A High Performance Parallel Strassen Implementation *

Brian Grayson
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
bgrayson@pine.ece.utexas.edu

Ajay Pankaj Shah
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
ajay@cs.utexas.edu

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

June 13, 1995

Abstract

In this paper, we give what we believe to be the first high performance parallel implementation of Strassen's algorithm for matrix multiplication. We show how under restricted conditions, this algorithm can be implemented plug compatible with standard parallel matrix multiplication algorithms. Results obtained on a large Intel Paragon system show a 10-20% reduction in execution time compared to what we believe to be the fastest standard parallel matrix multiplication implementation available at this time.

1 Introduction

In Strassen's algorithm, the total time complexity of the matrix multiplication is reduced by replacing it with smaller matrix multiplications together with a number of matrix additions, thereby reducing the operation count. A net reduction in execution time is attained only if the reduction in multiplications offsets the increase in additions. This requires the matrices to be relatively large before a net gain is observed. The advantage of using parallel architectures is that much larger problems can be stored in the aggregate memory of a parallel computer. This is offset by the added complexity that when smaller matrix multiplications are performed, communication overhead is more significant, reducing the performance of the matrix multiply. Thus, for parallel implementations

*This work is partially supported by the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project under NRA Grant NAG5-2497. Additional support came from the Intel Research Council. Brian Grayson is supported by a National Science Foundation Graduate Fellowship.

it is relatively easy to grow the problem to the point where the additional additions are insignificant, however, it is difficult to make sure that the net gain is not offset by an overall reduction in performance due to communication overhead.

This research was started when a paper by Luo and Drake [15] on the scalable implementation of Strassen's matrix multiply algorithm came to our attention. While the algorithm in that paper has attributes that are required to obtain efficient implementations of the method, the performance data in that paper was less than convincing, leading us to believe that a high performance implementation had not yet been achieved.

There are two developments that made us realize that we were in a position to duplicate and improve upon the results by Luo and Drake: first, with the advent of the Paragon, communication overhead was considerably less that that encountered on the iPSC/860 used by Luo and Drake. Second, we had just developed a more efficient standard parallel matrix multiply routine (SUMMA) [17] than the library routine used by Luo and Drake (PUMMA) [3]. Our SUMMA implementation has the added benefit that it requires much less work space than PUMMA, allowing us to run larger problems. Thus, while Luo and Drake concluded that the best approach was to use sequential Strassen implementations on each node, but use a standard parallel matrix multiply for the parallelization, we will show that under these new circumstances, the best approach is to parallelize Strassen, using standard parallel matrix multiplication at a lower level.

2 Background

We start by considering the formation of the matrix product

$$C = AB$$

where $C \in \mathbf{R}^{m\times n}$, $A \in \mathbf{R}^{m\times k}$ and $B \in \mathbf{R}^{k\times n}$. We will assume that m, n, and k are all *even* integers. By partitioning

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right),\ B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right),\ \text{and } C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

where $C_{kl} \in \mathbf{R}^{\frac{m}{2}\times\frac{n}{2}}$, $A_{kl} \in \mathbf{R}^{\frac{m}{2}\times\frac{k}{2}}$, and $B_{ij} \in \mathbf{R}^{\frac{k}{2}\times\frac{n}{2}}$, it can be shown [9, 16] that the following computations compute $C = AB$:

$$\begin{array}{rclrcl}
P_1 &=& (A_{11}+A_{22})(B_{11}+B_{22}) & P_2 &=& (A_{21}+A_{22})B_{11} \\
P_3 &=& A_{11}(B_{12}-B_{22}) & P_4 &=& A_{22}(B_{21}-B_{11}) \\
P_5 &=& (A_{11}+A_{12})B_{22} & P_6 &=& (A_{21}-A_{11})(B_{11}+B_{12}) \\
P_7 &=& (A_{12}-A_{22})(B_{21}+B_{22}) & & & \\
& & & & & \\
C_{11} &=& P_1+P_4-P_5+P_7 & C_{12} &=& P_3+P_5 \\
C_{21} &=& P_2+P_4 & C_{22} &=& P_1+P_3-P_2+P_6
\end{array} \tag{1}$$

While the cost of the original multiplication is ordinarily about mnk multiplies and mnk adds, this reformulation has a cost of $\frac{7}{8}mnk$ multiplies and $\frac{7}{8}mnk$ adds, plus the cost of performing the matrix adds in 1. Moreover, if each of the multiplies in 1 are themselves recursively performed in the same way, one can bring down the order of the computation from $O(n^3)$ to $O(n^{2.807})$ for square matrices [16].

Restriction: In the rest of this paper, we will assume that $n = m = k$. Moreover, every time a level of Strassen is applied, we will assume the dimension is even.

3 Parallel implementation of Strassen's algorithm

In general, one can use Strassen's algorithm in a multicomputer environment in several different ways, with different advantages and disadvantages. For example, one can use the Strassen algorithm only on each processor, for the local matrix multiplies. The advantage to this is that no interprocessor communication is required during the Strassen matrix multiplication stage. However, the relative cost of the extra additions is greatly reduced when the matrix size is large. This leads to the observation that the Strassen algorithm should give better speedup when it is used across all processors, with some other matrix multiply method used for multiplying smaller submatrices. We chose this second method for this research due to its higher potential for speedup.

3.1 Parallel algorithm

The following observation will become important in our parallel implementation: Consider the alternative blocking of the matrices

$$PAP^T = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right),\ PBP^T = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right),\ \text{and } PCP^T = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

where P and is a permutation matrix. Then

$$\begin{aligned} C &= P^T \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) P = P^T \left(\begin{array}{c|c} A_{11} & B_{12} \\ \hline A_{21} & B_{22} \end{array}\right) \left(\begin{array}{c|c} B_{11} & A_{12} \\ \hline B_{21} & A_{22} \end{array}\right) P & (2) \\ &= P^T \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) PP^T \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) P = AB & (3) \end{aligned}$$

We will use this to derive a supprisingly straight forward parallel implementation for Strassen's algorithm in the special case where the mesh of nodes is square.

> **Restriction:** We will only consider the case where the logical mesh of nodes is square. I.e. if there are p nodes, they will be viewed as forming an $r \times r$ logical mesh, with $r = \sqrt{p}$.

We will use the following partitioning of the matrices to distribute the data to nodes:

$$C = \left(\begin{array}{c||c||c||c||c} C^{0,0} & C^{0,1} & C^{0,2} & \cdots & C^{0,r-1} \\ \hline\hline C^{1,0} & C^{1,1} & C^{1,2} & \cdots & C^{1,r-1} \\ \hline\hline \vdots & \vdots & \vdots & & \vdots \\ \hline\hline C^{r-1,0} & C^{r-1,1} & C^{r-1,2} & \cdots & C^{r-1,r-1} \end{array}\right)$$

Matrices A and B are partitioned conformally. The assumption will be that $C^{i,j}$, $A^{i,j}$, and $B^{i,j}$ are all assigned to node (i, j). Hence, the double lines in the equation can be interpreted as borders between nodes.

Next, let us subdivide further:

$$C = \left(\begin{array}{c||c||c||c||c} \begin{array}{c|c} C^{0,0}_{11} & C^{0,0}_{12} \\ \hline C^{0,0}_{21} & C^{0,0}_{22} \end{array} & \begin{array}{c|c} C^{0,1}_{11} & C^{0,1}_{12} \\ \hline C^{0,1}_{21} & C^{0,1}_{22} \end{array} & \begin{array}{c|c} C^{0,2}_{11} & C^{0,2}_{12} \\ \hline C^{0,2}_{21} & C^{0,2}_{22} \end{array} & \cdots & \begin{array}{c|c} C^{0,r-1}_{11} & C^{0,r-1}_{12} \\ \hline C^{0,r-1}_{21} & C^{0,r-1}_{22} \end{array} \\ \hline\hline \begin{array}{c|c} C^{1,0}_{11} & C^{1,0}_{12} \\ \hline C^{1,0}_{21} & C^{1,0}_{22} \end{array} & \begin{array}{c|c} C^{1,1}_{11} & C^{1,1}_{12} \\ \hline C^{1,1}_{21} & C^{1,1}_{22} \end{array} & \begin{array}{c|c} C^{1,2}_{11} & C^{1,2}_{12} \\ \hline C^{1,2}_{21} & C^{1,2}_{22} \end{array} & \cdots & \begin{array}{c|c} C^{1,r-1}_{11} & C^{1,r-1}_{12} \\ \hline C^{1,r-1}_{21} & C^{1,r-1}_{22} \end{array} \\ \hline\hline \vdots & \vdots & \vdots & & \vdots \\ \hline\hline \begin{array}{c|c} C^{r-1,0}_{11} & C^{r-1,0}_{12} \\ \hline C^{r-1,0}_{21} & C^{r-1,0}_{22} \end{array} & \begin{array}{c|c} C^{r-1,1}_{11} & C^{r-1,1}_{12} \\ \hline C^{r-1,1}_{21} & C^{r-1,1}_{22} \end{array} & \begin{array}{c|c} C^{r-1,2}_{11} & C^{r-1,2}_{12} \\ \hline C^{r-1,2}_{21} & C^{r-1,2}_{22} \end{array} & \cdots & \begin{array}{c|c} C^{r-1,r-1}_{11} & C^{r-1,r-1}_{12} \\ \hline C^{r-1,r-1}_{21} & C^{r-1,r-1}_{22} \end{array} \end{array}\right)$$

where now $C^{i,j}_{kl}$ are $\frac{n}{2r} \times \frac{n}{2r}$. Again, A and B are subdivided conformally.

We can find a permutation P so that

$$PCP^T = \left(\begin{array}{c|c} \begin{array}{c||c||c} C_{1,1}^{0,0} & \cdots & C_{1,1}^{0,r-1} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{1,1}^{r-1,0} & \cdots & C_{1,1}^{r-1,r-1} \end{array} & \begin{array}{c||c||c} C_{1,2}^{0,0} & \cdots & C_{1,2}^{0,r-1} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{1,2}^{r-1,0} & \cdots & C_{1,2}^{r-1,r-1} \end{array} \\ \hline \begin{array}{c||c||c} C_{2,1}^{0,0} & \cdots & C_{2,1}^{0,r-1} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{2,1}^{r-1,0} & \cdots & C_{2,1}^{r-1,r-1} \end{array} & \begin{array}{c||c||c} C_{2,2}^{0,0} & \cdots & C_{2,2}^{0,r-1} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{2,2}^{r-1,0} & \cdots & C_{2,2}^{r-1,r-1} \end{array} \end{array} \right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right)$$

Let matrices A and B be permuted using the same permutation matrix. The double lines continue to denote borders between nodes, except that now C_{kl}, $k, l \in \{1, 2\}$ are each partitioned and distributed to nodes.

If we now look at a typical operation that is part of Strassen's algorithm

$$\begin{aligned} T_1 &= (A_{11} + A_{22}) \\ T_2 &= (B_{11} + B_{22}) \\ P_1 &= T_1 T_2 \end{aligned}$$

we notice that forming T_1 is equivalent to performing the operations $T_1^{i,j} = A_{1,1}^{i,j} + A_{2,2}^{i,j}$, where

$$T_1 = \left(\begin{array}{c||c||c||c||c} T_1^{0,0} & T_1^{0,1} & T_1^{0,2} & \cdots & T_1^{0,r-1} \\ \hline\hline T_1^{1,0} & T_1^{1,1} & T_1^{1,2} & \cdots & T_1^{1,r-1} \\ \hline\hline \vdots & \vdots & \vdots & & \vdots \\ \hline\hline T_1^{r-1,0} & T_1^{r-1,1} & T_1^{r-1,2} & \cdots & T_1^{r-1,r-1} \end{array} \right)$$

is partitioned and distributed like C_{kl}. Similarly, T_2 can be formed in parallel from B_{11} and B_{22}. Next, the multiplication $P_1 = T_1 T_2$ requires a parallel matrix multiply, leaving P_1 distributed like C_{kl}. The same can be shown for the formation of P_k, $i = 1, \ldots, 7$.

A second typical computation,

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

is also perfectly parallel, without requiring communication, since C_{kl}, $k, l \in \{1, 2\}$, and P_i, $i = 1, \ldots, 7$ all have corresponding elements assigned to the same nodes. Notice that now each matrix block C_{kl} must have even dimensions.

Restriction: Matrix dimension n must be an even multiple of $r = \sqrt{p}$.

Thus, our implementation does the following:

1. Partition the matrices such that each processor has an equal part of each quadrant of the original matrices.
2. Calculate each contibution to the final matrix by performing local matrix additions and parallel matrix multiplications.
3. For very large matrices, it may be appropriate to apply this method recursively, using the standard matrix multiplication at the lowest level.

Notice this approach allows many levels of Strassen's method to be performed *without the requirement of rearranging the matrix.* This means we have derived a parallel implementation of Strassen's algorithm that can be made plug-compatible with a standard parallel matrix multiplication routine.

Using the Strassen method for all levels is not necessarily the most efficient method for matrix multiply. In our implementation, which uses Strassen for large cross-processor matrix multiplies, the Scalable Universal Matrix Multiplication Algorithm (SUMMA) (see appendix) for smaller cross-processor matrix multiplies, and level 3 BLAS calls [5] for local matrix multiplies. The choice of how many levels of Strassen to perform is dependent on the matrix size, as is shown in the next section.

3.2 Performance Results

Our Strassen implementation differs from those attempted before in two ways: As shown in the previous section, we can avoid expensive and complicated data rearrangements by taking advantage of implicit permutations. Second, we actually achieve high performance, as will be shown in this section.

High performance dense linear algebra algorithm implementations are best measured by how many millions of floating point computations per second (MFLOPS) they achieve. To compute this, the operations count, $2n^3$ for standard matrix multiplication, is divided by the time required for the operation to complete. By comparing this performance to the peak performance of the computer, we gain insight into how efficiently the algorithm utilizes the architecture.

The platform we used for our experiments in a 512 node Intel Paragon system. This system is physically a 16×32 mesh of nodes. Each node consists of an Intel i860/XP computation processor and second i860/XP communication processor. On the system we used, the nodes have 32 Mbytes of primary memory each. All tests were performed using the OSF (release R1.3) operating system.

We based our Strassen implementation on an implementation of the SUMMA matrix multiplication algorithm, implemented using double-precision and NX point-to-point calls. In [17] we show that this algorithm achieves a performance of around 45 MFLOPS/node for large matrices. This is essentially the theoretical peak for standard matrix multiplication on a single node.

The MFLOPS performance reported for Strassen's algorithm also used the $2n^3$ operation count of the standard algorithm. Of course, Stassen's algorithm doesn't perform all these operations. Nonetheless, we believe this to be the best way to highlight the performance improvement attained.

In Figure 1 we show the performance attained by Strassen's algorithm on a single node. This gives an idea of what could be achieved if Strassen's algorithm was only used to accelerate the matrix multiplication on an individual node, on top of which a standard parallel matrix multiplication is implemented. The limiting factor is the trade-off between the reduction in multiplications vs. the increase in additions. Our SUMMA algorithm cannot take advantage of this approach, since it is not based on large square matrix multiplications on each node (see [17] for details).

In Fig. 2, we show the performance of implementations that use a varying number of levels of Strassen's algorithm. The standard matrix multiplication algorithm is given by `level = 0`. Notice a performance improvement is observed for `level=1,2,3`. SUMMA is shown to be scalable in [17], in the sense that when memory use per node is kept constant, efficiency (MFLOPS/node) remains essentially constant. An analysis of our Strassen implementation would show that its scalability is directly connected to the scalability of the standard parallel matrix multiplication algorithm used for the lowest level multiplication. This is observed in practice, as shown in Figs. 3 and 6.

Although our implementation has achieved speedup over SUMMA, there are more improvements that we are currently investigating. These include:

- A more effient broadcast in the SUMMA procedure using non-blocking calls.
- Overlapping computation and communication through the use of loop unrolling.
- Optimizing the matrix addition and subtraction subroutines.

These optimizations may affect the threshold of where SUMMA becomes just as efficient as Strassen's algorithm as well as how many levels of Strassen's algorithm can be used. Observe that if the standard parallel matrix multiplication algorithm is optimized so it achieves high performance for smaller matrices, the parallel Strassen's algorithm benefits, since potentially more levels of Strassen's algorithm can be effectively used.

4 Conclusion

The presented parallel algorithm for matrix multiplication is considerably simpler than those previously presented, all of which have been based on generalizations of the broadcast-multiply-roll algorithm. Nonetheless, performance is impressive, as is its flexibility.

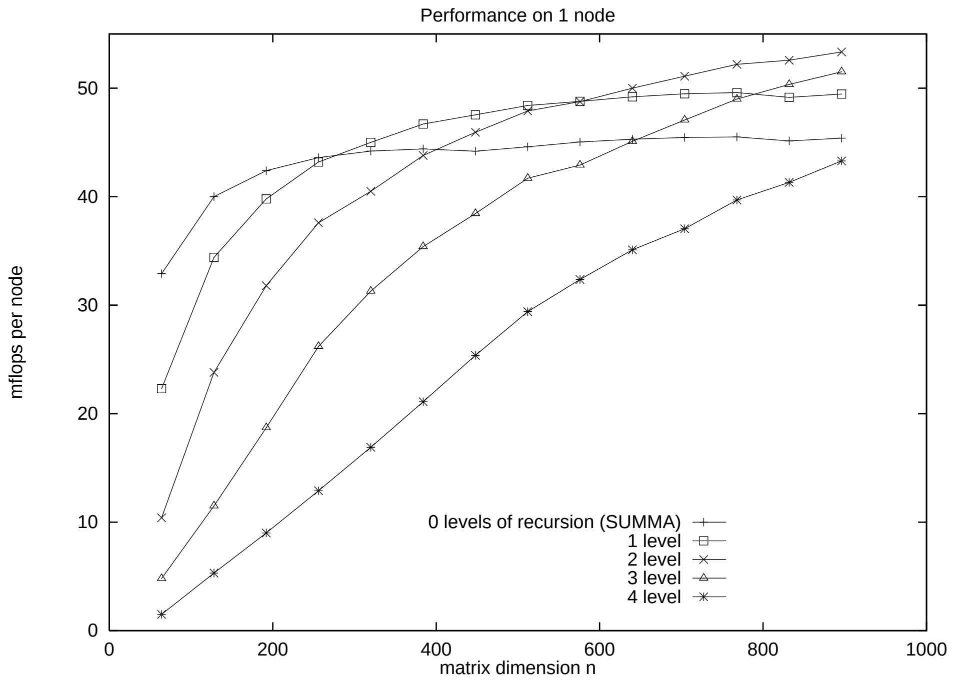

Figure 1: Performance of SUMMA and Strassen using NX on 1 node.

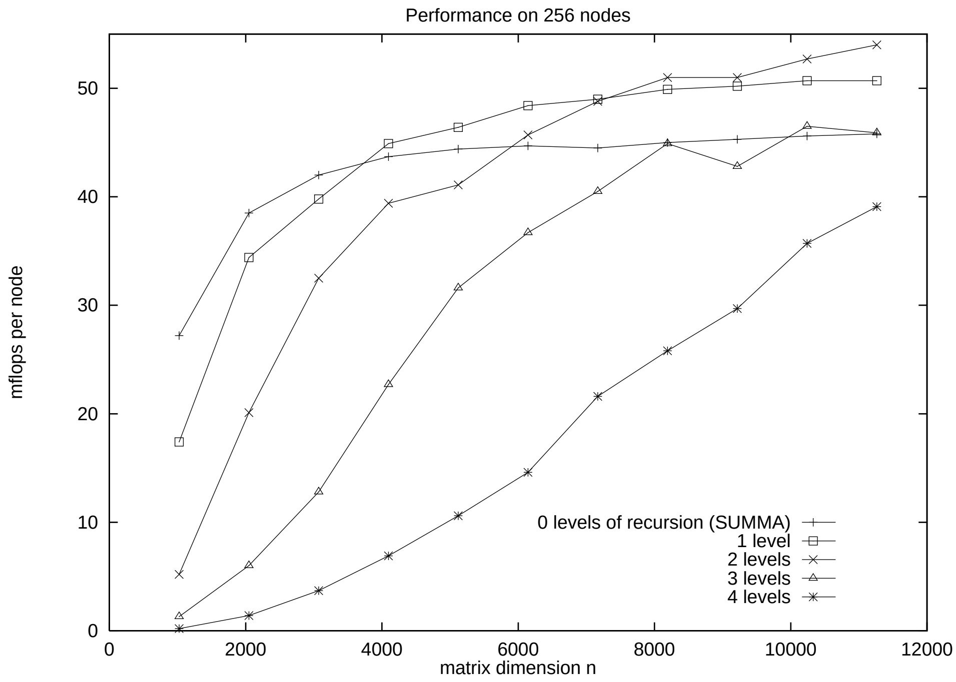

Figure 2: Performance of SUMMA and Strassen using NX on 256 nodes.

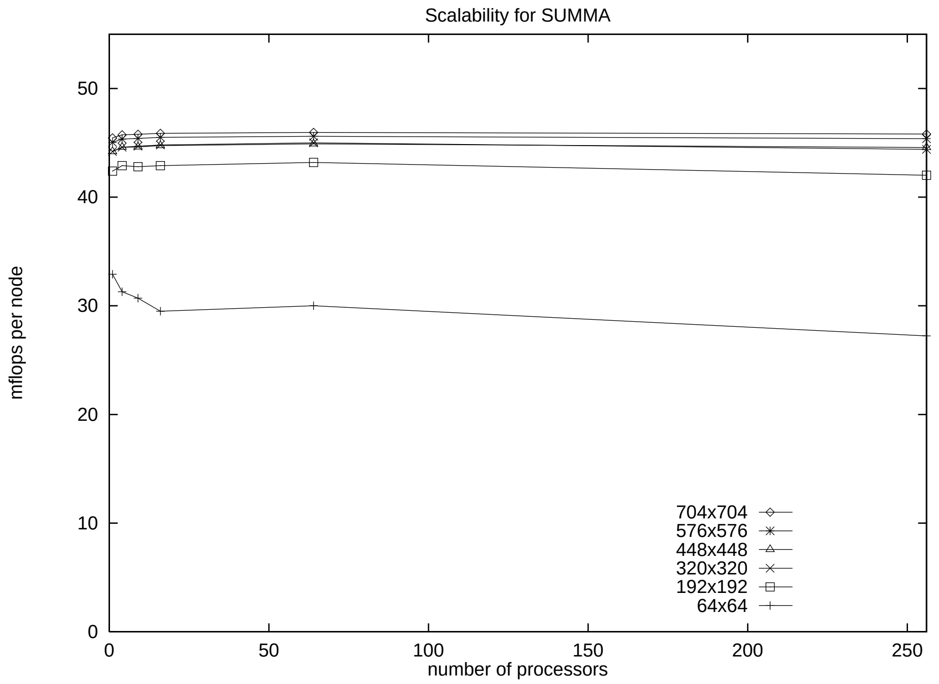

Figure 3: Scalability of the parallel SUMMA implementation. A line joins data points with equal per node memory usage. For example, on a 1×1 mesh, a matrix size of 512×512 is equivalent to a 1024×1024 problem on a 2×2 mesh, because in both cases, each node contains 512×512 matrices.

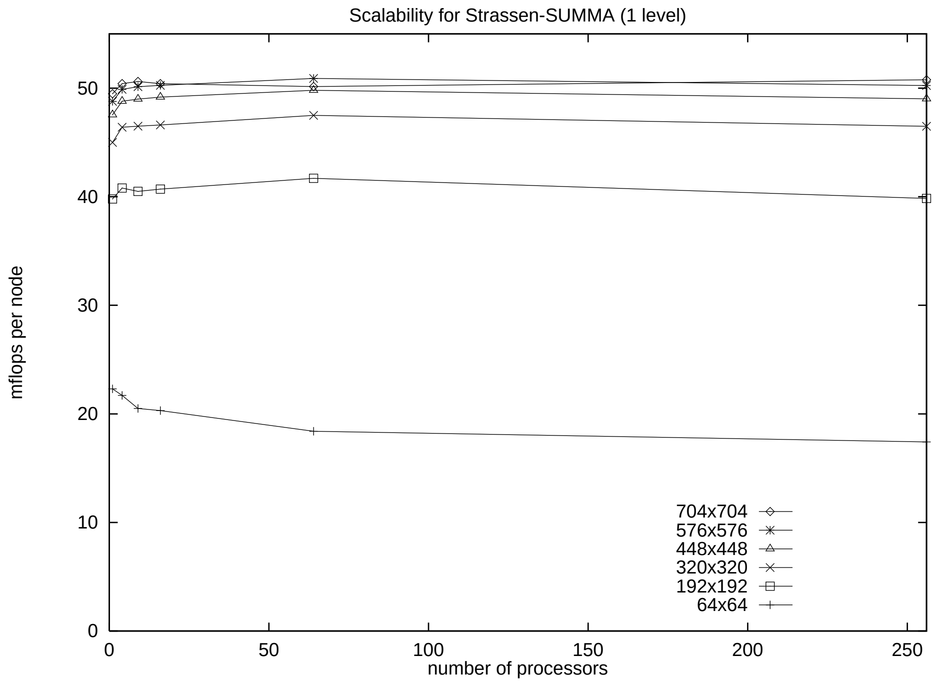

Figure 4: Scalability of the parallel Strassen implementation using one level of recursion.

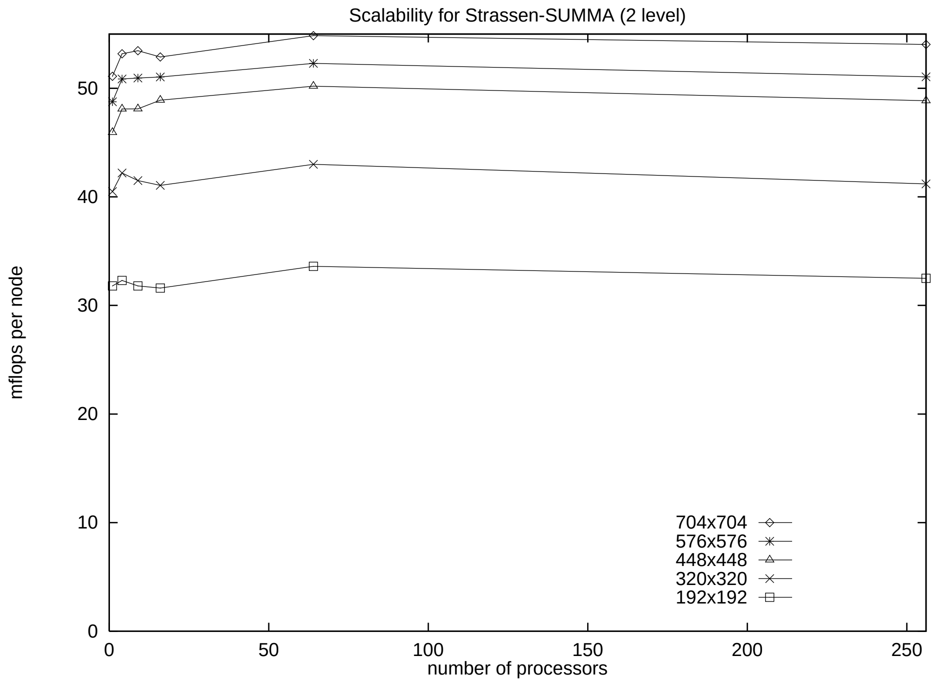

Figure 5: Scalability of the parallel Strassen implementation using two levels of recursion.

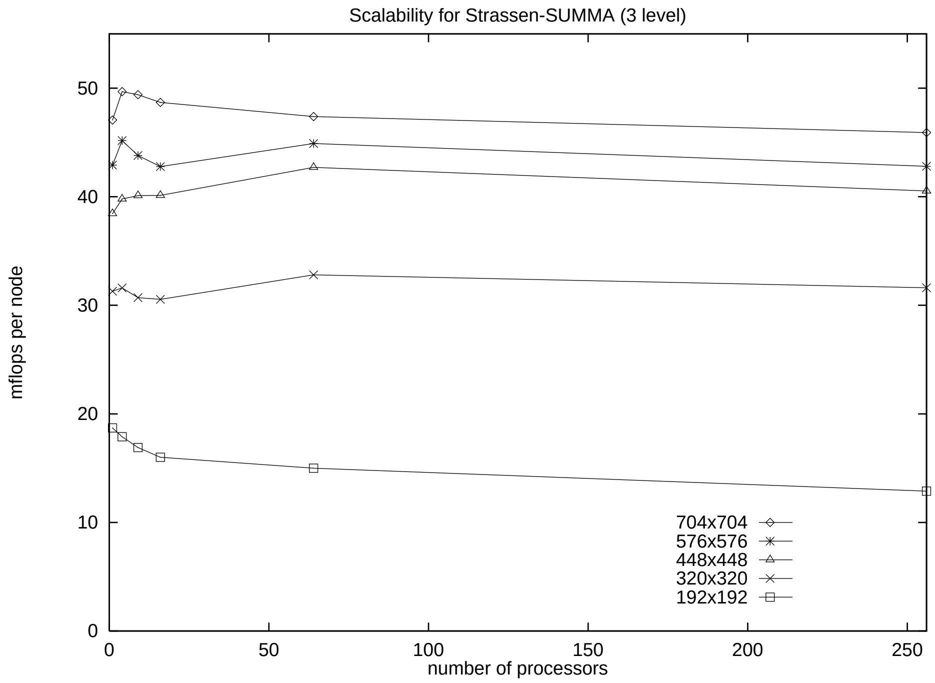

Figure 6: Scalability of the parallel Strassen implementation using three levels of recursion.

We are working on generalizing our results so that the approach will work for arbitrary matrix dimensions and arbitrary (non-square) mesh sizes. Preliminary results show that this can again be done with implicit permutations.

Our implementation can be easily adjusted to use other parallel matrix multiplication implementations for the lowest level multiplication. A number of implementations based on the "broadcast-multiply-role" method [7, 8] have been developed. For details see [3, 11, 12].

Acknowledgements

This research was performed in part using the Intel Paragon System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Intel Supercomputer Systems Division and the California Institute of Technology. This project started as a discussion in a graduate special topics class at UT-Austin. Thus a number of students who did not become coauthors made contributions. We apologize for not being able to acknowledge the specific contributions from these students. As mentioned, our implementation was based on a NX based implementation of SUMMA. This implementation was the result of a collaboration between Robert van de Geijn and Jerrell Watts.

References

[1] Cannon, L.E., *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. Thesis (1969), Montana State University.

[2] Choi, J., Dongarra, J. J., and Walker, D. W., "Level 3 BLAS for distributed memory concurrent computers", *CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*, Saint Hilaire du Touvet, France, Sept. 7-8, 1992. Elsevier Science Publishers, 1992.

[3] Choi, J., Dongarra, J. J., and Walker, D. W., "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, Vol 6(7), 543-570, 1994.

[4] Coppersmith, D., and Winograd, S., Matrix Multiplication via Arithmetic Progressions," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing,* pp. 1–6, 1987.

[5] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I., "A Set of Level 3 Basic Linear Algebra Subprograms," *TOMS*, Vol. 16, No. 1, pp. 1–16, 1990.

[6] Douglas, C., Heroux, M., and Slishman, G., "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm," *Journal of Computational Physics* 110, pp. 1-10, 1994.

[7] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K. and Walker, D. W., *Solving Problems on Concurrent Processors*, Vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.

[8] Fox, G., Otto, S., and Hey, A., "Matrix algorithms on a hypercube I: matrix multiplication," *Parallel Computing* **3** (1987), pp 17-31.

[9] Golub, G. H. , and C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, 2nd ed., 1989.

[10] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: Portable Programming with the Message-Passing Interface*, The MIT Press, 1994.

[11] Huss-Lederman, S., Jacobson E., and Tsao, A., "Comparison of Scalable Parallel Matrix Multiplication Libraries," in *Proceedings of the Scalable Parallel Libraries Conference*, Starksville, MS, Oct. 1993.

[12] Huss-Lederman, S., Jacobson, E., Tsao A., and Zhang, G., "Matrix Multiplication on the Intel Touchstone DELTA," *Concurrency: Practice and Experience*, Vol. 6 (7), Oct. 1994, pp. 571-594.

[13] Laderman, J., Pan, V., and Sha, X., "On Practical Algorithms for Accelerated Matrix Multiplication," *Linear Algebra and Its Applications,* pp. 557–588, 1992.

[14] Lin, C., and Snyder, L., "A Matrix Product Algorithm and its Comparative Performance on Hypercubes," in *Proceedings of Scalable High Performance Computing Conference*, (Stout, Q, and M. Wolfe, eds.), IEEE Press, Los Alamitos, CA, 1992, pp. 190–3.

[15] Luo, Q, and Drake, J. B., "A Scalable Parallel Strassen's Matrix Multiply Algorithm for Distributed Memory Computers",

[16] Strassen, V., "Gaussian Elimination is not optimal," *Numer. Math.* 13, pp 354–356, 1969.

[17] van de Geijn, R. and Watts, J., "SUMMA: Scalable Universal Matrix Multiplication Algorithm," TR-95-13, Department of Computer Sciences, University of Texas, April 1995. Also: LAPACK Working Note #96, University of Tennessee, CS-95-286, April 1995.