ESMF Array Sparse Matrix Multiplication Benchmark

Peggy Li

December 4, 2007

Objective

The objective of this task is to evaluate the performance of the Array Sparse Matrix Multiplication (ASMM) functions as implemented in ESMF and compare their performance with MCT (Model Coupling Toolkit) equivalents. The benchmark compares the performance of the ASMM initialization function of ESMF (ESMF_ArraySparseMatMulStore()) and MCT (m_SparseMatricPlus::init()), and the performance of the ASMM run function of ESMF (ESMF_ArraySparseMatMul()) and MCT (m_MatAttrVectMul::sMatAvMult()). It tests a special case in which the data from only one field is processed, and so does not reflect a typical use case. Further, timings for multiple fields may not be multiples of the times presented here.

We conducted the performance evaluation on the IBM Power5 Cluster at NCAR (bluevista), Cray XT3/XT4 at Oak Ridge National Laboratory (jaguar) and the SGI Altix Superclusters at NASA Ames (columbia) using 4 to 256 processors. We also ran a scalability benchmark with a large array on the Cray XT3/XT4 using up to 2048 processors. The IBM Cluster is running AIX 5.3.0.0 operating system with IBM XL compilers. The Cray XT3/XT4 is running UNICOS/lc 1.4.35 operating system with PGI 6.1.6 compilers and the SGI Altix is running Linux 2.6.5-7.276-sn2 with Intel compilers version 9.1.039 and SGI Message Passing Toolkit (MPT) version 1.12.0. The ESMF version used to run the benchmark is ESMF 3.1.0 beta snapshot #10 and the MCT version used is MCT 2.3. The scalability benchmark was performed on the Cray XT3/XT4 running CNL (Computer Node Linux) using a newer version of ESMF -- ESMF 3.1.0 beta snapshot #21.

Benchmark Program

Both the ESMF and MCT benchmark programs consist of three components, a source Grid Component, a destination Grid Component, and a Coupler component, and a driver program responsible for the creation, initialization, execution and termination of the components. Note there is no component concept in MCT. We organize the MCT code into init, run, and finalize functions mimicking the ESMF component structure. Both the source array and the destination array are 2D arrays. The source array is a 720x360 single precision float point array and the destination array is a 1080x640 single precision floating point array. The data values in the source array are based on a sample temperature field from an ocean model and the interpolation array is pre-calculated using bi-linear interpolation. The sparse matrix was read in by PE 0 and initialized in a centralized manner from PE0. We benchmarked one source and various combinations of destination array distributions - Row-only (R), Column-only (C), and Row-Column block distribution (RC). The block distribution for the source array and the destination array is slightly different depending on the number of processors. Table 1 depicts the various data distribution used in the benchmark. We measured the initialization calls,

ESMF_ArraySparseMatMulStore() and MCT m_SparseMatrixPlus::init once, and the average time of 50 run calls, ESMF_ArraySparseMatMul() and MCT

	Source Array	Dest Array		
# Procs	RC	RC	C	R
4	2x2	2x2	4x1	1x4
8	2x4	4x2	8x1	1x8
16	4x4	4x4	16x1	1x16
32	4x8	8x4	32x1	1x32
64	8x8	8x8	64x1	1x64
128	8x16	16x8	128x1	1x128
256	16x16	16x16	256x1	1x256

m_MatAttrVecMul::sMatAvMult().

Table 1 The data distributions

For the scalability benchmark on large PE counts, we use a 1024x1024 element single-precision floating point source array and a 2048x960 element destination array. The source array was partitioned using block distribution and the destination array was partitioned using Column-only and Row-only distributions. We measured both the memory usage and the timing performance using up to 2048 processors on the Cray XT3/XT4 machine.

The Results

Timing Comparison using different distributions on bluevista

Figure 1 to 2 are the timing comparison between ESMF and MCT on the IBM SP Cluster (bluevista). The source array is a 720x360 2D array using block distribution (RC). The destination array is a 1080x640 2D array using Column-only distribution (C).

From Figure 1, for the RC to C case, ESMF outperforms MCT in the Sparse Matrix Multiplication Initialization by 3 to 4 times (there is much less difference for other distributions, as shown in Figure 3). The initialization time is a function of the size of the array and the array distribution, but is almost independent of the number of processors used. The timing does increase slightly when the processor count is greater than or equal to 64. It is getting worse when the processor count is larger than 256 processors. The scalability issue will be discussed later.

In the MCT case, the user can choose between source-based or destination-based distributions. In source-based ("Xonly"), the weights are distributed so source information must be moved before interpolation can be done. In destination based ("Yonly"), destination data is moved to complete the calculation. Total data movement will depend on how large the arrays are and how different their two decompositions.

In our test case, the destination array is distributed by column only. In this case, the Yonly distribution works much better than Xonly distribution. In Figure 1, we show the performance for MCT Xonly (in navyblue) and MCT Yonly (in yellow); we see a 50%

time improvement by choosing a sparse matrix distribution similar to the destination array distribution.

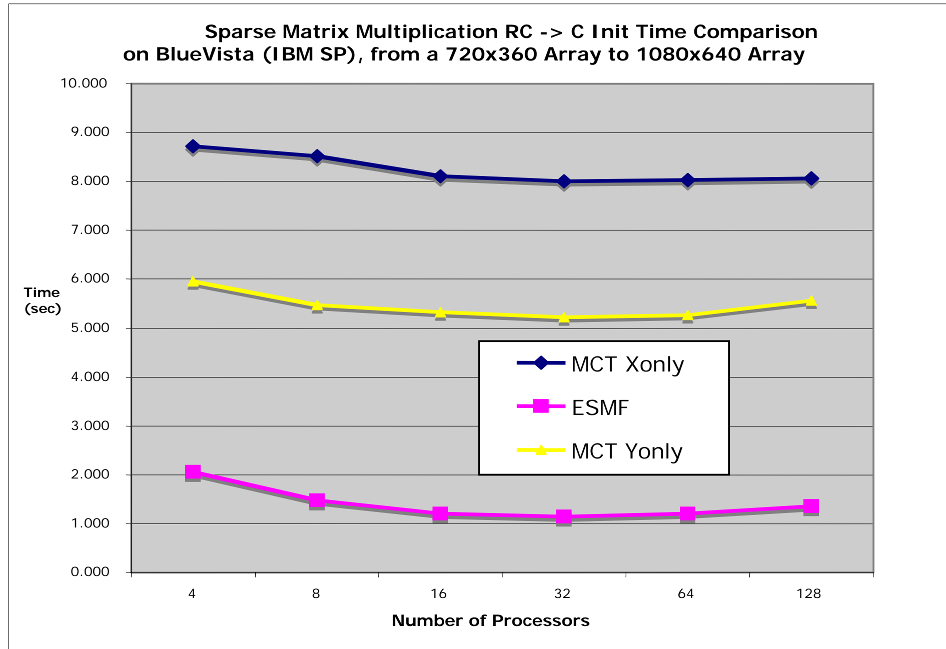

Figure 1 ASMM Init Time Comparison on bluevista

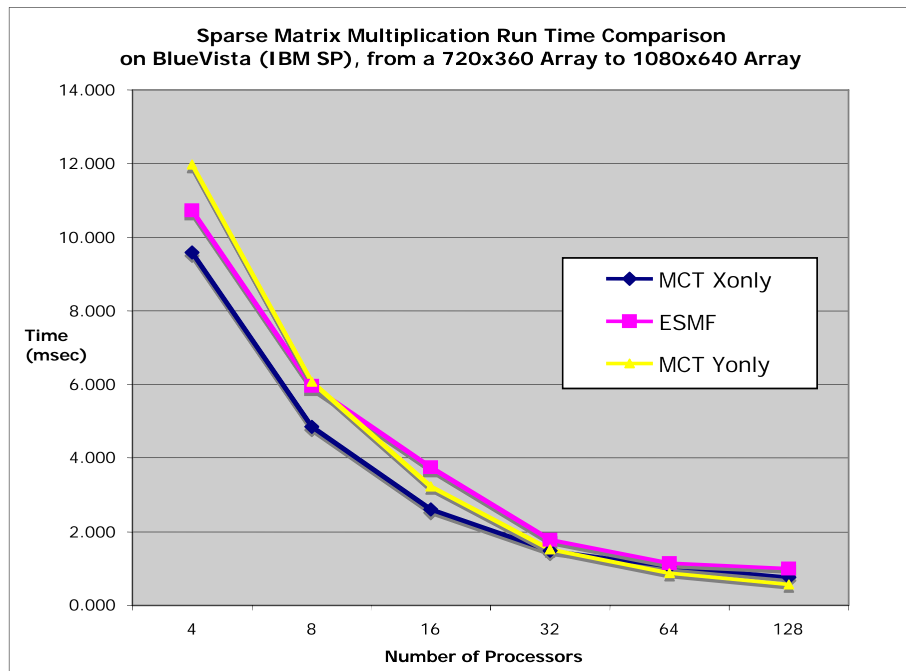

Figure 2 ASMM Run Time Comparison on bluevista

MCT defines it data distribution using index segments. A Row-only distribution can be represented by one index segment in MCT because the data in the local processor are contiguous in memory. On the other hands, a column only distribution will be represented by n small segments, when n is the y dimension of the array. The sparse matrix initialization time changes with the distribution pattern of the destination arrays. The Column-only distribution used by my benchmark turns out to be the slowest case in terms of the sparse matrix initialization time. So, we measured the SMM performance on bluevista using two additional distribution patterns, i.e., RC to R and RC to RC. The timing comparisons between MCT and ESMF are shown in Figure 3 and 4. The MCT runs used Xonly option for Sparse Matrix initialization, which performs better than the Yonly option in both the init and run time in these two cases.

Amongst the three destination array distributions, row-only distribution renders the fastest init time for MCT, followed by the block distribution and the column-only distribution (in Figure 1). The ESMF ASMM init performance is not as sensitive to the distribution pattern as MCT is for this particular source distribution and array sizes. Although the ESMF performance is slightly worse than the MCT using the best case distribution for the MCT code (RC to R), (the magenta line vs. the blue line). For other distributions, ESMF performs significantly better than MCT.

The ASMM run time difference between the two systems is much smaller (Figure 2 and Figure 4). The timing mainly depends on the total size of data been transferred on each processor. Among the three configurations, RC->RC has either the same or very similar data distribution from the source array to the destination array, thus the ASMM run time is the smallest of the three. This is true for both ESMF and MCT. The ASMM run time scales near linearly for both ESMF and MCT on all the configurations. It suggested that the code is optimized for scalability.

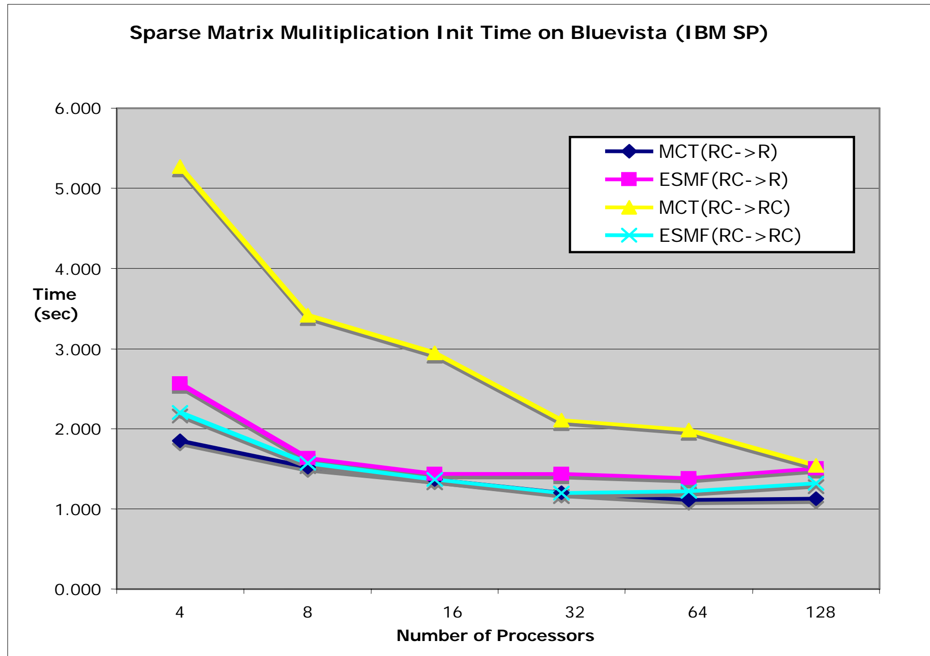

Figure 3 ASMM Init Time for other data distributions

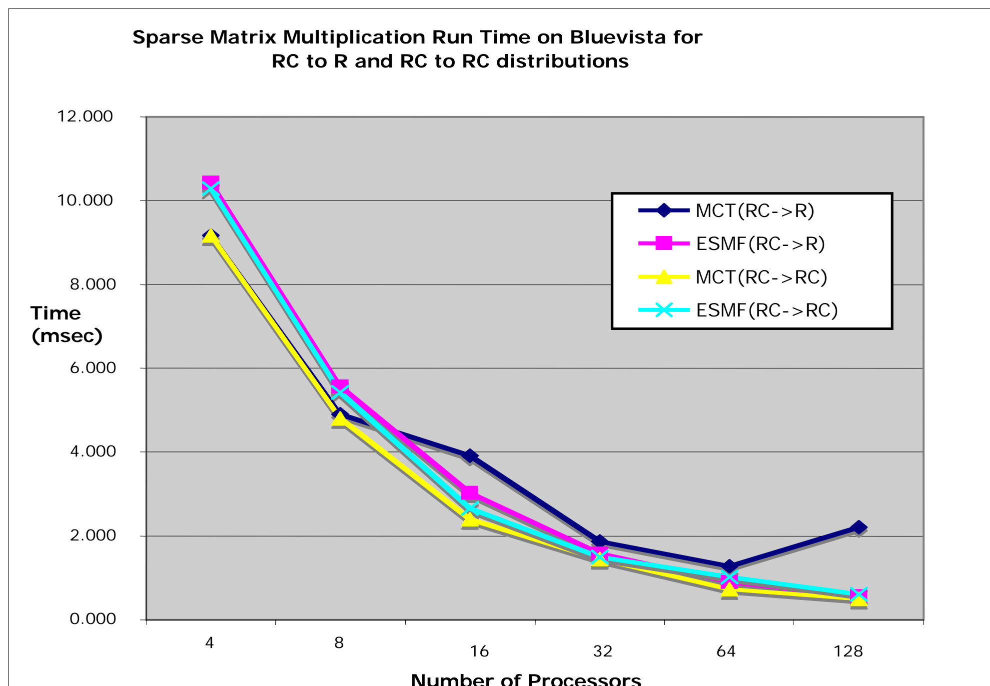

Figure 4 ASMM Run Time for more data distributions

Timing Comparison on different platforms

We ran the benchmark program for the RC to C case, in which ESMF performed best compared to MCT, on the Cray XT3/XT4 (jaguar) at Oak Ridge National Lab and on the SGI Altix (Columbia) at NASA Ames. Other distributions (RC to RC, RC to R) were not examined. The results are shown in Figure 5 to 8. We ran MCT using both Xonly and Yonly options. The results are consistent with the results observed on the IBM SP. The ESMF timing comparisons on three different platforms are shown in Figure 9 and 10. In summary, the Cray XT3/4 has the best performance for both the ASMM initialization and run time. However, it also shows the poorest scalability in ASMM Init with large number of processor counts.

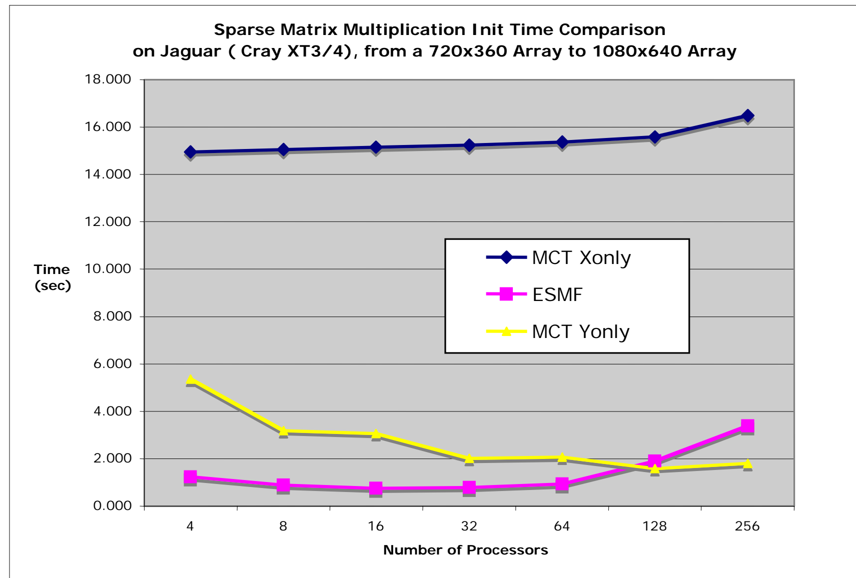

Figure 5 ASMM Init Time Comparison on Jaguar, RC to C

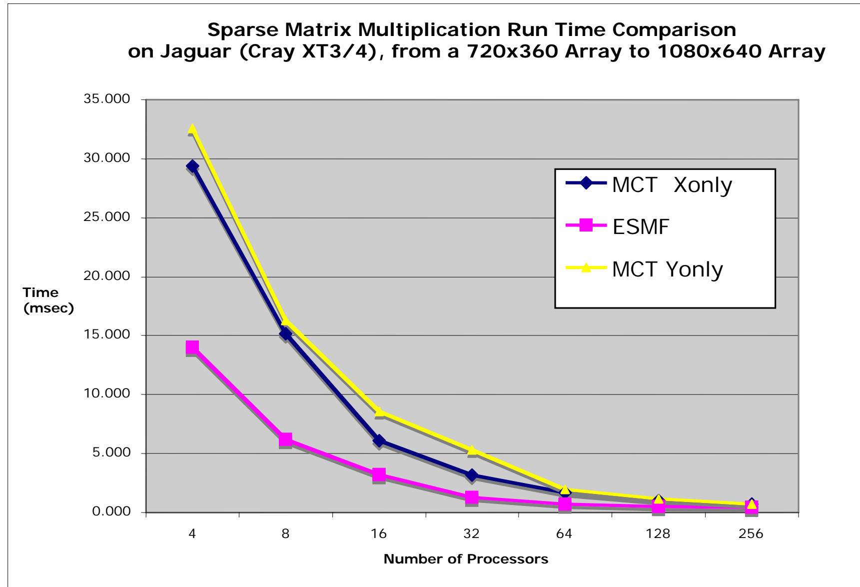

Figure 6 ASMM Run Time Comparison on Jaguar, RC to C

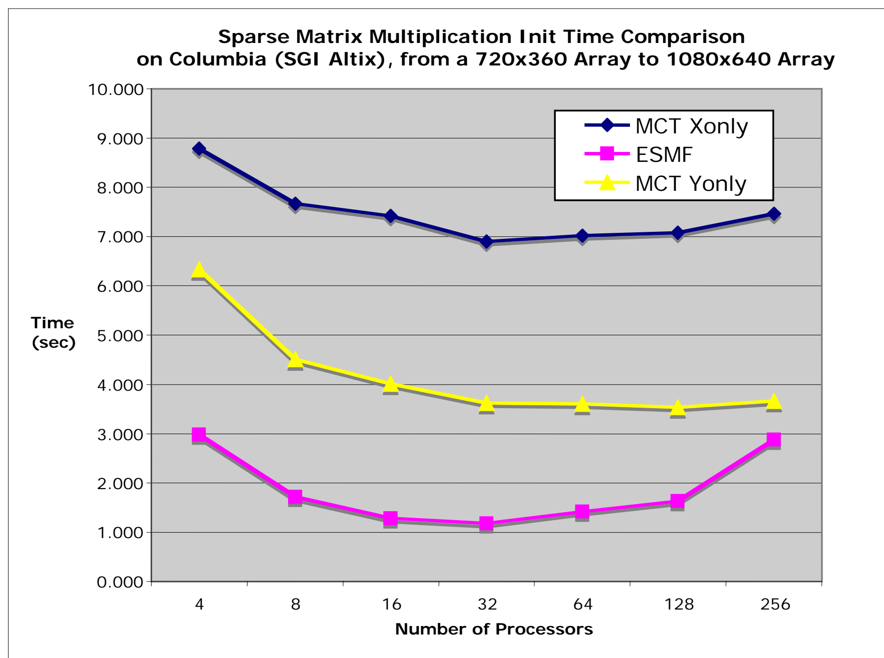

Figure 7 ASMM Init Time Comparison on Columbia, RC to C

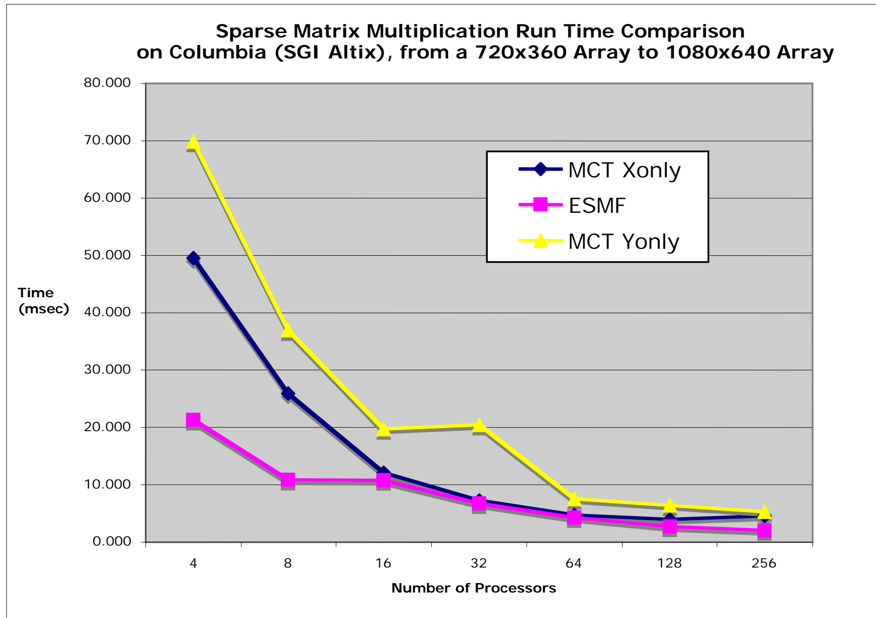

Figure 8 ASMM Run Time Comparison on Columbia, RC to C

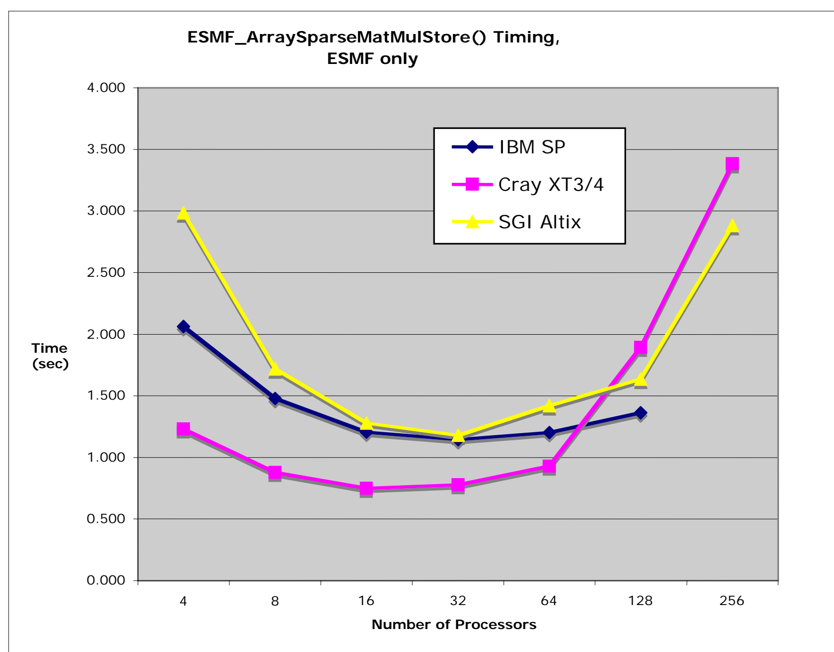

Figure 9 ASMM Init Timing on three different machines., ESMF only

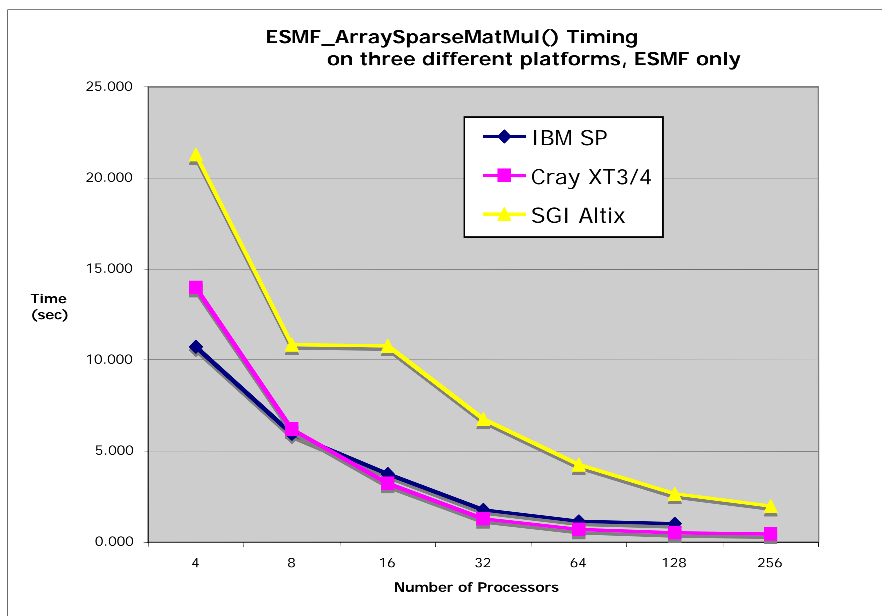

Figure 10 ASMM Run Time Comparison on three different machines, ESMF only

ASMM Scalability Study

We conducted the ASMM Scalability Benchmark on the Cray XT3/4 using up to 2048 processors. We use the same benchmark program as described above but bigger source and destination arrays. The source array is a 1024x1024 2D floating point array and the destination array is a 2400x960 2D floating point array. The source array is decomposed using the block distribution (RC) and the destination array is decomposed using Column-only (C) or Row-only (R) distributions. We had various problems running the ESMF benchmark on 1024 and more processors due to the internal buffer limitation set in the Cray MPI implementation. The working setting after several trials is: MPICH_PTL_UNEX_EVENTS = 40000 and MPICH_PTL_OTHER_EVENTS = 5000. The default settings for these two environment variables are 20,480 and 2048, respectively. We used a newer version of ESMF code, i.e. ESMF_3_1_0_beta_snapshot_21 where it has more optimization in ASMM for large PE counts. The benchmark runs were conducted after Jaguar's OS upgrade to CNL (Compute Node Linux).

Figure 11 and 12 show the timing results for RC to C distribution. We used Y only option for the MCT code since X only option performs much poorer when the destination array is distributed column wise. The MCT code got segmentation fault in m_SparseMatrixPlus::init() for 1024 and 2048 PEs. Therefore, we have no data for MCT beyond 512 PEs. This may be related to memory issues that were reported to be addressed in the MCT 2.4.0 release. The ESMF outperforms MCT in both the initialization and run time for all the configurations. The initialization time remains constant up to 512 PEs and starts to increase from 1024PEs. The run time scales down near linearly until 256 PEs and starts to increase for 1024PEs and beyond. The Y axis in

Figure 12 is in logarithmic scale, we can tell both MCT and ESMF scale linearly for up to 256PEs. Table 2 shows the timing results used in Figure 11 and 12.

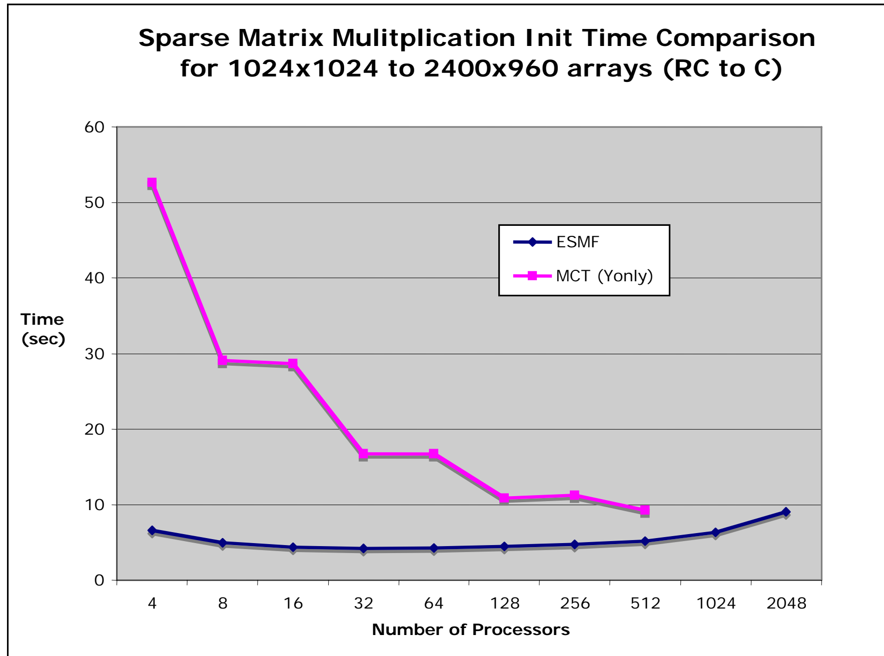

Figure 11 ASMM Init time comparison for large PE counts (RC to C distribution)

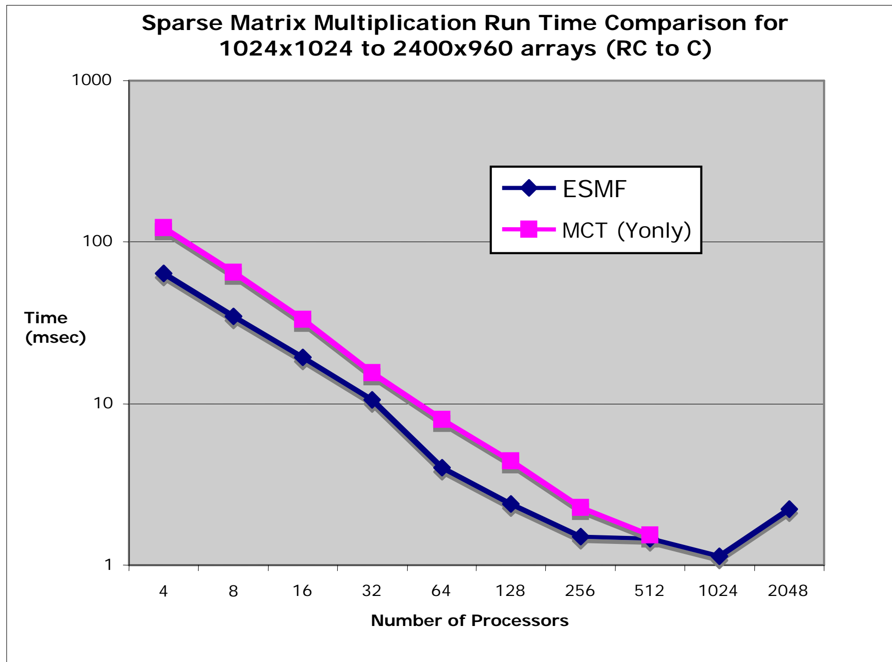

Figure 12 ASMM Run Time comparison for large PE counts (RC->C distribution)

Table 2 The ASMM Timing from a 2048x2048 array to a 2400x960 array using RC to C distribution

	SparseMatMulStore (sec)		SparseMatMulRun (msec)	
#processors	MCT (Yonly)	ESMF	MCT (Yonly)	ESMF
4	52.650	6.617	122.548	63.825
8	29.095	4.945	64.888	34.585
16	28.702	4.368	33.075	19.478
32	16.759	4.226	15.554	10.550
64	16.768	4.253	7.987	4.014
128	10.854	4.497	4.416	2.386
256	11.227	4.768	2.269	1.504
512	9.272	5.177	1.538	1.458
1024	seg fault	6.185		1.136
2048	seg fault	8.830		2.220

Figure 13 and 14 are the timing results for an RC->R distribution. We use X only option for the MCT code and it scales nicely all the way to 2048 PEs. The run time for RC to R distribution is faster than RC to C distribution because for both ESMF and MCT the destination array has only 960 rows, therefore some processors have no data at all when using more than 960 PEs. As a consequence, the total number of messages communicated is fewer than the RC to C distribution. From Figure 14, we can see the ASMM run routine scales linearly up to 512 PEs and starts to slow down form 1024. Table 3 shows the timing results used in Figure 13 and 14.

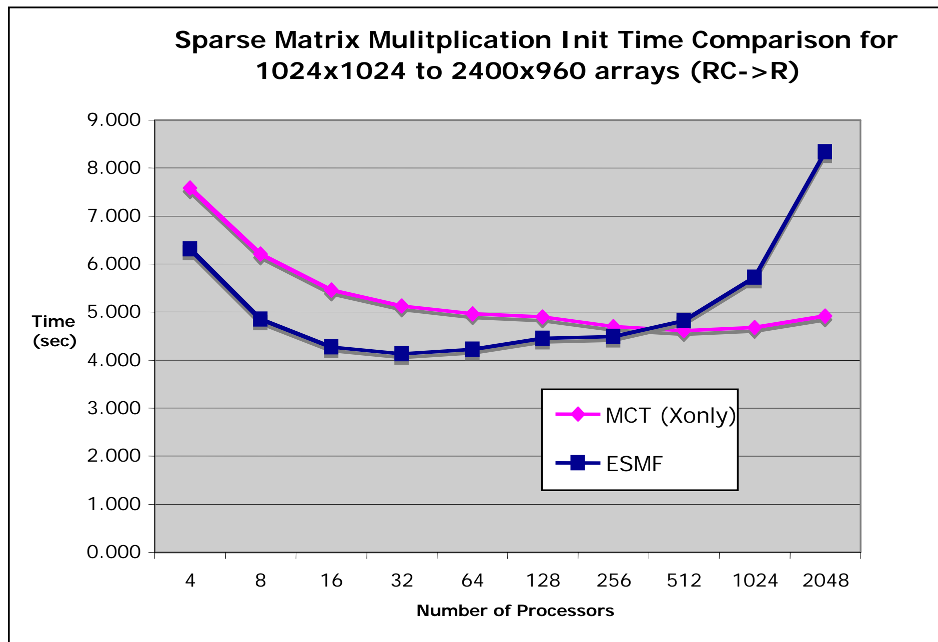

Figure 13 ASMM Init time for large PE counts (RC->R distribution)

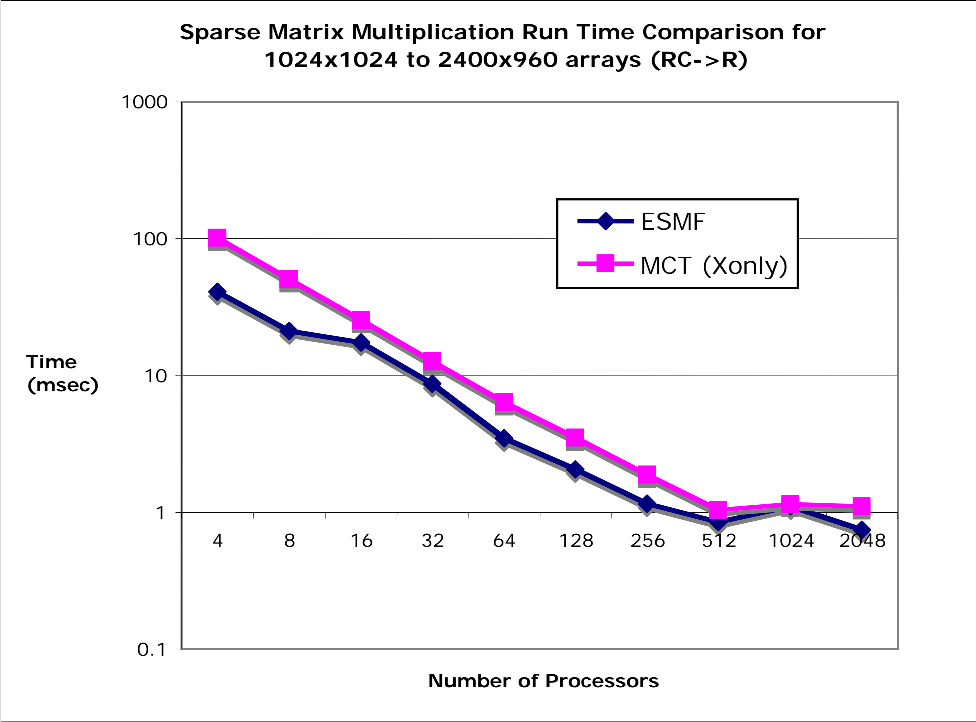

Figure 14 ASMM Run time for large PE counts (RC->R distribution)

Table 3 The ASMM Timing from a 2048x2048 array to a 2400x960 array using RC to R distribution

	SparseMatMulStore (sec)		SparseMatMulRun (msec)	
#processors	MCT (Xonly)	ESMF	MCT (Xonly)	ESMF
4	7.581	6.317	100.738	40.918
8	6.206	4.854	50.396	21.185
16	5.462	4.273	25.428	17.335
32	5.128	4.134	12.645	8.672
64	4.965	4.228	6.377	3.479
128	4.898	4.456	3.511	2.057
256	4.701	4.495	1.882	1.159
512	4.618	4.819	1.031	0.849
1024	4.679	5.662	1.143	1.107
2048	4.922	8.254	1.099	0.745

Memory Usage

The ESMF ASMM memory usage was measured using the XT3 function heap_info() before jaguar's OS upgrade. The ESMF used was ESMF_3_1_0_beta_snapshot_19. Heap_info() returns the number of segments and total number of bytes used in the heap. It reflects the dynamic memory allocated by ESMF. I first called heap_info() after all the ESMF_CompCreate() and ESMF_SetServices() calls and again after 50 iterations of ESMF_CompRun() routines. The results presented here are the memory usage from PE0. The memory used in the first call is mainly ESMF overhead to set up the components and their services. The difference of the two calls reflects the memory used to allocate the arrays, the sparse matrix and the ESMF overhead to handle sparse matrix multiplication and communication. Figure 15 shows the memory usage doubled from 512 PEs to 1024 PEs and tripled from 1024 to 2048 PEs. It demonstrated that the ESMF memory utilization for component setup does not scale well for 256 PEs and more. The ASMM memory usage scales pretty nicely for small number of PEs. It starts to increase with 512 PEs as shown in Figure 16.

Table 4 ESMF VM/Array/ASMM Memory Usage

	Before ASMMStore		After ASMM Run		ASMM Usage	
#processors	Kbytes	segments	kbytes	segments	kbytes	segments
8	100425	5273	138937	6298	38512	1025
16	100468	5482	120253	6513	19785	1031
32	100621	5915	111428	7008	10807	1093
64	101185	6778	109041	7869	7856	1091
128	103356	8503	109629	9713	6273	1210
256	111822	11959	116982	13172	5160	1213
512	145264	18877	151426	20316	6162	1439
1024	278198	32707	285845	34113	7647	1406
2048	808330	60352	819139	62355	10809	2003

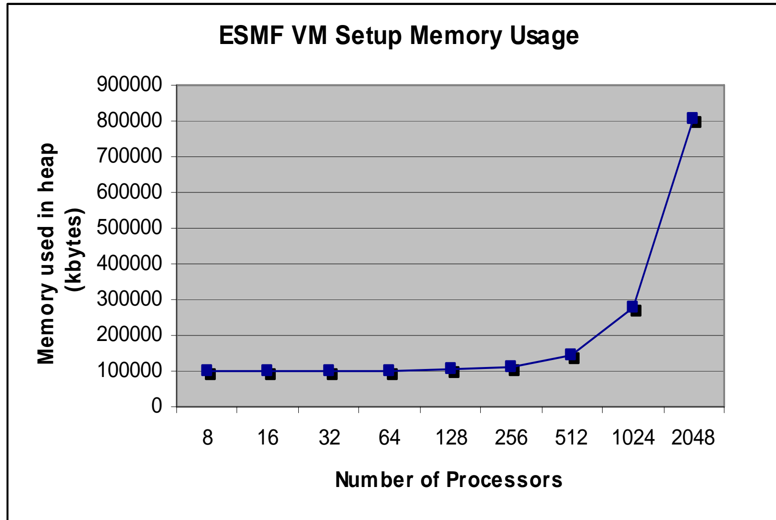

Figure 15 ESMF VM/Component Memory Usage

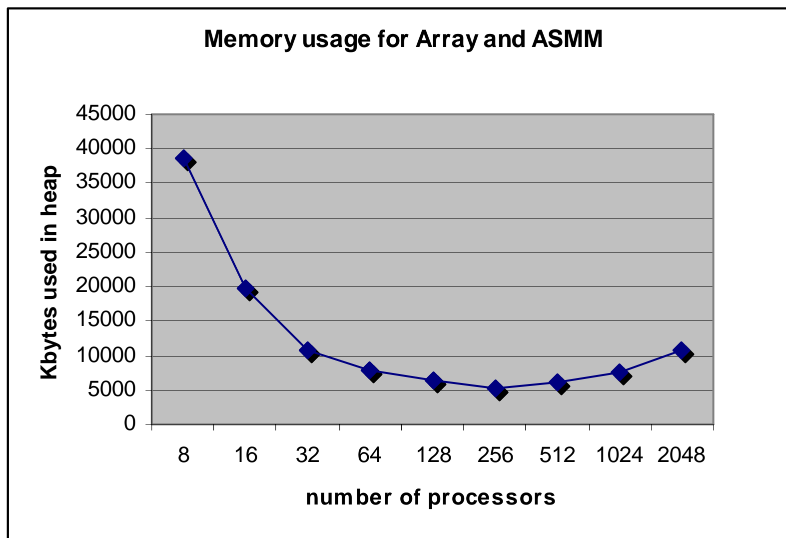

Figure 16 ESMF Array/ASMM Memory Usage

Conclusion

This benchmark measures the performance of the Array Sparse Matrix Multiplication functions in the ESMF version 3.1.0 (scheduled to release in December 2007). The performance of the ESMF ASMM has been optimized tremendously from version 3.0.1 to 3.1.0 during the benchmark process. For this study, we can see the ESMF ASMM functions are roughly comparable in performance to the corresponding MCT Sparse Matrix functions, with the results varying by platform, decomposition, and processor count. In the scalability test, we found that the ESMF ASMM functions perform reasonably well for up to 2048 PEs. We do see slight degradation in both the init and run routines for 512 and more PEs. Thus, we may have to revisit the performance issue when we need to run ESMF on more than 10,000 processors.

The memory usage in the ESMF VM implementation presents more serious scalability problem. This should be the next target for optimization.

Acknowledgements: Thanks very much to Rob Jacob of ANL for reviewing this report and providing valuable observations and comments.