EXPERIMENTS WITH STRASSEN'S ALGORITHM: FROM SEQUENTIAL TO PARALLEL

Fengguang Song, Jack Dongarra, and Shirley Moore
Computer Science Department
University of Tennessee
Knoxville, Tennessee 37996, USA
email: {song, dongarra, shirley}@cs.utk.edu

ABSTRACT
This paper studies Strassen's matrix multiplication algorithm by implementing it in a variety of methods: sequential, workflow, and in parallel. All the methods show better performance than the well-known scientific libraries for medium to large size matrices. The sequential recursive program is implemented and compared with ATLAS's DGEMM subroutine. A workflow program in the NetSolve system and two parallel programs based on MPI and ScaLAPACK are also implemented. By analyzing the time complexity and memory requirement of each method, we provide insight into how to utilize Strassen's Algorithm to speedup matrix multiplication based on existing high performance tools or libraries.

KEY WORDS
Strassen's Algorithm, Matrix Multiplication, Parallel Computing.

1 Introduction

Matrix multiplication is one of the most basic operations of scientific computing and is implemented as a key subroutine in the Level 3 BLAS [3]. Conventional algorithm to perform matrix multiplication is of time complexity $O(n^3)$ and has been investigated extensively. However, an alternative method of Strassen's algorithm is much less investigated. The complexity of Strassen's algorithm is $O(n^{2.807})$, which means it will run faster than the conventional algorithm for sufficiently large matrices. This interesting feature motivated us to perform various experiments on it.

A sequential and three parallel programs have been attempted to implement Strassen's algorithm. We write the sequential program by using the well-known Winograd's method [8, 10]. The sequential program stops its recursion on a certain level where it invokes the subroutine `DGEMM` provided by ATLAS [6]. Since the design of the program is straightforward, we only introduce its performance and instability issues, as well as how they vary with the recursion level. The three parallel programs include one workflow program and two MPI programs. The workflow program is implemented in the client-end on the NetSolve system [7]. It has a workflow controller to check and start the tasks in a task graph (Figure 4). All tasks are sent to the NetSolve servers to compute. When the dependent tasks are finished, the controller launches a new task immediately. The intensive computation is actually performed on the NetSolve servers, thus the client machine is available to run other tasks. Next we adopt two different approaches to design the parallel programs running on distributed memory systems. The first program uses a task-parallel approach, and the second one uses a data-parallel approach which uses the ScaLAPACK library to compute the submatrix multiplications [11, 12].

In the remainder of this paper we first present the related work. Next we briefly recall Strassen's algorithm and compare the sequential program's performance to ATLAS's `DGEMM` subroutine in Section 3. Sections 4, 5, 6 describe the NetSolve workflow, the task-parallel, and the data-parallel approaches, respectively. The experimental result and analysis are provided in Section 7. Finally we offer conclusions and future work in Section 8.

2 Related Work

Huss-Lederman developed an efficient and portable serial implementation of Strassen's algorithm called DGEFMM [8]. DGEFMM is designed to replace DGEMM and obtain better performance for all matrix sizes while minimizing the temporary storage. DGEMMS in the IBM ESSL library implements Strassen's algorithm but only performs the multiplication part of DGEMM, $C = op(A) \times op(B)$ [13].

There are various parallel methods to implement Strassen's algorithm on distributed memory architectures. The methods typically belong to three classes. The first class is to use the conventional algorithm at the top level (across processors) and Strassen's algorithm at the bottom level (within a processor). A commonly used one is Fox's Broadcast-Multiply- Roll (BMR) method [2]. Ohtaki [14] uses a similar method but is more focused on a distribution scheme particularly for heterogeneous clusters. The second class is to use Strassen's algorithm at both the top and bottom levels. Chou [4] decomposes the matrix A into 2×2 blocks of submatrices, then further decomposes each submatrix into four 2×2 blocks (i.e., 4×4 blocks). This way he is able to identify 49 multiplications and uses 7 or

Table 1. Strassen's Algorithm

Phase 1	$T_1 = A_{11} + A_{22}$	$T_6 = B_{11} + B_{22}$
	$T_2 = A_{21} + A_{22}$	$T_7 = B_{12} - B_{22}$
	$T_3 = A_{11} + A_{12}$	$T_8 = B_{21} - B_{11}$
	$T_4 = A_{21} - A_{11}$	$T_9 = B_{11} + B_{12}$
	$T_5 = A_{12} - A_{22}$	$T_{10} = B_{21} + B_{22}$
Phase 2	$Q_1 = T_1 \times T_6$	$Q_5 = T_3 \times B_{22}$
	$Q_2 = T_2 \times B_{11}$	$Q_6 = T_4 \times T_9$
	$Q_3 = A_{11} \times T_7$	$Q_7 = T_5 \times T_{10}$
	$Q_4 = A_{22} \times T_8$	
Phase 3	$T_1 = Q_1 + Q_4$	$T_3 = Q_3 + Q_1$
	$T_2 = Q_5 - Q_7$	$T_4 = Q_2 - Q_6$
Phase 4	$C_{11} = T_1 - T_2$	$C_{12} = Q_3 + Q_5$
	$C_{21} = Q_2 + Q_4$	$C_{22} = T_3 - T_4$

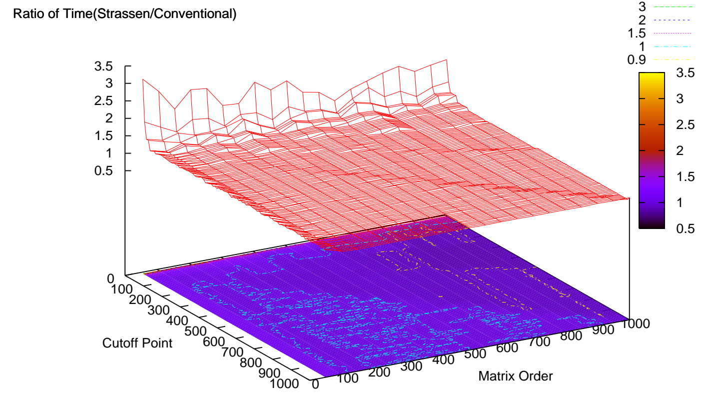

Figure 1. Serial implementation of Strassen's Algorithm compared to ATLAS.

49 processors to perform multiplications concurrently. Our task-parallel program uses a similar idea and proves that this simple approach can outperform ScaLAPACK by up to 20%. The last class is using Strassen's algorithm at the top level and the conventional one at the bottom level. As the matrix size increases, the cost of the extra matrix additions imposed by Strassen's algorithm becomes less compared to the saved matrix multiplication cost. Therefore Strassen's algorithm is better used across processors on the top level. Luo [5] proposes the *Strassen-BMR* method with Strassen's at the top and the BMR method at the bottom. Grayson [9] uses Strassen's algorithm at the top level and derives a new parallel algorithm specifically for a square mesh of nodes. Desprez [12] simultaneously exploits data and task parallelism employing Strassen's algorithm at the top and ScaLAPACK's `PDGEMM` at the bottom. Our data-parallel approach is also based upon ScaLAPACK, but is more generic and does not require that matrices A and B be in two disjoint sets of compute nodes.

3 Strassen's Algorithm

In 1969, Strassen introduced an algorithm to compute matrix multiplications [1]. In contrast to the conventional algorithm which involves 8 multiplications and 4 additions, Strassen's algorithm uses a clever scheme that involves 7 multiplications and 18 additions. The algorithm describes how to perform a single level recursion on 2×2 blocks:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

We can continue to apply Strassen's algorithm recursively to achieve the time complexity of $O(n^{log7}) = O(n^{2.807})$. The single level recursion of the algorithm is illustrated in Table 1, as in [12].

Unless the matrix has a dimension of 2^k you cannot apply Strassen's algorithm directly. A method called *dynamic peeling* is able to solve this problem effectively [10]. Dynamic peeling makes matrix dimensions even by stripping off an extra row or column as needed, and putting their contributions back to the final result later.

Strassen's algorithm requires additional temporary storage and complicates cache blocking; therefore in practice it is slower than the conventional algorithm for small-size matrices. From Table 1, ten temporary variables are needed in phase 1, seven in phase 2, and four in phase 3. Jacobson [8] suggests an optimized method of memory usage which requires only three temporaries.

Strassen's algorithm suffers from an instability problem due to cancellation errors in the algorithm. Stopping the recursion early and performing the bottom-level matrix multiplication with the conventional method can eliminate the problem. We use ATLAS's `DGEMM` as the fundamental subroutine. Figure 1 and Figure 2 depict the speedup and relative error, respectively, of the serial implementation of the algorithm with respect to matrix dimension and cut-off point. The experiments were performed on a Pentium4 3.2GHz desktop. In Figure 1, we observe that Strassen's algorithm is faster than ATLAS's `DGEMM` around the bottom right area. It is about 90% of the execution time of `DGEMM`. Also as shown in Figure 2, as the recursion level increases, the relative error becomes more and more severe.

4 Workflow Program Using NetSolve

NetSolve is a network enabled solver system [7]. It has a client/agent/server design. The client first sends a request to the agent. The agent then discovers appropriate servers to service the request dynamically. After that, the client communicates with the provided servers directly to send the input and receive the output. An advantage of using

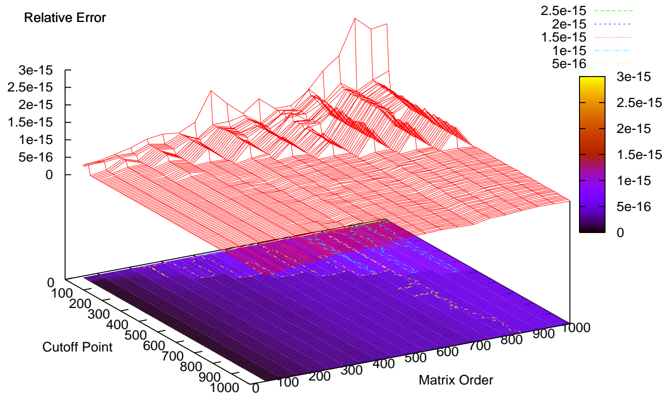

Figure 2. Relative error of Strassen's Algorithm.

NetSolve is that a user does not have to worry about the software maintenance issue and needs only a few resources on the client-end. Figure 3 shows the architecture of NetSolve.

It is straightforward to represent Strassen's algorithm in a task graph. A task graph is a directed graph whose edges represent dependence relationship and nodes represent tasks (either sequential or parallel programs). The node located at the arc tail should be executed before the node at the arc head. Figure 4 displays the corresponding task graph of one-level recursion of Strassen's algorithm. Please refer to Table 1 for the task notations.

4.1 Program Design

We look at the task graph in Figure 4 as a workflow diagram where a branch can be executed independently of other branches. To minimize the execution time, each node may start its computation whenever it is possible. We wrote a C program to implement the essentially data-driven computation. There are two types of operations in Strassen's algorithm: T1-T10 and C11-C22 are matrix additions, Q1-Q7 are matrix multiplications for which services are provided by NetSolve servers.

The client program calls a nonblocking function `netslnb` to invoke remote services on the servers. It uses a controller to traverse all of the nodes in the graph and manages to keep the data dependency. A set of flags indicating whether a node is started or completed help the controller to make appropriate decisions. Whenever the input to a node is available, the controller starts the node by sending a request to the agent to request a matrix addition or multiplication. The client is also able to support multi-level recursion of Strassen's algorithm.

The controller is essentially a `while` loop as shown in Figure 5. The function `test_run_xxx` checks whether a node's input is available and launches the job whenever

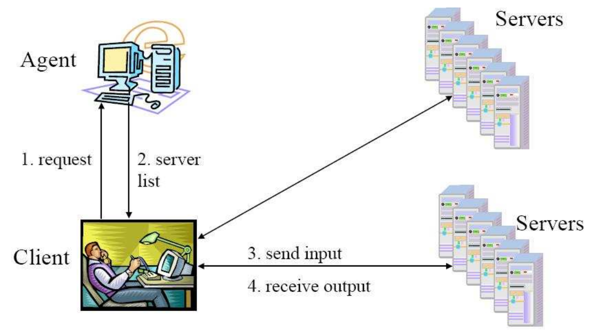

Figure 3. NetSolve architecture.

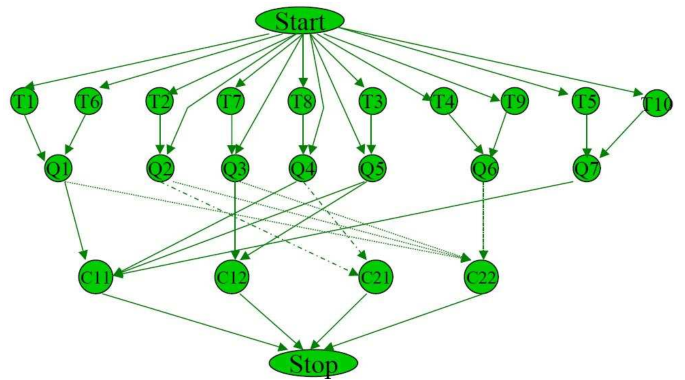

Figure 4. Task graph of Strassen's Algorithm.

possible.

4.2 Memory Usage

If we use one-level recursion of Strassen's algorithm in the client program, matrices A, B and C will use $3N^2$ elements of memory. There are also intermediate results required to be stored in temporary locations. Phase one needs ten temporary variables (for T1 to T10), phase two seven temporary variables (for Q1 to Q7), and phase three needs four variables (for C11 to C22). The total amount of storage is $3N^2 + 21(\frac{N}{2})^2 = 8.25N^2$. The server side requires less memory than the client side, namely $3(\frac{N}{2})^2$. In addition, if two-level recursion of Strassen's algorithm is used, the client program uses $3N^2 + 21(\frac{N}{4})^2 = 4.3125N^2$ memory space and the server program uses $3(\frac{N}{4})^2$ memory space.

5 Task-Parallel Approach Using MPI

We use a straightforward method to parallelize Strassen's algorithm. Based on the algorithm, there exist seven matrix multiplications (i.e., Q1-Q7). If we employ seven processes to compute the multiplications, the program design

```
void do_schedule() {
 /* Kick off Phase 1 nodes all at once */
 start_phase1_jobs(global_data);

 while(!is_alljobs_done(flags)) {
  /* Phase 2 nodes */
  test_run_Q1(global_data, flags);
  test_run_Q2(global_data, flags);
  test_run_Q3(global_data, flags);
  test_run_Q4(global_data, flags);
  test_run_Q5(global_data, flags);
  test_run_Q6(global_data, flags);
  test_run_Q7(global_data, flags);
  /* Phase 3 nodes */
  test_run_Phase3_T1(global_data, flags);
  test_run_Phase3_T2(global_data, flags);
  test_run_Phase3_T3(global_data, flags);
  test_run_Phase3_T4(global_data, flags);
  /* Phase 4 nodes */
  test_run_Phase4_C11(global_data, flags);
  test_run_Phase4_C12(global_data, flags);
  test_run_Phase4_C21(global_data, flags);
  test_run_Phase4_C22(global_data, flags);
  sleep(1);
 }
}
```

Figure 5. Workflow controller in the NetSolve client.

becomes very simple. The basic idea is that we divide the task graph (Figure 4) into seven parts and each part is handled by a process. Figure 6 depicts the task partitioning across the seven processes. Nodes with the same color belong to a single process. Please note that positions of nodes in Figure 6 are identical to those in Figure 4. Since Figure 6 directly reflects our program design, it is straightforward to write an MPI program based on the graph. For instance, process P0 performs computations of T1, T6, Q1 and C11 which correspond to the red nodes on the left.

5.1 Memory Usage

As shown in Figure 6, processes 0, 5 and 6 store one more intermediate result than the other processes and thus need more memory. We only consider the memory cost required by processes 0, 5 and 6. Each of them needs $\frac{1}{2}N^2$ space to store two blocks of A, $\frac{1}{2}N^2$ to store two blocks of B, and $\frac{1}{4}N^2$ to store one block of C. Additionally, six temporary variables for intermediate results require $6(\frac{N}{2})^2$ elements of memory. In total, the memory requirement is equal to $2.75N^2$.

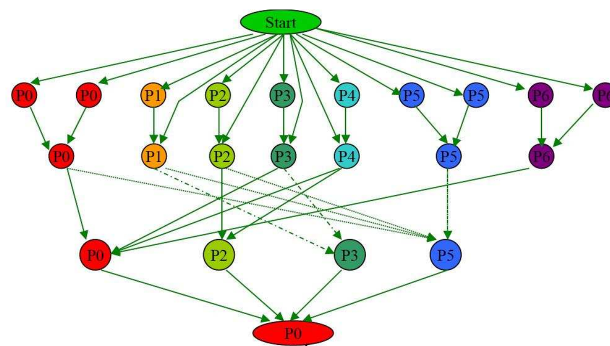

Figure 6. Using seven MPI processes to implement Strassen's Algorithm.

6 Data-Parallel Approach Using ScaLAPACK

As described in Section 2, there are typically three classes of methods to parallel Strassen's algorithm. In this section, we describe a method belonging to the third class, where Strassen's algorithm is used at the higher level and the parallel ScaLAPACK `PDGEMM` is used at the lower level. Different from the task-parallel program introduced in Section 5, this is a data-parallel program that performs the same operations in parallel on different subsets of the data.

Because our program calls the ScaLAPACK subroutines, we must adopt the same data distribution scheme as that of ScaLAPACK, that is, two-dimensional block-cyclic distribution.

6.1 Program Design

The processor grid is square $p \times p$, where p is of dimension 2^i. Square matrices of A, B, C are distributed across the p^2 processors located from the top left [0,0] to the bottom right [p-1, p-1]. Each process has a local matrix that consists of elements from different blocks of a matrix in the 2-D block-cyclic way. Since no process owns the whole matrix, matrix multiplications of Q1-Q7 have to be performed in a distributed way. This is done by calling PBLAS's `PDGEMM`. Interestingly, matrix additions and subtractions in phase 1 and 3 can be computed without any communication at all (i.e., T1-T10 and C11-C22). This is one advantage of using the 2-D block-cyclic scheme for Strassen's algorithm.

In the program, each process executes the T, Q, and C operations defined in Table 1 sequentially. All processes have the identical source code but operate on different submatrices. Matrix addition operations are local to each process, and matrix multiplications are performed across multiple processes by calling `PDGEMM`. The flow chart diagram in Figure 7 describes how the program works.

It is easy to port the program to various platforms since it is built on a standard library. Developers don't have

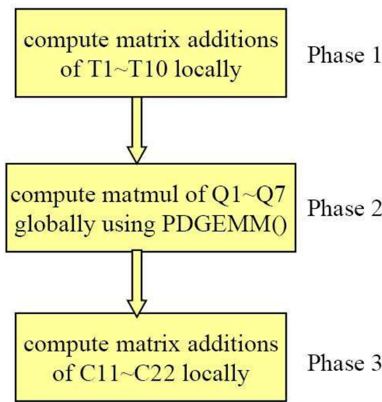

Figure 7. Flow chart diagram of the data-parallel method using ScaLAPACK.

to worry about the detailed parallelism implementation.

6.2 Time Cost Analysis

Desprez [12] introduced a theoretical model to determine the performance of PDGEMM. The model is given as follows:

$$T_{pdgemm} = \frac{2N^3}{p^2} t_m + N^2(2g) + \frac{2N}{N/p}(2L),$$

where t_m is the time per floating operation, g is a function for the *gap* and L is a function for the *latency*. The first item represents the computation time, and the last two items represent the communication time.

Our method computes seven sub-matrix multiplications in the lower level. Therefore its time cost is equal to:

$$\begin{aligned} T_{strassen} &= 7(\frac{2(N/2)^3}{p^2} t_m + (N/2)^2(2g) + \frac{N}{N/2p}(2L)) \\ &= \frac{7}{4}\frac{N^3}{p^2} t_m + \frac{7}{2}N^2 g + 28pL \end{aligned}$$

Since there is no communication at all for matrix additions, and their computation time is small compared to matrix multiplications', we omit the matrix addition time in the equation for $T_{strassen}$.

The above model predicts that we could achieve an improved performance for matrices of sufficiently large size if the communication time is relatively small. This can be easily seen by comparing the equation of T_{pdgemm} to the equation of $T_{strassen}$. The experimental result in Figure 10 (described in Section 7.3) validates the analytical model.

6.3 Memory Usage

Because we compute the matrix product on a square processor grid of $p \times p$ and use the scheme of 2-D block cyclic data distribution, all the temporary variables are of size $\frac{N}{2p}$. Each process uses $21(\frac{N}{2p})^2$ space. Considering the space to store matrices A, B, C, the total amount of memory space is $3(\frac{N}{p})^2 + 21(\frac{N}{2p})^2 = 8.25(\frac{N}{p})^2$.

7 Experimental Results and Analysis

All the experiments were performed on a local Linux cluster. The cluster consists of 32 nodes, each of which has dual PentiumIV Xeon 2.4GHz processors and 2GB RAM. In this section, we will present the experimental results for the workflow and parallel programs that implement Strassen's algorithm.

7.1 Result for the Workflow Approach

We provide NetSolve servers with two services: `matadd` and `matmul`, to add and multiply matrices, respectively. A NetSolve agent and three NetSolve servers are launched on four compute nodes. The client program is running on the fifth machine. The experiment multiplies matrices of size 512, 1024, 2048, and 4196 respectively. Since we are considering in-core computations and the memory capacity is 2GB, the biggest matrix size allowed is around 4000. We consider the client program to be sequential and compare it with ATLAS DGEMM. Figure 8 shows the total execution time of the workflow program and its breakdown into upload overhead, download overhead, and computation time. The upload overhead refers to the communication time for the client to send requests to the agent and to send input to servers. The download overhead refers to the communication time to receive result from servers. The computation time is the total execution time minus the communication time (i.e., upload overhead + download overhead).

We observe that the performance of Strassen's algorithm is not as good as the ATLAS program. The reason why it is slower is because it takes significant time to transfer data between the client and servers. For instance, the communication time occupies around 60% of the total time when n = 512, 1024, and 90% when n = 2048, 4096. However, when the matrix dimension is large enough (e.g. n=4096), Strassen's method becomes faster than ATLAS.

The client often keeps most of the CPU resource free even though it is doing intensive $O(n^3)$ computations. This justifies our original intention to attempt this method. If the time taken to send/receive data to NetSolve servers (i.e., $O(n^2)$) is less than the time taken to compute them locally (i.e., $O(n^3)$), we are able to achieve a better performance than DGEMM by using the workflow method. A further investigation shows that the agent overhead, which lies between 1 and 2 seconds is costly. An improved NetSolve agent could make the workflow prototype run much faster.

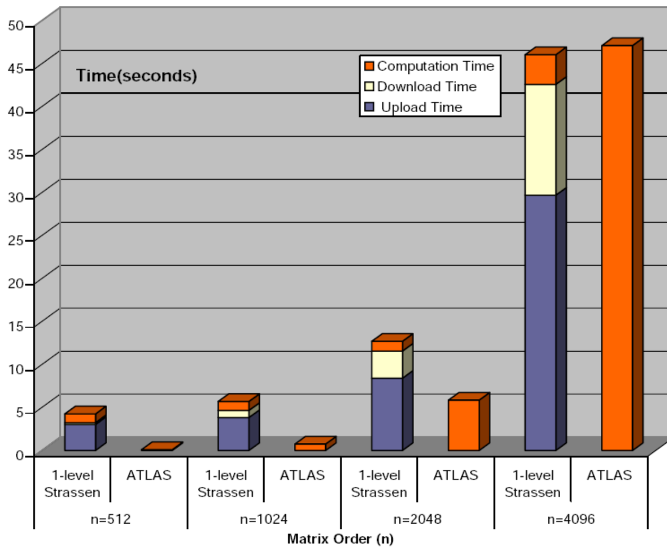

Figure 8. The workflow approach on a NetSolve client. When $n = 4096$, the S-method outperforms ATLAS. The computation cost on the client is always small.

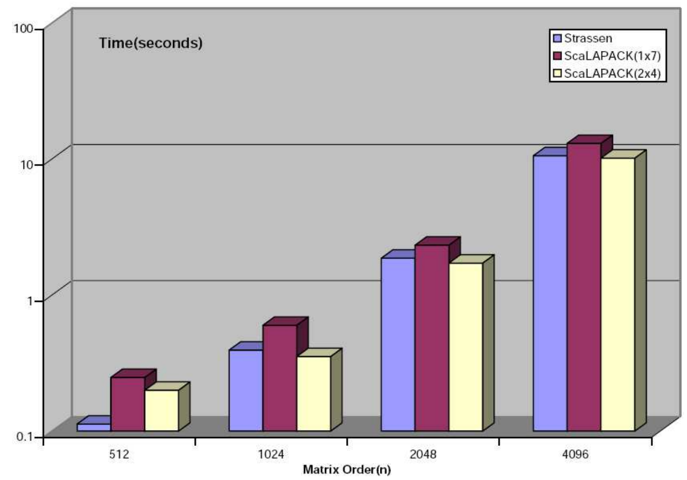

Figure 9. The task-parallel approach using MPI. It is compared to ScaLAPACK's `PDGEMM` with seven or eight processes. The S-method is always faster than `PDGEMM` with seven processes and close to that with eight processes.

7.2 Result for the Task-Parallel Approach Using MPI

We use seven MPI processes to compute the matrix multiplication. We compare it with ScaLAPACK's `PDGEMM` which adopts two different processor grids of 1×7 and 2×4. The 2×4 grid uses the 2-D block-cyclic scheme and is more load-balanced than 1×7. We conduct experiments on matrices of size 512, 1024, 2048, and 4196. For each matrix, three programs (Strassen's method, ScaLAPACK with seven processes, ScaLAPACK with eight processes) are executed. It is obvious to see in Figure 9 that the performance of Strassen's method is always better than that of ScaLAPACK. In particular, the S-method provides a performance even close to the ScaLAPACK 2×4 program. Note that the 2×4 program uses one more processor than the S-method. Although the idea is simple, our experimental result shows that the task-parallel S-method is more efficient than ScaLAPACK. We plan to extend the approach to support running more than seven processors.

7.3 Result for the Data-Parallel Approach Using ScaLAPACK

A number of experiments were performed to compute the matrix multiplications for matrices of size n = 512, 1024, 2048, 4096, 9182. We used a processor grid of 2×2 to run the experiments. The performance of ScaLAPACK is also presented in comparison with the S-method. In Figure 10, ScaLAPACK performs 10% better than the Strassen's program when n = 512, 1024, 2048. When n = 4096 they are the same, and when n = 8192 Strassen's method shows a better performance than ScaLAPACK. With the increment of the matrix dimension (more intensive computation), the S-method performs better than ScaLAPACK by saving one submatrix multiplication. This phenomenon also validates our theoretical model described in Section 6.2. Furthermore, based on the model, we expect that a faster network would improve the S-method greatly for a smaller matrix size rather than the current n = 4096.

8 Conclusions and Future Work

We have attempted three approaches to implement Strassen's algorithm. Although Strassen's algorithm has the problem of instability, stopping the recursion early can solve it easily. The workflow approach utilizes NetSolve servers to do the matrix multiplication by sending tasks to servers. When the matrix size n = 4096, the client does the computation faster than ATLAS while using a very small portion of its CPU resource. In addition, two parallel implementations of Strassen's algorithm running on distributed memory systems are presented. The first one is a task-parallel application where each process takes care of a different task. It always provides a better performance than ScaLAPACK using 1×7 processors and is even close to that of ScaLAPACK with one more processor. The second parallel implementation is a data-parallel application that calls `PDGEMM` at the bottom recursion level of Strassen's algorithm. A theoretical performance model is also provided and validated. It predicts that this approach will perform better than ScaLAPACK when n is sufficiently large. This method is scalable and easy to port to other platforms be-

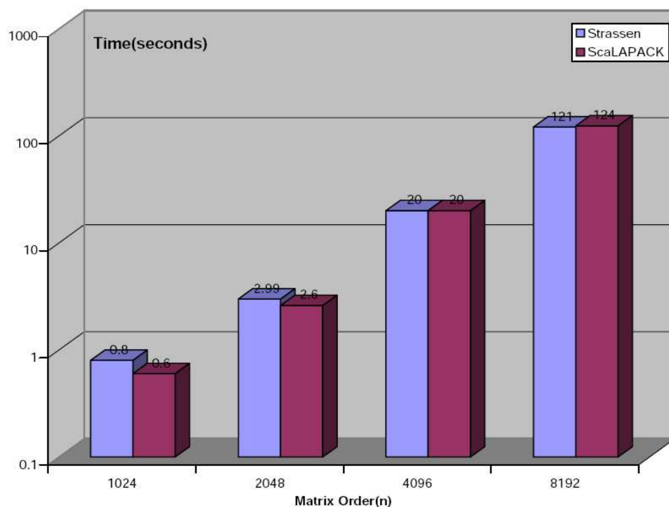

Figure 10. The data-parallel approach using ScaLAPACK. The S-method becomes better than ScaLAPACK after n = 4096.

cause the low-level subroutine of PDGEMM is from a standard portable library. All of the methods we tried have shown an improved performance by using Strassen's algorithm. We hope to use them on different types of platforms accordingly: either on a desktop computer or in a distributed-memory system. Hence it is worthwhile to put more effort into this method so that the basic operation of matrix multiplication could be improved further.

Our experience suggests that it is not trivial to make Strassen's algorithm perform better on medium-size matrices, not to mention small-size ones. But there is still much we can do to continue improving the programs' performance. For the NetSolve method, doing matrix additions locally rather than remotely would reduce the communication cost greatly. With an extension to the MPI method, we could use more than seven processors to do the computation. A mixed task- and data-parallel approach would be able to give the ScaLAPACK method more parallelism and improve the program performance further.

References

[1] Gaussian Elimination Is Not Optimal, V. Strassen. Numer. Math., 13:354-356, 1969.

[2] Matrix Algorithms on the Hypercube I: Matrix Multiplication, G. C. Fox, A. I. Hey and S. Otto. Parallel Computing 4:17-31, 1987.

[3] A Set of Level 3 Basic Linear Algebra Subprograms, J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff. ACM Trans. Math. Software, 16:1-17, 1990.

[4] Parallelizing Strassen's Method for Matrix Multiplication on Distributed-Memory MIMD Architectures, C. Chou, Y. Deng, G. Li and Y. Wang. Computers for Mathematics with Applications, 30(2):49, 1995.

[5] A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed-Memory Computers, Q. Luo and J. Drake. Proceedings of the 1995 ACM Symposium on Applied Computing, 221-226, Nashville, 1995.

[6] Automatically Tuned Linear Algebra Software (ATLAS), http://www.netlib.org/atlas.

[7] NetSolve/GridSolve, http://icl.cs.utk.edu/netsolve.

[8] Implementation of Strassen's Algorithm for Matrix Multiplication, S. Huss-Lederman and E. Jacobson. Proceedings of the 1996 ACM/IEEE conference on Supercomputing, Pittsburgh, 1996.

[9] A High Performance Parallel Strassen Implementation, B. Grayson and R. de Geijn. Parallel Processing Letters, Vol 6, No. 1, 3-12, 1996.

[10] Efficient Implementation of Strassen's Algorithm for Matrix Multiplication, M. Spence and V. Kanodia. Technical Report, Rice University.

[11] ScaLAPACK Users' Guide, L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R.C. Whaley. ScaLAPACK User's Guide, SIAM, Philadelphia, 1997.

[12] Mixed Parallel Implementation of the Top Level Step of Strassen and Winograd Matrix Multiplication Algorithms, F. Desprez and F. Suter. Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), San Francisco, 2001.

[13] IBM Engineering and Scientific Subroutine Library Version 3 Release 3 Guide and Reference. Document Number SA22-7272-04, IBM.

[14] Parallel Implementation of Strassen's Matrix Multiplication Algorithm for Heterogeneous Clusters, Y. Ohtaki, D. Takahashi, T. Boku and M. Sato. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, 2004.