RC24809 (W0906-064) June 15, 2009
Computer Science

IBM Research Report

Sparse Matrix Factorization on Massively Parallel Computers

Anshul Gupta
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Seid Koric
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801

Thomas George
Department of Computer Science
Texas A & M University
College Station, TX 77843

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Sparse Matrix Factorization on Massively Parallel Computers

Anshul Gupta
Mathematical Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
anshul@watson.ibm.com

Seid Koric
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801
skoric@ncsa.uiuc.edu

Thomas George
Department of Computer Science
Texas A & M University
College Station, TX 77843
tgeorge@cs.tamu.edu

Abstract

Although direct methods for solving sparse systems of linear equations have much higher asymptotic computational and memory requirements compared to iterative methods, systems arising in some applications, such as structural analysis, can often be too ill-conditioned for iterative solvers to be effective. We cite real applications where this is indeed the case, and using matrices extracted from these applications to conduct experiments on three different massively parallel architectures, show that a well designed sparse factorization algorithm can attain very high levels of performance and scalability. We present strong scalability results for test data from real applications on up to 8,192 cores, along with both analytical and experimental weak scalability results for a model problem on up to 16,384 cores—an unprecedented number for sparse factorization. For the model problem, we also compare experimental results with multiple analytical scaling metrics and distinguish between some commonly used weak scaling methods.

1 Introduction

The core computation in a large number of applications in science, engineering, and optimization involves solving large sparse systems of linear equations of the form $Ax = b$, where $A \in \mathbb{C}^{N \times N}$ is the sparse coefficient matrix, $b \in \mathbb{C}^N$ is the right hand-side vector, and $x \in \mathbb{C}^N$ is the vector of unknowns for which a solution is sought. These systems are typically solved using two classes of methods: (1) *iterative methods* that start with an initial approximation of the solution and iteratively refine it until the desired accuracy is achieved, and (2) *direct methods* that compute a factorization $A = LU$, where $L, U \in \mathbb{C}^{N \times N}$ are lower and upper triangular, respectively, and the solution x can be obtained by trivially solving the triangular systems $Ly = b$ and $Ux = y$.

A practically important class of sparse linear systems is one in which A is Hermitian and positive definite (referred to as symmetric positive-definite or SPD when A is real). A symmetric matrix can be regarded as the adjacency matrix of a graph. It is well known [13, 35] that the total work involved in factoring the adjacency matrix of an N-node graph whose n-node subgraphs have $O(n^{2/3})$-node separators is $O(N^2)$ with a nested-dissection [13] ordering. The corresponding memory requirement is $O(N^{4/3})$. All matrices that arise from finite-element or finite-difference discretizations of partial differential equations over three-dimensional (3-D) domains belong to this class. Similar lower bounds for the time and space requirements in the two-dimensional case are $O(N^{3/2})$ and $O(N \log(N))$, respectively. Clearly, both the time and the memory requirements grow superlinearly with the size of the system being solved. On the other hand, there are preconditioners for Krylov subspace iterative methods that, at least in theory, can solve such systems in $O(N)$ space and $O(N)$ time. In

practice however, the sparse systems arising in many applications can be too ill-conditioned for the iterative solvers to be effective. Also, in practice, $O(N)$ time is rarely achieved with an $O(N)$-space preconditioner because the number of iterations required for satisfactory convergence invariably grows with the size of the system. Finally, despite a higher asymptotic complexity, direct solvers turn out to be much faster than iterative solver for moderately sized systems arising in many practical applications [11, 14, 15]. This is primarily because (1) a relatively inexpensive symbolic factorization phase computes the static structure of the factors to enable the subsequent numerical factorization phase to proceed with minimal use of expensive integer and pointer operations, and (2) supernodal [13] and multifrontal [8, 37] techniques ensure that practically all floating-point computation is performed by cache-friendly level 2 and level 3 basic linear algebra subprograms (BLAS) [6, 7]. Direct solvers are extremely efficient in situations where several systems need to be solved with the same coefficient matrix. Direct solvers and the associated algorithms can also be useful in the context of iterative solvers; for example, as coarse-grid solvers for multigrid methods [38] or for computing preconditioners based on incomplete factorization [39]. Therefore, a high-performance and scalable parallel direct solver is an invaluable scientific computing tool.

In this paper, we experimentally and analytically explore the performance and scalability properties of the direct solver for symmetric positive definite (SPD) matrices in Watson Sparse Matrix Package (WSMP) [18]. Using matrices extracted from real applications that currently rely on direct solvers due to their robustness, we show that well designed algorithms for factorization [20] and triangular solution [21, 27] can attain very high levels of performance and scalability. We present strong scalability results on up to 8,192 CPUs for test data from real applications on three different massively parallel architectures. We observed sparse factorization speeds up to 4.6 Teraflops for a problem for which the factors require less than 3% of the memory of the machine. We also present both analytical and experimental weak scalability results for a model problem on up to 16,384 cores of a Blue Gene/P (BG/P) [32] machine. For our largest test case, the factorization required less than 2% of the available memory and attained 7.05 Teraflops. To the best of our knowledge, this is the highest performance and the utilization of the maximum number of cores for sparse matrix factorization reported to date. For the model problem, we also compare experimental results with multiple analytical scaling metrics and distinguish between some commonly used weak scaling methods.

The remainder of the paper is organized as follows. Section 2 introduces our experimental set up. In Section 3, we present our experience with iterative solvers on our suite of test problems. In Section 4, we compare the performance and scalability of three distributed-memory parallel direct solvers. In Section 5, we present performance and scalability results for WSMP's symmetric direct solver on Cray XT4 and IBM BG/P computers. In Section 6, we present performance and scalability results for a comprehensive set of matrices derived from a model 3-D problem on the BG/P. In Section 7, we present weak scalability analyses of symmetric sparse factorization algorithm under various criteria for increasing the problem size with the size of the machine and compare analytical and experimental results. Section 8 contains concluding remarks.

2 Experimental Setup

In this section, we present the details of our experimental setup. These include an introduction to the test matrices, the solver packages tried, and the hardware platforms used.

Matrix	N	NNZ	Application
Bstone-1	1,017,397	74,144,859	Structural Analysis
Lmco	665,017	107,514,163	Structural Analysis
Nastran-b	1,508,088	111,614,436	Structural Analysis
Sgi_1M	1,522,431	125,755,875	Structural Analysis
Ten-b	1,371,166	108,009,680	3-D Metal Forming

Table 1: SPD test matrices with their order (N) and number of non-zeros (NNZ).

2.1 Test data

The bulk of the experiments reported in this paper are performed on five test matrices extracted from real applications that currently use a direct solver. Table 1 gives some basic details of these matrices. The matrix *Bstone-1* was generated by an in-house FEM software for structural analysis of automobile tires at Bridgestone Corporation. The matrix *Lmco* was generated by the application of Lockheed Martin Corporation's DIAL finite element system to a structural analysis problem. The matrix *Nastran-b* also comes from a structural analysis problem while performing implicit nonlinear analysis using MSC Nastran [5]. *Sgi_1M* is a global stiffness matrix extracted from MSC Nastran while performing linear static analysis of a machine bracket. The matrix *Ten-b* is generated by Tenaris' in-house finite-element simulation software for modeling complex 3-D bulk metal forming processes.

In addition to these matrices, we have used matrices derived from cubic 3-dimensional 7-point stencil finite-difference grids of varying sizes to present weak scaling results in Section 6.

2.2 Software packages

Following is a brief description of the various iterative and direct solver packages used in this study:

Hypre (version 2.0.0): Hypre [9], developed at Lawrence Livermore National Laboratory, is an iterative solver package designed for solving large, sparse systems on massively parallel computers. Hypre includes parallel preconditioners based on incomplete factorization (Euclid [26]), sparse approximate inverses (ParaSails [4]), and algebraic multigrid (BoomerAMG [25]).

MUMPS (version 4.7.3): MUMPS [2, 3] is a distributed-memory parallel direct solver package for symmetric and unsymmetric systems and is based on the parallel multifrontal method [8].

SuperLU_DIST (version 2.3): SuperLU_DIST [33, 34] is the distributed-memory parallel version of the SuperLU family of solvers. It is based on supernodal right-looking LU factorization and is designed for general systems. Unlike symmetric solvers, it computes separate lower and upper triangular factors. Therefore, in practice, it would typically not be used to solve symmetric positive definite systems, which are the focus of our study. We have included SuperLU in our experiments to compare the scalability of various direct factorization algorithms.

WSMP (version 8.7): The Watson Sparse Matrix Package (WSMP) [18] contains hybrid distributed- and shared-memory parallel direct solvers based on the multifrontal algorithm for both symmetric [19, 20] and unsymmetric [16] systems.

Hardware Feature	Blue Gene/P	Cray XT4	IBM p5-575 Cluster
Processors	32-bit PPC 450	64-bit AMD Opteron	64-bit Power5+
Cores per node	4 (one quad-core)	4 (one quad-core)	16 (8 dual-cores)
Core frequency	850 MHz	2.3 GHz	1.9 GHz
Memory per node	4 GB	4 GB	32 GB
L1 data cache	32 KB	64 KB	32K KB
L2 cache	14 stream prefetching	512 KB private/core	1.9 MB shared/dual-core chip
L3 cache	8 MB shared/node	2 MB shared/node	36 MB shared/dual-core chip
Theoretical peak	3.4 GFlops/core	9.2 GFlops/core	7.6 GFlops/core
Max no. of cores used	16384	4096	1024

Table 2: Key hardware features of the three supercomputers used for measuring the performance of WSMP's sparse Cholesky factorization.

2.3 Hardware platforms

We used three very different hardware platforms to test the performance and scalability of WSMP's direct solver for the symmetric matrices described in Section 2.1. Table 2 gives the basic details of each of these platforms. The Blue Gene/P (BG/P) system is located at IBM T.J. Watson Research Center. We used the Cray XT4 system at University of Bergen's Center for Computational Science. We used two IBM p5-575 clusters, one at Texas A&M University and one at the National Center for Supercomputing Applications (NCSA) at University of Illinois. Both clusters have the same nodes; however, the cluster at Texas A&M University has a faster switch. The experiments comparing the direct and iterative solvers reported in Section 3 were performed on the Texas A&M cluster, while those comparing the scalability of different direct solvers reported in Section 4 were performed on the NCSA cluster.

3 Comparison with Iterative Solvers

In this section, we explore how WSMP's direct solver compares with the *best-case* scenario for iterative solvers for solving systems with the five coefficient matrices listed in Table 1. George et al. [14] have conducted a fairly rigorous comparison of the implementations of several state-of-the-art preconditioning methods for SPD systems in a number of iterative solver packages. From the results of that study, we concluded that the

Preconditioner	Orderings	Parameters	No. of Configurations
IC(k)	RCM, ND	*Level of fill:* 0, 1, 2, 4, 6, 8	12
BoomerAMG	RCM, ND NONE	*Maximum number of levels:* 25 *Number of aggressive coarsening levels:* 0, 10 *Coarsening schemes:* Falgout, HMIS, PMIS *Strong threshold:* 0.25, 0.5, 0.8. 0.9	72
ParaSails	RCM, ND NONE	*Number of levels:* 0, 1, 2 *Threshold:* 0, 0.01, 0.1, -0.75, -0.9 *Filter:* 0, 0.001, 0.05, -0.9	81

Table 3: Preconditioners and their parameters that were used.

Matrix	Solver	Failure rate	Av. Mem	Med. Mem	Min. Mem	Max. Mem
Nastran-b	hypre-IC(k)	1/12	1.81e+10	1.92e+10	4.08e+09	3.89e+10
	hypre-AMG	2/72	3.35e+09	1.70e+09	1.03e+09	2.84e+10
	hypre-PSAILS	52/81	4.48e+09	4.63e+09	2.30e+09	7.47e+09
	wsmp-DIRECT	0/1	9.12e+09	9.12e+09	9.12e+09	9.12e+09
Sgi_1M	hypre-IC(k)	0/12	2.54e+10	1.98e+10	4.22e+09	5.10e+10
	hypre-AMG	0/72	4.33e+09	1.98e+09	1.15e+09	4.42e+10
	hypre-PSAILS	15/81	3.67e+09	2.81e+09	2.41e+09	8.83e+09
	wsmp-DIRECT	0/1	1.57e+10	1.57e+10	1.57e+10	1.57e+10

Table 4: Memory statistics for the direct and iterative solvers on 64 processors.

Matrix	Solver	Failure rate	Av. Time	Med. Time	Min. Time	Max. Time
Nastran-b	hypre-IC(k)	1/12	161	137	45.1	378
	hypre-AMG	2/72	148	109	79.7	948
	hypre-PSAILS	52/81	166	156	28.8	384
	wsmp-DIRECT	0/1	20.7	20.7	20.7	20.7
Sgi_1M	hypre-IC(k)	0/12	202	110	33.9	585
	hypre-AMG	0/72	101	66.5	46.4	783
	hypre-PSAILS	15/81	91.7	29.7	16.3	399
	wsmp-DIRECT	0/1	32.2	32.2	32.2	32.2

Table 5: Time statistics for the direct and iterative solvers on 64 processors.

level-of-fill based incomplete Cholesky factorization (IC(k) [26]), sparse approximate inverse preconditioner (ParaSails [4]), and the algebraic multigrid preconditioner (BoomerAMG [25]) available in the Hypre [9] package offered the best chances of providing a most robust, high-performance, and memory efficient solution in a highly parallel distributed-memory environment.

The performance and effectiveness of most preconditioners is not only problem dependent, but is also highly sensitive to various tunable parameters of the preconditioner as well as the choice of matrix preprocessing steps such as ordering and scaling etc. Having chosen the "best" [14] iterative solver package for SPD systems, we attempted to use the best configurations of its preconditioners for each matrix for a comparison with the direct solver. Using different combinations of various parameters, as shown in Table 3, we crafted 12 configurations for the IC(k) preconditioner, 72 for BoomerAMG, and 81 for ParaSails[1] and attempted to solve all five systems on 64 processors of an IBM p5-575 cluster. The conjugate gradient (CG) solver was used. Diagonal scaling was performed on all matrices and a right-hand side vector of all 1's was used. A maximum of 2000 iterations were permitted and were stopped when the relative norm of residual dropped below 10^{-5}.

None of the configurations of any of the three preconditioners could solve systems with *Bstone-1*, *Lmco*, and *Ten-b* as coefficient matrices. The statistics for the memory and time used by various preconditioners for the other two matrices are shown in Tables 4 and 5, respectively. The tables show that for *Nastran-b*, Hypre-AMG had the most significant memory advantage over the direct solver. Both the median and the minimum memory was small and only two of its 72 configurations failed. However, even its best configuration was about four times slower than the direct solver. Hypre-PSAILS had a large number failures. Although its best configuration

[1]Not all combinations of parameters shown in Table 3 were used because some were not practical.

Matrix	Solver	Failure rate	Av. Mem	Med. Mem	Min. Mem	Max. Mem
Nastran-b	hypre-IC(k)	0/2	4.00e+09	4.00e+09	4.00e+09	4.00e+09
	hypre-AMG	0/4	1.44e+09	1.21e+09	1.04e+09	2.33e+09
	hypre-PSAILS	0/4	5.82e+09	5.82e+09	5.82e+09	5.83e+09
	wsmp-DIRECT	0/1	8.68e+09	8.68e+09	8.68e+09	8.68e+09
Sgi_1M	hypre-IC(k)	0/2	4.24e+09	4.24e+09	4.24e+09	4.24e+09
	hypre-AMG	0/4	1.64e+09	1.36e+09	1.16e+09	2.68e+09
	hypre-PSAILS	0/4	5.94e+09	5.94e+09	5.94e+09	5.94e+09
	wsmp-DIRECT	0/1	1.56e+10	1.56e+10	1.56e+10	1.56e+10

Table 6: Memory statistics for the direct and iterative solvers on 256 processors.

Matrix	Solver	Failure rate	Av. Time	Med. Time	Min. Time	Max. Time
Nastran-b	hypre-IC(k)	0/2	5.95	5.95	5.95	5.96
	hypre-AMG	0/4	32.3	32.5	24.2	39.9
	hypre-PSAILS	0/4	5.92	5.85	5.24	6.76
	wsmp-DIRECT	0/1	7.03	7.03	7.03	7.03
Sgi_1M	hypre-IC(k)	0/2	3.83	3.83	3.81	3.86
	hypre-AMG	0/4	17.4	17.6	14.6	19.7
	hypre-PSAILS	0/4	4.12	4.02	3.38	5.07
	wsmp-DIRECT	0/1	11.1	11.1	11.1	11.1

Table 7: Time statistics for the direct and iterative solvers on 256 processors.

performed well, the median and the average values were much worse. For *Sgi_1M*, all preconditioners worked reasonably well. Hypre-AMG and Hypre-PSAILS were competitive in median and minimum times and had a significant memory advantage over the direct solver.

Based on the results of our 64-CPU experiments, we chose the best configurations with respect to time and memory for *Nastran-b* and *Sgi_1M* and tried them on 256 CPUs for both the matrices. The 256-processor memory and time results are summarized in Tables 6 and 7, respectively. Since these were the best configurations for 64 CPUs, they performed very well compared to the direct solver. Two trends could be observed from these tables. First, going from 64 to 256 processors, all three preconditioners scale better than the direct solver. In fact, IC(k) tends to show superlinear speedup, for which we do not have an explanation yet (the number of iterations actually increased with the number of processors). The second observation is that the total memory usage of Hypre-PSAILS creeps up with the number of processors. As a result, it does not have a significant memory advantage over the direct solver on 256 CPUs, although, by virtue of its better scalability, the solution time of at least its best configuration is smaller than that of the direct solver for both the matrices.

To summarize the results of this section, iterative solvers are effective for only one or two of the five matrices, depending on the preconditioner and the number of processors used. Even in these cases, a careful selection of parameters is necessary, and it may not always be practical to perform extensive experimentation to select the best set of parameters in real-life situations. Note that these results are not meant to be indicative of iterative solvers' performance in general. The test data is not random, but is obtained from hand-picked applications that use direct solvers. The purpose of these experiments is to show that a preference for direct solvers in these applications is justified and that direct solvers, despite their cost, are indispensable for many ap-

#CPUs	Factorization Time [seconds]					
	wsmp−1T	wsmp−2T	wsmp−4T	wsmp−8T	SLU	MUMPS
1	68.	68.	68.	68.	465.	104.
8	18.	23.	20.	18.	91.0	29.4
16	12.	16.	16.	16.	52.7	17.3
32	7.1	11.	9.5	9.5	34.3	13.2
64	5.6	7.5	6.6	5.5	29.7	9.51
128	4.7	4.7	4.5	3.5	22.1	11.2
256	3.4	3.5	3.2	2.8	18.8	15.6
512	2.4	2.6	2.1	2.3	19.0	19.3
1024	1.8	2.1	2.1	1.7		

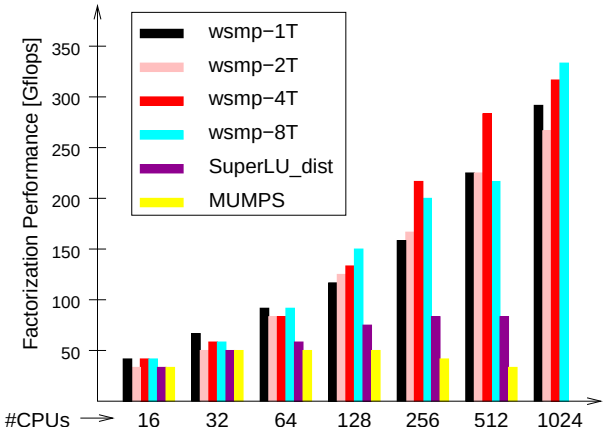

Figure 1: Factorization times and speeds of four different configurations of WSMP, SuperLU, and MUMPS for matrix *Bstone-1* on Power5+ cluster.

plications. Therefore, it is desirable to develop highly parallel algorithms for them and study their performance and scalability properties.

4 Comparison with Other Direct Solvers

In this section, by means of experiments on an IBM p5-575 cluster, we compare WSMP's symmetric direct solver with two well known distributed-memory parallel direct solvers, namely MUMPS [2, 3] and SuperLU_DIST [33, 34]. These three direct solvers use different parallelization approaches for sparse factorization. Two types of parallelism are available for factoring sparse matrices. The first, *task parallelism*, views the entire computation as a task-dependency graph and seeks to perform independent tasks in parallel. The task dependency graph for factoring symmetric sparse matrices happens to be a tree, which is known as the elimination tree [36]. *Data parallelism* is the second type of parallelism available and pertains to multiple processes performing a computational task such as panel factorization or an update operation in parallel with the data involved in the operation distributed among the processes. Gupta et al. [20] analytically showed that one of the factors that determines the scalability of parallel sparse factorization is whether task parallelism, or data parallelism, or both are exploited. When data parallelism is exploited, the scalability is quite sensitive to data distribution. A two-dimensional distribution, where both the rows and the columns of the matrix or its submatrices are partitioned, leads to more scalable parallel formulations than a one-dimensional distribution involving partitioning of either rows or columns only. WSMP uses both task parallelism and data parallelism, gradually switching from the former to the latter as the number of independent tasks declines and their size increases. In the data parallel part, it uses a two-dimensional distribution. MUMPS also uses both task and data parallelism; however, with the exception of the root supernode of the elimination tree, it uses a one-dimensional partitioning in the data-parallel part. SuperLU_DIST uses data parallelism only with a two-dimensional distribution of data. Another recent distributed-memory parallel symmetric direct solver PaStiX [10, 24] is based on data parallelism with a two dimensional partitioning. We are currently in the process of evaluating PaStiX, but expect its scalability to be similar to that of SuperLU_DIST due to the similarity of their parallelization approaches. Preliminary experiments show PaStiX to be significantly faster than SuperLU_DIST, but to scale at a slightly worse rate.

#CPUs	Factorization Time [seconds]					
	wsmp−1T	wsmp−2T	wsmp−4T	wsmp−8T	SLU	MUMPS
8	101.	97.9	98.7	102.	732.	144.
16	56.7	57.9	55.3	59.7	374.	93.0
32	32.9	31.9	32.5	32.1	203.	59.8
64	16.7	20.4	18.8	19.3	128.	50.8
128	12.9	10.7	11.8	12.8	74.5	46.8
256	7.55	6.42	6.68	8.21	50.1	52.3
512	4.59	3.60	4.72	4.97	52.2	88.5
1024	2.43	2.22	2.32	2.93		

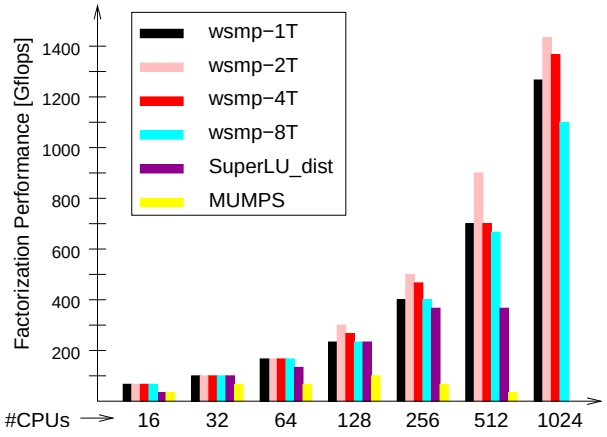

Figure 2: Factorization times and speeds of four different configurations of WSMP, SuperLU, and MUMPS for matrix *Lmco* on Power5+ cluster.

#CPUs	Factorization Time [seconds]					
	wsmp−1T	wsmp−2T	wsmp−4T	wsmp−8T	SLU	MUMPS
8	83.4	96.4	89.8	72.9	1264	141.
16	43.8	57.8	47.3	46.7	607.	99.4
32	28.0	35.7	30.4	29.4	331.	67.0
64	16.2	22.5	18.8	20.0	298.	45.1
128	9.71	12.9	11.2	12.5	119.	34.3
256	6.04	7.30	6.60	7.57	86.3	33.9
512	4.18	5.63	4.49	5.10	87.4	48.2
1024	2.75	3.24	2.58	2.77		

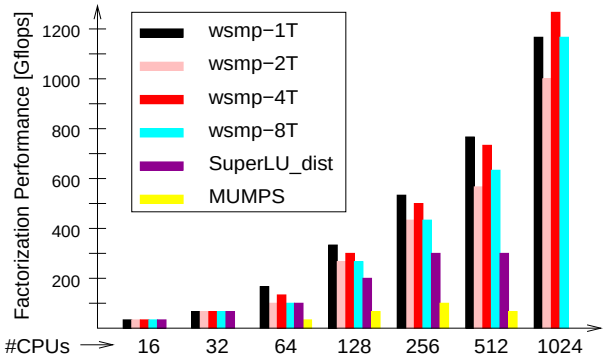

Figure 3: Factorization times and speeds of four different configurations of WSMP, SuperLU, and MUMPS for matrix *Nastran-b* on Power5+ cluster.

MUMPS and SuperLU_DIST are pure distributed-memory parallel solvers and perform best if the number of MPI processes is the same as the number of CPUs. WSMP and PaStiX, on the other hand, can use multi-threaded MPI processes. As a result, a given number of CPUs can be utilized via multiple combinations of MPI processes and threads. We experimented with four such combinations for WSMP and show the results of using 1, 2, 4, and 8 threads per MPI process in this section.

Figures 1–5 show the performance of WSMP, SuperLU, and MUMPS for factoring the five test matrices[2] on up to 1024 CPUs of the p5-575 cluster. The data columns and the bars corresponding to WSMP-kT in these figures denote that k threads were used by each of the p/k MPI processes for utilizing p CPUs. Prior to factorization, the matrices were permuted using the default ordering scheme for each package. By default, WSMP uses its native ordering [17], SuperLU uses AMD [1], and MUMPS uses PORD [40]. Note that unlike WSMP and MUMPS, which factor only a triangular part of the symmetric matrix, SuperLU performs an LU factorization of the entire matrix because it has been designed for general sparse matrices. As a result, it performs twice the number of operations needed to factor a symmetric matrix, and its performance should be viewed in the light of this. However, the focus of the experiments in this section is to study the relative

[2]In most cases, MUMPS did not finish factoring *Ten-b* within the two-hour limit specified for the batch jobs.

#CPUs	Factorization Time [seconds]					
	wsmp−1T	wsmp−2T	wsmp−4T	wsmp−8T	SLU	MUMPS
16	105.	116.	117.	119.	1612	233.
32	64.9	67.7	66.4	58.4	875.	136.
64	38.5	34.1	36.6	39.1	500.	87.5
128	25.8	20.3	20.8	21.0	278.	67.6
256	13.7	11.5	12.8	14.8	179.	56.5
512	7.81	7.62	8.22	8.45	158.	68.4
1024	4.48	3.97	4.18	5.30	157.	

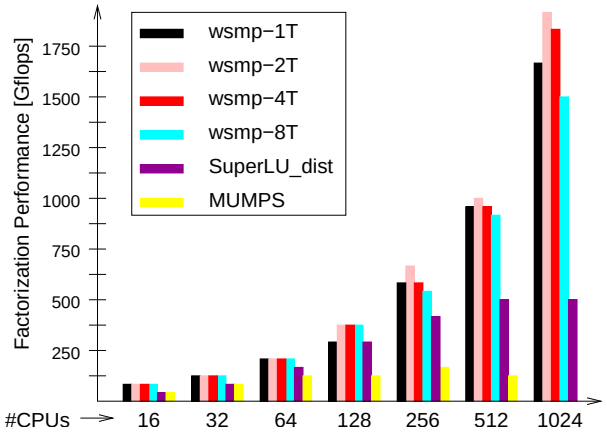

Figure 4: Factorization times and speeds of four different configurations of WSMP, SuperLU, and MUMPS for matrix *Sgi_1M* on Power5+ cluster.

#CPUs	Factorization Time [seconds]				
	wsmp−1T	wsmp−2T	wsmp−4T	wsmp−8T	SLU
16	415.	469.	458.	477.	6813
32	235.	280.	254.	252.	3514
64	143.	153.	145.	155.	1845
128	83.4	85.5	83.7	89.5	994.
256	58.0	43.9	46.7	54.2	565.
512	31.8	24.0	26.2	29.5	367.
1024	17.5	13.8	14.7	18.0	360.

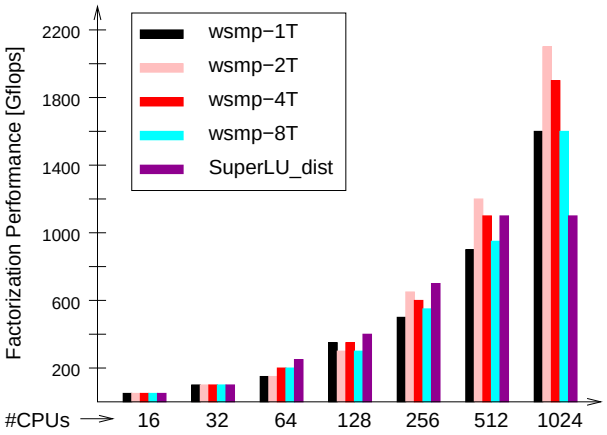

Figure 5: Factorization times and speeds of four different configurations of WSMP, SuperLU, and MUMPS for matrix *Ten-b* on Power5+ cluster.

scalability of the solvers; i.e., their capacity to deliver increasing speedups with increasing number of CPUs.

The results in Figures 1–5 show that WSMP's factorization scales the best, followed by that of SuperLU_DIST. The highest factorization speed of nearly 2.2 Teraflops was obtained for the *Ten-b* matrix with 512 MPI processes using two threads each. The number of threads per process affects the performance, and using multiple threads per process is almost always beneficial. In most cases, using 2 or 4 threads per MPI process delivers better performance than using single threaded processes. Performance appears to begin declining when 8 or more threads are used.

5 Performance on Cray XT/4 and Blue Gene/P

Having observed the speedups on up to 1024 CPUs of a p5-575 cluster, we tested the solver on two other massively parallel platforms, the Cray XT4 and IBM Blue Gene/P, on which we could access an even higher number of processors. On both these machines, each node consists of four cores in an SMP configuration—AMD Opteron on the XT4 and PowerPC 450 on BG/P. Therefore, we used one multithreaded MPI process on

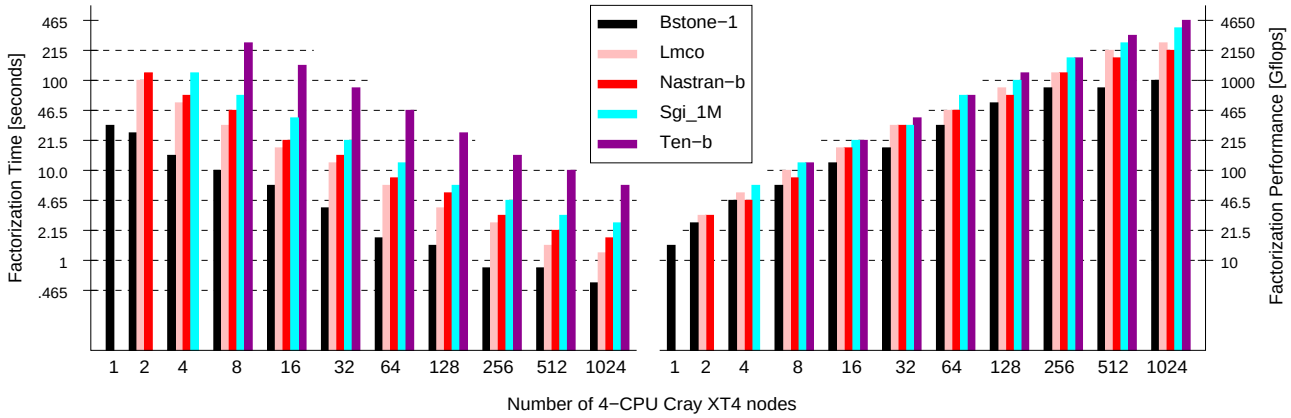

Figure 6: Factorization times and speeds for *Bstone-1*, *Lmco*, Nastran-b, *Sgi_1M*, and *Ten-b* on logarithmic scales on Cray XT/4.

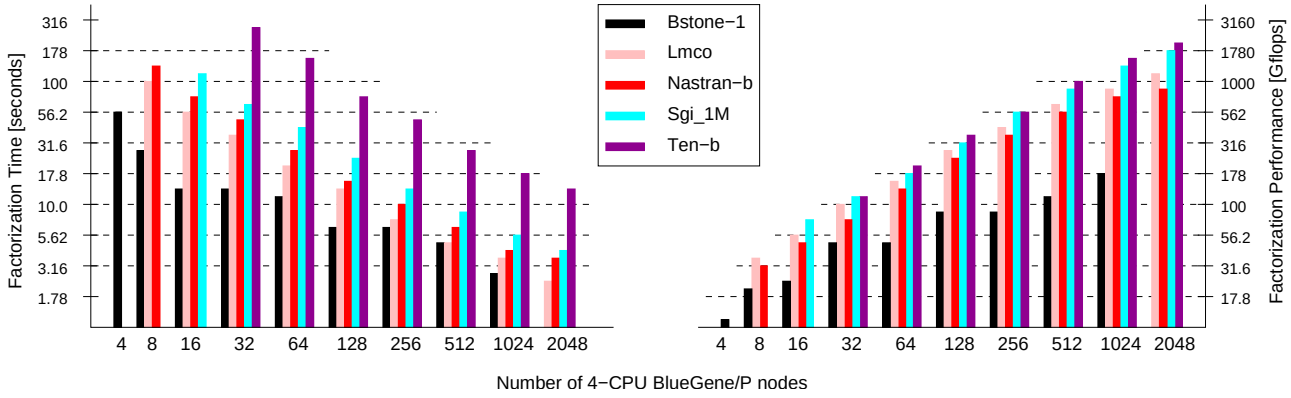

Figure 7: Factorization times and speeds for *Bstone-1*, *Lmco*, Nastran-b, *Sgi_1M*, and *Ten-b* on logarithmic scales on IBM Blue Gene/P.

each node and allowed it to use all four CPUs.

Figure 6 shows the factorization times and speed in Gigaflops for each of the five matrices in our test suite on the Cray XT4 on logarithmic scales. All experiments were conducted with 4 GB memory per node on 32 or more nodes and with 8 GB per node on fewer nodes. As the figure shows, with the exception of the smallest matrix *Bstone-1*, the speedup continues to increase all the way up to 1024 nodes (4096 CPUs). The highest performance of 4.6 Teraflops was obtained for the largest matrix *Ten-b*.

Figure 7 depicts the same data for the BG/P. Once again, with the exception of *Bstone-1*, the factorization time continues to decline up to 2048 nodes (8192 CPUs). Since BG/P nodes are slower than the XT4 nodes, the performance is proportionally lower. On an average, on the same number of nodes, the factorization code seems to run about three times faster on the XT4 than on BG/P. A peak performance of just over 2 Teraflops was obtained for *Ten-b* on 2048 nodes. The scalability of the code, however, seems to be similar on both machines. For example, Table 8 shows the speedups of going to 1024 nodes from 32 nodes on both machines for all five matrices. These are the maximum and the minimum number of nodes for which we have data for all matrices on both machines, and the speedup numbers are quite close.

Machine	Bstone-1	Lmco	Nastran-b	Sgi_1M	Ten-b
IBM BG/P	5.04	10.2	11.4	12.6	13.5
Cray XT4	6.33	10.5	8.65	11.3	13.2

Table 8: Ratio of factorization time on 32-nodes to that on 1024 nodes of BG/P and XT4.

6 Weak Scaling for a Model Problem

In the previous sections, we used five fixed matrices to study the performance and scalability of WSMP's symmetric sparse factorization. Solving a fixed problem using an increasing number of processors almost always results in diminishing efficiencies (typically defined as the ratio of speedup to the number of processors). In this section, we use matrices derived from artificial cubic 3-dimensional 7-point stencil finite-difference grids of varying sizes for a more in-depth investigation of the scaling properties of this solver. Note that, unlike the matrices used earlier in the paper, the model finite difference problem is easily solvable using iterative methods and a direct solver would not be used in practice. The purpose of the experiments described in this section is simply to use a set of matrices with varying size but similar structure.

Table 9 shows the performance of the factor and solve phases of WSMP's symmetric sparse direct solver over a fairly comprehensive set of matrices derived from the model cubic 3-D finite difference problem. The experiments were performed on a BG/P system. Each row of data corresponds to a given number p of BG/P nodes, where p ranges from 1 to 4096. WSMP uses as many MPI processes as the number of nodes used, and each multithreaded process uses all the four CPUs on each node. The table contains five columns of data corresponding to increasing values of n, the dimension of the 3-D cubic grid. Each major column of data (separated by double vertical lines) contains three subcolumns: the first one shows the value of n in bold font and the factorization efficiency E in a smaller font, the second one shows factorization time in seconds and speed in Gigaflops or Teraflops, and the third one shows the time and speed of the triangular solution phase for a single right-hand side vector. The values of n used in the table range between 20 and 232. Successive values of n are chosen such that the number of operations required for factorization increases by a factor of $2\sqrt{2}$ (explanation to follow in Section 7.2). The problem size or the total amount of work for the triangular solution phase grows as $O(p)$ along the columns. Five values of n were tried for each p. The values of n used in each row are shifted so that strong scalability results can be viewed along top-right to bottom-left diagonals and weak scalability results with the factorization problem size growing as $O(p^{1.5})$ can be viewed along the columns of the table.

Table 9 shows that factorization times for a fixed problem continue to decline up to 4096 nodes (16,384 cores) and the solve operation, which always takes well under a second to complete, continues to speed up until 2048 nodes. For the largest matrix (corresponding to $n = 232$) and the largest BG/P configuration with $p = 4096$, a maximum sustained speed of 7.05 Teraflops was obtained. To the best of our knowledge, this is the highest performance for sparse matrix factorization reported to date. A triangular factor of the matrix corresponding to $n = 232$ contains about 28 billion nonzeros. This amounts to less than 2% of the memory of a 4096-node BG/P with 4 Gbytes of memory per node. Hence, our largest test case is relatively small for a machine of this size, which can potentially handle much larger problems and deliver significantly higher speeds and efficiency for sparse matrix factorization. Currently, WSMP's preordering phase requires gathering all indices of the sparse matrix on a single node, which limits its ability to handle the matrix corresponding to the next value of $n = (2\sqrt{2} \times 232^6)^{1/6} \approx 276$. This limitation could have been overcome by using a fully

No. of nodes (p)	n E	Factor	Solve	n E	Factor	Solve	n E	Factor	Solve	n E	Factor	Solve	n E	Factor	Solve
1	**20** .29	.125 s 2.10 GF	.014 s .25 GF	**24** .40	0.28 s 2.97 GF	0.03 s 0.3 GF	**29** .49	0.74 s 3.59 GF	0.05 s 0.4 GF	**34** .61	1.64 s† 4.53 GF	0.08 s 0.4 GF	**41** .69	4.57 s 5.07 GF	0.15 s 0.5 GF
2	**24** .31	.185 s 4.50 GF	.015 s 0.5 GF	**29** .39	0.46 s 5.74 GF	0.03 s 0.7 GF	**34** .52	1.11 s‡ 7.70 GF	0.04 s 0.9 GF	**41** .59	2.72 s 8.68 GF	0.07 s 1.1 GF	**48** .69	6.09 s 10.2 GF	0.13 s 1.2 GF
4	**29** .29	0.33 s 8.44 GF	.015 s 1.2 GF	**34** .36	0.69 s 10.7 GF	0.03 s 1.3 GF	**41** .44	1.80 s 13.1 GF	0.05 s 1.5 GF	**48** .55	4.02 s 16.1 GF	0.08 s 2.1 GF	**58** .65	10.9 s 19.2 GF	0.14 s 2.5 GF
8	**34** .26	0.48 s 15.5 GF	0.02 s 1.6 GF	**41** .35	1.17 s 20.6 GF	0.04 s 2.1 GF	**48** .42	2.58 s$\oint$ 24.9 GF	0.06 s 2.8 GF	**58** .50	7.13 s 29.7 GF	0.10 s 3.3 GF	**69** .57	17.7 s 33.6 GF	0.20 s 3.7 GF
16	**41** .25	0.77 s 29.2 GF	0.03 s 2.7 GF	**48** .33	1.66 s† 38.9 GF	0.04 s 4.1 GF	**58** .41	3.82 s# 48.2 GF	0.06 s 5.5 GF	**69** .48	9.71 s 56.8 GF	0.09 s 7.6 GF	**82** .54	24.0 s 63.7 GF	0.19 s 7.6 GF
32	**48** .24	1.11 s‡ 57.1 GF	0.04 s 4.1 GF	**58** .30	2.64 s 70.4 GF	0.05 s 7.3 GF	**69** .39	5.95 s 92.9 GF	0.07 s 9.6 GF	**82** .45	13.8 s 107. GF	0.11 s 13. GF	**97** .53	36.5 s 124. GF	0.18 s 17. GF
64	**58** .25	1.75 s 117. GF	0.04 s 8.6 GF	**69** .33	3.81 s# 155. GF	0.06 s 12. GF	**82** .39	9.46 s 182. GF	0.09 s 17. GF	**97** .47	21.9 s$ 221. GF	0.14 s 23. GF	**116** .53	56.6 s 251. GF	0.22 s 29. GF
128	**69** .25	2.58 s$\oint$ 232. GF	0.06 s 13. GF	**82** .31	5.93 s 293. GF	0.08 s 19. GF	**97** .39	13.4 s 366. GF	0.11 s 28. GF	**116** .45	34.8 s 424. GF	0.17 s 40. GF	**138** .50	88.0 s 470. GF	0.27 s 48. GF
256	**82** .23	3.80 s# 436. GF	0.07 s 21. GF	**97** .31	8.32 s 579. GF	0.10 s 31. GF	**116** .38	20.6 s 712. GF	0.14 s 45. GF	**138** .44	50.2 s 838. GF	0.22 s 64. GF	**164** .50	127. s 941. GF	0.33 s 79. GF
512	**97** .21	5.86 s 0.81 TF	0.10 s .03 TF	**116** .28	13.4 s 1.07 TF	0.14 s .05 TF	**138** .35	31.4 s 1.31 TF	0.20 s .07 TF	**164** .42	75.1 s 1.58 TF	0.27 s .10 TF	**195** .46	199. s 1.74 TF	0.43 s .12 TF
1024	**116** .19	10.5 s 1.42 TF	0.14 s .05 TF	**138** .25	22.0 s$ 1.92 TF	0.19 s .07 TF	**164** .32	49.1 s 2.44 TF	0.26 s .10 TF	**195** .39	116. s 2.96 TF	0.38 s .13 TF	**232** .45	289. s 3.40 TF	0.56 s .18 TF
2048	**138** .16	17.3 s 2.39 TF	0.20 s .07 TF	**164** .22	35.5 s 3.31 TF	0.27 s .09 TF	**195** .28	81.6 s 4.22 TF	0.37 s .13 TF	**232** .33	185. s 5.04 TF	0.52 s .19 TF			
4096	**164** .13	30.7 s 3.92 TF	0.29 s .09 TF	**195** .18	63.0 s 5.33 TF	0.39 s .13 TF	**232** .23	139. s 7.05 TF	0.57 s .18 TF						

Table 9: Performance of factor and solve steps for various BG/P node counts and matrix sizes. The matrices are derived from an $n \times n \times n$ 7-point stencil 3-D finite difference mesh for several values of n ranging from 20 to 232.

distributed ordering such as ParMETIS [28]. Another consideration in using relatively small problem sizes for the large number of experiments required to gather the data for Table 9 was to conserve computational resources.

Just below each value of n in Table 9 is a fraction that represents the efficiency of factoring the matrix of dimension $N = n^3$ using the number of nodes p associated with that row of the table. Traditionally, parallel efficiency is defined as the ratio of parallel speedup to the number of processors. However, while scaling the problem size with the number of processors (or nodes), it is usually not possible to measure the speedup, which is the ratio of the parallel execution time to the best serial time for solving the same problem. The reason is that the problem attempted on p nodes may be too large to fit in the memory of a single node. In order to overcome this difficulty, we solved systems with increasing values of n on a single node and measured the speed $s_1(n)$ in Gigaflops. The largest system that could be solved on a single node corresponded to $n = 74$. The speed $s_1(74)$ was clocked at about 7.4 Gigaflops, which is little more than half of the theoretical peak for a BG/P node. We assume that $s_1(74)$ is the maximum speed that a single BG/P node can deliver on this problem. We then computed the factorization efficiency for each (p, n) pair as $s_p(n)/ps_1(74)$, where $s_p(n)$ is the measured speed in Gigaflops of factoring the matrix corresponding to the $n \times n \times n$ grid on p nodes. This efficiency computation also mitigates, to some extent, a usual complication in reconciling experimental and analytical performance and scalability results. The analysis typically considers factors like communication overhead and load-imbalance, but does not usually account for the fact that the speed at which computations are performed can change with the number of processors. Sometimes, the speed of computation improves with an increasing number of processors due to a larger net cache size; however, in most numerical applications, including sparse and dense matrix factorization, the computation speed declines with an increasing number of processors because of progressively shorter temporal and spatial streams of computation and data between the communication steps. As a result of computing the efficiency relative to the maximum obtainable speed on a single node, the single node efficiencies for $n < 74$ are less than the ideal value of 1.0, thus capturing the impact of smaller blocks of data. This makes it fairer to compare these with the efficiencies when the number of processors and n are increased.

7 Weak Scaling Analysis of the 3-D Model Problem

In this section, we review some prior analytical scalability results for solving sparse systems arising from problems involving discretization of 3-D domains, derive scalability results for additional scaling criteria, and compare the analytical results with the experimental results presented in Section 6.

7.1 Definitions and assumptions

We start by introducing the following terminology, which will be used in the remainder of the section. **Number of processes** (p), which is equivalent to the number of nodes in the context of this paper, is the number of MPI processes used to solve a given instance of the problem. Due to the hierarchical nature of parallelism used in the sparse factorization implementation being studied, we will use the number of processes or nodes in lieu of the number of processors, which is often used in the analysis of parallel algorithms. Each node may have multiple CPUs that are utilized by means of a multithreaded single-node code in our implementation. We regard a node as the basic processing element that hosts a single MPI process and communicates with other nodes. **Problem size** (W) is a measure of total amount of work required to solve the problem and is usually expressed

in terms of the number of basic operations needed to solve the problem. When expressed in order terms, it is the sequential time complexity of the problem. Since the problem size is related to the serial execution time by a constant, we also use W to denote the execution time on a single process. A coefficient matrix of dimension $N = n^3$ resulting from an $n \times n \times n$ grid requires $O(N^2)$ or $O(n^6)$ operations [13, 35]. Similarly, the problem size for the triangular solution phase is $O(N^{4/3})$ or $O(n^4)$. **Parallel Execution Time** (T_P) is the time to solve a problem in parallel and is a function of the number of processes p and the problem size W. When the problem size is expressed as a function of a parameter of the input size, such as the matrix dimension N in our case, then T_P can be expressed as a function of p and N. Gupta et al. [20] have shown that for the model problem being considered in this section, a lower bound on T_P of WSMP's sparse factorization algorithm is $\Omega(\frac{N^2}{p} + \frac{N^{4/3}}{\sqrt{p}})$. The first term is due to the computation that each node performs and the second term is a lower bound on the time spent in communication. **Total Parallel Overhead** (T_o) is the sum of all the overhead incurred due to parallel processing by all the processes. In our case, a lower bound on the total overhead due to the communication term in T_P would be $\Omega(N^{4/3}\sqrt{p})$. **Speedup** (S) is the ratio of the serial execution time W of the fastest serial algorithm to the parallel execution time T_P. **Efficiency** (E) is usually expressed as the ratio of speedup (S) to the number of processors (p). Thus, $E = \frac{W}{pT_P} = \frac{1}{1+\frac{T_o}{W}}$.

7.2 Scalability analysis

It is well known that for a problem instance of a fixed size, the speedup of a parallel algorithm does not continue to increase with increasing number of processes and tends to saturate or peak at a certain value. The size of the problem must be increased with the number of processes for an increasing number of processes to be utilized efficiently—a concept loosely known as *weak scaling.* Various criteria have been studied for determining the rate at which the problem size must be increased (many of these are surveyed by Kumar and Gupta [29]). In this section, we discuss a few of these in the context of factorization of sparse symmetric matrices resulting from discretizing 3-D domains.

7.2.1 Fixed efficiency scaling

Kumar and Rao [30] developed a scalability metric relating the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors. This metric is known as the *isoefficiency function.* Gupta et al. [20] have shown that the problem size must increase at least as fast as $\Omega(p^{1.5})$ for WSMP's sparse Cholesky factorization to maintain a fixed efficiency.

7.2.2 Fixed memory scaling

Increasing the problem size with the number of processes such that the memory requirement for each process remains constant [42, 43] is a practical way to scale the problem size since the total available memory is proportional to the number of processes, at best. For the problem at hand, the total memory requirement is $O(N^{4/3})$ and the problem size W is $O(N^2)$. Incidentally, $N^{4/3} \propto p$ implies $N^2 \propto p^{1.5}$. Therefore, fixed-memory scaling is equivalent to the fixed-efficiency scaling.

7.2.3 Fixed workload scaling

Gustafson et al. [22, 23] were the the first to experimentally demonstrate that by scaling up the problem size, near-linear speedup could be obtained on up to 1024 processors. They introduced the *scaled speedup* metric,

which is defined as the speedup obtained when the problem size is increased linearly with the number of processes. Scaled speedup increases linearly with the number of processes only when the isoefficiency function is linear, which is not the case for parallel sparse factorization. In this case, $E = \frac{1}{(1+\frac{T_o}{W})} = \frac{1}{1+Kp^{1/6}}$ for some constant K because $T_o \propto N^{4/3}\sqrt{p}$ and $W \propto N^2 \propto p$. Thus, the efficiency can be expected to progressively decline as the problem is scaled in proportion to p.

7.2.4 Fixed time scaling

Under fixed time scaling, first studied in detail by Worley [45], the problem size is increased with p such that the parallel execution time T_P remains constant. Sun and Gustafson [41] describe a similar concept, called *sizeup*, which is the ratio of the size of the problem solved on the parallel computer to the size of the problem solved on the sequential computer in a given fixed amount of time. For our problem, the expression for T_P has two asymptotic terms, $\frac{N^2}{p}$ and $\frac{N^{4/3}}{\sqrt{p}}$. None of these should grow asymptotically in order to keep T_P fixed. The more restrictive of the two conditions on problem size growth with respect to p results from the second term and dictates that $N^{4/3} \propto \sqrt{p}$, or the problem size $W \propto N^2$ grows no faster than $O(p^{3/4})$ to keep T_P from increasing.

7.2.5 Fixed speed scaling

Sun and Rover [44] define *isospeed* or fixed-speed scaling, where the problem size is increased with p such that the average unit speed of computation remains constant. The isospeed metric for a parallel algorithm-machine combination can only be determined experimentally by means of data of the form shown in Table 9.

7.3 Worst case scalability analysis

The various weak scalability analyses presented in Section 7.2 are based on Gupta et al.'s [20] $\Omega(N^{4/3}\sqrt{p})$ lower bound on the total communication overhead. These analyses represent the best-case scenario for two main reasons. First, Gupta et al.'s derivation assumes a hypercube or a two-dimensional (2-D) mesh interconnection for the target parallel computer. In our case, however, the BG/P machine employs a 3-D torus network. Secondly, they consider communication overhead only and do not account for the overhead due to load-imbalance. The latter, unfortunately, is hard to analyze because it depends on the permutation generated by the ordering heuristic[3].

It may be possible to reorganize the communication in the algorithm to make optimal use of the torus network; however, in our current implementation, the above bound on the communication overhead is valid either for a hypercube interconnect, or on a network in which message traversal time is independent of the source and destination nodes, as long as there is sufficient bandwidth to handle all the communication volume. We use a recursive bisection based mapping of MPI processes to the BG/P nodes such that the subtree-to-subcube mapping [20] of the elimination tree works naturally on the 3-D submeshes of the machine. However, within the nodes of a $k \times k \times k$ 3-D submesh, the communication pattern for the factorization steps is that of a $k^{3/2} \times k^{3/2}$ 2-D mesh. Folding a $k^{3/2} \times k^{3/2}$ 2-D logical mesh inside a $k \times k \times k$ 3-D physical mesh or torus causes congestion by a factor of $k^{1/6}$ along some edges of the physical mesh. BG/P uses adaptive routing for messages longer than a certain limit. However, the adaptive routing algorithm can choose only one of the

[3] A permuted version of the original matrix is typically factored in order to minimize memory and CPU time [13].

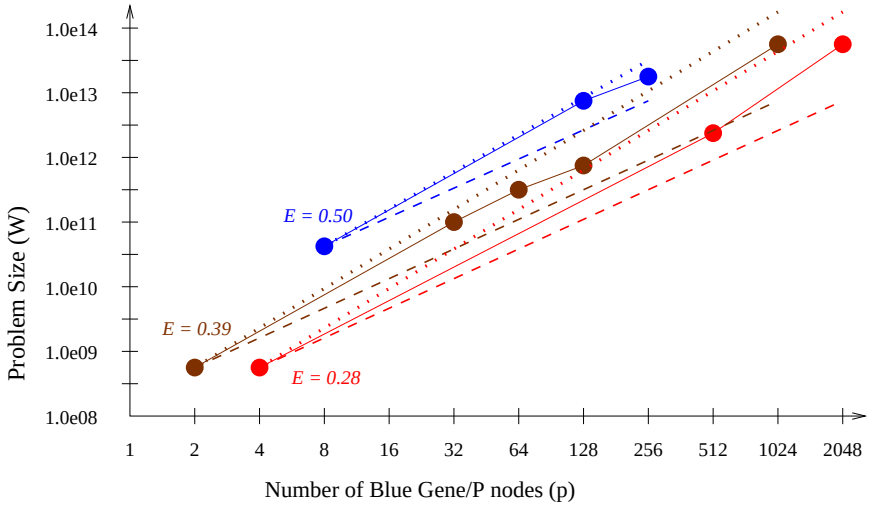

Figure 8: Some observed isoefficiency curves from the data in Table 9. The dashed and dotted lines show hypothetical curves corresponding to isoefficiency functions of $O(p^{1.5})$ and $O(p^2)$, respectively.

shortest paths between the communicating nodes. While folding a 2-D logical mesh into a 3-D mesh or torus, the congestion occurs along paths between pairs of nodes that share two of the three spatial coordinates. Since there is only one shortest path between such pairs of nodes, adaptive routing is ineffective. This congestion could potentially add a factor of $O(p^{1/6})$ to the total overhead and make it $O(N^{4/3}p^{2/3})$. Note that, this is the worst case scenario because it assumes that computation and communication are perfectly synchronized on all nodes and all $k^{1/6}$ messages along the paths with potential congestion overlap completely. In reality, the extra overhead due to congestion is likely to be probabilistic in nature, with virtually all messages getting through without much congestion for small values of k, but experiencing increasing collisions as k increases.

The bounds for various weak scaling methods under the worst-case communication overhead assumption can be derived along the lines of the analyses in Section 7.2. To maintain a fixed efficiency, the ratio of T_o and W must constant, which leads to an isoefficiency function of $O(p^2)$ when $T_o = O(N^{4/3}p^{2/3})$. If the $O(N^2)$ problem size is increased as $O(p^2)$ in order to maintain a fixed efficiency, then the $O(N^{4/3})$ memory requirement grows as $O(P^{4/3})$ and the superlinearly growing memory requirement makes it impossible to maintain fixed efficiency within fixed memory per process. Scaling the problem with fixed workload per process will result in a efficiency decline at the rate of $\frac{1}{1+Kp^{1/3}}$ for some constant K. Finally, for fixed time scaling, the problem size can be increased only as $O(\sqrt{p})$.

7.4 Comparison of empirical and analytical results

In Sections 7.2 and 7.3, we have derived expressions for several weak scaling metrics under the best- and the worst-case communication overhead assumptions. In this section, we compare these analytical results with the experimental data collected in Table 9.

Figure 8 shows three isoefficiency plots based on the (p, n) pairs that we could find in Table 9 with almost the same efficiency values. The figure also shows two hypothetical isoefficiency curves starting from the point with the smallest p of each observed curve. These curves, represented by dashed and dotted lines, correspond to isoefficiency functions of $O(p^{1.5})$ and $O(p^2)$, respectively, which are the isoefficiency functions derived from

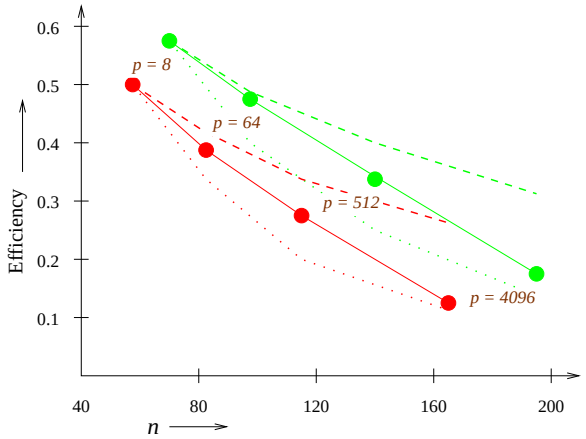

Figure 9: Efficiency as a function of 3-D grid dimension n when the problem size is scaled in proportion to the number of nodes p. The dashed and dotted lines correspond to the analytical best- and worst-case rates of efficiency declines in the fixed workload per node scaling.

the worst- and the best-case expressions for communication overhead. The figure shows that the observed rate at which the problem size increases with p to maintain a fixed efficiency lies between $O(p^{1.5})$ and $O(p^2)$. A close observation of the efficiency numbers along the columns of Table 9 (where the problem size is increasing at the rate of $p^{1.5}$) reveals that efficiency stays mostly constant or declines very slowly in the middle rows of the table. These rows roughly correspond to values of p between 8 and 512, and the observed isoefficiency appears to be close to the best-case $O(p^{1.5})$ growth rate of problem size in this range of p. For $p > 1024$, the isoefficient problem size growth rate appears to approach the worst-case of $O(p^2)$, possibly due to increased congestion, as discussed in Section 7.3.

Recall that we have computed the efficiency values shown in Table 9 by dividing the observed per process speed by 7.4 Gigaflops, the highest speed observed on a single process. Therefore, the isospeed and isoefficiency scalings are identical.

Figure 9 shows how the efficiency declines as the problem size is increased in proportion to the number of processes p, such that each process performs the same amount of work. The top curve corresponds to (p, n) values of (8,69), (64,97), (512,138), and (4096,195) and the bottom curve corresponds to (8,58), (64,82), (512,116), and (4096,164). These points represent a fixed work load per process because as p increases by a factor of 8, n increases by a factor of $\sqrt{2}$. Recall that $W \propto N^2 = n^6$. Sections 7.2 and 7.3 give the best- and the worst-case rates of efficiency declines, which the figure depicts by the dashed and the dotted lines, respectively. Once again, the observed data lies between the best- and worst-case analytical values.

To observe fixed time scaling, we chose (p, n) pairs from Table 9 with almost the same factorization times (see underlined factorization times in the table). The factorization time is 1.11 seconds for (2,34) and (32,48), it is roughly 3.8 seconds for (16,58), (64,69), and (256,82), and it is 2.58 seconds for (8,48) and (128,69). More groups like this can be found and it can be verified easily that in each case, the problem size increases at the rate of $p^{3/4}$. For example, for the 1.11 and 2.58 second case, a 16-fold increase in p is accompanied by an 8-fold increase in the problem size. Similarly, for the 3.8 second case, each 4-fold increase in p is accompanied by an increase in problem size by a factor of $2\sqrt{2}$. Note that an $O(p^{3/4})$ fixed time rate of problem size increase was predicted by the analysis in Section 7.2. If we observe any (p, n) pairs corresponding to the worst-case fixed time analysis of Section 7.3, say (64,116) and (4096,164), we see that the factorization T_P reduces from

56.6 seconds to 30.7 seconds. This suggests that the problem size can increase at a rate faster than $O(\sqrt{p})$ to maintain a fixed T_P, even for the highest values of p used.

8 Concluding Remarks and Future Work

We have presented a fairly comprehensive set of experimental and analytical performance and scalability results for a direct sparse linear solver, with particular emphasis on sparse factorization—the dominant phase of the solver in terms of time and memory consumption. We have demonstrated unprecedented levels of both performance and scalability. This performance was obtained without any architecture-specific tuning of the software. In fact, we have shown similar scalability results on three very different massively parallel computers. In the future, we plan to investigate the performance and scalability of this solver on commodity and popular x86_64 clusters. We also plan to study its scalability on Petascale systems, such as NCSA's upcoming Blue Waters [12] system and compare its performance and scalability with PaStiX [10, 24]. We hope that some of these investigations will help us understand whether the decline in efficiency at more than 1024 nodes (4096 CPUs) of BG/P observed in Section 6 is a result of some interaction of algorithmic and architectural properties like congestion caused by the 2-D mesh communication pattern within process subgroups, or if it is due to some other factor. We also plan to use performance analysis and tracing tools [31] to further understand and improve performance.

Acknowledgements

We would like to thank Ritske Huismans, Eirik Thorsnes, Halvor Utby, and the University of Bergen for providing access to the Cray XT4, NCSA and the Texas A&M Supercomputing Center for access to the p5-575 clusters, and IBM for access to the BG/P. We would also like to thank Yasuo Osawa and Yoshinori Shimoda for the matrix *Bstone-1*, Naga Nagarajan for *Lmco*, Doug Petesch for *Nastran-b*, and Miguel Cavaliere and Dolores Demarco for *Ten-b*.

References

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] Patrick R. Amestoy, Iain S. Duff, Jacko Koster, and J. Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] Patrick R. Amestoy, Iain S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computational Methods in Applied Mechanical Engineering*, 184:501–520, 2000.

[4] Edmond Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *International Journal of High Performance Computing Applications*, 15(1):56–74, 2001.

[5] MSC Software Corporation. *MSC Software Producs: Engineering Analysis*. Santa Ana, CA. *http://www.mscsoftware.com/products/core_products.cfm?Q=131&Z=396*.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[8] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, 1983.

[9] Robert D. Falgout and Ulrike Meier Yang. *hypre*, High Performance Preconditioners: Users manual. Technical report, Lawrence Livermore National Laboratory, 2006. Paper appears in ICCS 2002.

[10] Mathieu Faverge and Pierre Ramet. Dynamic scheduling for sparse direct solver on NUMA architectures. In *Proceedings of PARA'2008*, Trondheim, Norway.

[11] John Fettig, Seid Koric, and Nahil Sobh. A comparison of direct and iterative linear sparse solvers in computational solid mechanics. In *International Conference on Preconditioning Techniques for Large Sparse Matrix Problems*, Napa, CA, 2003.

[12] National Center for Supercomputing Applications. *Blue Waters: Sustained Petascale Computing*. University of Illinois, IL. *http://www.ncsa.uiuc.edu/BlueWaters*.

[13] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, NJ, 1981.

[14] Thomas George, Anshul Gupta, and Vivek Sarin. An empirical analysis of iterative solver performance for SPD systems. Technical Report RC 24737, IBM T. J. Watson Research Center, Yorktown Heights, NY, January 2009. A short version submitted to *ACM Transactions on Mathematical Software*.

[15] John R. Gilbert and Sivan Toledo. An assessment of incomplete-LU preconditioners for nonsymmetric linear systems. *Informatica*, 24(3):409–425, 2000.

[16] Anshul Gupta. A shared- and distributed-memory parallel sparse direct solver. *Applicable Algebra in Engineering, Communication, and Computing*, 18(3):263–277, 2007.

[17] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development*, 41(1/2):171–183, January/March, 1997.

[18] Anshul Gupta. WSMP: Watson sparse matrix package (Part-I: Direct solution of symmetric sparse systems). Technical Report RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000. *http://www.cs.umn.edu/~agupta/wsmp*.

[19] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSMP: A high-performance serial and parallel symmetric sparse linear solver. In Bo Kagstrom, Jack J. Dongarra, Eric Elmroth, and Jerzy Wasniewski, editors, *PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems*. Springer Verlag, 1998.

[20] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[21] Anshul Gupta and Vipin Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95 Proceedings*, December 1995.

[22] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[23] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[24] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, 2002.

[25] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS*, 41(1):155–177, 2002.

[26] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2000.

[27] Mahesh Joshi, Anshul Gupta, George Karypis, and Vipin Kumar. Two-dimensional scalable parallel algorithms for solution of triangular systems. In *Proceedings of the 1997 International Conference on High Performance Computing (HiPC)*, 1997.

[28] George Karypis and Vipin Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report TR 97-060, Department of Computer Science, University of Minnesota, 1997.

[29] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[30] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[31] Gary Lakner et al. *IBM Systems Blue Gene Solution: Performance Analysis Tools*. IBM, 2008. *http://www.redbooks.ibm.com/abstracts/redp4256.html*.

[32] Gary Lakner and Carlos P. Sosa. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM, 2008. *http://www.redbooks.ibm.com/abstracts/sg247287.html*.

[33] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Supercomputing '98 Proceedings*, 1998.

[34] Xiaoye S. Li and James W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[35] R. J. Lipton, D. J. Rose, and Robert E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[36] Joseph W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[37] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[38] J. W. Ruge and K. Stüben. Multigrid methods. In Stephen F. McCormick, editor, *Frontiers in Applied Mathematics*, volume 3, pages 73–130. SIAM, Philadelphia, PA, 1987.

[39] Yousef Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, 2003.

[40] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT Numerical Mathematics*, 41(4):800–841, 2001.

[41] Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17(2):1093–1109, 1991.

[42] Xian-He Sun and Lionel M. Ni. Another view of parallel speedup. In *Supercomputing '90 Proceedings*, pages 324–333, 1990.

[43] Xian-He Sun and Lionel M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993.

[44] Xian-He Sun and Diane Thiede Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, 1994.

[45] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990.