packages see [102].

The AMG method determines coarse grids (coarsening), inter-grid transfer operators (restriction and interpolation), and coarse-grid equations (matrices $A^{(n)h}$) based on the entries of the finest grid matrix A^h. The coarse-grid equations are defined by $A^{(n)h} = I^{(n)h}_h A^h I^h_{(n)h}$, where $I^{(n)h}_h$ is the restriction operator and $I^h_{(n)h}$ is the interpolation operator. The most common approach is to use the Galerkin operator to define the coarse-grid equations where the restriction operator $I^{(n)h}_h$ is just a transpose of the interpolation operator , i.e. $(I^h_{(n)h})^T$. Coarse grid selection and interpolation are the key factors that determine the convergence of the method and affect each other in many ways. The quality of the AMG method is indicated by the *convergence factor*, which gives an estimate of the number of iterations needed for convergence and *complexity*, which gives an estimate on the operations needed per iteration. Both these metrics have to be considered when defining the coarsening and interpolation procedures. Higher complexity resulting from larger stencils and more robust interpolation schemes increase the computation per iteration but as well improve the convergence.

AMG is widely used as a preconditioner with Krylov methods in which case the complexity can be kept lower. Practical experience has clearly shown that

AMG is also a very good preconditioner with acceleration methods such as conjugate gradient, BI-CGSTAB or GMRES. This development was driven by the observation that it is often not only simpler but also more efficient to use accelerated multigrid approaches (i.e multigrid method as a preconditioner in Krylov methods) rather than to try to optimize the interplay between the various multigrid components in order to improve the convergence of stand-alone multigrid cycles. AMG's convergence factor is typically limited by the slow convergence of just a few exceptional error components, which were not properly transfered to coarser grids by coarsening/interpolation process, while the majority of the error components are reduced very quickly. Acceleration by Krylov methods typically eliminates such frequencies very efficiently, hence making this combination very powerful.

AMG is implemented as a preconditioner for the Krylov methods in BLIS. Only two level coarsening is available in the present implementation. Classical Ruge-Stüben (RS) coarsening and direct interpolation [72, 81] are used. The other popular approach used for determining the coarse grid (coarsening) is by smooth aggregation [92, 93]. RS coarsening works on the concept of *strong influence* and *strong dependence*. A point i depends strongly on j or j strongly influences i if

$$-a_{ij} \geq \theta \max_{k \neq i}(-a_{ik})$$

The size of the strength threshold θ can have a significant influence on complexities, particularly stencil size, as well as convergence and this parameter can be chosen by the BLIS users. In RS coarsening, the following two conditions are to be fulfilled:

C1: For each point j that strongly influences an F-point i, j is either a C-point or it strongly depends on a C-point k that also strongly influences i.

C2: The C-points should be a maximal independent subset of all points, i.e. no two C-points are connected to each other, and if another C-point is added the independence is lost.

C1 is designed to insure the quality of interpolation, while C2 is designed to restrict the size of the coarse grids. In general, it is not possible to fulfill both conditions, therefore C1 is enforced, while C2 is used as a guideline. RS coarsening consists of two passes. In the first pass, each point i is assigned a measure λ_i, which equals the number of points that are strongly influenced by i. Then a point with a maximal λ_i (there usually will be several) is selected as the first coarse point. Now all points that strongly depend on i become F-points. For all points that strongly influence these new F-points, j is incremented by the number of new F-points that j strongly influences in order to increase j's chances

of becoming a C-point. This process is repeated until all points are either C- or F-points. Since this first pass does not guarantee that condition C1 is satisfied, it is followed by a second pass, which examines all strong $F - F$ connections for common coarse neighbors. If C1 is not satisfied new C-points are added.

The property of *smooth error* is used to define the interpolation operator in AMG. In the geometric case, the most important property of smooth error (low frequency error components) is that it is not effectively reduced by a relaxation method (Jacobi or Gauss-Seidel methods). Weighted Jacobi relaxation method can be expressed as

$$x \leftarrow x + \omega D^{-1}(b - Ax) \tag{3.5}$$

where D is the diagonal of A. As detailed in Sect. 1.2.1, x is an approximation to the exact solution x^*. The error propagation for this iteration is given as

$$e \leftarrow (I - \omega D^{-1}A)e \tag{3.6}$$

Algebraically smooth error means that the size of $e^{(k+1)}$ is not significantly less than that of $e^{(k)}$. Measuring the error in the A-norm (if A is symmetric positive definite) gives

$$\|e\|_A = (Ae, e)^{1/2}$$

using this norm and equation 3.6, we see that the algebraically smooth error is characterized by

$$\|(I - \omega D^{-1}A)e\|_A \approx \|e\|_A$$

this translates into

$$(D^{-1}Ae, Ae) \ll (e, Ae)$$

writing the above equation in components

$$\sum_{i=1}^{n} \frac{r_i^2}{a_{ii}} \ll \sum_{i=1}^{n} r_i e_i, \quad where \quad r = Ae$$

This implies that, on an average, algebraically smooth error e satisfies

$$|r_i| \ll a_{ii}|e_i| \tag{3.7}$$

This is the key underlying assumption of classical AMG which means that algebraically smooth error has very small residuals. To derive interpolation, this property is used in the form (the property in Equation 3.7 is extremized by making the residual to be zero)

$$Ae \approx 0$$

If i is a F-point to which we wish to interpolate, N_i the neighborhood of i, i.e. the set of all points, which influence i. Then the ith equation becomes

$$a_{ii}e_i + \sum_{j \in N_i} a_{ij}e_j = 0$$

Classical/RS interpolation as described in [72] proceeds by dividing N_i into the set of coarse neighbors, C_i, the set of strongly influencing neighbors, F_i^s, and the set of weakly influencing neighbors, F_i^w. Using those distinctions as well as condition C1, which guarantees that a neighbor in F_i^s is also strongly influenced by at least one point in C_i yields the following interpolation formula

$$w_{ij} = \frac{1}{a_{ii} + \sum_{k \in F_i^w} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik}a_{kj}}{\sum_{m \in C_i} a_{km}} \right)$$

This interpolation formula fails whenever C1 is violated, since there would be no m and one would divide by zero. One can somewhat remedy this by including elements of F_i^s that violate C1 in F_i^w, but this will affect the quality of the interpolation and lead to worse convergence. Nevertheless often good results can be achieved with this interpolation, if the resulting AMG method is used as a preconditioner for a Krylov method as is the case in BLIS. One advantage of this interpolation formula is that it only involves immediate neighbors and thus is easier to implement in parallel, since it requires only one layer of ghost points located on a neighbor processor.

The computationally intensive parts in AMG are building the coarse grid operators and relaxation on different grids (Jacobi or Gauss-Seidel methods, i.e essentially based on sparse matrix vector product). With the blocking scheme as in BLIS, it is clear that the relaxation/smoothing operator based on Sparse MVP is very efficient on vector systems. If the sparse matrix-matrix operations necessary to build the coarse grid operator can be implemented efficiently, this method would be very well suited for vector systems. An advantage on the vector systems is the huge memory available (16 GB / CPU on the SX-8) when compared to clusters of PCs (typically $<$ 4 GB / CPU).

Figures 3.29 and 3.30 show the quality of AMG preconditioning for laminar flow problems described in Sect. 4.1.2 with 8K and 40K unknowns respectively.

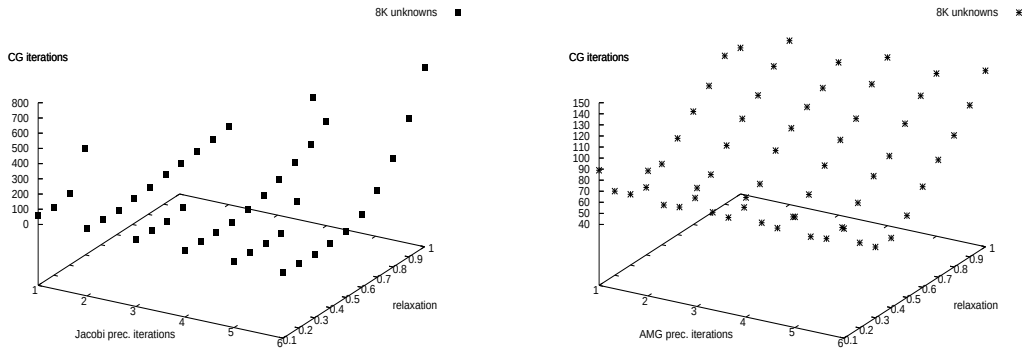

Figure 3.29: Convergence in BiCGSTAB with Block Jacobi as preconditioner (left) and with AMG as preconditioner (right) for solving a fluid problem with 8K unknowns.

The problems were run using 8 CPUs of NEC SX-8. The vertical axis lists the iterations in the Krylov subspace method (BiCGSTAB in this case) and the horizontal axes list the no. of Jacobi iterations or AMG relaxation iterations and the value of the relaxation parameter, ω, used in these methods. The strength threshold for coarsening, θ, was chosen to be 0.25 for these problems. The convergence behavior is more predictable and smooth in case of the AMG preconditioning when compared with block Jacobi preconditioning due to the effective reduction of the low frequency error components. It can also be noted that AMG preconditioner is not as sensitive to the relaxation parameter as the Jacobi preconditioner. The results in the figures are obtained using only one additional coarser level for AMG preconditioning. Ruge-Stüben coarsening and interpolation is used along with Block Jacobi relaxation for these tests.

Table 3.6 lists the performance of various parts of AMG preconditioning for two different problem sizes. The time measured in the set-up is divided into two parts. The first part includes the time for determining the coarse grid (coarsening) and building the interpolation and restriction operators. The second part includes the time spent in building the coarse grid operator. It can be seen from the table that most of the time in the set-up is spent in the second part where two sparse matrix-matrix products are computed. It should also be noted that SX-8 is slower in this kernel than Itanium 2 due to the poor MFlop/s rate. The listed time is for the scalar version of the kernel run on SX-8. The problem with the vector version (unrolled and vectorized) is that it is much more slower (in time) than the scalar variant though it shows a higher sustained FP performance. The reason for this is still under investigation. It can also be noted that the time in the solution (mainly relaxation) is nearly 4 times faster on the SX-8 than on the Itanium 2.

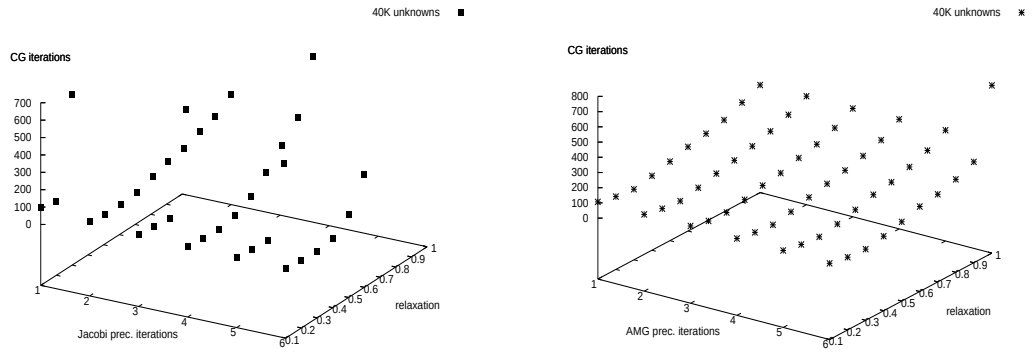

Figure 3.30: Convergence in BiCGSTAB with Block Jacobi as preconditioner (left) and with AMG as preconditioner (right) for solving a fluid problem with 40K unknowns.

Machine	**Problem size**	**Time in Setup1 (Coarsening+ Interpolation+ Restriction)**	**Time in Setup2 Coarse grid operator =(R)(A)(P)**	**Time in Solution**
NEC SX-8	8008	0.017 sec (500 MFlop/s)	0.495 sec (12 MFlop/s)	0.333 sec (2.5 GFlop/s)
Itanium 2	8008	0.03 sec	0.21 sec	0.64 sec
NEC SX-8	40260	0.093 sec (517 MFlop/s)	12.568 sec (3.2 MFlop/s)	2.615 sec (2.6 GFlop/s)
Itanium 2	40260	0.22 sec	4.28 sec	9.21 sec

Table 3.6: Performance comparison for AMG kernels between NEC SX-8 and Itanium 2.

3.8 Available features in the solver

Presently, BLIS (Block-based Linear Iterative Solver) is working with finite element applications that have 1, 3, 4 or 6 unknowns to be solved per mesh point. Block JAD sparse storage format is used to store the dense blocks of the matrix. This assures sufficient average vector length for operations done using the sparse matrix object (Preconditioning, Sparse MVP). The single CPU performance of sparse MVP, which is the key operation in sparse linear iterative solvers, with a matrix consisting of 4x4 dense blocks is around 7.2 GFlop/s (about 45% vector peak) on the NEC SX-8 (see Fig. 3.31). The performance in this kernel increases with the increase in the size of the dense blocks as explained in Sect. 3.3.2.

BLIS is developed using C and is based on MPI for parallelism. A Fortran

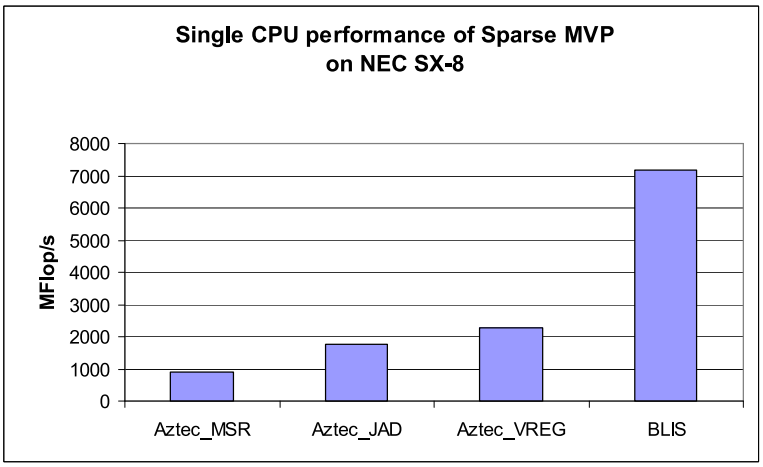

Figure 3.31: Single CPU performance of Sparse MVP on NEC SX-8.

interface is also available to be used by application codes developed in Fortran. The solver includes the commonly used Krylov subspace methods such as CG, BiCGSTAB and GMRES(m). Block scaling, block Jacobi, colored block symmetric Gauss-Seidel, block ILU(0) on sub-domains and a two-level Algebraic Multigrid method are the available matrix preconditioners. Exchange of halos in sparse MVP can be done using MPI non-blocking or MPI persistent communication. Matrix blocking functionality is provided in the solver in order to free the users from preparing blocked matrices. Users can use this functionality to convert point based CRS matrices (needed by most public domain sparse solvers) to the block JAD format, which is used with in BLIS. This helps users to easily migrate to BLIS from public domain solvers without much effort. For more information on using BLIS please refer Appendix A.

Chapter 4

Usage of BLIS in applications

In this chapter, performance results are presented for some engineering application codes that are using BLIS. The development of BLIS was started in the process of running large flow and structural simulations on the vector system, NEC SX-8, using Computer Aided Research Analysis Tool (CCARAT) developed at the Institute of Structural Mechanics (IBS), University of Stuttgart. The main bottleneck to performance was the public domain sparse linear solvers used in CCARAT. Most of the time in the whole simulation (in some cases upto 90%) was spent in the linear solver on the vector system. This lead to the development of a vectorized sparse solver BLIS. The solver is developed as an independent library and can be used by any FEM application codes. Some other codes that have started to couple BLIS are Finite Element based Numerical Flow Simulation System (FENFLOSS), developed at the Institute of Fluid Mechanics and Hydraulic Machinery (IHS), University of Stuttgart and Finite Element Ocean Model (FEOM), developed at Alfred Wegener Institute for Polar and Marine Research (AWI), Bremenhaven.

4.1 CCARAT

CCARAT (Computer Aided Research Analysis Tool) is a research finite element program designed to solve a large range of problems in Computational Fluid and Solid Mechanics: e.g. multifield, multiscale and multidiscretisation schemes, shape and topology optimization problems and material modeling and element technology aspects. It is developed and maintained at the Institute of Structural Mechanics (IBS), University of Stuttgart. Various types of sophisticated structural elements (e.g. shell, brick and beam) and GLS stabilized fluid elements in 2d and 3d on a moving mesh (Arbitrary Lagrangian-Eulerian formulation) are included in the code. For the linear system of equations several external solvers (e.g. AZTEC, SPOOLES, UMFPACK and TRILINOS) as well as an in-house parallel semi-algebraic multilevel scheme based on the aggregation concept are

available.

Fluid structure interaction with CCARAT

The considered class of applications fall under the category of transient interaction of incompressible viscous flows and nonlinear flexible structures. Partitioned analysis techniques enjoy more popularity than the fully coupled monolithic approaches, as they allow the independent use of suitable discretization methods for physically and/or dynamically different partitions. In this approach a non-overlapping partitioning is employed, where the physical fields (fluid and structure) are coupled at the interface, i.e the wetted structural surface. A third computational field, the deforming fluid mesh, is introduced through an Arbitrary Lagrangian-Eulerian (ALE) description. Each individual field is solved by semi-discretization strategies with finite elements and implicit time stepping algorithms. The iterative substructuring schemes used are accelerated via the gradient method(method of steepest descent) or via the Aitken method.

4.1.1 Vectorization of element integration

The original flow of instructions for the element integration in the finite element method is shown in the left part of the Fig. 4.1. The main problem with this implementation on vector systems is the length of the innermost loop, which depends on the number of nodes of each element and hence is a very small number. To make the innermost loop longer, computationally similar elements are grouped into sets and then all the calculations necessary to build the element matrices are performed simultaneously for all elements in one set. The modified flow of instructions is shown in the right part of Fig. 4.1. The length of this loop is equal to the parameter *looplength* which is specified at compile time. It has to be noted that the number of the last set of elements can be different from *looplength* and it depends on the total number of elements present in the problem, which need not be a multiple of *looplength*. This parameter is set to be a multiple of hardware vector length on vector systems. This kind of grouped element processing also improves performance on scalar processors due to better caching of data. As the present scalar processors add more vector SIMD units, vectorization is absolutely necessary even for the future scalar architectures. The implementation can also handle a discretization of different types of element. All the elements that are grouped into a set are always of the same type.

Benchmark example: Beltrami flow

This flow example problem was developed by Ethier and Steinman for benchmarking purposes, although it is unlikely to be physically realized. The problem

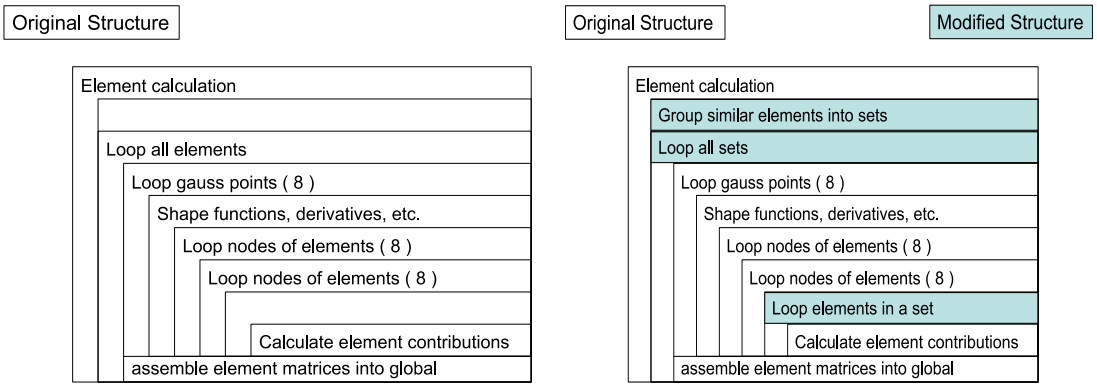

Figure 4.1: Original (left) and modified (right) flow of instructions in element integration.

is solved on a domain Ω = [-1,1] x [-1,1] x [-1,1]. The exact solutions for the velocity and the pressure read as follows.

$$u_1 = -a[e^{ax_1} sin(ax_2 \pm dx_3) + e^{ax_3} cos(ax_1 \pm dx_2)]e^{-\nu d^2 t} \tag{4.1}$$

$$u_2 = -a[e^{ax_2} sin(ax_3 \pm dx_1) + e^{ax_1} cos(ax_2 \pm dx_3)]e^{-\nu d^2 t} \tag{4.2}$$

$$u_3 = -a[e^{ax_3} sin(ax_1 \pm dx_2) + e^{ax_2} cos(ax_3 \pm dx_1)]e^{-\nu d^2 t} \tag{4.3}$$

$$\begin{aligned} p = -\frac{a^2}{2}[e^{2ax_1} + e^{2ax_2} + e^{2ax_3} + 2sin(ax_1 \pm dx_2)cos(ax_3 \pm dx_1)e^{a(x_2+x_3)} \\ +2sin(ax_2 \pm dx_3)cos(ax_1 \pm dx_2)e^{a(x_3+x_1)} \\ +2sin(ax_3 \pm dx_1)cos(ax_2 \pm dx_3)e^{a(x_1+x_2)}]e^{-2\nu d^2 t} \end{aligned} \tag{4.4}$$

where a and d are open parameters separating a family of solutions. They will be fixed according to Either and Steinman [28] with the values a=$\pi/4$ and d=$\pi/2$. The characteristic feature of this flow is a series of counter-rotating vortices intersecting one another at oblique angles. The time-dependent terms in equations 4.1-4.3 indicate the exponential decay in time of the initial flow configuration.

Uniform discretizations with 4x4x4, 8x8x8, 16x16x16 and 32x32x32 HEX8 elements are used. All the discretizations are run with the original and re-implemented versions of the 3D fluid element code in CCARAT. Fig. 4.2 shows the performance improvement with the modified element implementation for this benchmark problems on a typical vector system (NEC SX-6+) and on a typical

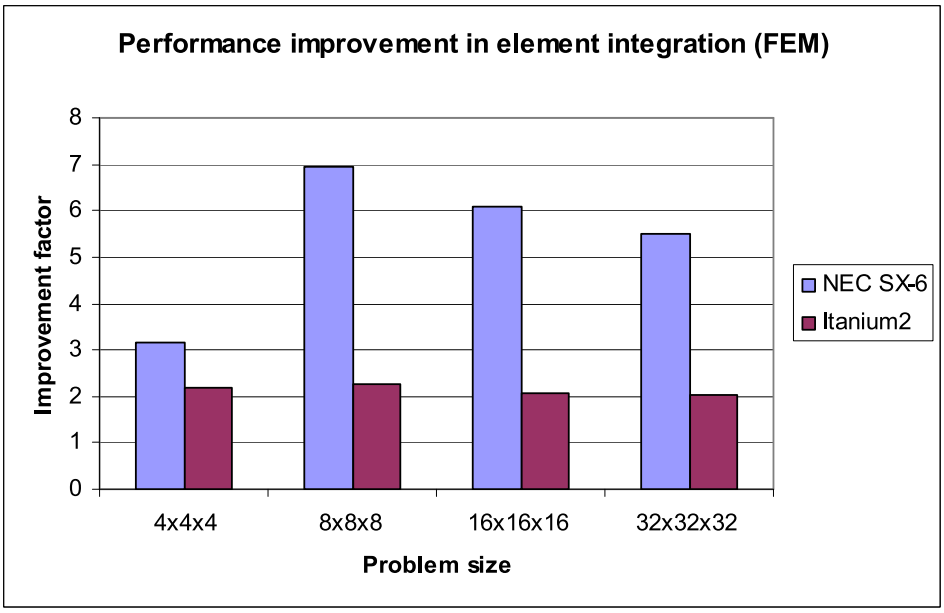

Figure 4.2: Performance improvement (factor) with the modified element implementation on NEC SX-6 and HP Itanium 2 systems.

scalar system (HP Itanium2). A performance improvement of about factor 5 is seen on the SX-6+ for large problems and on the Itanium 2 processor, the modified implementation is twice as fast than the original. For more details on the implementation and results see [61, 62, 83]. The *looplength* parameter is selected to be 512 (twice the hardware vector register size) on the NEC SX-6+ and to be 20 on the HP Itanium 2 during code compilation, which are the optimum values for the respective architectures. The value on the cache based systems (most commodity processors) depends on the cache sizes (the largest on-chip cache size being the most important) as well as the number of registers.

4.1.2 BLIS performance

This section provides the performance analysis of the newly developed solver, BLIS, using FEM simulations on both scalar and vector architectures. Firstly, scaling of BLIS on NEC SX-8 is presented for a laminar flow problem with different discretizations. Then, performance of a FSI example and a pure fluid examples are compared between two different hardware architectures. The machines used for the performance tests are a cluster of NEC SX-8 SMPs and a cluster of Intel 3.2 GHz Xeon EM64T processors as described in Sect. 1.3. The network interconnect available on NEC SX-8 is a proprietary multi-stage crossbar called IXS and Infiniband on the Xeon cluster. Vendor tuned MPI library is used on the SX-8 and Voltaire MPI library on the Xeon cluster.

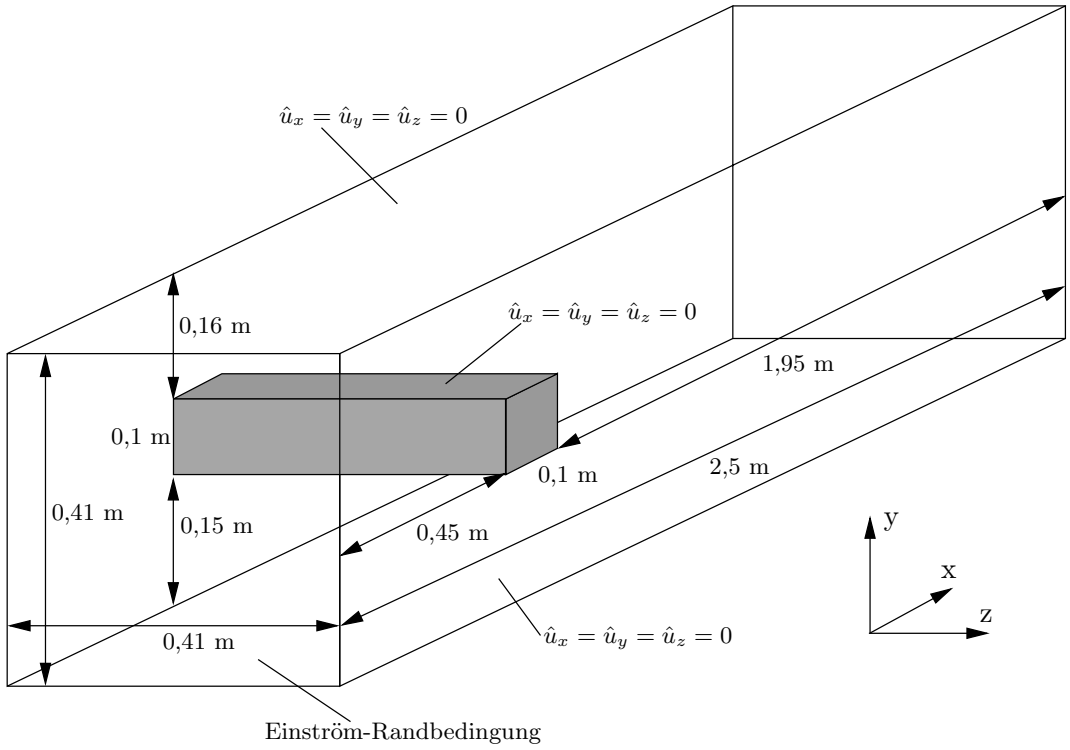

Figure 4.3: The geometry and boundary conditions of the flow around a cylinder with a square cross-section.

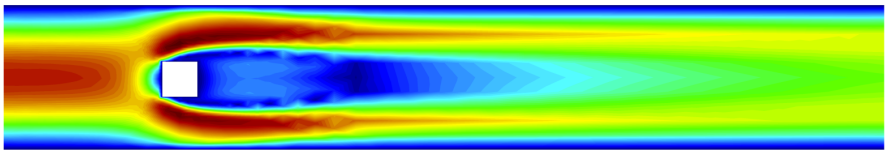

Figure 4.4: Velocity in the midplane of the channel.

Example used for scaling tests

In this example the laminar, unsteady 3-dimensional flow around a cylinder with a square cross-section is examined. The setup was introduced as a benchmark example by the DFG Priority Research Program "Flow Simulation on High Performance Computers" to compare different solution approaches of the Navier-Stokes equations [77]. The fluid is assumed to be incompressible Newtonian with a kinematic viscosity $\nu = 10^{-3}\ m^2/s$ and a density of $\rho = 1.0\ kg/m^3$. The rigid cylinder (cross-section: 0.1 m x 0.1m) is placed in a 2.5 m long channel with a square cross-section of 0.41 m by 0.41 m. On one side a parabolic inflow condition

Discretization	No.of elements	No. of nodes	No. of unknowns
1	33750	37760	151040
2	81200	88347	353388
3	157500	168584	674336
4	270000	285820	1143280
5	538612	563589	2254356
6	911250	946680	3786720

Table 4.1: Different discretizations of the introduced example.

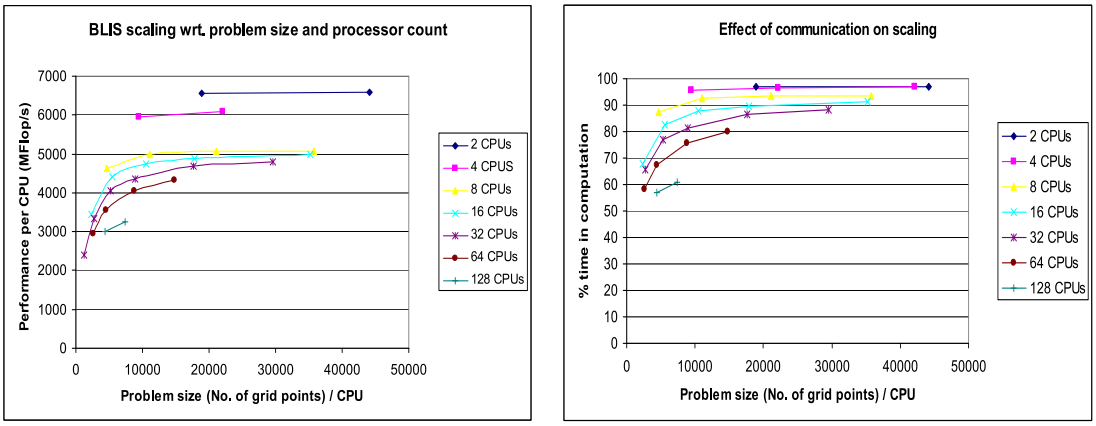

Figure 4.5: Scaling of BLIS wrt. problem size on NEC SX-8 (left) Computation to communication ratio in BLIS on NEC SX-8 (right).

with the mean velocity $u_m = 2.25\ m/s$ is applied. No-slip boundary conditions are assumed on the four sides of the channel and on the cylinder as shown in Fig. 4.3.

BLIS scaling on NEC SX-8

Scaling of the solver on NEC SX-8 was tested for the above mentioned numerical example using stabilized 3D hexahedral fluid elements implemented in CCARAT. Table 4.1 lists all the six discretizations of the example used.

Figure 4.5 plots weak scaling of BLIS for different processor counts. Each curve represents performance using particular number of CPUs with varying problem size. All problems were run for 5 time steps where each non-linear time step needs about 3-5 newton iterations for convergence. The number of iterations needed for convergence in BLIS for each newton step varies largely between 200-2000 depending on the problem size (number of equations). The plots show the drop in sustained floating point performance of BLIS from over 6 GFlop/s to 3 GFlop/s depending on the number of processors used for each problem size.

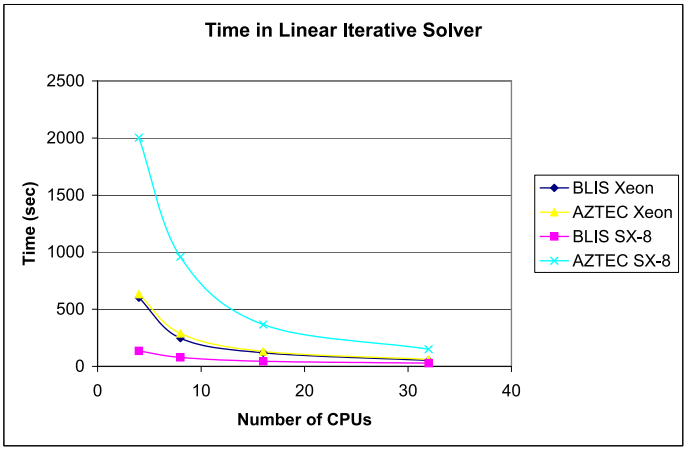

Figure 4.6: Elapsed time (seconds) in linear solver.

The right plot of Fig. 4.5 explains the reason for this drop in performance in terms of drop in computation to communication ratio in BLIS. It has to be noted that major part of the communication with the increase in processor count is spent in MPI global reduction calls which need global synchronization (as shown in Fig. 3.27). As the processor count increases, it takes a larger problem size to reach a saturated level of sustained performance. This behavior can be directly attributed to the time spent in communication that can be noted from the right plot. These plots are hence important as they accentuate the problem with Krylov subspace algorithms where large problem sizes are needed to sustain high performance on large processor counts. This is a drawback for certain class of applications where the demand for HPC (High Performance Computing) is due to the largely transient nature of the problem. For instance, even though the problem size is moderate in some Fluid-Structure interaction examples, thousands of time steps are necessary to simulate the transient effects.

Performance comparison on scalar and vector machines

Furthermore, strong scaling performance of BLIS and AZTEC on both Xeon and SX-8 systems is plotted in Fig. 4.6 using a moderate problem size (353K unknowns) of the example used for the scaling tests earlier. Elapsed time (seconds) in linear iterative solver for 5 time steps is listed on the y-axis of the plot in Fig. 4.6. Fastest solution method (BiCGSTAB) and preconditioning (ILU on the Xeon cluster and block Jacobi on NEC SX-8) were used on the respective machines.

BLIS is about 20% faster than AZTEC on the Xeon cluster due to block based approach which results in better cache usage. BLIS is about 10 times faster than AZTEC on the NEC SX-8 for lower processor counts when the problem size per processor provides sufficient average vector length. The solution of the linear

Machine	Solver	Precond.	Total CG iters.	MFlop/s per CPU	CPU time
SX-8	BLIS3,4	BJAC	597	6005	**66**
	AZTEC	ILU	507	609	564
Xeon	BLIS3,4	BILU	652	-	294
	AZTEC	ILU	518	-	**346**

Table 4.2: Performance comparison in solver between SX-8 and Xeon for a fluid structure interaction example with 25168 fluid equations and 26352 structural equations.

system is about 4-5 times faster on the NEC SX-8 than on the Xeon cluster for lower processor counts. As the number of processors increases, the performance of BLIS on NEC SX-8 drops due to decrease in computation to communication ratio and also because of a drop in average vector length. Contrarily, this is an advantage (better cache utilization) on the Xeon cluster due to a smaller problem size on each processor. This can be noticed from the super linear speedup on the Xeon cluster.

Then, performance of a fluid structure interaction (FSI) example is compared on these two architectures with AZTEC and BLIS. In the example tested, the structural field is discretized using BRICK elements. So, matrix block size 3 is used for structural field and block size 4 functionality in BLIS is used to solve the fluid field. The problem was run for 1 FSI time step which needed 21 solver calls (linearization steps). It can be noted from the results in Table 4.2 that the number of iterations needed for convergence in the solver vary between different preconditioners and also between different architectures for the same preconditioner. The reason for this variation between architectures is due to the difference in partitioning. Also the preconditioning in BLIS and AZTEC cannot be exactly compared as BLIS operates on blocks which normally result in superior preconditioning than point-based algorithms.

Even with all the above mentioned differences, the comparison is done on the basis of time to solution for the same problem on different systems and using different algorithms. The time to solution for 1 FSI time step using BLIS is clearly much better on the SX-8 when compared to AZTEC. This is the main reason for developing a new general vectorized sparse solver. It is also interesting to note that the time for solving the linear systems (which is the most time consuming part of any unstructured implicit finite element or finite volume simulation) is clearly more than a factor 5 faster on the SX-8 when compared to the Xeon cluster.

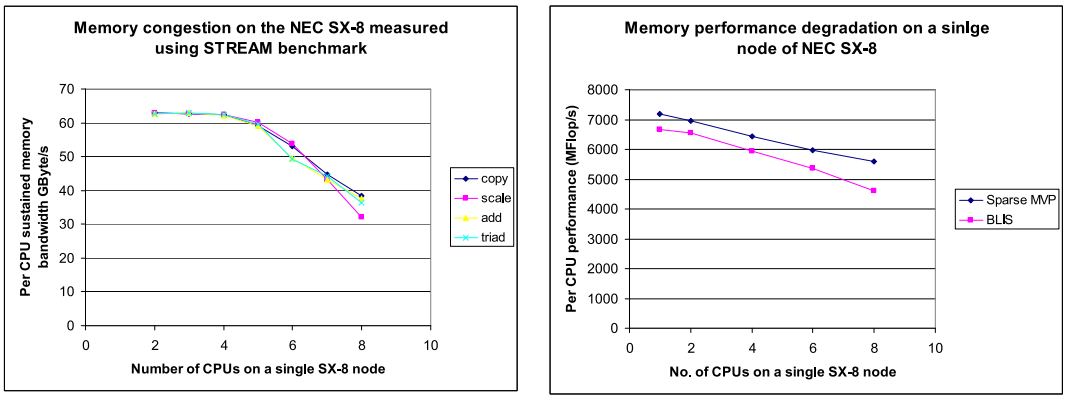

Figure 4.7: Memory degradation on the NEC SX-8 with direct (STREAM benchmark - left) and indirect memory addressing (right).

Memory bandwidth congestion on the NEC SX-8

All shared memory machines show degradation in memory performance per core as more and more cores try to access the memory with in the node. In case of hierarchical memory this effect can be reduced due to the presence of caches that act like local buffers. As long as the problem size is small so that it fits into the caches, shared memory scalar systems usually show lower degradation in memory intense kernels with the increase in the active cores/node.

For floating point bound kernels, this effect may not be visible if the FP pipelines can be fed with data even with the degraded memory performance. This may result in a perfect scaling for the kernel on the whole SMP node even though there may be actually some memory degradation present. The best way to measure the degradation is to use memory bound kernels like the sparse MVP or the FFTs. Embarrassingly parallel STREAM benchmark, which is part of the HPC Challenge Benchmark suite, is also widely used for this purpose.

Figure 4.7 shows the degradation in memory performance on the NEC SX-8. The left part of the figure shows the degradation on a single node for all the four components of the STREAM benchmark accessing real arrays with at least 300 million elements to/from the main memory. The right part of the figure shows the degradation in memory performance for the block sparse MVP kernel with block size four. As a result of this, the performance in the whole solver is also effected as sparse MVP is usually the most time consuming portion of a linear iterative solver. It is important to note that for direct memory access, a degradation in memory performance (theoretical peak memory bandwidth per CPU on NEC SX-8 is 64 GByte/s) is noticeable if more than five CPUs are used on a single node. On the other hand, the memory degradation in case of indirect memory access (sparse MVP) is nearly linear starting from two CPUs. The reason for this is the irregular access to the memory banks which happens in

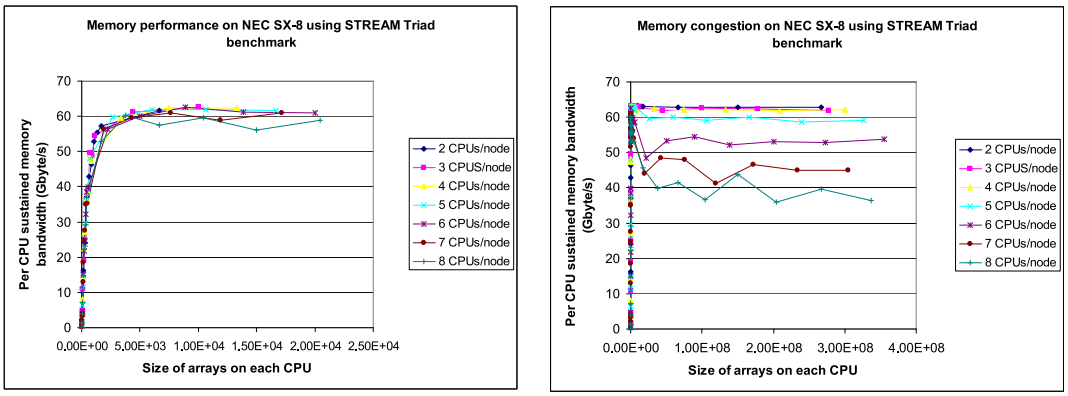

Figure 4.8: Memory performance on a single node NEC SX-8 with STREAM Triad for small problem sizes (left) and large problem sizes (right).

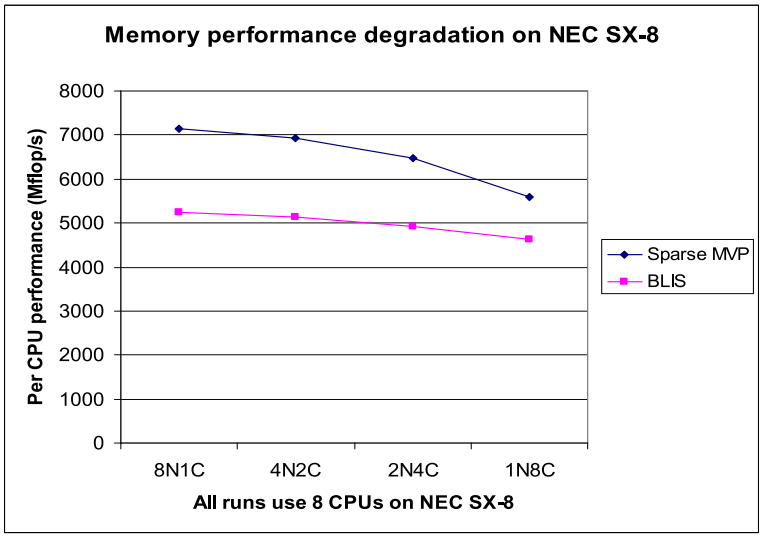

Figure 4.9: Performance degradation in BLIS with increasing number of CPUs per node.

case of indirect memory accesses. These effects can be amplified in case of larger vector nodes like the NEC SX-9 system which will have 16 CPUs/node. This is also a concern on scalar systems with large nodes, but these effects are reduced by hierarchical memory design and cache blocking algorithms.

Figure 4.8 plots the STREAM Triad benchmark run on different CPUs/node with increasing array size. The left plot shows the results for small problem sizes and the right plot for larger problem sizes. Upto an array size of 20K, the sustained memory bandwidth is about 60 GByte/s even when all the eighth CPUs/node are used on the NEC SX-8. Performance degradation becomes no-

ticeable when more than five CPUs are used for array sizes of over 10M. Normally, most user applications that are vectorized have large vectors and this memory degradation becomes relevant.

Lastly, in Fig. 4.9, the effect of this memory performance degradation on a memory bound kernel such as the sparse MVP in BLIS (block size 4) is highlighted. In order to show the importance of memory performance for libraries like BLIS, the solver is run on different number of shared memory nodes on the SX-8 while keeping the number of CPUs constant at eight. On the x-axis of the plot, various combinations of nodes and CPUs accounting to eight CPUs/node are listed. 8N1C stands for 8 nodes and 1 CPU/node. So, this run should have no memory degradation due to congestion but the MPI communication is always outside of the node (over the network) and hence should be slower than when the MPI communication is partly or wholly with in the node. The run 1N8C stands for 1 node and 8 CPUs/node which will have the maximum memory degradation but on the other hand the best MPI performance as all communication is within the node. It is apparent from the plot that the memory performance is far more crucial to the performance of the whole solver than the MPI communication performance.

4.1.3 Simulations using BLIS

CFD application: flow over a rigid membrane roof

In this numerical example a simplified 3-dimensional representation of a cubic building with a flat rigid membrane roof is studied (Fig. 4.10). The building is situated in a horizontal flow with an initially exponential profile defined by

$$\hat{u} = 200\ m/s \times (\frac{z}{350})^{0.22} \times (sin(\pi(\frac{t}{5.0} - 0.5)) + 1) \times 0.5$$

where z is the vertical spatial direction and t represents time. The maximum velocity reached in the flow is 151.8 m/s. The fluid is Newtonian with dynamic viscosity $\nu_F = 0.1\ Ns/m^2$ and density $\rho_F = 1.25\ kg/m^3$.

The fluid domain is discretized by 247,040 GLS-stabilized Q1Q1 elements. This discretization results in $1,039,140$ degrees of freedom for the complete system.

Performance and results

First, the problem is run for 5 time steps (about 2-3 linear iterations per time step) with different solvers (Aztec and BLIS with block size 4) using different methods (GMRES and BiCGSTAB) and different preconditioners (Block Jacobi, Block ILU, ILU) on a single node of SX-8 (8 CPUs) in order to find out the fastest method for this problem. The time to solution of these test runs is listed

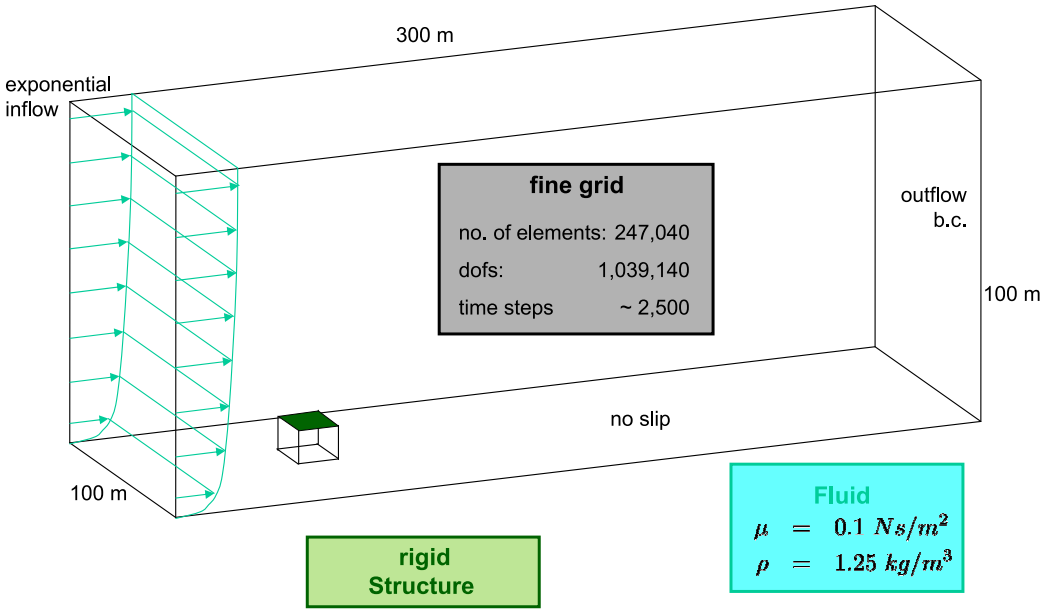

Figure 4.10: Rigid membrane roof: geometry and material parameters

Solver-Method-Preconditioner	Wall Time (sec)	MFlop/s per CPU	Vector operation ratio	Average vector length
AZTEC-GMRES-ILU	605	367	77.06	130.8
AZTEC-BiCGSTAB-ILU	489	184	67.46	126.0
BLIS-GMRES-BILU	290	1024	91.47	65.9
BLIS-GMRES-BJAC	75	3489	97.76	241.7
BLIS-BiCGSTAB-BILU	103	884	90.83	64.4
BLIS-BiCGSTAB-BJAC	24	3471	98.12	229.7

Table 4.3: Performance in linear solvers for the membrane roof example.

in Table 4.3. BiCGSTAB method using Block Jacobi preconditioning in BLIS is the best method in terms of time to solution. This is more than a factor 20 faster than the fastest method with AZTEC solver on the NEC SX-8.

The best method (BiCGSTAB + BJAC in BLIS) was used to run the full calculation for 2,500 time steps with $\Delta t = 0.01\ s$, resulting in a simulation time of $\sim 25\ s$. For each time step, 2-3 linear iterations (i.e. solver calls) are necessary to solve the non-linear flow equations. The wall time needed on 4 nodes of SX-8 (32 CPUs) for the whole simulation is about 8.22 hours which adds to about 263 hours of CPU time. The total time in the solver is about 33% of the whole simulation time due to the efficient implementation in BLIS. The sustained per CPU floating point performance was about 2.6 GFlop/s in the solver (vector operation ratio is 96.20% and average vector length is 206.8) and about 1.75

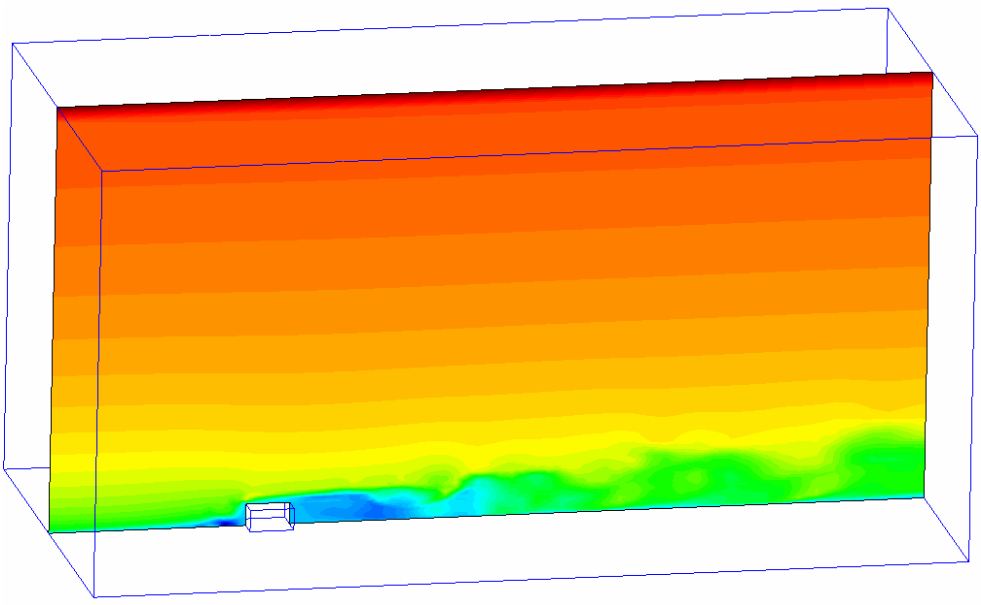

Figure 4.11: Sectional view of a x-velocity field over the rigid roof

GFlop/s for the whole simulation. Higher performance can be obtained with larger problem sizes per CPU (i.e when running the same problem on a single node SX-8 as in the test phase) but the demand for high performance computing for these class of problems is due to the extremely large number of time steps.

The results (velocity in x-direction) for the rigid roof for $t = 10.0\ s$ are visualized in figure 4.11. In the future, the structure would be modeled using 3-d BRICK elements and a fluid structure interaction simulation is planned.

Structural mechanics application: Micromechanical simulation of the cancellous bone structure

The presented micromechanical Finite-Element model was set up for the purpose of calculating orthotropic material constants for cancellous bone on the scale of clinical imaging data with a method introduced in [91].

The simulated test object was a 10mm x 10mm x 10mm bone spcimen from the distal region of a primed human femur. From this specimen a Micro-Computer-tomography was taken at an isotropic spatial resolution of 0.014mm resulting in a volume data set of 817x865x504 Voxels. Out of this data set, a subset of 150x150x150 Voxels was selected and an iso-surface was calculated to segment the porous structure of the cancellous bone.

The iso-surface calculation resulted in a triangulated surface which was further reduced and restructured with the aim of gaining a mesh of high quality triangles suitable for a volume meshing algorithm.

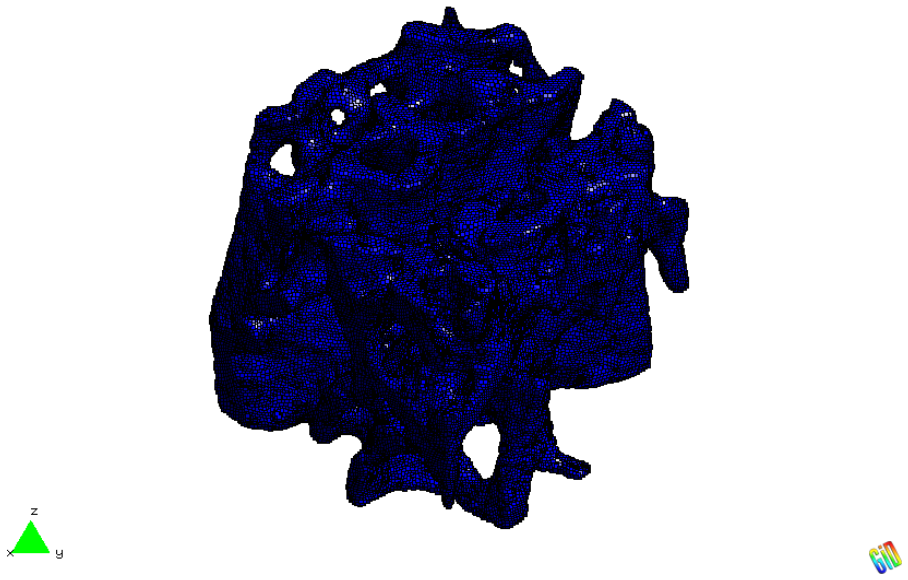

Figure 4.12: Discretized bone geometry.

At the end of the model setup the reconstructed part of the specimen had a physical dimension of 2.81mm x 2.81mm x 2.81mm and the volume mesh consisted out of 1,333,017 linear tetrahedrons connected through 282,653 nodes. The whole model setup resulted in a total of 847,959 degrees of freedom.

As boundary conditions, the displacements on five sides of the cube were set to zero in the normal direction to the cube's designated side. On the sixth side a compressive load of 300 N/mm^2 was applied. The Youngs Modulus was chosen to be 14000 MPa and Poisson Ratio to be 0.3 [91].

Performance and results

This linear static simulation is run using CG method on 8 CPUs of NEC SX-8 with different preconditioners in Aztec and BLIS with block size 3. The results are listed in Table 4.4. The time to solution for one solver call along with other performance metrics on the SX-8 can be found in the table. The results are normalized with respect to the AZTEC solver. The fastest method using BLIS is about a factor of 26 faster than the AZTEC solver. This is mainly due to longer average vector lengths and better vector operation ratio. The sustained performance in BLIS using CG with block Jacobi preconditioning is over 4 GFlop/s per CPU on the NEC SX-8.

The displacement results as a contour fill are shown in Fig. 4.13 and as contour lines are shown in Fig. 4.14. Finer discretizations of the same problem will be run in the future.

Solver-Method-Preconditioner	Improvement factor	Wall Time (sec)	MFlop/s per CPU	Vector operation ratio	Average vector length
AZTEC-CG-ILU	1.0	1234	573	90.19	88.6
BLIS-CG-BILU	2.11	584	869	95.03	61.4
BLIS-CG-BJAC	26.25	47	4123	99.52	248

Table 4.4: Performance in linear solvers for the bone simulation.

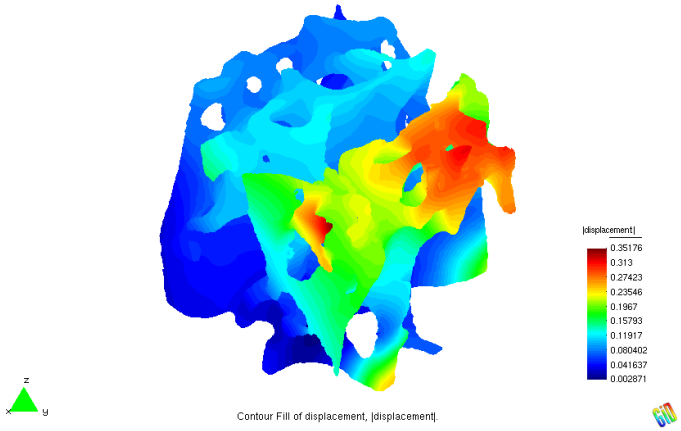

Figure 4.13: Mean Displacement contour fill for a 300 N/mm^2 compressive loading.

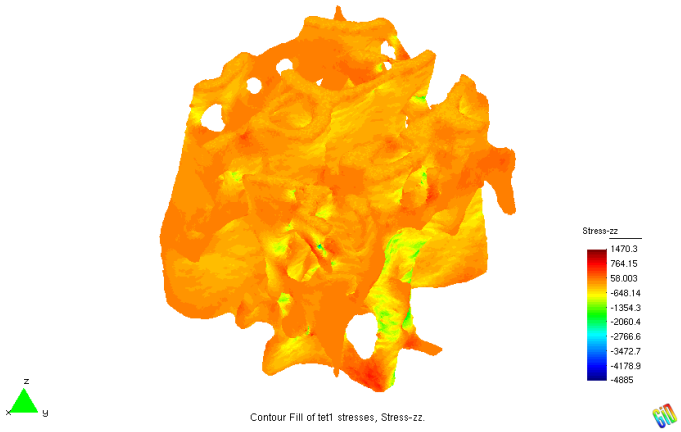

Figure 4.14: Stresses in Z direction for a 300 N/mm^2 compressive loading.

Chapter 5

Innovative hardware architectures and challenges

This chapter is dedicated to the new developments in Hardware architectures and the implications for sparse linear iterative solvers. This is particularly interesting as the sparse linear solvers need a high memory performance (both bandwidth and latency). The present disturbing trend in hardware development is to add more cores (arithmetic units) to the chip without a equivalent improvement in the memory bandwidth and latency. This trend is encouraged by some commercial applications like video games and also the fact that increasing memory bandwidth is much more expensive than increasing the FP performance.

Even some scientific applications that depend mostly on kernels that block data in cache (like for e.g. DGEMM) may not need a very high bandwidth to the main memory. This is in fact the main reason for the development of the hierarchical approach towards memory on the chip, i.e. caches. This is very helpful for those kernels that have high spatial and/or temporal locality. The latency to the main memory which is between one to two orders of magnitude higher than to the first level cache are hidden by loading the data into cache and then operating repeatedly on the cached data. The only problem with this approach is with kernels that do not have locality in the algorithm such as the sparse linear solver. The dense linear algebra kernels can be blocked and hence the usage of BLAS2 and BLAS3 calls is possible resulting in much higher locality in the data. Blocking sparse code is not straight forward and hence generally not implemented in many sparse solver libraries.

Another interesting trend is the increasing availability of hardware accelerators such as Clearspeed, MDGRAPE, etc. But each such architecture addresses only a subset of the problems and is not a general purpose solution. Also, new hardware designs such as the STI CELL BE, GPGPU, etc. are increasingly been used for scientific computing. This needs a lot of programming effort in order to fully use the capabilities on these hardware architectures. The usage of programming language extensions for this purpose is common and this makes the extra

programming effort not portable. As this trend grows, there may be standardization of such extensions. The main concern with these new designs is the memory performance. Though new ways of accessing the memory have been designed, it still does not solve the problem of memory and FP performance parity.

5.1 Memory bandwidth challenge

Even with these new approaches to computing, one clear trend is the reduction of sustained memory bandwidth. This is the central theme for memory bound applications such as the implicit FEM simulations that need a sparse linear iterative solver. As the increase in peak memory performance is much lower than the increase in the peak FP performance, the sustained performance of the memory bound applications is growing not with the peak FP growth but with the memory performance growth. This issue was well studied in the paper [101].

5.2 Multi-core performance with BLIS

The potential of multi-core architectures has been investigated for memory intensive applications. Both the dual-socket quad-core Intel Xeon and quad-socket dual-core AMD Opteron with HyperTransport network have been compared with a single node NEC SX-8 for scalability. The 2.13 GHz Intel Xeon has 1MB L2 cache per core and the 2.61 GHz AMD Opteron has a (slightly less) 512KB L2 cache per core.

Figures 5.1 to 5.12 plot the scaling of sparse linear solver on the above mentioned multicore systems and on a single node NEC SX-8. Both row-oriented (BCRS) and pseudo diagonal-oriented (BJAD) block storage formats are used for this purpose. The time mentioned in the figures is for one preconditioned BiCGSTAB iteration with Block Jacobi (BJAC) or Block ILU (BILU) as a preconditioner. The scaling of the problem on to the whole node (8 cores in case of Opteron and Xeon and 8 CPUs in case of SX-8) is plotted on the right side of each figure. Plotting only the scaling data without the CPU time or FP performance would not give a complete picture of the relative performance between the systems. The reason for testing with so many different combinations is due to the variety of the architectures. On scalar systems such as the Opteron and Xeon, the performance of BILU preconditioner is usually better with respect to time to solution. This is not the case on SX-8 as detailed in Sect. 3.4. With the sparse storage formats, BJAD is preferred on the SX-8. The reason to use different storage formats is that BCRS is sometimes slightly better on the scalar systems when compared to BJAD.

As can be noted from the plots, the scaling on Opteron and SX-8 are much better than on the Xeon node. The main reason for Opteron to scale is because

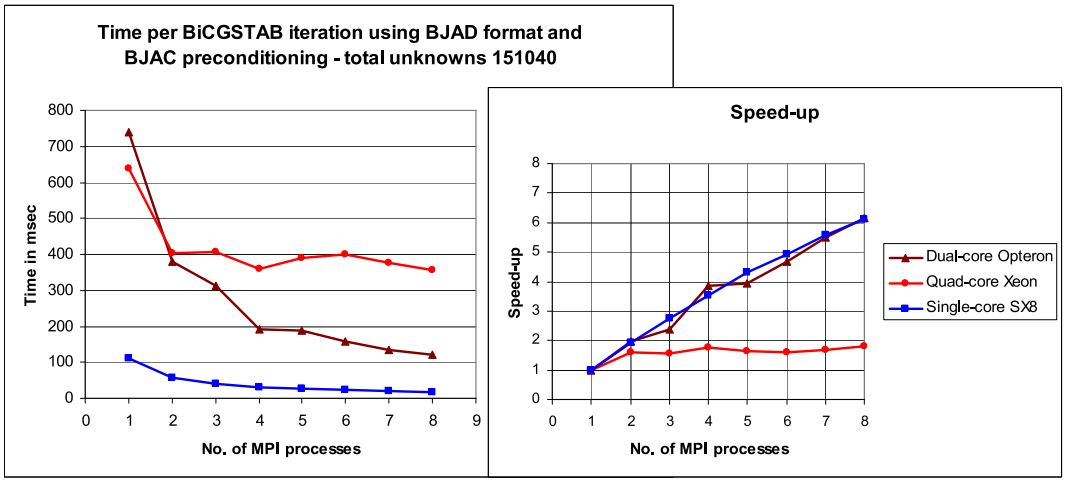

Figure 5.1: Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 151k equations.

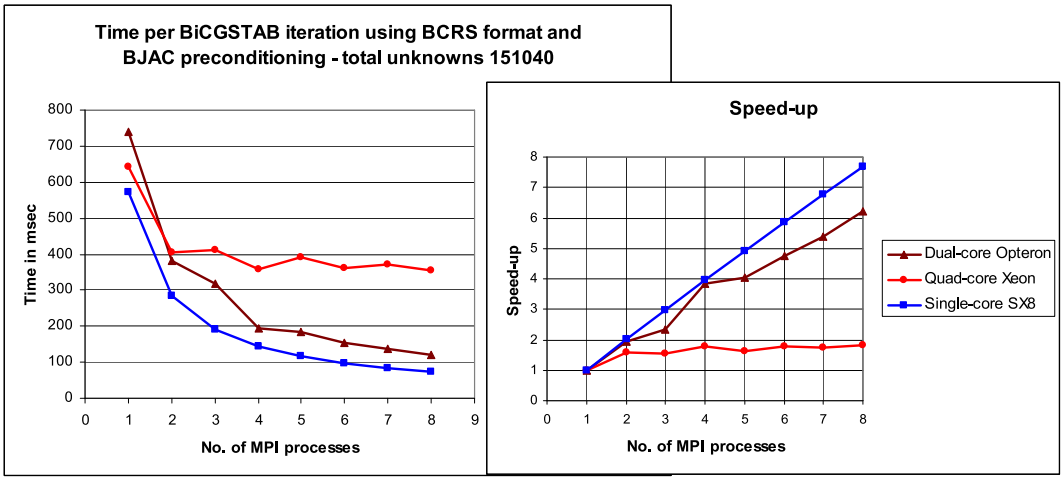

Figure 5.2: Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 151k equations.

the memory bandwidth scales with the number of sockets, which are connected using the HyperTransport interconnect within the node. The bottleneck on the Xeon node is the single Front Side Memory Bus (FSB) which supplies data to two quad core sockets. It can be concluded from the data that one core per socket saturates the available memory bandwidth on the Xeon node and limits the scaling to two with larger problems. This is mainly due to the lack of a scalable network on the Xeon node that is similar to HyperTransport interconnect on the Opteron node. Only in case of very small problems that fit within the 1 MB L2 cache per core, the performance scales beyond two on the Xeon node. With the

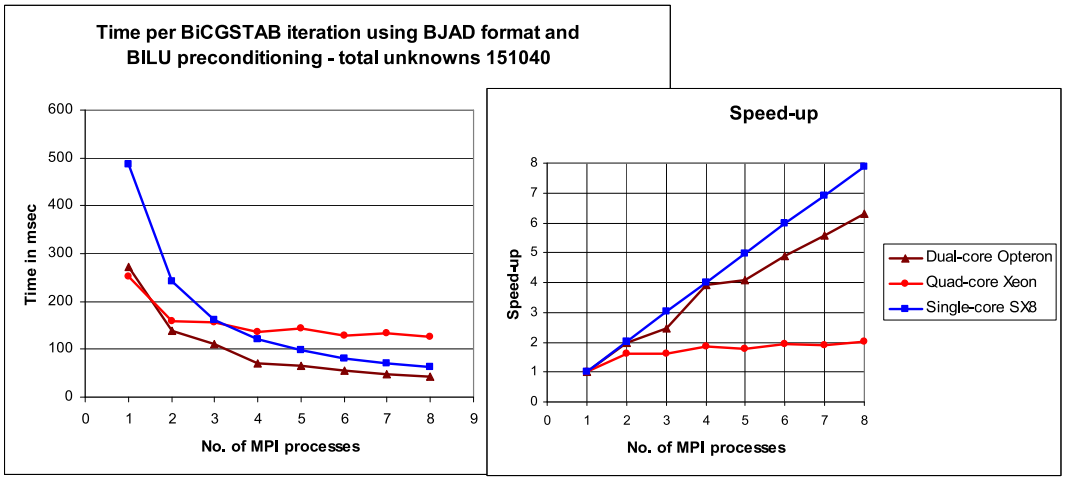

Figure 5.3: Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 151k equations.

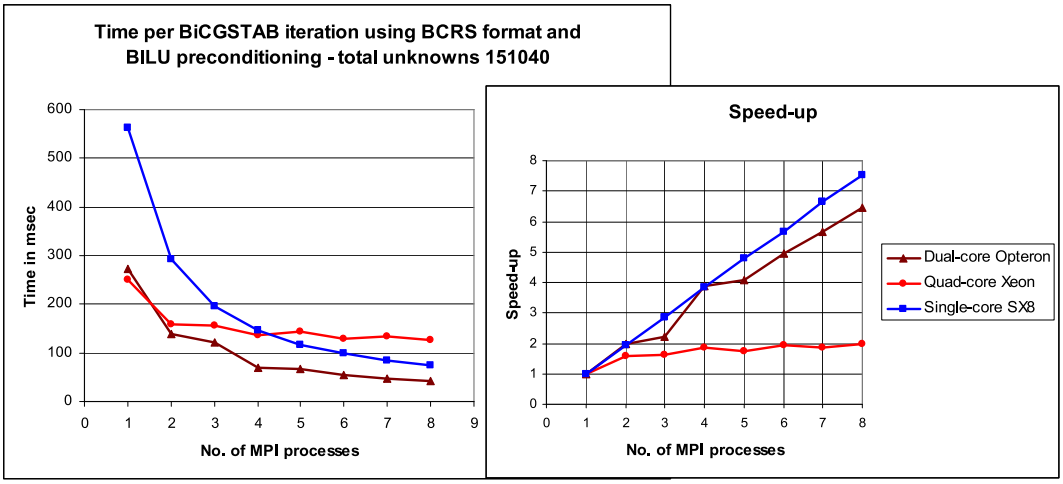

Figure 5.4: Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 151k equations.

smallest problem size (8K unknowns) measured, a speed up of more than two can be seen on the Xeon node in Figs. 5.9 to 5.12. This is mainly due to better cache usage. But small problem sizes result in a small average vector length and effect scaling on the SX-8. Another characteristic that can be noted from the plots is the oscillation in performance between odd and even number of cores usage on the scalar nodes. This is due to disproportionate usage of different components of the interconnect/memory subsystem. It can also be noticed that the scaling does not vary on the scalar multicore systems for different storage formats or preconditioners. On the SX-8 though, a small degradation in scaling can be seen

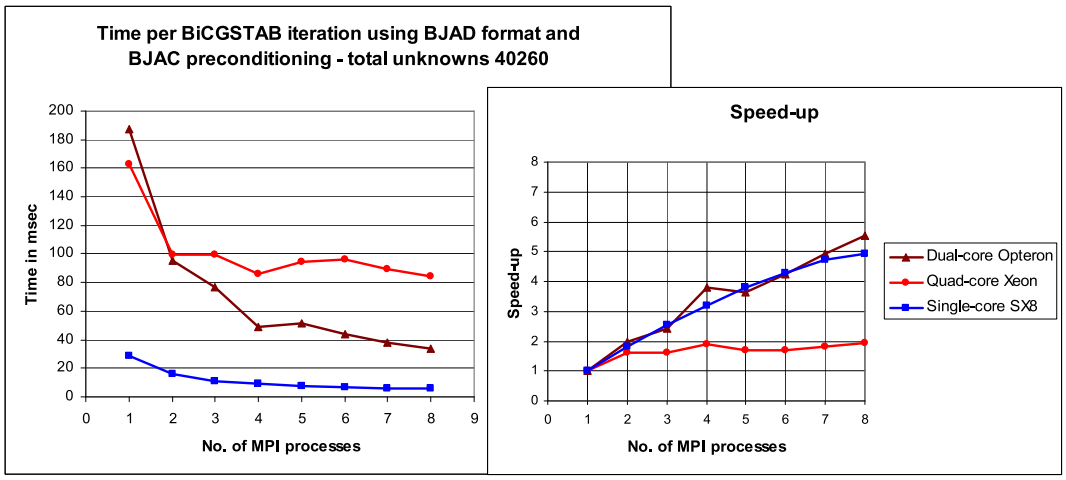

Figure 5.5: Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 40k equations.

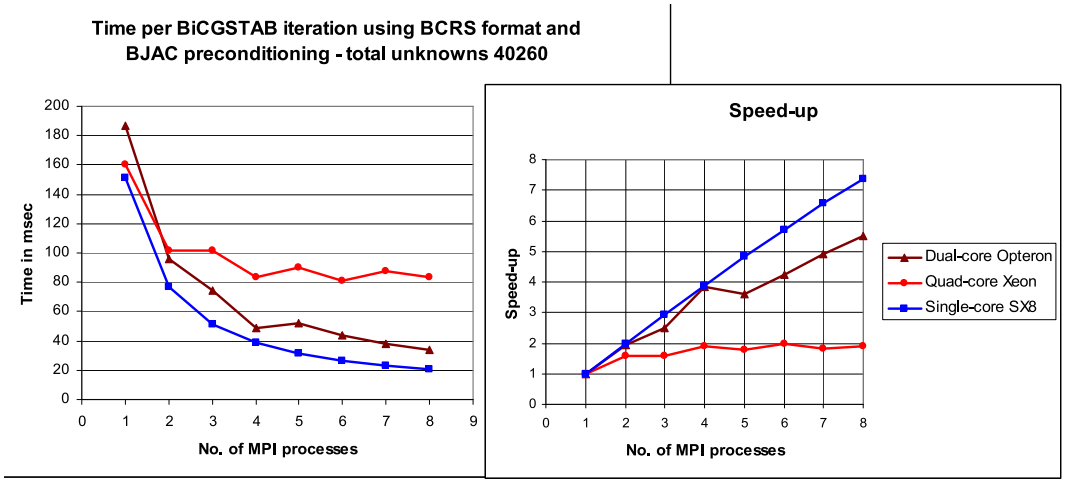

Figure 5.6: Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 40k equations.

with BJAD format and BJAC preconditioning due to the reasons elaborated in Sect. 4.1.2. This degradation is absent with BCRS format as the bottleneck then is not sustained memory bandwidth but average vector length.

The importance of a scalable interconnect such as HyperTransport on the Opteron node is evident in the plots. Though the single core performance on Xeon is superior to Opteron because of larger caches, Opteron overtakes Xeon in performance on the whole node (8 cores) because of scalable intranode interconnect. This behavior would be typical for all the memory bound applications and underlines the importance of the scalable memory bandwidth via intranode

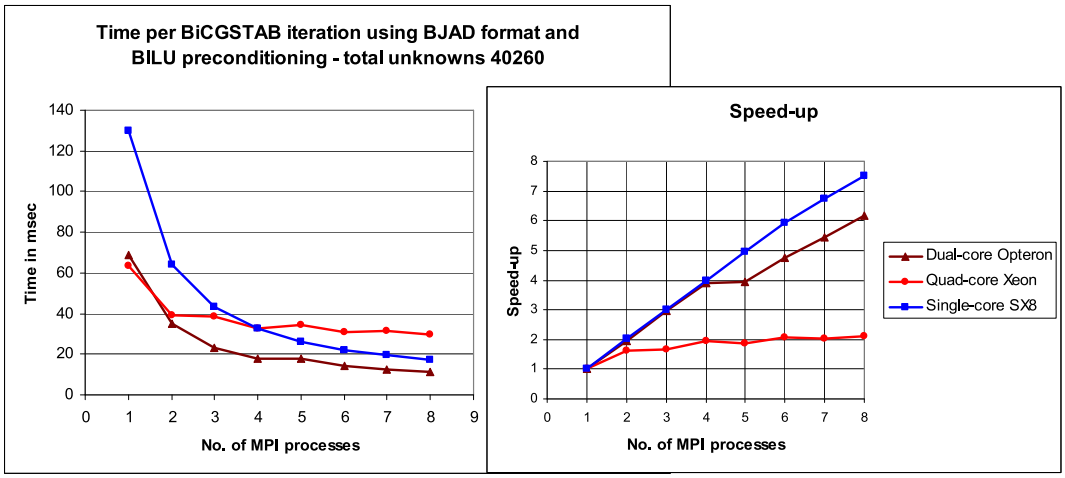

Figure 5.7: Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 40k equations.

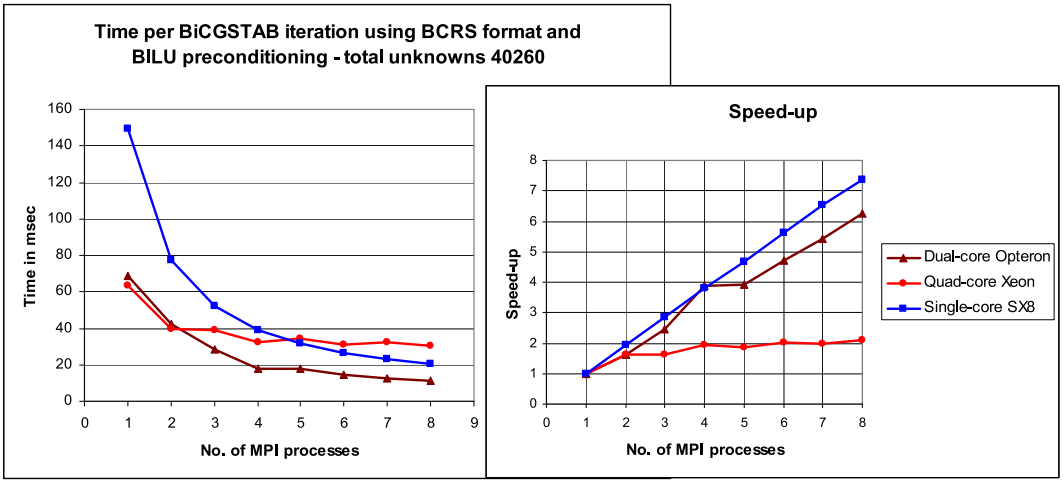

Figure 5.8: Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 40k equations.

network over the peak FP rating of the node.

5.3 STI CELL BE performance with BLIS

The CELL processor was jointly developed by Sony, Toshiba and IBM (STI). It is a novel design mainly targeting the graphics intensive applications and is now being extended to computationally intensive applications. In order to target scientific applications, the speed of double precision arithmetic is being tremen-

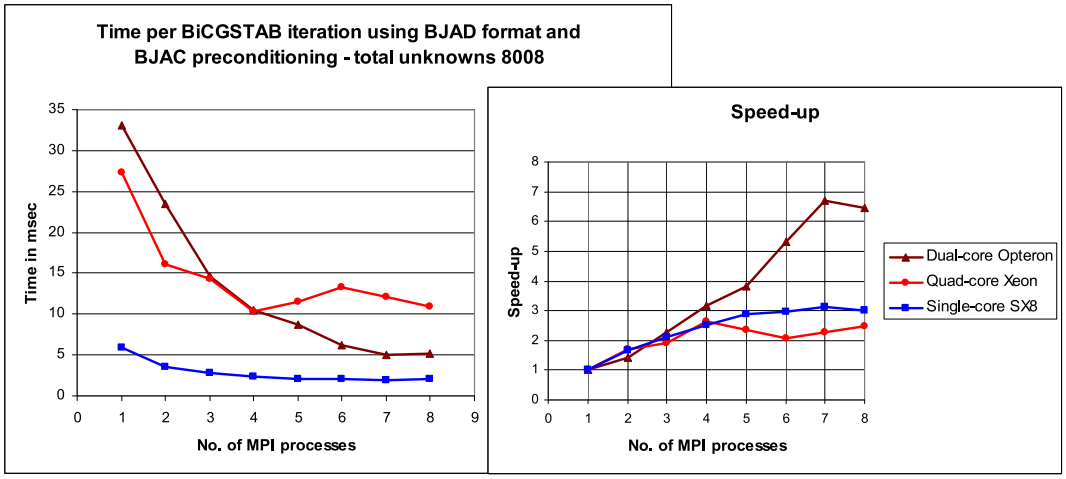

Figure 5.9: Time per BiCGSTAB iteration with BJAC preconditioning using BJAD format on multicore/SMP systems for solving 8k equations.

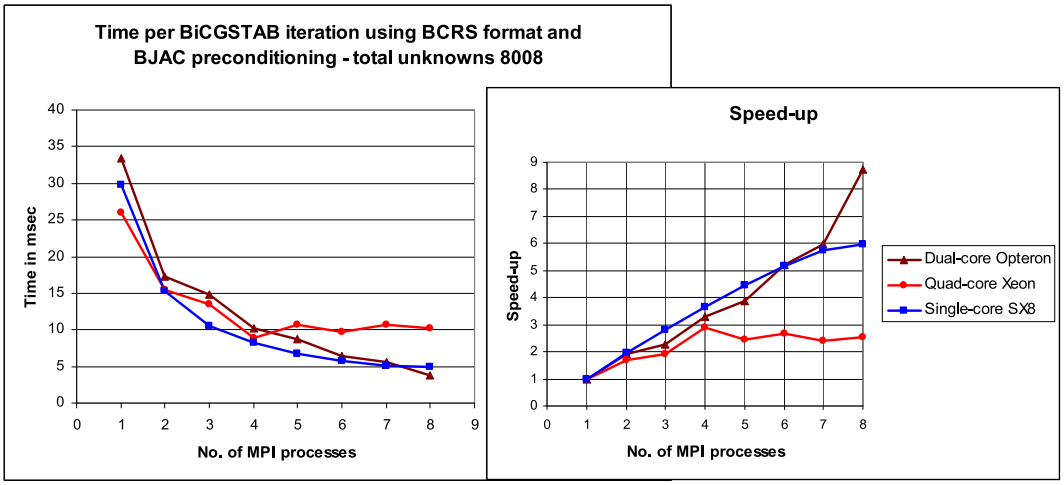

Figure 5.10: Time per BiCGSTAB iteration with BJAC preconditioning using BCRS format on multicore/SMP systems for solving 8k equations.

dously improved in the newer designs of the chip narrowing the gap between the peak performance of single precision and double precision arithmetic on the SPU cores.

An attempt was made to test the capability of the CELL processor with blocked sparse MVP kernel as available in BLIS. This test kernel had a block size of 4x4 with one million rows and 100 non-zeros per row. The tests were run on the HLRS installation of the STI CELL cluster as described in Sect. 1.3.4. VMX instructions were also implemented to compare the performance to the scalar version. Table 5.1 shows the time in the sparse MVP on the PowerPC processor

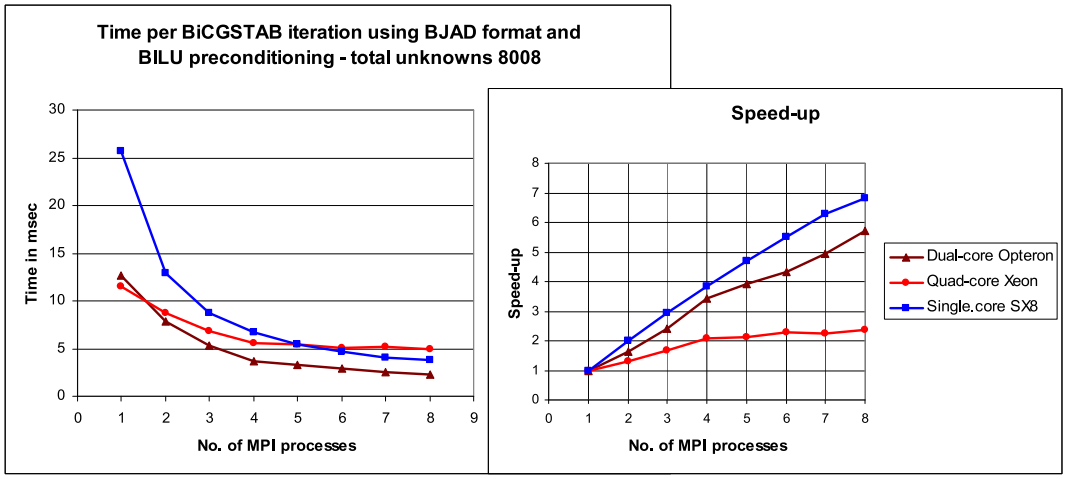

Figure 5.11: Time per BiCGSTAB iteration with BILU preconditioning using BJAD format on multicore/SMP systems for solving 8k equations.

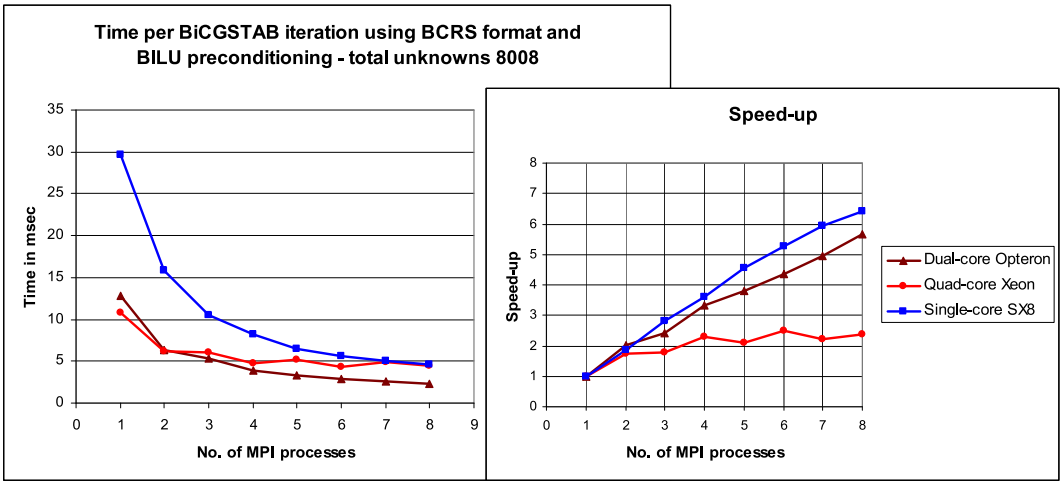

Figure 5.12: Time per BiCGSTAB iteration with BILU preconditioning using BCRS format on multicore/SMP systems for solving 8k equations.

for both the scalar implementation and using VMX instructions.

The peak single precision performance on all the 8 SPUs together is 204 GFlop/s and the bandwidth to memory is 25.6 GByte/s. Time in SPU implementation is also shown in the table which is about 5 times faster than the scalar implementation on the PowerPC in single precision calculations at the moment. About 95% of the time in the SPU variant is spent in the memory transfer from from the main memory to the local store using list DMA transfers. This can be further improved by using double buffering. Other studies also show a limited success of this architecture on sparse kernels [100]. It is very clear that the

Instructions	**Time in Sparse MVP (sec)**	**Speed-up**
Scalar	2.7	1
VMX	1.9	1.42
PPU-SPU(single)	0.53	5

Table 5.1: Performance comparison for scalar, VMX and SPU implementations on the STI CELL BE

bottleneck on this system for sparse kernels is the bandwidth to main memory, which is only 25.6 GByte/s for a very high FP rating. Hence, this architecture is not really suitable for sparse codes which need a good memory bandwidth. The main problem to using this system is the huge programming effort needed to run application as the memory transfers between the main memory on the PPU and local stores on the SPUs have to be explicitly programmed by the users. The advantage of this approach is the flexibility to overlay data access and computation in an optimal way, which otherwise is normally done by the compiler. Another important disadvantage is that the programs developed for the CELL processor are not portable onto other architectures.

Chapter 6

Conclusions and future work

The result of this research is an efficient implementation of a vectorized sparse linear solver called BLIS (Block-based Linear Iterative Solver). The key problems with most public domain solvers which leads to their dismal performance on the vector systems was highlighted in this work. The main reason being the design considerations of most software projects in general and linear solvers in particular in the last few decades. The mainstream high performance computing hardware in this time period saw a transition from the classical vector systems such as Cray 1 to the purely scalar architectures. Lately, there is again some interest in vector processing due to their power efficiency and suitability for some classes of applications, such as processing graphics information. Most modern scalar processors have a significant vector co-processor on the chip. Most compilers also provide some programming language extensions in order to effectively use the vector co-processor. As the available die space increases according to Moore's law, such vector units are expected to grow in both size and functionality. The main limitation on most modern scalar processors is a single-precision only vector co-processor and this is expected to change soon in the future.

Performance comparison of contemporary high performance computing systems using HPCC and IMB benchmarks shows that the vector systems are superior to most of the present scalar systems in memory intensive benchmarks as well as network benchmarks. Figure 2.2 shows that the balance of computation to network bandwidth and computation to memory bandwidth are both superior on vector systems than the scalar systems. Also, the absolute network bandwidth is much higher for the vector systems. This can be seen from the results of the point-to-point as well as collective Intel MPI Benchmarks. Vector systems for certain benchmarks were even over an order of magnitude better than the scalar systems.

The sparse solver (BLIS) developed during this work was specially targeted for classical vector systems like the NEC SX or the Cray X series. The main challenges of memory latency and bandwidth were addressed via blocking the sparse matrices. Special storage format was used in order to improve the average

vector length in the sparse MVP kernel. Coloring was used both to vectorize certain algorithms as well as to improve MPI communication performance. Finally, block-based Algebraic Multigrid was implemented as a preconditioner for Krylov methods in the solver. All these considerations make the solver particularly suitable for vector systems.

The difference in performance between a public domain solver (AZTEC) and BLIS on the NEC SX-8 was compared using some FEM examples. The scaling of BLIS was also studied on the NEC SX-8 using various discretizations of a flow problem. The fastest method in BLIS is between 10 - 20 times faster than the best method with AZTEC for the tested problems on the NEC SX-8. Moderate performance improvement was also measured on the scalar systems due to better cache locality. Lastly, the suitability of new hardware architectures for memory bound kernels such as in sparse solvers was studied in this work. Results show the memory bandwidth limitation on both the general purpose and special purpose (graphics processing) multicore architectures for such kernels.

Future work

The present solver package can be extended to include even more block sizes. To generate code on the fly for different block sizes as done in SPARSITY [47] is a reasonable approach. But this needs to be done in a way to take care of all the issues on vector systems as discussed in Sections 3.2 and 3.3. Due to the vectorization approach and coloring functionality already available in BLIS, it can easily be extended to run efficiently on the vector co-processors of the commodity scalar architectures. Implementing SIMD instructions should be straight forward due to the block-based approach used in the solver. The block-based approach additionally reduces the amount of indirect addressing needed and also improves locality while accessing sparse data. This is absolutely crucial for both the future scalar and vector architectures where memory latency and bandwidth improvement is not on par with the improvement in the FP performance [101]. Hence, any effort to reduce or improve memory access in the kernels would be important for all future hardware architectures.

Another important area of potential research is in reducing the complexity of solving sparse linear systems. Non-stationary iterative methods (Krylov subspace methods) need MPI_Allreduce in each iteration, which make these methods inefficient on very large processor counts (as detailed in Sect. 3.7). Using robust preconditioners with Krylov methods to limit the iterations needed for convergence is absolutely necessary when using large processor counts. Multigrid methods are also very important when running large problems using huge computer resources due to their optimal complexity. These methods do not need any global MPI collective call, but rather only neighbour point-to-point communication. A basic 2 level block-based Algebraic Multigrid preconditioner is already implemented in BLIS. This can be extended with a more general implementation.

Bibliography

[1] SPEC MPI2007: http://www.spec.org/mpi2007.

[2] Intel MPI Benchmarks: Users Guide and Methodology Description, Intel GmbH, Germany, 2004.

[3] Panel on HPC Challenge Benchmarks: An Expanded View of High End Computers, SC2004, (November 12, 2004).

[4] PARKBENCH, Netlib Repository, http://www.netlib.org/parkbench.

[5] Official HPCC website: http://icl.cs.utk.edu/hpcc.

[6] GeoFEM User Manual Ver.6.0 E, http://geofem.tokyo.rist.or.jp/manual_en.

[7] Lis 1.1.2 Users Manual, JST, http://www.ssisc.org/lis/index_en.html.

[8] Cray XT3 MPP Delivers Scalable Performance, white paper, January, 2005.

[9] P. Agarwal and et al. ORNL Cray X1 evaluation status report. Technical Report LBNL-55302, Lawrence Berkeley National Laboratory, May 1, 2004.

[10] C. C. Ashcraft and R. G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM J. Sci. Stat. Comput.*, 9(1):122–151, 1988.

[11] O. Axelsson. *Iterative solution methods*. Cambridge University Press, New York, NY, USA, 1994.

[12] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NAS Division, March, 1994.

[13] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[14] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[15] G. Birkhoff. Solving elliptic problems: 1930-1980. In M. Schultz, editor, *Elliptic problem solvers*, pages 17–38, New York, 1981. Academic Press.

[16] S. Borowski, S. R. Tiyyagura, and U. Küster. Matrix Assembly without Coloring on Vector Machines. In *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM 06)*, Crete, Greece, Sept 15-19 2006.

[17] A. Brandt. Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, Lecture Notes in Physics, Volume 18, pages 82–89, 1973.

[18] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial: second edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[19] P. Christen. *A Parallel Iterative Linear System Solver with Dynamic Load Balancing*. PhD thesis, Institut fr Informatik, Universitt Basel, 1999.

[20] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *Proceedings of the Fifth International Conference on Computational Science (ICCS 2005)*, Atlanta, USA, 2005. Springer-Verlag.

[21] E. F. D'Azevedo, P. A. Forsyth, and W.-P. Tang. Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems. *SIAM J. Matrix Anal. Appl.*, 13(3):944–961, 1992.

[22] J. Demmel, M. Heath, and H. van der Vorst. Parallel Numerical Linear Algebra. *Acta Numerica*, 2:53–62, 1993.

[23] J. Dongarra. *Solving linear systems on vector and shared memory computers*. SIAM, Philadelphia, PA, 1993.

[24] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15(1):1–18, 2003.

[25] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, March, 1990.

[26] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. Algorithm 679; a set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):18–28, March, 1990.

[27] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. Technical Report BIT, 29, 1989.

[28] C. Ethier and D. Steinman. Exact Fully 3D Navier Stokes Solution for Benchmarking. *International Journal for Numerical Methods in Fluids*, 19:369–375, 1994.

[29] R. D. Falgout. An Introduction to Algebraic Multigrid Computing. *Computing in Science and Engg.*, 8(6):24–33, 2006.

[30] R. D. Falgout and U. M. Yang. hypre: A Library of High Performance Preconditioners. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 632–641, London, UK, 2002. Springer-Verlag.

[31] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. Watson, editor, *Numerical Analysis, Dundee, 1975*, Lecture Notes in Math., Vol. 506, pages 73–89, Berlin, 1976. Springer.

[32] S. P. Frankel. Convergence rates of iterative treatments of partial differential equations. *MTAC*, pages 65–75, 1950.

[33] S. Fujino, M. Mori, and T. Takeuchi. Performance of hyperplane ordering on vector computers. *J. Comput. Appl. Math.*, 38(1-3):125–136, 1991.

[34] J. Gebis and D. Patterson. Embracing and Extending 20th-Century Instruction Set Architectures. *Computer*, 40(4):68–75, 2007.

[35] G. H. Golub and C. F. V. Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[36] G. H. Golub. and D. P. O'Leary. Some History of the Conjugate Gradient and Lanczos Algorithms: 1948-1976. *SIAM Review*, 31:50–102, 1989.

[37] W. Hackbusch. *Multigrid Methods and Applications*. Springer-Verlag, Berlin, 1985.

[38] M. A. Heroux. AztecOO User Guide. Technical Report SAND2004-3796, Sandia National Laboratories, 2004.

[39] M. A. Heroux. Epetra Performance Optimization Guide. Technical Report SAND2005-1668, Sandia National Laboratories, 2005.

[40] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[41] M. A. Heroux and J. M. Willenbring. Trilinos Users Guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[42] M. R. Hestenes and E. Stiefel. Method of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[43] S. Hossain. On Efficient Storage of Sparse Matrices. In *2005 Istanbul Computational Science and Engineering Conference (ICCSE 2005)*, ITU, Istanbul, 2005.

[44] A. S. Householder. *The theory of matrices in numerical analysis*. Blaisdell, New York, 1964.

[45] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, Computer Science Division, University of California at Berkeley, 2000.

[46] E.-J. Im and K. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. *Lecture Notes in Computer Science*, 2073:127–136, 2001.

[47] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, (1)18:135–158, 2004.

[48] M. T. Jones and P. E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, 1995.

[49] W. Kahan. *Gauss-Seidel Methods of Solving Large Systems of Linear Equations*. PhD thesis, University of Toronto, Canada, 1958.

[50] G. Karypis and V. Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Technical Report Version 4.0, Department of Computer Science, University of Minnesota, September 20, 1998.

[51] A. E. Koniges, R. Rabenseifner, and K. Solchenbach. Benchmark Design for Characterization of Balanced High-Performance Architectures. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), Workshop on Massively Parallel Processing (WMPP), Vol. 3*, San Francisco, USA, April 23-27, 2001. IEEE Computer Society Press.

[52] L. Kornblueh. ECHAM5 - An Atmospheric Climate Model and the Extension to a Coupled Model. In M. Resch, T. Bnisch, S. Tiyyagura, T. Furui, Y. Seo, and W. Bez, editors, *High Performance Computing on Vector Systems 2006*, pages 171–181. Springer, 2007.

[53] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53, 1952.

[54] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. *http://icl.cs.utk.edu/hpcc/pubs/*, March, 2005.

[55] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computing. http://www.cs.virginia.edu/stream/, Department of Computer Science, University of Virginia.

[56] E. Montagne and A. Ekambaram. An Optimal Storage Format for Sparse Matrices. *Information Processing Letters*, 90:87–92, 2004.

[57] K. Nakajima. Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator. In *SC*, page 13. ACM, 2003.

[58] K. Nakajima. Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the Earth Simulator. *Parallel Computing*, 31(10-12):1048–1065, 2005.

[59] K. Nakajima. Parallel Preconditioning Methods with Selective Fill-Ins and Selective Overlapping for Ill-Conditioned Problems in Finite-Element Methods. In Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *International Conference on Computational Science (3)*, volume 4489 of *Lecture Notes in Computer Science*, pages 1085–1092. Springer, 2007.

[60] K. Nakajima, H. Nakamura, and H. Okuda. Highly Stable Localized ILU Preconditioning for Unstructured Grids. In P. M. A. Sloot, M. Bubak, and L. O. Hertzberger, editors, *HPCN Europe*, volume 1401 of *Lecture Notes in Computer Science*, pages 904–906. Springer, 1998.

[61] M. Neumann, U. Küttler, S. R. Tiyyagura, W. A. Wall, and E. Ramm. Computational Efficiency of Parallel Unstructured Finite Element Simulations. In M. Resch, T. Bönisch, K. Benkert, T. Furui, Y. Seo, and W. Bez, editors, *High Performance Computing on Vector Systems, March 2005*, volume XIV, pages 89–107. Springer, 2006.

[62] M. Neumann, S. R. Tiyyagura, W. A. Wall, and E. Ramm. Efficiency Aspects for Advanced Fluid Finite Element Formulations. In *Thirteenth Conference on Finite Elements for Flow Problems*, Swansea, Wales, April 4-6, 2005.

[63] A. Nishida, H. Kotakemori, T. Kajiyama, and A. Nukada. Scalable software infrastructure project. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 140, New York, NY, USA, 2006. ACM.

[64] H. Okuda and G. Yagawan. Large-Scale Parallel Finite Element Analysis for Solid Earth Problems by GeoFEM. Technical Report GeoFEM.2001-001, Research Organization for Information Science and Technology, Tokyo, 2001.

[65] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC 2004)*, Pittsburgh, USA, 2004.

[66] L. Oliker, A. Canning, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, and R. van der Wijngaart. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC 2003)*, Phoenix, Arizona, USA, 2003.

[67] T. Pohl, F. Deserno, N. Threy, U. Rde, P. Lammers, G. Wellein, and T. Zeiser. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *SC*, page 21. IEEE Computer Society, 2004.

[68] R. Rabenseifner. Hybrid Parallel Programming on HPC Platforms. In *Proceedings of the Fifth European Workshop on OpenMP, EWOMP '03*, pages 185–194, 2003.

[69] R. Rabenseifner and A. E. Koniges. Effective Communication and File-I/O Bandwidth Benchmarks. In J. Dongarra and Y. Cotronis, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 8th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2001*, Santorini, Greece, Sep. 23-26, 2001. IEEE Computer Society Press.

[70] R. Rabenseifner, S. R. Tiyyagura, and M. Müller. Network Bandwidth Measurements and Ratio Analysis with the HPC Challenge Benchmark Suite (HPCC). In B. D. Martino, D. K. Mueller, and J. Dongarra, editors, *Proceedings of the 12th European PVM/MPI Users' Group Meeting (EURO PVM/MPI 2005)*, LNCS 3666, pages 368–378, Sorrento, Italy, Sep. 18-21 2005. Springer.

[71] R. Rabenseifner and G. Wellein. Comparison of Parallel Programming Models on Clusters of SMP Nodes. In H. Bock, E. Kostina, H. Phu, and R. Rannacher, editors, *Proceedings of the International Conference on High Performance Scientific Computing*, pages 409–426. Springer, 2004.

[72] J. Ruge and K. Stüben. Algebraic multigrid. In S. F. McCormick, editor, *Multigrid Methods, Frontiers in Applied Mathematics*, volume 3, pages 73–130. Springer, 1987.

[73] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations. Technical Report RIACS-90-20, NASA Ames Research Center, Moffet Field, CA, 1994.

[74] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition.* SIAM, Philadelphia, PA, 2003.

[75] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[76] P. Sadayappan and V. Visvanathan. Efficient sparse matrix factorization for circuit simulation on vector supercomputers. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(12):1276–1285, 1989.

[77] M. Schäfer and S. Turek. Benchmark Computations of Laminar Flow Around a Cylinder. *Notes on Numerical Fluid Mechanics*, 52:547–566, 1996.

[78] J. R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, Pittsburgh, PA, USA, 1994.

[79] P. Stathis, S. Vassiliadis, and S. Cotofana. A Hierarchical Sparse Matrix Storage Format for Vector Processors. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 61.1, Washington, DC, USA, 2003. IEEE Computer Society.

[80] P. T. Stathis. *Sparse Matrix Vector Processing Formats*. PhD thesis, Technische Universiteit Delft, 2004.

[81] K. Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128(1-2):281–309, 2001.

[82] D. Takahashi and Y. Kanada. High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers. *Journal of Supercomputing*, 15(2):207–228, Feb., 2000.

[83] S. R. Tiyyagura. Efficient Implementation for Fluid and Structural Elements. Master's thesis, Institute of Structural Mechanics, University of Stuttgart, Germamy, May 2004.

[84] S. R. Tiyyagura and U. Küster. Block-Based Approach to Solving Linear Systems. In Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *International Conference on Computational Science (1)*, volume 4487 of *Lecture Notes in Computer Science*, pages 128–135. Springer, 2007.

[85] S. R. Tiyyagura, U. Küster, and S. Borowski. Performance Improvement of Sparse Matrix Vector Product on Vector Machines. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Proceedings of the Sixth International Conference on Computational Science (ICCS 2006)*, LNCS 3991, pages 196–203, Reading, UK, May 28-31 2006. Springer.

[86] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Aztec User's Guide: Version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.

[87] R. S. Tuminaro, J. Shadid, and S. Hutchinson. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency - Practice and Experience*, 10(3):229–247, 1998.

[88] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5(1-2):45–54, 1987.

[89] H. A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10(6):1174–1185, 1989.

[90] H. A. van der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

[91] B. van Rietbergen, H. Weinans, R. Huiskes, and A. Odgaard. A new method to determine trabecular bone elastic properties and loading using micromechanical finite-elements models. *Journal of Biomechanics*, 28(1):69–81, 1995.

[92] P. Vanek, M. Brezina, and J. Mandel. Convergence of Algebraic Multigrid Based on Smoothed Aggregation. *Numerische Mathematik*, 88:559–579, 2001.

[93] P. Vanek, J. Mandel, and M. Brezina. Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems. *Computing*, 56(3):179–196, 1996.

[94] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1962.

[95] S. Vassiliadis, S. Cotofana, and P. Stathis. Vector ISA Extension for Sparse Matrix-Vector Multiplication. In *European Conference on Parallel Processing*, pages 708–715, 1999.

[96] R. Vuduc, J. W. Demmel, and K. A. Yelick. The Optimized Sparse Kernel Interface (OSKI) Library Users Guide. Technical Report Version 1.0.1h, BeBOP Group, University of California, Berkeley, 2007.

[97] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, Computer Science Division, University of California at Berkeley, 2003.

[98] R. W. Vuduc and H.-J. Moon. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block. *Lecture Notes in Computer Science*, 3726:807–816, 2005.

[99] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).

[100] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[101] W. A. Wulf and S. A. Mckee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[102] U. M. Yang. Parallel Algebraic Multigrid Methods - High Performance Preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solutions of PDEs on Parallel Computers*, Lecture Notes in Computational Science and Engineering, pages 209–236. Springer-Verlag, 2006.

[103] D. M. Young. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type*. PhD thesis, Harvard University, Mathematics Department, Cambridge, MA, USA, 1950.

[104] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.

[105] D. M. Young. An historical review of iterative methods. In *Proceedings of the ACM conference on History of scientific and numeric computation*, pages 117–123, New York, NY, USA, 1987. ACM.

Appendix A

BLIS Interface

BLIS presently works with FE problems having 1, 3, 4 or 6 unknowns at each grid point. Another constraint is that the Dirichlet conditions should also to solved. This is necessary as the solver assumes equal number of unknowns at each grid point (same block size for the whole matrix).

A.1 Implemented algorithms

Presently, the following functionality is available in BLIS.

- Krylov subspace methods
  - BiCGSTAB
  - GMRES(m) with classical/modified Gram-Schmidt orthogonalization
  - CG

- Preconditioners for the above methods
  - Block scaling
  - Block Jacobi preconditioning
  - Colored block symmetric Gauss-Seidel preconditioning
  - Block ILU(0) on subdomains
  - Classical AMG (Ruge-Stüben) with 2 grids

A.2 BLIS user options

The following options can be specified by the users when using BLIS.

typedef struct Options_ {
int method; /*solution method*/
 - 1 for BiCGSTAB, 2 for GMRES(m) and 3 for CG
int scaling; /*matrix scaling*/

- 0 for No and 1 for Yes
int preconditioner; /*preconditioner*/
- 0 for None, 1 for Jacobi and 4 for ILU(0)
int max_iter; /*maximum iterations if not converged*/
- maximum iterations when not converged
double err_tol; /*error tolerance to check convergence*/
- RHS scaled error tolerance ($||rhs - A * x||/||rhs||$) to check convergence
- usually between 1.0e-4 and 1.0e-8
int space_size; /*for GMRES(m) - restarted*/
- subspace size in restarted GMRES method
- suggested optimal value = 50
int orthog; /*Orthogonalization of Krylov vectors in GMRES(m) method*/
- 1 for Standard Gram-Schmidt and 2 for Modified Gram-Schmidt
- suggested optimal value = 1
int prec_iter; /*number of Jac or SGS iterations for preconditioning*/
- number of preconditioner iterations when using splitting methods
- suggested optimal value = 5
double prec_relax; /*relaxation in each preconditioner iteration*/
- Relaxation for splitting method preconditioners
- suggested optimal value = 0.35 for Jacobi preconditioner
int status; /*for conveying breakdown in the selected method*/
-(OUTPUT) 0 for Success, 1 for Numerical Breakdown and 2 when not converged
int iters; /*returns actual no of iterations in the method*/
- (OUTPUT) Number of actual iterations in the selected solution method
int subiters; /*returns actual no of subiterations in the gmres method*/
- (OUTPUT) Number of actual subiterations in the GMRES method (not much useful)
} Options;

A.3 List of C functions

All BLIS function names start with "blis*", where the "*" has to be replaced by the block size, i.e. for block size 3, all the BLIS functions start with the name blis3.

****

/*this function calls the BLIS solver*/

void blis*_solver(Matrix *A_local, double *b, double *x, Matrix *Scal, Matrix *Prec, Colored_Matrix *Precsgs, AMG_set *AMG_matrices, Communication *Comm, Options *solver_options)

Type of parameter	Parameter name	Description
input	*A_local*	block JAD matrix
input	*b*	rhs vector
input/output	*x*	solution vector
input	*Scal*	scaling block JAD matrix
input	*Prec*	ILU factors
input	*Precsgs*	colored block JAD matrix
input	*AMG_matrices*	Matrices used in AMG method
input	*Comm*	communication data object
input/output	*solver_options*	user options object

****

/*this function allocated memory for the scaling matrix block JAD object*/

void blis*_scal_allocate(Matrix *Dinv_local, int num_rows, int blksize)

Type of parameter	Parameter name	Description
output	*Dinv_local*	scaling block JAD matrix
input	*num_rows*	local matrix # rows
input	*blksize*	block size in the sparse matrice

****

/*this function destroys the scaling matrix block JAD object*/

void blis*_scal_destroy(Matrix *Dinv_local)

Type of parameter	Parameter name	Description
input	*Dinv_local*	scaling block JAD matrix

****

/*this function initializes the scaling matrix block JAD object*/

void blis*_scaling_init(Matrix *A_local, Matrix *Dinv_local)

Type of parameter	Parameter name	Description
input	*A_local*	block JAD matrix
output	*Dinv_local*	scaling block JAD matrix

****

/*this function initializes the BiCGSTAB preconditioner (not working

reliably yet)*/

void blis*_prec_bicgstab_init(int num_rows, int num_external)

Type of parameter	Parameter name	Description
input	*num_rows*	local matrix # rows
input	*num_external*	# external ghost points to be received

****

/*this function destroys the BiCGSTAB preconditioner*/

void blis*_prec_bicgstab_destroy()

No function parameters

****

/*this function allocates the matrix used for ILU(0) preconditioning*/

void blis*_precond_allocate(Matrix *Prec, int num_rows, int num_nz, int blksize)

Type of parameter	Parameter name	Description
output	*Prec*	ILU factors
input	*num_rows*	local matrix # rows
input	*num_nz*	local matrix # non-zeros
input	*blksize*	block size in the sparse matrice

****

/*this function destroys the matrix used for ILU(0) preconditioning*/

void blis*_precond_destroy(Matrix *Prec)

Type of parameter	Parameter name	Description
input	*Prec*	ILU factors

****

/*this function initializes the matrix used for ILU(0) preconditioning*/

void blis*_precond_init(Matrix *Prec, Matrix *A)

Type of parameter	Parameter name	Description
output	*Prec*	ILU factors
input	*A*	block JAD matrix

****

/*this function allocates the colored block JAD matrix used for SGS preconditioning*/

void blis*_precsgs_allocate(Colored_Matrix *Precsgs, COLORING *coloring, int blksize)

Type of parameter	Parameter name	Description
output	*Precsgs*	colored block JAD matrix
input	*coloring*	graph coloring object
input	*blksize*	block size in the sparse matrice

****

/*this function deletes the colored block JAD matrix*/

void blis*_precsgs_destroy(Colored_Matrix *Precsgs)

Type of parameter	Parameter name	Description
input	*Precsgs*	colored block JAD matrix

****

/*this function initializes the colored block JAD matrix*/

void blis*_precsgs_init(Colored_Matrix *Precsgs, COLORING *coloring, Matrix *A)

Type of parameter	Parameter name	Description
output	*Precsgs*	colored block JAD matrix
input	*coloring*	graph coloring object
input	*A*	block JAD matrix

****

/*this function selects the coarse points based on the mesh graph and strength*/

void blis*_amg_coarsen_ruge_init(Matrix *A_local, int *coarsening, double strength)

Type of parameter	Parameter name	Description
input	*A_local*	block CRS matrix
output	*coarsening*	list of coarse and fine points
input	*strength*	cut-off strength

****

/*this function allocates an AMG interpolation matrix*/

void blis*_amg_interp_allocate(Matrix *A_local, Matrix *AMG_interp)

Type of parameter	**Parameter name**	**Description**
input	*A_local*	block CRS matrix
output	*AMG_interp*	CRS Interpolation matrix

****

/*this function deletes an AMG interpolation matrix*/

void blis*_amg_interp_delete(Matrix *AMG_interp)

Type of parameter	**Parameter name**	**Description**
input	*AMG_interp*	CRS Interpolation matrix

****

/*this function initializes an AMG interpolation matrix*/

void blis*_amg_interp_init(Matrix *A_local, Matrix *AMG_interp, int *coarsening, double strength)

Type of parameter	**Parameter name**	**Description**
input	*A_local*	block CRS matrix
output	*AMG_interp*	CRS Interpolation matrix
input	*coarsening*	list of coarse and fine points
input	*strength*	cut-off strength

****

/*this function allocates a transposed AMG interpolation matrix*/

void blis*_amg_crs_transp_allocate(Matrix *AMG_interp, Matrix *AMG_interp_transp)

Type of parameter	**Parameter name**	**Description**
input	*AMG_interp*	CRS Interpolation matrix
output	*AMG_interp_transp*	CRS Restriction matrix

****

/*this function deletes a transposed AMG interpolation matrix*/

void blis*_amg_crs_transp_delete(Matrix *AMG_interp_transp)

Type of parameter	Parameter name	Description
input	*AMG_interp_transp*	CRS Restriction matrix

****

/*this function initializes a transposed AMG interpolation matrix*/

void blis*_amg_crs_transp_init(Matrix *AMG_interp, Matrix *AMG_interp_transp)

Type of parameter	Parameter name	Description
input	*AMG_interp*	CRS Interpolation matrix
output	*AMG_interp_transp*	CRS Restriction matrix

****

/*this function allocates a coarse grid operator(matrix)*/

void blis*_amg_coarsegrid_allocate(Matrix *AMG_transp, Matrix *A_local, Matrix *AMG_coarsegrid)

Type of parameter	Parameter name	Description
input	*AMG_transp*	CRS Restriction matrix
input	*A_local*	block CRS matrix
output	*AMG_coarsegrid*	coarse grid operator(block CRS matrix)

****

/*this function deletes a coarse grid operator(matrix)*/

void blis*_amg_coarsegrid_delete(Matrix *AMG_coarsegrid)

Type of parameter	Parameter name	Description
input	*AMG_coarsegrid*	coarse grid operator(block CRS matrix)

****

/*this function initializes a coarse grid operator(matrix)*/

void blis*_amg_coarsegrid_init(Matrix *AMG_transp, Matrix *A_local, Matrix *AMG_coarsegrid)

Type of parameter	Parameter name	Description
input	*AMG_transp*	CRS Restriction matrix
input	*A_local*	block CRS matrix
output	*AMG_coarsegrid*	coarse grid operator(block CRS matrix)

****

/*this function renumbers the indices on each processor from global to process local numbering*/

void blis*_comm_renumbering(Matrix *A_local, int *global_rows_proc, int *recv_list, int *send_list, int send_length, int red_n)

Type of parameter	Parameter name	Description
input/output	*A_local*	block CRS matrix
input	*global_rows_proc*	list of global rows on local process/partition
input	*recv_list*	list of all ghost points to be received from the neighbours
input/output	*send_list*	list of all ghost points to be send to the neighbours
input	*send_length*	# ghost points to be sent
input	*red_n*	global # rows

****

/*this function schedules communication using the partitioning data (e.g. from tools like Metis)*/

void blis*_comm_scheduling(Communication *Comm, int red_n, int *red_xadj, int *red_adjncy, int *red_part, int nparts)

Type of parameter	Parameter name	Description
output	*Comm*	communication data object
input	*red_n*	# global nodal points
input	*red_xadj*	CRS row vector of the mesh connectivity graph
input	*red_adjncy*	CRS index vector of the mesh connectivity graph
input	*red_part*	global partitioned information list
input	*nparts*	# partitions

****

/*this function arranges(colors) the order of communication between processors*/

void blis*_comm_coloring(Communication *Comm)

Type of parameter	Parameter name	Description
input/output	*Comm*	

A.4 Fortran interface

A primitive Fortran interface is available in BLIS along with an example. Some functionality, such as the colored symmetric Guass Seidel preconditioner, is not yet available in the Fortran interface. The following functionality can be accessed presently via the interface.

- Krylov subspace methods
  - BiCGSTAB
  - GMRES(m)
  - CG
- Preconditioners for the above methods
  - Diagonal Scaling
  - Jacobi
  - subdomain ILU(0)

A.4.1 Implementation details

Holding C user defined data objects in Fortran

All the memory allocated in C using the BLIS Fortran interface is held in Fortran as 64-bit addresses (this is portable on 32 and 64-bit architectures).

Array indexing

Due to the differences in array notation between C and Fortran, some arrays in Fortran need to be modified before passing them to BLIS. BLIS assumes C array notation for all the arrays that store the indices of other arrays. This is particularly true with the send and receive lists used in communication that hold the array indices of the ghost variables. This is also true for the sparse matrix arrays that store the row/diagonal pointers and the column indices. So, the above mentioned arrays that hold the index information have to be decremented by one before they are passed to BLIS using the Fortran interface.

A.4.2 List of functions

All the Fortran interface calls start with the name "blis*inter", where "*" has to be replaced with the block size.

****

/*this function calls the BLIS solver*/

void blis*inter_solver(unsigned long long int *matrixobject, double *b,

double *x, unsigned long long int *scalingobject, unsigned long long int *precobject, unsigned long long int *precsgsobject, unsigned long long int *commobject, unsigned long long int *optionsobject)

Type of parameter	Parameter name	Description
input	*matrixobject*	block JAD matrix
input	*b*	rhs vector
input/output	*x*	solution vector
input	*scalingobject*	scaling block JAD matrix
input	*precobject*	ILU factors
input	*precsgsobject*	colored block JAD matrix
input	*commobject*	communication data object
input/output	*optionsobject*	user options object

****

/*this functions builds the user optionsobject*/

void blis*inter_buildoptions(unsigned long long int *optionsobject, int *max_iter, int *scaling, int *preconditioner, int *method, int *status, int *prec_iter, int *space_size, int *orthog, double *err_tol, double *prec_relax)

Type of parameter	Parameter name	Description
output	*optionsobject*	user options object
input	*max_iter*	maximum iterations if not converged
input	*scaling*	matrix diagonal scaling
input	*preconditioner*	preconditioner type
input	*method*	solution method
input	*status*	for conveying breakdown in the selected method
input	*prec_iter*	# Jacobi or SGS iterations for preconditioning
input	*space_size*	for GMRES(m) - restarted
input	*orthog*	orthogonalization of Krylov vectors in GMRES(m) method
input	*err_tol*	error tolerance to check convergence
input	*prec_relax*	relaxation in each precond. iter.

****

/*this function prints the status of the solution*/

void blis*inter_printstatus(unsigned long long int *optionsobject)

Type of parameter	Parameter name	Description
input	*optionsobject*	user options object

****

/*this function schedules communication using the partitioning data (e.g. from tools like Metis)*/

void blis*inter_comm_scheduling(int *red_n, int *red_xadj, int *red_adjncy, int *red_part, int *nparts, int *ext_rows, int *blksize, unsigned long long int *commobject)

Type of parameter	Parameter name	Description
input	*red_n*	# global nodal points
input	*red_xadj*	CRS row vector of the mesh connectivity graph
input	*red_adjncy*	CRS index vector of the mesh connectivity graph
input	*red_part*	global partitioned information list
input	*nparts*	# partitions
output	*ext_rows*	# external ghost points to be received
input	*blksize*	block size in the sparse matrice
output	*commobject*	communication object

****

/*this function renumbers the indices on each processor from global to process local numbering*/

void blis*inter_comm_renumbering(int *indices, int *global_rows_proc, int *num_rows, int *total_num_rows, int *num_nnz, unsigned long long int *commobject)

Type of parameter	Parameter name	Description
input/output	*indices*	local block CRS index vector
input	*global_rows_proc*	list of global rows on local process/partition
input	*num_rows*	local matrix # rows
input	*total_num_rows*	global # rows
input	*num_nnz*	local matrix # non-zeros
input/output	*commobject*	communication object

****

/*this function initializes communication using already existing Fortran data*/

void blis*inter_buildcommunicator(int *num_neigh, int *my_neigh, int *send_length, int *send_list, int *recv_length, int *recv_list, int *blksize, unsigned long long int *commobject)

Type of parameter	Parameter name	Description
input	*num_neigh*	# neighbours to me (domain)
input	*my_neigh*	list of my neighbours
input	*send_length*	list of # ghost points to be send to the neighbours
input	*send_list*	list of all ghost points to be send to the neighbours
input	*recv_length*	list of # ghost points to be received from the neighbours
input	*recv_list*	list of all ghost points to be received from the neighbours
input	*blksize*	block size in the sparse matrice
output	*commobject*	communication object

****

/*this function allocates the matrix used for scaling*/

void blis*inter_scal_allocate(unsigned long long int *scalingobject, int *num_rows, int *blksize)

Type of parameter	Parameter name	Description
output	*scalingobject*	scaling block JAD matrix
input	*num_rows*	local matrix # rows
input	*blksize*	block size in the sparse matrice

****

/*this function destroys the matrix used for scaling*/

void blis*inter_scal_destroy(unsigned long long int *scalingobject)

Type of parameter	Parameter name	Description
input	*scalingobject*	scaling block JAD matrix

****

/*this function allocates the matrix used for ILU(0) preconditioning*/

void blis*inter_precond_allocate(unsigned long long int *precobject, int *num_rows, int *num_nz, int *blksize)

Type of parameter	Parameter name	Description
output	*precobject*	ILU factors
input	*num_rows*	local matrix # rows
input	*num_nz*	local matrix # non-zeros
input	*blksize*	block size in the sparse matrice

****

/*this function destroys the matrix used for ILU(0) preconditioning*/

void blis*inter_precond_destroy(unsigned long long int *precobject)

Type of parameter	Parameter name	Description
input	*precobject*	ILU factors

****

/*this function initializes the matrix used for ILU(0) preconditioning*/

void blis*inter_precond_init(unsigned long long int *precobject, double *values, int *row_ptrs, int *indices, int *num_rows, int *num_nz, int *blksize)

Type of parameter	Parameter name	Description
output	*precobject*	ILU factors
input	*values*	local block CRS values vector
input	*row_ptrs*	local block CRS row pointer vector
input	*indices*	local block CRS index vector
input	*num_rows*	local matrix # rows
input	*num_nz*	local matrix # non-zeros
input	*blksize*	block size in the sparse matrice

****

/*this function converts the sparse matrix from block CRS to block JAD format*/

void blis*inter_crstojadblk(unsigned long long int *matrixobject, double *values, int *row_ptrs, int *indices, int *num_rows, int *num_nz, int *ext_rows, int *blksize)

Type of parameter	Parameter name	Description
input/output	*matrixobject*	block CRS matrix
input	*values*	local block CRS values vector
input	*row_ptrs*	local block CRS row pointer vector

input	*indices*	local block CRS index vector
input	*num_rows*	local matrix # rows
input	*num_nz*	local matrix # non-zeros
input	*ext_rows*	# external ghost points to be received
input	*blksize*	block size in the sparse matrice

****

/*this function frees memory used for the block JAD matrix*/

void blis*inter_release_matrix(unsigned long long int *matrixobject)

Type of parameter	**Parameter name**	**Description**
input	*matrixobject*	block JAD matrix

Curriculum Vitae

Personal data

Sunil Reddy Tiyyagura
3-2-373, Flat 203
My Avenue Apts, Kachiguda
500027, Hyderabad
India

Tel.: +91(040)24652645
E-Mail: suniltiyya@yahoo.com

D.O.B.: 12. 05.1979 in Tenali, India
Married, Indian

Education

04/1984–03/1995	St. Joseph's Public School, Hyderabad, India
05/1995–05/1997	Ratna Junior College, Hyderabad, India
08/1998–06/2002	Bachelor of Mechanical Engineering, Chaitanya Bharati Institute of Technology, Hyderabad, India
10/2002–06/2004	MSc. Computational Mechanics (COMMAS), University of Stuttgart, Germany
10/2004–10/2008	PhD, HLRS, University of Stuttgart, Germany

Work Experience

11/2002–09/2004	Student Researcher at HLRS, University of Stuttgart, Germany
10/2004–10/2008	Technical Staff Member, Teraflop Workbench Project, HLRS, Germany
11/2008–present	Senior Software Engineer, Synopsys Inc., India

Languages English, German (conversational), Telugu, Hindi

Stuttgart, 08. 07.2010