Performance Evaluation and Enhancement of *SuperLU_DIST* 2.0*

Xiaoye S. Li
Lawrence Berkeley National Laboratory

Yu Wang
Illinois Institute of Technology

August 29, 2003

Abstract

We present the runtime comparison of the two versions of *SuperLU_DIST*, using up to 128 processors of the IBM SP at NERSC. One version provides the global input interface, and another provides the distributed input interface. The comparison includes the total runtime of the solver with both 32-bit and 64-bit addressing modes, the time breakdown for different phases of the solver. We also present an in-depth comparison of four sparse matrix-vector multiplication methods in the context of iterative refinement. Finally, we describe our Fortran 90 interface that enhances the usability of the software.

1 Introduction

SuperLU_DIST [2, 4] is a distributed-memory sparse direct solver for large sets of linear equations $AX = B$. It has gone through two versions of release: Ver 1.0 and Ver 2.0. The major difference between them is that Ver 1.0 uses a *global* input interface, whereas Ver 2.0 uses a *distributed* input interface. In other words, in Ver 1.0, every processor needs to have the whole global input matrices A and B, and the sparse matrix A is in compressed column format (a.k.a. Harwell-Boeing format). In Ver 2.0, each processor only owns a block of consective rows of the input matrices, and the local part of sparse A is in compressed row format. Interprocessor communication is needed to convert the input distribution to the internal data distribution. In this report, we evaluate the performance of these two versions of *SuperLU_DIST* and attemp to further improve the performance of *SuperLU_DIST* Ver 2.0.

The rest of this paper is organized as follows. In Section 2, we first review the main implementation differences between the two versions of the software. In Section 3, we discuss the results obtained from the extensive experiments we have conducted to evaluate the performance of *SuperLU_DIST*. Section 4 describes the Fortran 90 interface we have developed. A brief conclusion is given in Section 5.

2 SuperLU_DIST

In this section, we briefly review the *G*ESP algorithm (Gaussian Elimination with Static Pivoting [5]) used in *SuperLU_DIST*. Please refer to [4] for more details. The algorithm consists of the

*This work was supported in part by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098, and was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020, NSF Grant No. ACI-9813362.

Xiaoye S. Li, xsli@lbl.gov, Lawrence Berkeley National Lab, MS 50F-1650, One Cyclotron Rd., Berkeley, CA 94720.
Yu Wang, wangyu1@iit.edu, Department of Computer Science, Illinois Institute of Technology, 10 W. 31st St., Chicago, IL 60616

following steps:

Algorithm 1 *The GESP algorithm*

1. Perform row/column equilibration: $A \leftarrow D_r A D_c$, where D_r and D_c are diagonal matrices chosen so that the largest entry of each row and column is ± 1.
2. Perform row permutation: $A \leftarrow P_r A$, where P_r is chosen to make the diagonal large compared to the off-diagonal.
3. Find a column permutation P_c to preserve sparsity: $A \leftarrow P_c A P_c^T$
4. Perform symbolic analysis to determine the nonzero structures of L and U.
5. Distribute L and U data structures onto the processor grid ($P = nprow \times npcol$).
6. Factorize $A = LU$ with control of diagonal magnitude.
7. Perform triangular solutions using L and U.
8. If needed, perform iterative refinement.

As can be seen in Algorithm 1, the input matrix A goes through a sequence of transformations before it is factorized. Comparing Ver 1.0 and Ver 2.0, some of the above steps are identical, including steps 2, 3, 4 and 6. Their algorithmic descriptions were given elsewhere [1, 4].

We now describe the implementation differences between the two versions. These include steps 1, 5, 7 and 8. For convenience, we associate with each step a label which will be used later when we present the timing results. We also list the corresponding double-precision routine name in each version.

Step 1 EQUIL (Ver 1.0: *dgsequ*, Ver 2.0: *pdgsequ*)
In both versions, the scaling factors (diagonals of D_r and D_c) are replicated on each processor. In Ver 1.0, the scaling factors are computed by processor 0, and then broadcasted to all the other processors. In Ver 2.0, they are computed in parallel.

Step 5 DISTRIBUTE (Ver 1.0: *ddistribute*, Ver 2.0: *pddistribute*)
In both versions, after symbolic factorization, the supernodal graph of L and the skeleton graph of U are replicated on all the processors. Based on these graphs, each processor sets up the local L and U data structures by extracting only the part of the graphs the processor owns. The difference between the two versions lies in how the initial values of A are loaded into the L and U data structures. In Ver 1.0, this is a simple matter of extracting the local part of A from the global A, because global A is available on every processor. In Ver 2.0, however, data re-distribution is needed, because the input A is by one-dimensional block mapping, whereas L and U are by two-dimensional block-cyclic mapping. This re-distribution involves all-to-all communication.

Step 7 SOLVE (Ver 1.0: *pdgstrs_Bglobal*, Ver 2.0: *pdgstrs*)
In the beginning of the solution phase, we need to distribute the right-hand side B on the processors who own the diagonal blocks of L and U. We call these processors diagonal processors. In the end, the solution vector resides on the diagonal processors. We need to re-distribute it to the right-hand side B. Ver 1.0 and 2.0 differ in the way the re-distribution works, because B is distributed differently on input. In Ver 1.0, B is globally available on each processor. So each diagonal processor can simply extracts the local part of B to start the solve. In the end, the diagonal processors "broadcast" their local parts to all the other processors, so X is globally available on each processor. In Ver 2.0, B is distributed by block rows among all the processors. Before triangular solve, we re-distribute B to the diagonal processors. After the solve, we perform another re-distribution from the diagonal processors to all processors, so that X is distributed by block rows among all the processors.

Step 8 REFINE (Ver 1.0: *pdgsrfs_ABXglobal*, Ver 2.0: *pdgsrfs*)
The kernels of iterative refinement are sparse matrix-vector multiplication (SMVM) to compute residual $r = b - Ax$ and triangular solve to compute the correction term. The triangular solve works the same as described above. For Ver 1.0, SMVM is easy, because each processor has the global data A and x, it perform multiplication using only part of the global data. For Ver 2.0, each processor computes a block of r that it owns. There is no need to communicate A if we use inner-product version of the multiplication algorithm. However, there is a need to communicate x. The algorithm consists of a setup phase and an actual multiplication phase. In the setup phase, we processor the graph of A, and performs all-to-all communication to determine for each processor which x components need to be sent to which other processors. After this phase, the communication schedule for x is saved on each processor. In the actual multiplication, we use this communication schedule to transfer x. Note that the setup is needed only once before the iterative refinement loop.

3 Performance Evaluation

In this section, we evaluate the performance of *SuperLU_DIST* on the IBM POWER3 SMP machine at NERSC. The processors are distributed among 380 compute nodes with 16 processors per node. Each compute node has between 16 and 64 GBytes of memory. Each processor is clocked at 375 MHz and has a peak performance of 1.5 Gflops. In all of our runs, we use as many processors as possible per node. We use a set of large matrices from various application domains. The matrices are available from University of Florida Sparse Matrix collection[1], the forthcoming Rutherford-Boeing Sparse Matrix Collection [7] [2], and the industrial partners of the PARASOL Project [3]. The last three matrices in the table are from the next-generation accelerator design project at SLAC. The characteristics of these matrices are presented in Table 1. It includes the matrix order, the number of nonzeros in the matrix A, the number of nonzeros in the factors L and U using minimum degree (`MMD`) ordering [8] on $A^T + A$, the number of supernodes N, and the number of floating-point operations. For the three largest matrices from SLAC, we also used the nested dissection (`ND`) ordering from Metis [9]. The last column gives the average number of nonzeros per supernode, $nnz(L+U)/N$, which is a certain measure of the sparsity of the filled matrix.

Matrix	Order	$nnz(A)$	$nnz(L+U)$ $\times 10^6$	N	$Flops(F)$ $\times 10^9$	$nnz(L+U)/N$ $\times 10^3$
bbmat	38744	1771722	36.1	7128	26.28	5.06
fidapm11	22294	623554	25.6	2610	24.11	9.81
wang4	26064	177196	10.7	6961	8.79	1.54
twotone	120750	1224224	11.4	38939	7.59	0.29
mixing-tank	29957	1995041	44.6	2538	76.84	17.57
inv-extrusion-1	30412	1793881	30.2	4129	32.46	7.31
ecl32	51993	380415	41.9	19168	60.74	2.19
ir	186230	2910202	(`MMD`) 132.7	25778	148.48	5.15
			(`ND`) 89.8	31498	66.19	2.85
dds.quadratic	380698	15844364	(`MMD`) 642.5	36877	2120.75	17.42
			(`ND`) 325.0	41362	491.04	7.86
dds15	834575	13100653	(`MMD`) 875.3	114929	1576.43	7.62
			(`ND`) 526.6	141060	600.58	3.73

Table 1: Characteristics of the benchmark matrices.

[1] http://www.cise.ufl.edu/research/sparse/matrices/
[2] http://www.cse.clrc.ac.uk/nag/hb/
[3] EU ESPRIT IV LTR Project 20160, matrices available at http://www.parallab.uib.no/parasol/

3.1 Overall Comparison Between Ver 1.0 and Ver 2.0

In the first set of experiments, we run both versions of *SuperLU_DIST* with the test matrices using different number of processors. The running times in seconds of each step of *SuperLU_DIST* are shown in Tables 2 to 6. Since steps 2 (row permutation), 3 (column permutation), 4 (symbolic analysis) and 6 (numerical factorization) are the same in both versions, in most of the tables (Tables 3 to 6) we only list the running times of the other four steps. To show the relative running time of all steps, we also list the running time of steps 2, 3, 4, 6 in Table 2. In the second columns of all these tables, "dis" denotes Ver 2.0 in which each processor has the distributed input matrix and "glb" denotes Ver 1.0 in which each processor has the global input matrix. All running time listed in the talbes are in seconds.

Matrix	Version	Step 2 (MC64)	Step 3 (MMD)	Step 4 (SymFact)	Step 6 (Factor)	Mflops in Step 6
bbmat	glb	1.83	6.97	1.35	7.44	3424.84
	dis	1.74	6.99	1.35	7.39	3555.82
fidapm11	glb	2.23	1.08	0.85	4.83	5557.03
	dis	2.21	1.04	0.81	4.65	5190.35
wang4	glb	0.12	0.35	0.35	2.27	3870.40
	dis	0.11	0.34	0.35	2.42	3634.56
twotone	glb	0.91	6.48	0.96	18.01	421.41
	dis	0.82	6.47	0.94	17.47	434.44
mixing-tank	glb	1.34	1.17	1.32	9.37	8509.45
	dis	1.32	1.18	1.33	9.30	8261.58
inv-extrusion-1	glb	5.30	1.54	1.47	6.14	4681.55
	dis	5.43	1.53	1.28	7.40	4383.50
ecl32	glb	0.26	1.38	1.35	9.31	6524.30
	dis	0.24	1.39	1.36	9.26	6561.07
ir	glb	1.65	4.69	3.99	18.78	7910.93
	dis	1.42	4.95	4.01	19.24	7723.66
dds.quadratic	glb	7.24	11.05	18.10	67.69	31441.25
	dis	6.36	11.11	18.14	67.40	31573.26
dds15	glb	7.42	34.67	25.56	104.36	15196.59
	dis	6.49	36.68	25.82	106.06	14952.87

Table 2: The running time of steps 2 (row permutation), 3 (column permutation), 4 (symbolic analysis) and 6 (numerical factorization) of Ver 1.0 and Ver 2.0. The last column shows the Mflops in the numerical factorization step. Here, for the first seven matrices $P = 4 \times 4$, while for the last three large matrices $P = 8 \times 16$.

We first look at the time difference of the individual steps. In the EQUIL step, Ver 1.0 is clearly slower, because there is no parallelism, and the broadcast time increases with increasing numbers of processors. For Ver 2.0, we get some speedup when the number of processors is small, but time is flat for larger numbers of processors because the communication starts to dominate the computation. Note that EQUIL usually takes very small fraction of the total runtime, so it is not critical to improve its speed.

In the DISTRIBUTE step, both versions achieve good speedup. Ver 1.0 has more local memory access, without the need for any communication. Whereas Ver 2.0 has less data access, but needs to do much all-to-all communication for data re-distribution. Ver 2.0 is expected to be slower than Ver 1.0. But it is interesting to see that Ver 2.0 is only slightly slower in many cases, even with 128 processors and for the largest problems. The similar situation also happens for the SOLVE step.

In the REFINE step, in most cases, Ver 2.0 is clearly a little bit faster than Ver 1.0. The reason may come from two aspects: (1) in Ver 1.0, for SMVM, it also has a setup phase to prepare the local

EQUIL	Version	$P = 1 \times 1$	$P = 2 \times 2$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
bbmat	glb	0.12	0.14	0.18	-	-
	dis	0.12	0.05	0.06	-	-
fidapm11	glb	0.05	0.05	0.07	-	-
	dis	0.05	0.02	0.04	-	-
wang4	glb	0.02	0.03	0.03	-	-
	dis	0.02	0.02	0.04	-	-
twotone	glb	0.13	0.16	0.21	0.25	0.26
	dis	0.12	0.08	0.09	0.14	0.14
mixing-tank	glb	0.14	0.14	0.18	0.19	0.20
	dis	0.13	0.05	0.05	0.06	0.07
inv-extrusion-1	glb	0.12	0.13	0.16	0.18	0.18
	dis	0.12	0.11	0.08	0.06	0.07
ecl32	glb	0.04	0.06	0.07	0.08	0.09
	dis	0.04	0.03	0.05	0.07	0.07
ir	glb	0.25	0.30	0.44	0.48	0.52
	dis	0.23	0.18	0.19	0.20	0.21
dds.quadratic	glb	-	-	1.33	1.82	1.92
	dis	-	-	0.39	0.45	0.48
dds15	glb	-	-	1.61	2.24	2.32
	dis	-	-	0.55	0.88	0.97

Table 3: The running time of EQUIL step of Ver 1.0 and Ver 2.0.

DISTRIBUTE	Version	$P = 1 \times 1$	$P = 2 \times 2$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
bbmat	glb	5.69	2.00	1.01	-	-
	dis	6.29	2.03	1.00	-	-
fidapm11	glb	3.07	1.20	0.44	-	-
	dis	3.19	1.05	0.52	-	-
wang4	glb	2.63	0.82	0.32	-	-
	dis	2.66	0.81	0.33	-	-
twotone	glb	50.74	13.27	3.53	1.23	0.88
	dis	44.16	11.73	3.50	1.36	1.05
mixing-tank	glb	5.71	1.64	0.67	0.48	0.41
	dis	6.77	1.89	0.70	0.43	0.43
inv-extrusion-1	glb	4.14	1.50	0.71	0.44	0.45
	dis	5.08	1.76	0.71	0.52	0.52
ecl32	glb	16.73	4.09	1.32	0.56	0.38
	dis	15.14	4.11	1.46	0.94	0.50
ir	glb	35.61	10.36	3.35	1.14	1.11
	dis	33.71	10.08	3.29	1.41	1.22
dds.quadratic	glb	-	-	9.53	4.64	3.52
	dis	-	-	9.63	4.79	3.83
dds15	glb	-	-	37.74	12.47	9.87
	dis	-	-	35.85	12.55	9.98

Table 4: The running time of DISTRIBUTE step of Ver 1.0 and Ver 2.0.

SOLVE	Version	$P = 1 \times 1$	$P = 2 \times 2$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
bbmat	glb	1.61	0.64	0.43	-	-
	dis	1.65	0.66	0.45	-	-
fidapm11	glb	0.96	0.36	0.21	-	-
	dis	0.93	0.36	0.19	-	-
wang4	glb	0.61	0.33	0.28	-	-
	dis	0.64	0.33	0.30	-	-
twotone	glb	2.78	1.95	1.62	1.22	1.87
	dis	3.01	1.89	1.63	1.44	1.86
mixing-tank	glb	1.30	0.46	0.23	0.15	0.13
	dis	1.31	0.46	0.23	0.16	0.19
inv-extrusion-1	glb	1.23	0.50	0.30	0.18	0.21
	dis	1.29	0.53	0.32	0.20	0.26
ecl32	glb	1.98	0.91	0.79	0.52	0.70
	dis	1.98	0.88	0.69	0.58	0.93
ir	glb	5.03	2.12	1.53	1.22	1.29
	dis	5.44	2.19	1.55	1.19	1.37
dds.quadratic	glb	-	-	2.98	1.79	2.00
	dis	-	-	3.07	1.89	2.08
dds15	glb	-	-	8.36	5.04	6.68
	dis	-	-	8.80	5.28	7.08

Table 5: The running time of SOLVE step of Ver 1.0 and Ver 2.0.

REFINE	Version	$P = 1 \times 1$	$P = 2 \times 2$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
bbmat	glb	8.78 (8.77)	4.02 (3.92)	2.59 (3.13)	-	-
	dis	8.75 (8.59)	3.43 (3.38)	2.28 (2.29)	-	-
fidapm11	glb	1.11 (1.04)	0.44 (0.46)	0.26 (0.31)	-	-
	dis	1.01 (0.97)	0.39 (0.37)	0.20 (0.34)	-	-
wang4	glb	1.31 (1.33)	0.88 (0.70)	0.69 (0.66)	-	-
	dis	1.32 (1.32)	0.68 (0.69)	0.61 (0.61)	-	-
twotone	glb	6.14 (5.98)	7.68 (4.11)	5.61 (3.75)	3.45 (3.07)	3.86 (3.79)
	dis	6.32 (6.03)	3.86 (5.82)	3.30 (3.49)	2.77 (3.96)	3.72 (5.23)
mixing-tank	glb	1.79 (1.55)	0.67 (0.73)	0.36 (0.57)	0.27 (0.65)	0.26 (0.52)
	dis	1.58 (1.44)	0.55 (0.50)	0.27 (0.25)	0.19 (0.19)	0.24 (0.23)
inv-extrusion-1	glb	4.29 (4.17)	1.85 (2.00)	1.10(1.54)	0.76 (1.29)	0.82 (1.45)
	dis	5.60 (5.51)	2.25 (2.21)	1.34 (1.32)	0.88 (1.12)	1.31 (1.15)
ecl32	glb	4.12 (4.03)	3.43 (1.83)	2.17 (1.52)	1.33 (1.33)	1.74 (1.71)
	dis	4.06 (4.20)	1.80 (1.78)	1.42 (1.40)	1.19 (1.19)	1.76 (1.68)
ir	glb	11.14 (11.10)	7.03 (5.22)	4.33 (4.36)	2.98 (3.54)	3.00 (3.92)
	dis	11.54 (10.81)	4.58 (4.56)	3.17 (3.31)	2.42 (2.41)	2.80 (2.90)
dds.quadratic	glb	-	-	9.30 (9.74)	5.26 (8.54)	5.32 (8.74)
	dis	-	-	6.35 (6.28)	3.89 (3.96)	4.24 (4.54)
dds15	glb	-	-	35.04 (21.28)	19.47 (15.57)	15.82 (19.01)
	dis	-	-	17.50 (17.45)	10.50 (10.92)	13.92 (13.76)

Table 6: The running time of REFINE step of Ver 1.0 and Ver 2.0.

TOTAL	Version	$P = 1 \times 1$	$P = 2 \times 2$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
bbmat	glb	89.09	35.58	21.91	-	-
	dis	90.83	35.50	21.62	-	-
fidapm11	glb	57.78	19.89	9.98	-	-
	dis	54.57	18.83	9.78	-	-
wang4	glb	19.07	7.39	4.44	-	-
	dis	19.20	7.18	4.56	-	-
twotone	glb	248.20	85.83	37.43	25.44	25.25
	dis	245.69	83.87	34.53	25.13	25.28
mixing-tank	glb	122.70	37.80	14.73	10.31	9.67
	dis	121.66	38.47	14.96	10.61	10.18
inv-extrusion-1	glb	83.13	30.09	16.79	14.54	15.06
	dis	98.07	36.22	18.69	15.18	16.07
ecl32	glb	120.06	37.76	16.57	11.74	12.31
	dis	118.47	37.06	15.99	11.98	12.41
ir	glb	285.09	96.05	44.58	35.67	35.21
	dis	285.92	94.52	43.73	35.02	36.09
dds.quadratic	glb	-	-	290.81	136.33	117.40
	dis	-	-	281.31	135.84	118.24
dds15	glb	-	-	351.48	225.84	206.71
	dis	-	-	331.24	218.75	209.07

Table 7: The total running time of Ver 1.0 and Ver 2.0.

part of A and transfer it to a distributed modified sparse row (MSR) matrix. This phase maybe much more time consuming than the setup phase of SMVM in Ver 2.0. This is comfirmed by the experimental results in Section 3.3. (2) in both Ver 1.0 and Ver 2.0, before and after the triangular solve in the refinement step, there are some procedures to distribute or gather data betweeen the diagonal processors and all processors. However, due to the different storage sturctures for the right hand side (one is global, the other is distributed), these procedures can be more time consuming in Ver 1.0 than in Ver 2.0. In other words, Ver 1.0 needs more comunications in these procedures. The data in parentheses are the running times of optional SMVM methods for both versions, which we will explain in Section 3.3.

Finally, we compare the overall performance of the two versions. Table 7 gives the total running times of both versions, including all eight steps of Algorithm 1. The results show that there is no much difference in runtime between Ver 1.0 and Ver 2.0. It is clear that the amount of more communication in Ver 2.0 can well offset the cost of the amount of more local memory access in Ver 1.0. For some cases, Ver 2.0 is even faster than Ver 1.0.

3.2 Comparison Between 32-bit and 64-bit Pointers and Integers

In this section, we compare the time difference between compiling the code in 32-bit and in 64-bit. *SuperLU_DIST* is implemented in such a way that it can be compiled with the following three modes:

Mode 1 Using 32 bit addressing and 32 bit integer (default `int`)

Mode 2 Using 64 bit addressing and 32 bit integer (default `int`)
This is needed if one MPI task needs to access more than 2 GBytes memory on the IBM SP. The compiler flag is "`-q64`".

Mode 3 Using 64 bit addressing and 64 bit integer (default `long int`)
This is needed when the matrix is very large. The compiler flag is "`-q64 -D_LONGINT`".

We tested all these modes on two largest matrices: *dds.quadratic* and *dds15*. The results are given in Table 8. In most cases, the code is slower with more bits used, as expected. From Mode 1 to Mode 2, the slowdown is no more than 17% (dds.15 on 128 processors). From Mode 2 to Mode 3, the slowdown is no more than 79% (dds.15 on 64 processors). From Mode 1 to Mode 3, the slowdown can be as large as a factor of two (dds.15 on 64 processors).

TOTAL TIME (in seconds)	dds.15			dds.quadratic		
	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$	$P = 4 \times 4$	$P = 8 \times 8$	$P = 8 \times 16$
Ver 1.0						
Mode 1	351.48	230.06	209.86	290.81(281.54)	143.27	123.56
Mode 2	401.56	263.54	244.37	280.68(278.06)	150.84	138.02
Mode 3	394.84	472.64	300.07	281.82(281.33)	162.17	149.12
Ver 2.0 with **dis-smvm**						
Mode 1	330.60	220.08	209.81	291.87(282.03)	139.13	123.70
Mode 2	377.02	242.06	232.64	281.28(276.36)	147.70	132.66
Mode 3	378.13	395.76	280.65	280.14(280.43)	159.99	144.30
Ver 2.0 with **dis-smvm-brdcst**						
Mode 1	331.16	218.65	208.18	289.41(287.75)	138.59	120.55
Mode 2	384.57	252.28	241.73	281.99(277.04)	151.11	132.96
Mode 3	372.79	370.60	272.17	280.73(280.69)	160.70	142.81

Table 8: The total running time of *SuperLU_DIST* using different addressing modes.

3.3 Different Methods for Parallel Sparse Matrix-Vector Multiplication

In this section, we examine more closely the performance of parallel sparse matrix-vector multiplication (SMVM). This is used in our iterative refinement routine to compute the residual $r = b - Ax$. It is also of great interest by itself because it is a key kernel in most iterative solvers. Consider $y \leftarrow Ax$. Recall that in Ver 2.0 the input matrix A is distributed by block rows. For each i, we need to compute $y_i = \sum_{j=1}^{n} a_{i,j}x_j$, where $a_{i,j}$ are on the same processor for all j. But some x_j may be on some other processor, so there is a need to communicate the x components. We implemented two different algorithms communicating x in Ver 2.0. The first algorithm was proposed in [6], and consists of an initialization phase and an actual multiplication phase. In the initialization phase, each processor processes the local graph of A (i.e., all $a_{i,j}$), and determines all the j's such that x_j is nonlocal. It then informs those processors who own x_j so that they know they need to send x_j to this processor. This phase involves an all-to-all communication so in the end every processor knowns which of my local x_j needs to be sent to which other processors. Some optimization is performed to reduce communication. For example, if a processor needs to send several x_j's to one other processor, these x_j's are packed into one message, so that each processor sends no more than one message to any other processor. Note that the initialization phase is time-consuming, so we run it only once and save the communication pattern. In the actual multiplication phase, each processor sends the corresponding local parts of x to the processors who need them. Each processor also receives all those nonlocal parts of x, and together with the local part of x, it then performs the multiplication. This algorithm is called **dis-smvm**. As a comparison, we also implemented a simpler algorithm, in which each processor just broadcasts its local part of x to all the other processors, then all the processors get the global x. This algorithm is called **dis-smvm-brdcst**.

In Ver 1.0 with the golbal interface, matrix A is stored in compressed column storage. Our first algorithm consists of an initialization phase, in which matrix A is converted into the distributed compressed row format. After this transposition, the actual multiplication becomes embarrassingly parallel. We call this algorithm **glb-smvm**. Our second algorithm is completely sequential (since

A and B are globally available on each processor), and the multiplication is performed replicatedly on each processor. No communication is involved. We call this algorithm **glb-smvm-sequential**.

In Table 9 we give timings of the above four algorithms with three largest matrices, using 64 and 128 processors, both minimum degree and nested dissection orderings. The timings are divided into the time of the entire iterative refinement routine, the time for SMVM initialization, and the time for one multiplication. Note that the initialization step of **dis-smvm** gathers the information of which parts of x to send/receive and where to send/receive, whereas the initialization step of **dis-smvm-brdcst** only needs to gather the information of the number of nonlocal rows, which takes much less time. However, in each iteration, the **dis-smvm-brdcst** uses much more time than **dis-smvm**. The reason is that the number of messages and message volume transferred in **dis-smvm** are much less than that in **dis-smvm-brdcst**. This can be seen from Table 10, which gives the ranges of the size and number of messages transferred by each processor with **dis-smvm**, whereas with **dis-smvm-brdcst**, every processor needs to send a message to every other processor. The third column of Table 10 shows the ranges of the number of nonzeros local to each processor. Since the input matrix A is evenly distibuted by block rows, the number of rows on each processor is almost the same, while the number of nonzeros is not (which can also be seen in Figures 1 to 3). The fourth and fifth columns of Table 10 give the percentage of x entries transferred by each processor. Figures 1 to 3 show the detailed distribution of the size and number of messages transferred by each processor with **dis-smvm**, using 64 processors with three largest matrices: *ir*, *dds.quadratic*, *dds15*. We observe that the curves of the message size recieved and the number of local nonzeros have similar pattern in Figures 1 and 2. It shows that if there are more nonzeros in a processor, it may require more entries of x from the other processors. In Figure 3, this observation is not clear. This is because even if the number of nonzeros on one processor is large, many of them maybe in the same column, which means these nonzeros (in the same column) only require one element of x.

Note that in Table 6, for Ver 2.0 (marked with "dis"), the timings outside the parentheses are from **dis-smvm**, the timings in parentheses are from **dis-smvm-brdcst**. When the number of processors is large, in most cases **dis-smvm** is much better than **dis-smvm-brdcst**. When the number of processors is small, the running times of these two methods are similar.

Now we examine the SMVM timings for Ver 1.0 in Table 9. Note that for each multiplication, the sequentail method is much slower than the parallel one. However, it does not need an initializtion. The data in parentheses in Table 6 are the running time of using **glb-smvm-sequential** instead of using **glb-smvm**. In some cases, especially when matrix is sparse or number of processors is small, the sequential method is faster. Because even though the sequential method takes much more time than the parallel one in each steps, it saves the communications used for distributing B to the diagonal processors (before triangular solve) and gathering the X from the diagonal processors onto all processors (after triangular solve). Therefore, if the matrix is sparser, the overhead of the sequential sparse matrix-vector multiply is less; similarly, if the number of processors is small, the overhead is also small. However, when the matrix is less sparse or the number of processors is much larger, the overhead of the sequential method maybe larger than the gain from the communication saved. Clearly, there is a trade-off here.

4 Fortran 90 Interface

Due to Fortran's popularity in scientific computing, we also developed a Fortran interface of *SuperLU_DIST* for Fortran 90 users. Using this interface, they can call *SuperLU_DIST* to slove linear systems in their Fortran applications. All the interface files and an example driver program are located in the SuperLU_DIST/FORTRAN/ subdirectory. Table 11 lists all the files.

Note that in this interface, all objects (such as grid, options, etc.) in *SuperLU_DIST* are *opaque*: their size and structure are not visible to the Fortran user. These opaque objects are allocated, deallocated and operated in the C side and not directly accessible from Fortran side.

MMD ordering						
	$P = 8 \times 8$			$P = 8 \times 16$		
dis-smvm (Ver 2.0)	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	2.40	0.01-0.02	0.00-0.01	2.84	0.01-0.02	0.00-0.02
dds.quadratic	3.94	0.02-0.03	0.01-0.02	4.33	0.04-0.05	0.01-0.02
dds15	11.77	0.04-0.06	0.01-0.03	14.72	0.03-0.06	0.01-0.02
dis-smvm-brdcst	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	2.49	0.00-0.01	0.02-0.03	3.05	0.00-0.02	0.02-0.04
dds.quadratic	3.98	0.00-0.01	0.04-0.05	4.44	0.00-0.01	0.04-0.05
dds15	11.31	0.00-0.02	0.08-0.10	13.96	0.00-0.01	0.08-0.10
glb-smvm (Ver 1.0)	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	2.91	0.16-0.18	0.00	3.02	0.16-0.18	0.00
dds.quadratic	5.35	0.74-0.81	0.01	5.38	0.72-0.82	0.00-0.01
dds15	19.82	0.73-0.82	0.01	15.88	0.72-0.81	0.00-0.01
glb-smvm-sequential	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	3.71	0.00	0.35-0.38	3.93	0.00	0.35-0.38
dds.quadratic	8.42	0.00	1.38-1.43	8.87	0.00	1.38-1.54
dds15	16.65	0.00	1.56-1.75	19.28	0.00	1.56-1.69
ND ordering						
	$P = 8 \times 8$			$P = 8 \times 16$		
dis-smvm (Ver 2.0)	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	2.63	0.01-0.03	0.00-0.02	3.19	0.01-0.04	0.00-0.03
dds.quadratic	3.61	0.03-0.04	0.01-0.03	4.17	0.02-0.05	0.01-0.02
dds15	16.88	0.04-0.06	0.01-0.03	17.91	0.04-0.08	0.01-0.03
dis-smvm-brdcst	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	2.62	0.00	0.02-0.03	3.13	0.00-0.01	0.03-0.04
dds.quadratic	3.91	0.00-0.01	0.04-0.05	4.24	0.00-0.01	0.05-0.06
dds15	16.63	0.00-0.02	0.08-0.11	18.77	0.00-0.01	0.08-0.10
glb-smvm (Ver 1.0)	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	3.20	0.15-0.17	0.00	3.30	0.15-0.17	0.00
dds.quadratic	5.18	0.71-0.79	0.01-0.02	5.13	0.70-0.81	0.00-0.01
dds15	30.38	0.69-0.77	0.01-0.02	20.08	0.68-0.78	0.00-0.01
glb-smvm-sequential	Refine	Init-SMVM	one SMVM	Refine	Init-SMVM	one SMVM
ir	3.96	0.00	0.32-0.33	4.04	0.00	0.32-0.35
dds.quadratic	8.46	0.00	1.40-1.45	8.75	0.00	1.41-1.47
dds15	21.47	0.00	1.50-1.67	23.21	0.00	1.50-1.57

Table 9: The running times of the four SMVM algorithms on 64 and 128 processors.

	n	Local Nozeros	% of x Send	% of x Recv	Msgs Send	Msgs Recv
MMD ordering						
ir ($P = 64$)	186230	35823-48244	2.7%-4.4%	2.7%-5.8%	3-26	5-26
($P = 128$)		17212-24359	2.0%-2.9%	1.4%-4.1%	7-43	10-43
dds.quadratic	380698	230938-277626	2.2%-4.0%	1.8%-5.9%	5-40	8-28
($P = 128$)		113768-145896	1.4%-2.6%	1.0%-4.7%	8-55	9-48
dds15	834575	192160-214392	1.7%-3.2%	1.2%-3.9%	4-19	4-23
($P = 128$)		87555-107285	1.0%-2.0%	0.9%-2.6%	8-32	7-37
ND ordering						
ir ($P = 64$)	186230	35823-48244	3.3%-4.9%	3.0%-5.8%	4-33	7-29
($P = 128$)		17212-24359	2.0%-3.3%	1.4%-4.1%	8-52	10-48
dds.quadratic	380698	230938-277626	2.8%-4.3%	2.3%-6.3%	5-40	7-25
($P = 128$)		113768-145896	1.5%-2.6%	1.2%-4.7%	8-57	9-39
dds15	834575	192160-214392	1.8%-3.2%	1.7%-3.9%	4-21	4-18
($P = 128$)		87555-107285	1.2%-2.0%	1.0%-2.6%	7-32	7-36

Table 10: The size and number of messages transferred by each processor in **dis-smvm**, using 64 and 128 processors.

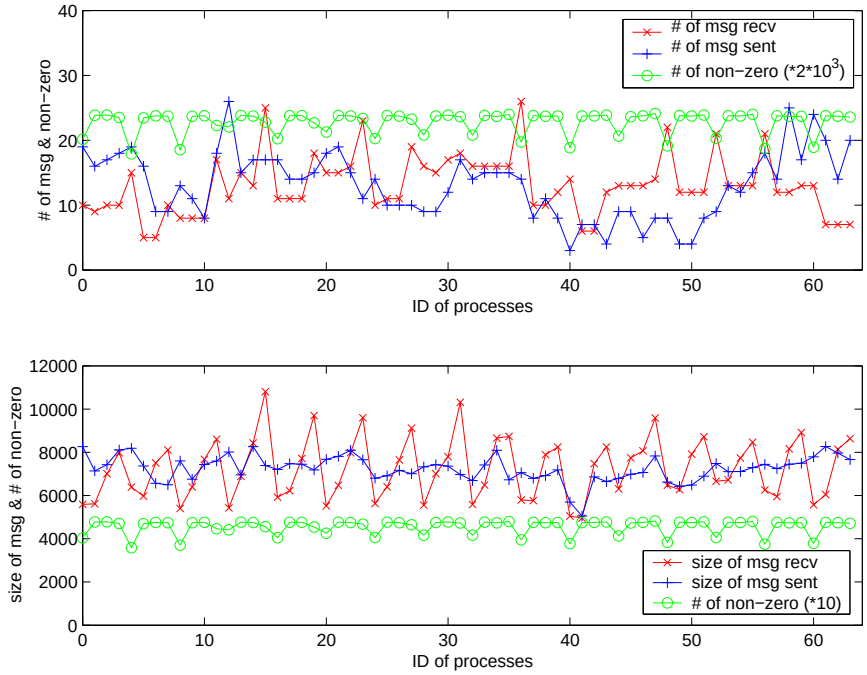

Figure 1: The number and size of messages transfered by each processor in **dis-smvm**, and the number of local nonzeros on each processor. Here we use 64 processors with matrix *ir*.

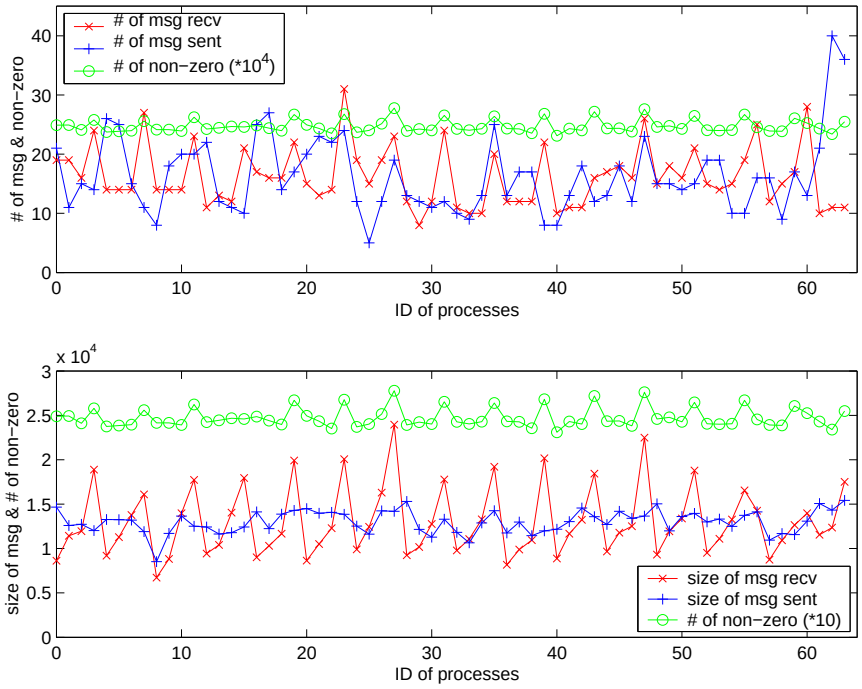

Figure 2: The number and size of messages transfered by each processor in **dis-smvm**. Here we use 64 processors on the matrix *dds.quadratic*.

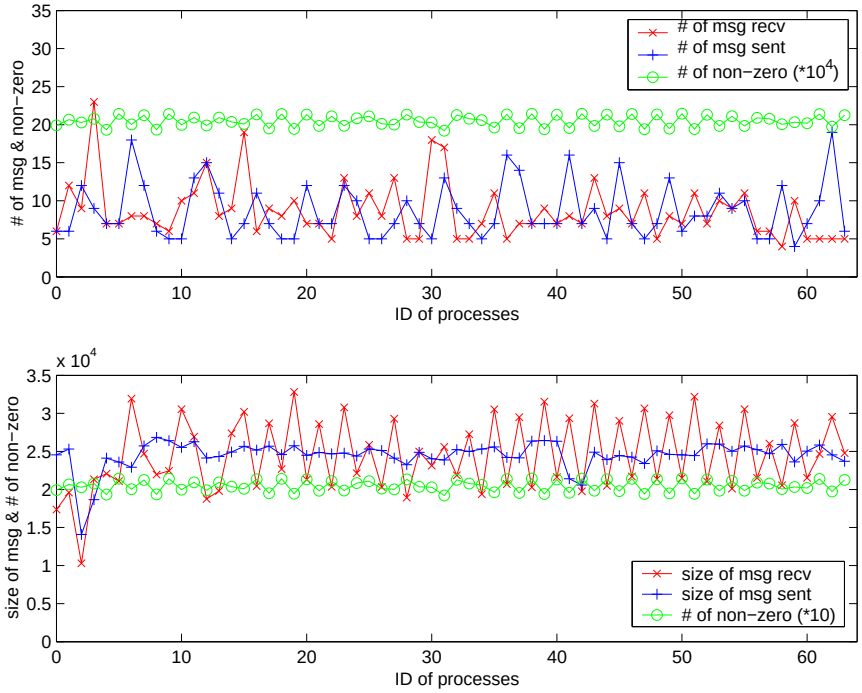

Figure 3: The number and size of messages transfered by each processor in **dis-smvm**. Here we use 64 processors on the matrix *dds15*.

f_pddrive.f90	An example Fortran driver routine.
superlu_mod.f90	Fortran 90 module that defines the wrapper functions to access *SuperLU_DIST* data structures. These functions have with optional arguments so the user does not have to provide the full set of components.
superlupara.f90	It contains parameters that correspond to *SuperLU_DIST*'s enums.
hbcode1.f90	Fortran function for reading a sparse Harwell-Boeing matrix from the file.
superlu_c2f_wrap.c	C wrapper functions, callable from Fortran. The functions fall into three classes: 1) Those that allocate a structure and return a handle, or deallocate the memory of a structure. 2) Those that get or set the value of a component of a struct. 3) Those that are wrappers for *SuperLU_DIST* functions.
dcreate_matrix_dist.c	C function for distributing the matrix in a distributed compressed row format. Note that here we adjust the 1-based indexing to 0-based indexing which is required by the C routines.

Table 11: All the interface files and an example driver routine

They can only be accessed via *handles* that are exist in user space. In Fortran, all handles have type INTEGER. Specifically, in our interface, the size of Fortran handle is defined by superlu_ptr in superlupara.f90. For different systems, the size might need to be changed. Then using these handles, Fortran user can call C wrapper routines to manipulate the opaque objects. For example, you can call f_create_gridinfo(grid_handle) to allocate memory for structure grid, and return a handle grid_handle. All callable interface (wrapper) routines are listed in Appendix.

The sample program illustrates the basic steps required to use *SuperLU_DIST* in Fortran to solve systems of equations. These include how to set up the processor grid and the input matrix, how to call the linear equation solver. This program is listed below, and is also available as f_pddrive.f90 in the subdirectory. Note that the routine must include the moudle superlu_mod which contains the definitions of all parameters and the Fortran wrapper functions. A Makefile is provided to generate the executable. A README file in this subdirectory shows how to run the example.

```
      program f_pddrive
!
! Purpose
! =======
!
! The driver program F_PDDRIVE.
!
! This example illustrates how to use F_PDGSSVX with the full
! (default) options to solve a linear system.
!
! Seven basic steps are required:
!   1. Create C structures used in SuperLU
!   2. Initialize the MPI environment and the SuperLU process grid
!   3. Set up the input matrix and the right-hand side
!   4. Set the options argument
```

```
!   5. Call f_pdgssvx
!   6. Release the process grid and terminate the MPI environment
!   7. Release all structures
!
      use superlu_mod
      include 'mpif.h'
      implicit none
      integer maxn, maxnz, maxnrhs
      parameter ( maxn = 10000, maxnz = 100000, maxnrhs = 10 )
      integer rowind(maxnz), colptr(maxn)
      real*8  values(maxnz), b(maxn), berr(maxnrhs)
      integer n, m, nnz, nrhs, ldb, i, ierr, info, iam
      integer nprow, npcol
      integer init

      integer(superlu_ptr) :: grid
      integer(superlu_ptr) :: options
      integer(superlu_ptr) :: ScalePermstruct
      integer(superlu_ptr) :: LUstruct
      integer(superlu_ptr) :: SOLVEstruct
      integer(superlu_ptr) :: A
      integer(superlu_ptr) :: stat


! Create C structures used in SuperLU
      call f_create_gridinfo(grid)
      call f_create_options(options)
      call f_create_ScalePermstruct(ScalePermstruct)
      call f_create_LUstruct(LUstruct)
      call f_create_SOLVEstruct(SOLVEstruct)
      call f_create_SuperMatrix(A)
      call f_create_SuperLUStat(stat)

! Initialize MPI environment
      call mpi_init(ierr)

! Initialize the SuperLU process grid
      nprow = 2
      npcol = 2
      call f_superlu_gridinit(MPI_COMM_WORLD, nprow, npcol, grid)

! Bail out if I do not belong in the grid.
      call get_GridInfo(grid, iam=iam)
      if ( iam >= nprow * npcol ) then
         go to 100
      endif
      if ( iam == 0 ) then
         write(*,*) ' Process grid ', nprow, ' X ', npcol
      endif

! Read Harwell-Boeing matrix
      if ( iam == 0 ) then
```

```
         call hbcode1(m, n, nnz, values, rowind, colptr)
      endif

! Distribute the matrix to the gird
      call  f_dcreate_matrix_dis(A, m, n, nnz, values, rowind, colptr, grid)

! Setup the right hand side
      nrhs = 1
      call  get_CompRowLoc_Matrix(A, nrow_loc=ldb)
      do i = 1, ldb
         b(i) = 1.0
      enddo

! Set the default input options
      call f_set_default_options(options)

! Set one or more options
!      call set_superlu_options(options,Fact=FACTORED)

! Initialize ScalePermstruct and LUstruct
      call get_SuperMatrix(A,nrow=m,ncol=n)
      call f_ScalePermstructInit(m, n, ScalePermstruct)
      call f_LUstructInit(m, n, LUstruct)

! Initialize the statistics variables
      call f_PStatInit(stat)

! Call the linear equation solver
      call f_pdgssvx(options, A, ScalePermstruct, b, ldb, nrhs, &
                     grid, LUstruct, SOLVEstruct, berr, stat, info)

      if (info == 0) then
         write (*,*) 'Backward error: ', (berr(i), i = 1, nrhs)
      else
         write(*,*) 'INFO from f_pdgssvx = ', info
      endif

! Deallocate SuperLU allocated storage
      call f_PStatFree(stat)
      call f_Destroy_CompRowLoc_Matrix_dis(A)
      call f_ScalePermstructFree(ScalePermstruct)
      call f_Destroy_LU(n, grid, LUstruct)
      call f_LUstructFree(LUstruct)
      call get_superlu_options(options, SolveInitialized=init)
      if (init == YES) then
         call f_dSolveFinalize(options, SOLVEstruct)
      endif

! Release the SuperLU process grid
100   call f_superlu_gridexit(grid)

! Terminate the MPI execution environment
```

```
      call mpi_finalize(ierr)

! Destroy all C structures
      call f_destroy_gridinfo(grid)
      call f_destroy_options(options)
      call f_destroy_ScalePermstruct(ScalePermstruct)
      call f_destroy_LUstruct(LUstruct)
      call f_destroy_SOLVEstruct(SOLVEstruct)
      call f_destroy_SuperMatrix(A)
      call f_destroy_SuperLUStat(stat)

      stop
      end
```

Similar to the driver routine pddrive.c in C, seven basic steps are required to call a *SuperLU_DIST* routine in Fortran:

1. Create C structures used in SuperLU: grid, options, ScalePermstruct, LUstruct, SOLVEstruct, A and stat. This is achieved by the calls to the C wrapper *"create"* routines f_create_XXX, where XXX is the name of the corresponding strucuture.

2. Initialize the MPI environment and the SuperLU process grid. This is achieved by the calls to mpi_init and the C wrapper routines f_superlu_gridinit. Note that when calling f_superlu_gridinit, it requires the numbers of row and column of the grid. In this example, we set them to 2s.

3. Set up the input matrix and the right-hand side. This example uses *SuperLU_DIST* Ver 2.0 kernel, so we need to convert input matrix to the distributed compressed row format. Process 0 first reads the input matrix stored on disk in Harwell-Boeing format [3] by calling Fortran routine hbcode1. Then all processes call a C wrapper routine f_dcreate_matrix_dis to distribute the matrix to the grid in a distributed compressed row format. The right-hand side matrix in this example is set to one column of all ones. Note that, before setting the right-hand side, we use get_CompRowLoc_Matrix to get the number of local rows in the distributed matrix A.

4. Set the input arguments: options, ScalePermstruct, LUstruct, stat. The input argument options controls how the linear system would be sloved –use equilibration or not, how to order the rows and the columns of the matrix, use iterative refinement or not. The routine f_set_default_options sets the options argument so that the slover performs all the functionality. You can also set it up according to your own needs, using a call set_superlu_options. LUstruct is the data struture in which the distributed L and U factors are stored. ScalePermstruct is the data struture in which several vectors describing the transformations done to matrix A are stored. stat is a structure collecting the statistcs about runtime and flop count. These three structures can be set by calling the C wrapper *"init"* routines f_XXXInit.

5. Call the C wrapper routine f_pdgssvx for linear equation solver.

6. Release the process grid and terminate the MPI environment. After the computation on a process grid has been completed, the process grid should be released by a call f_spuerlu_gridexit. When all computations have been completed, the C wrapper routine mpi_finalize should be called.

7. Release all structures. First we need to deallocate storage SuperLu allocated using a set of *"free"* calls. Note that this should be called before f_spuerlu_gridexit since some of *"free"* calls use the grid. Then we call the C wrapper *"destroy"* routines f_destroy_XXX to destroy all C structures. Note that f_destroy_gridinfo should be called after f_spuerlu_gridexit.

5 Conclusions

In this report, we evaluated the performance of the two versions of *SuperLU_DIST*, one with global input interface (Ver 1.0), and another with distributed input interface (V 2.0), using up to 128 processors of the IBM SP parallel machine. Clearly, Ver 2.0 has memory advantage over Ver 1.0. In terms of runtime, our results on the IBM SP show that there is very little difference between the two versions, even though Ver 2.0 requires much more communication for data redistribution. This is because the cost of extra communication in Ver 2.0 can well offset the cost of the extra local memory access in Ver 1.0. It remains to see whether this also holds for a more loosely coupled cluster environment, where interprocessor communication is relatively more expensive.

Furthermore, we compared the performance of four different sparse matrix-vector multiplication methods in the context of iterative refinement. Each method has its own merit depending on the input matrices and number of processors used. We also compared the runtime difference between various addressing modes. Usually, 32-bit is faster, but no more than a factor of two. In most cases, the difference is small. However, 64-bit is needed when the matrix is very large.

Finally, we developed a Fortran 90 interface for Ver 2.0 so that the Fortran users can use this solver easily. Our implementation uses opaque objects and includes Fortran wrapper functions to access those objects created in C.

6 Acknowledgment

We would like to thank William F. Mitchell for sharing his codes and ideas when we were preparing the Fortran interface.

References

[1] I. S. Duff and J. Koster, The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices, *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

[2] James W. Demmel, John R. Gilbert, and Xiaoye S. Li, SuperLU Users' Guide, Technical Report LBNL-44289, Ernest Orlando Lawrence Berkeley National Laboratory, September 1999.

[3] I. S. Duff, R. G. Grimes, and J. G. Lewis, Users' Guide for the Harwell-Boeing Sparse Matrix Collection (release 1), Technical Report RAL-92-086, Rutherford Appleton Laboratory, December 1992.

[4] Xiaoye S. Li and James W. Demmel, SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems, *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[5] Xiaoye. S. Li and James W. Demmel, Making Sparse Gaussian Elimination Scalable by Static Pivoting, In *Proceedings of SC98: High Performance Networking and Computing Conference*, Orlando, Florida, November 1998.

[6] Ray S. Tuminaro, John N. Shadid, and Scott A. Hutchinson, Parallel Sparse Matrix-Vector Multiply Software for Matrices with Data Locality, Technical Report SAND95-1540J, Sandia National Laboratory, July 1995.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis, The Rutherford-Boeing Sparse Matrix Collection, Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS.

[8] Liu, J. W. H. 1985. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software 11,* 2, 141–153.

[9] Karypis, G. and Kumar, V. 1998. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota.

Appendix: User-callable functions in Fortran interface

Callable Functions from C wrapper spuerlu_c2f_wrap.c:

```
/* functions that create memory for a struct and return a handle */
void f_create_gridinfo(fptr *handle)
void f_create_options(fptr *handle)
void f_create_ScalePermstruct(fptr *handle)
void f_create_LUstruct(fptr *handle)
void f_create_SOLVEstruct(fptr *handle)
void f_create_SuperMatrix(fptr *handle)
void f_create_SuperLUStat(fptr *handle)

/* functions that free the memory allocated by the above functions */
void f_destroy_gridinfo(fptr *handle)
void f_destroy_options(fptr *handle)
void f_destroy_ScalePermstruct(fptr *handle)
void f_destroy_LUstruct(fptr *handle)
void f_destroy_SOLVEstruct(fptr *handle)
void f_destroy_SuperMatrix(fptr *handle)
void f_destroy_SuperLUStat(fptr *handle)

/* functions that get or set values in a C struct. */

void f_get_gridinfo(fptr *grid, int *iam, int *nprow, int *npcol)
void f_get_SuperMatrix(fptr *A, int *nrow, int *ncol)
void f_set_SuperMatrix(fptr *A, int *nrow, int *ncol)
void f_get_CompRowLoc_Matrix(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row)
void f_set_CompRowLoc_Matrix(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row)
void f_get_superlu_options(fptr *opt, int *Fact, int *Trans, int *Equil,
                           int *RowPerm, int *ColPerm, int *ReplaceTinyPivot,
                           int *IterRefine, int *SolveInitialized,
                           int *RefineInitialized)
void f_set_superlu_options(fptr *opt, int *Fact, int *Trans, int *Equil,
                           int *RowPerm, int *ColPerm, int *ReplaceTinyPivot,
                           int *IterRefine, int *SolveInitialized,
                           int *RefineInitialized)

/* wrappers for SuperLU functions */
void f_set_default_options(fptr *options)
void f_superlu_gridinit(int *Bcomm, int *nprow, int *npcol, fptr *grid)
void f_superlu_gridexit(fptr *grid)
void f_ScalePermstructInit(int *m, int *n, fptr *ScalePermstruct)
```

```
void f_ScalePermstructFree(fptr *ScalePermstruct)
void f_PStatInit(fptr *stat)
void f_PStatFree(fptr *stat)
void f_LUstructInit(int *m, int *n, fptr *LUstruct)
void f_LUstructFree(fptr *LUstruct)
void f_Destroy_LU(int *n, fptr *grid, fptr *LUstruct)
void f_dCreate_CompRowLoc_Matrix_dist(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row, double *nzval,
                                      int *colind, int *rowptr, int *stype,
                                      int *dtype, int *mtype)
void f_Destroy_CompRowLoc_Matrix_dist(fptr *A)
void f_dSolveFinalize(fptr *options, fptr *SOLVEstruct)
void f_pdgssvx(fptr *options, fptr *A, fptr *ScalePermstruct, double *B,
               int *ldb, int *nrhs, fptr *grid, fptr *LUstruct,
               fptr *SOLVEstruct, double *berr, fptr *stat, int *info)
void f_dcreate_matrix_dis(fptr *A, int *m, int *n, int *nnz, double *nzval,
                          int *rowind, int *colptr, fptr *grid)
void f_check_malloc(int *iam)
```

Callable Functions from Fortran wrapper spuerlu_mod.f90:

```
subroutine get_GridInfo(grid, iam, nprow, npcol)
integer(superlu_ptr) :: grid
integer, optional :: iam, nprow, npcol

subroutine get_SuperMatrix(A, nrow, ncol)
integer(superlu_ptr) :: A
integer, optional :: nrow, ncol

subroutine set_SuperMatrix(A, nrow, ncol)
integer(superlu_ptr) :: A
integer, optional :: nrow, ncol

subroutine get_CompRowLoc_Matrix(A, nrow, ncol, nnz_loc, nrow_loc, fst_row)
integer(superlu_ptr) :: A
integer, optional :: nrow, ncol, nnz_loc, nrow_loc, fst_row

subroutine set_CompRowLoc_Matrix(A, nrow, ncol, nnz_loc, nrow_loc, fst_row)
integer(superlu_ptr) :: A
integer, optional :: nrow, ncol, nnz_loc, nrow_loc, fst_row

subroutine get_superlu_options(opt, Fact, Trans, Equil, RowPerm, &
                               ColPerm, ReplaceTinyPivot, IterRefine, &
                               SolveInitialized, RefineInitialized)
integer(superlu_ptr) :: opt
integer, optional :: Fact, Trans, Equil, RowPerm, ColPerm, &
                     ReplaceTinyPivot, IterRefine, SolveInitialized, &
                     RefineInitialized

subroutine set_superlu_options(opt, Fact, Trans, Equil, RowPerm, &
                               ColPerm, ReplaceTinyPivot, IterRefine, &
                               SolveInitialized, RefineInitialized)
```

```
integer(superlu_ptr) :: opt
integer, optional :: Fact, Trans, Equil, RowPerm, ColPerm, &
                     ReplaceTinyPivot, IterRefine, SolveInitialized, &
                     RefineInitialized
```