To appear in Proceedings of Supercomputing '91, Albuquerque, NM, Nov 1991

A New Approach for Automatic Parallelization of Blocked Linear Algebra Computations

H. T. Kung and Jaspal Subhlok
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper describes a new approach for automatic generation of efficient parallel programs from sequential blocked linear algebra programs. By exploiting recent progress in fine-grain parallel architectures such as iWarp, and in libraries based on matrix-matrix block operations such as LAPACK, the approach is expected to be effective in parallelizing a large class of linear algebra computations. An implementation of LAPACK on iWarp is under development. In the implementation, block routines are executed on the iWarp processor array using highly parallel systolic algorithms. Matrices are distributed over the array in a way that allows parallel block routines to be used wherever the original program calls a sequential block routine. This data distribution scheme significantly simplifies the process of parallelization, and as a result, efficient parallel versions of programs can be generated automatically. We discuss experiences and performance results from our preliminary implementation, and present the design of a fully automatic system.

1 Introduction

A key technology critical to the widespread use of parallel processing is software tools capable of automatically generating parallel programs. It is well known that developing these tools is a challenging task, especially for distributed memory parallel machines.

In this paper we propose a new approach to developing tools for automatic parallelization of linear algebra computations. We believe that this approach can be applied to building practically useful tools for parallelizing linear algebra programs using block routines. Based on automatic parallelization of block operations, the approach is effective for three reasons:

(1) Efficient systolic algorithms exist for parallelizing block operations such as matrix multiplication [10], [13], [14].

(2) Fine-grain distributed memory parallel machines capable of efficient execution of systolic algorithms have become available such as iWarp [2],[3]. (iWarp is commercially available from Intel.)

(3) Libraries written using block routines for linear algebra computations have been developed such as LAPACK [6],[8].

Thus our approach takes advantage of advances in several areas, including parallel algorithm design, parallel architectures, and library development. Recent advances in areas (2) and (3) have made the approach practically useful.

At Carnegie Mellon we have done a feasibility study of this approach. We have implemented the block LU decomposition routine from LAPACK on a 64-processor iWarp machine. Based on results of this study, we believe that a software tool capable of automatically translating blocked linear algebra programs into parallel code can be developed, and the resulting parallel code will be efficient on a fine-grain parallel machines like iWarp. We are currently developing a tool for automatic porting of programs in LAPACK onto iWarp.

In Section 2 we describe the approach and illustrate it by examples. In Section 3 a theoretical analysis of the expected performance of the approach on iWarp is presented. Section 4 discusses the goals and status of our implementation on iWarp. In Sections 5 we describe the implementation of LU decomposition as an example and present performance results. In Section 6 a detailed performance analysis is given. In Section 7 we discuss the ongoing development of the automatic tool. Section 8 contains summary and conclusions.

2 Overview of the Approach

2.1 Basic Idea

Given a program written using block routines, we will perform each block operation on a $P \times P$ processor array. The actual target of our current implementation is an 8×8 iWarp

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035, and in part by the Office of Naval Research under Contract N00014-90-J-1939.

array, and thus for this implementation $P = 8$. A systolic algorithm is used to do each block operation in parallel over the whole array. Application programs written with block routines can be used without change.

Most linear algebra problems can be programmed with block operations such that most of the computation is in block routines. Fine-grain parallel machines can efficiently execute block operations by using systolic algorithms. Hence our approach is expected to yield good performance. Moreover, since the parallelization process mainly involves only parallelizing block operations, it is mechanical and relatively easy to automate.

The principle of this approach is analogous to that of using a vector processor to execute computations written in vector operations. Now, instead of a vector processor, we use a fine-grain parallel machine to execute matrix-matrix block operations in the computation.

2.2 Data Distribution

Each matrix involved in the computation is equally divided into two dimensional (2-D) block submatrices. Each block is uniformly distributed over the processor array, as shown in Figure 1. Thus every processor has a sub-block of every block.

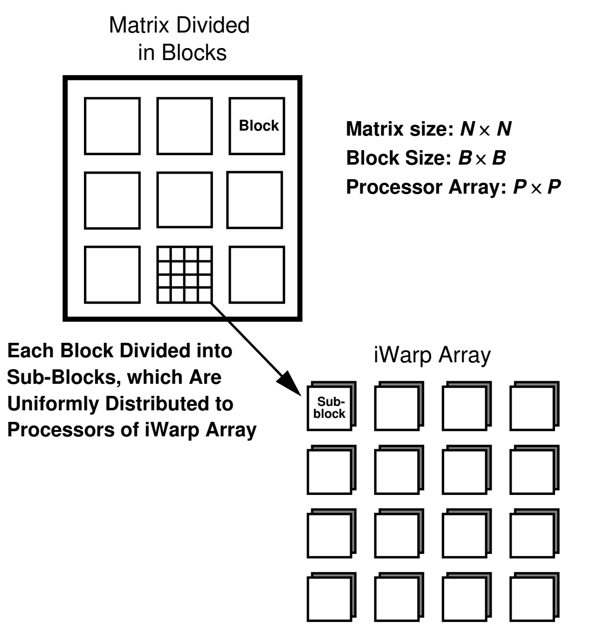
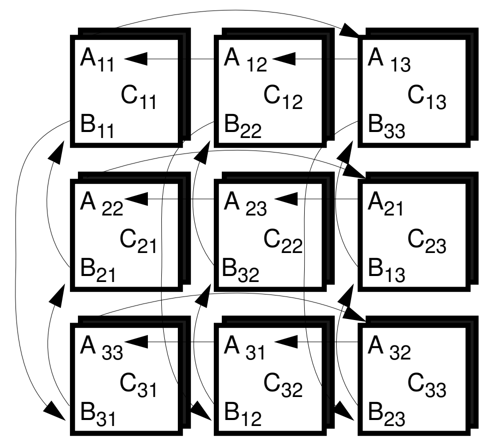


Figure 1 Dividing matrix into blocks and distributing sub-blocks of each block over processor array

2.3 Systolic Algorithms

With the data distribution scheme described above, various 2-D systolic algorithms can be used to implement block operations. Figure 2 depicts a systolic algorithm for matrix multiplication.

Sub-blocks in each processor are shifted by a fixed number of steps, bounded by P, the dimension of the processor array, before the computation is initiated. Then a sequence of local matrix multiplies and sub-block shifts, followed by a sequence of shifts analogous to those before starting the routine, accomplish the matrix multiplication and leave the operands and the results in place.



Initial location of sub-blocks and data movement patterns for matrix multiplication $C = A \times B$

Figure 2 Systolic algorithm for matrix multiplication

On iWarp, the computation and communication steps discussed above can be overlapped, giving near peak performance even for small sub-block sizes. This has also been demonstrated on Warp [1], a prototype systolic array machine that led to iWarp. More precisely, it has been shown that for many matrix operations, systolic algorithms can achieve near linear speedups on Warp [16].

3 Theoretical Performance Analysis

This section shows that with the approach outlined above, we can expect to achieve large speedups even for matrices whose size is not much larger than the size of the processor array. To be concrete, we give a theoretical performance analysis for the LU decomposition, a frequently used LAPACK routine.

Figure 3 illustrates how LU-decomposition is done in LAPACK using block operations. The basic, repeating step involves two matrix multiplications (i. e., $L_1 \times U_1$ and $L_2 \times U_2$), solution of a triangular linear system, and LU decomposition of a block column starting at ✻ on the diagonal. This step needs to be performed once for each of the *N/B* positions that ✻ may occupy, starting from the upper left corner to the bottom right corner. Note that the matrix multiplications, L1 × U1 and L2 × U2, can be done efficiently using the systolic algorithm outlined in Section 2.3.

However, the LU decomposition of a block column cannot be done in a completely parallel way, partly because it involves finding the maximum column element for pivoting. Furthermore, because of data dependence, LU decompositions of different block columns cannot be done in parallel. Therefore *B*, the block size, should be small; otherwise the degree of parallelism available in the entire program would be small.

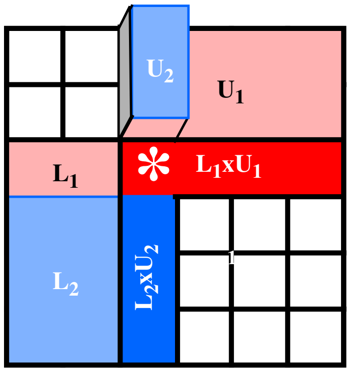


Figure 3 LU-decomposition using block operations

On the other hand, B may need to be large relative to P in order to ensure that the interprocessor communication overhead can be kept small compared to the computation time. Let

$$B = \alpha P$$

that is, the sub-blocks are $\alpha \times \alpha$. To ensure that the communication time is not larger than the computation time, the minimum value of α should satisfy:

$$\alpha = r$$

where r is the number of floating-point multiplications (or the number of floating-point additions) each processor in the processor array can perform in the time the processor inputs and outputs a word in both X and Y directions. This is because in the systolic algorithm depicted in Figure 2, each processor needs to input and output α^2 words for each of the two sub-block operands, while performing α^3 floating-point multiplications. Since iWarp supports fine-grain parallelism, the value of r (and thus α) is as small as 1.

With these notations, by some straightforward calculations, the following upper bound on speedup can be derived for performing the LU-decomposition on an $N \times N$ matrix:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}} \cong \frac{N^3}{B^3 N/B + N^3/P^2}$$

$$= \frac{N^2}{\alpha^2 P^2 + N^2/P^2}$$

The above formula is accurate in its first order terms when N is significantly larger than P or B, e.g., when $N = O(P^2)$.

For a fixed N, one can check that the speedup is maximized when $P^2 = N/\alpha$. Thus,

$$\text{Maximal Speedup} = \frac{N}{2\alpha} \quad \text{or} \quad \frac{N}{2r}$$

for any processor array of any size. We see that the maximal speedup is inversely proportional to r. Therefore our approach takes advantage of fine-grain parallel machines such as iWarp for which r is as small as 1.

Another way of looking at this result is that more and more processors can be usefully utilized until the number of processors is equal to N/α. At this time the maximal speedup is reached and is $P^2/2$. Thus the efficiency of using the processors is 50% at the time when the maximal speedup is reached, and is above 50% in other cases.

Now suppose that P^2, the number of processors, is fixed and we let N, the problem size, grow. Then from the speedup formula above, we see that if α or r is small then we will approach the speedup of P^2, or the peak performance of the parallel machine rapidly as N increases. This is another way of expressing the fact that our approaches takes advantage of fine-grain parallel machines which have small values of r. Figure 4 shows the expected performance of an 8×8 iWarp array for blocked LU decomposition of matrices of various sizes. Note that the performance approaches the peak of the 64-processor iWarp, which is about 1.2 GFLOPS, even for moderate matrix sizes. In the figure this is contrasted to the performance curve [9] for a coarse-grain parallel machine which has a peak performance larger than 1.2 GFLOPS

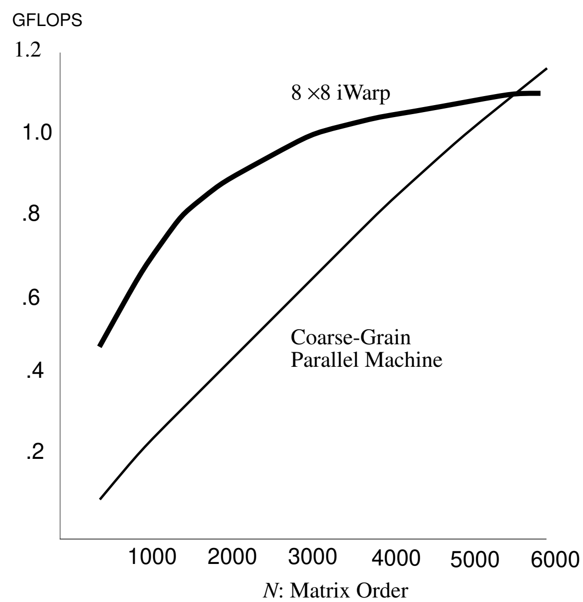


Figure 4 Expected performance of iWarp compared to a conventional coarse-grain distributed memory machine

4 Implementation on iWarp

The approach described in this paper is being implementated on a 8×8 iWarp array at Carnegie Mellon University. The implementation has the following objectives:

(1) Develop efficient parallel routines for all basic matrix-matrix operations referred to as level 3 BLAS operations. Develop a communication library that can efficiently handle common communication patterns that occur when parallelizing linear algebra code.

(2) Develop a sample set of LAPACK routines using the computation and communication libraries discussed above. Study and analyze the performance obtained.

(3) Develop a tool to automatically parallelize blocked linear algebra routines that use calls to BLAS routines for compute intensive operations.

We have developed parallel routines for matrix multiplication and solution of triangular systems of equations with multiple right hand sides. These routines are fairly efficient although we are looking into ways to improve the performance further. Using these, we have developed an implementation of LU decomposition. We discuss these in more detail in the next section.We are in the process of identifying what functionality should be included in the communication library. A discussion of the automatic tool under development is in Section 7.

5 Case Study: LU Decomposition

We have implemented the blocked LU decomposition algorithm using the general scheme described above. Here we describe this implementation and present performance results. In the next section, we will discuss the results in more detail.

5.1 Algorithm Overview

A high level description of the program is outlined in Figure 5. For a detailed description of this and other block algorithms for LU decomposition, see [4].

```
for i = 0 to last column block do

    update diagonal block (i,i) and subdiagonal
    blocks with a block matrix multiplication.

    perform an unblocked LU decomposition of block
    column i.

    update block row i with a block matrix multiplication

    compute the block row i with a block routine that
    solves a triangular system with multiple right
    hand sides.

endfor
```

Figure 5 Blocked LU decomposition algorithm

The exact amount of computation that is done in the different steps in the program depends on the block size. However, for reasonable block sizes, maximum computation is done in the matrix multiplication routine.

5.2 Implementation

We discuss the implementation of the main steps in the program above. All these subprograms act on contiguous blocks of data, and each block is distributed uniformly over the processor array in two dimensions. We state the names of the equivalent BLAS routines in parentheses, and will use them in the rest of the paper, although the description may not match precisely with the BLAS routine specifications since we are still in the process of implementing them.

Matrix Multiply (SGEMM):

Implemented as a block version of the 2-D systolic matrix multiplication discussed earlier in the paper. First the submatrices are aligned, a step that can involve moving a submatrix at most distance *P/2* on a $P \times P$ processor array. Then a sequence of local matrix multiplications and block shifts completes the computation. Each block shift involves moving a submatrix operand to an adjacent processor. In the end, the operand matrices are moved to their canonical location in a step analogous to the alignment step before the main computation.

The local matrix multiplication was coded as an assembler routine. The iWarp architecture is designed to support loops with no overhead when the loop bounds are known before loop execution starts. Thus the matrix multiplication loop should run at peak speed of 20 MFLOPS for each processor. However, the version of iWarp system that was available introduced a significant loop overhead for reasons discussed later. A special matrix multiplication for 16×16 matrices with the inner loop unrolled was used to get better performance on the matrix multiplication.

Unblocked LU Decomposition (SGETRF2):

A simple version of LU decomposition with row pivoting was used in this step. One column of processors is involved in computing the maximum of a column in the matrix. Next, in a communication step, the global maximum is computed and broadcast to all processors. This step is followed by row interchange between the current row and the selected pivot row. One column of L, the matrix of multipliers, is then computed and broadcast along the processor rows. The entries of the pivot row are sent systolically along processor columns. The relevant parts of the matrix are updated using calls to an assembler coded routine that updates a vector with a multiple of another vector, a version of the SAXPY routine.

Triangular System Solver (STRSM):

This routine solves the system of equations $A*X = B$, where A is a unit lower triangular matrix and B is a matrix containing multiple right hand sides for the equation system. The solution matrix X is overwritten on B.

First all parts of A in a given processor row are replicated within the processor row. For the LU decomposition algorithm, A is always a single block with one sub-block in each processor, so this is not an expensive step, at least in this context. The remaining data movement is restricted to sending rows of matrix B (same as X) along the processor columns. In every step a new row of X is computed and sent to processors in the same column. The remaining matrix is updated using calls to a single processor matrix-vector routine SGEMV.

5.3 Performance

The measured performance resulting from the current implementation using a 16×16 matrix sub-block in each processor is shown in Figure 6. The machine configuration used is an 8×8 iWarp array. Single precision arithmetic is used in all cases. Each processor has a peak performance of 20 MFLOPS giving the configuration an overall theoretical peak of 1.28 GFLOPS.

We expect our performance numbers to improve and we discuss that in a later section.

N : Matrix Order	MFLOPS
512	150.8
1024	263.6
1536	330.0
2048	375.8

Figure 6 Performance of LU decomposition resulting from the current implementation

The best performance was around 375 MFLOPS for a matrix of order 2048. The performance figures shown are not necessarily the best figures for a given matrix size, since the absolute performance does depend on the block size. According to our experimentation, a 16×16 sub-block size is close to being optimal for matrix sizes between about 1000 and 2000.

For the case of an order 2048 matrix, an approximate analysis shows that processors spent around 60% of time in matrix multiplication, 26.9% in unblocked LU decomposition and 13.1% in the triangular equation system solver. Of the time spent in matrix multiplication, about 64% was spent in the computation kernel, and rest was spent as communication and software overhead.

We consider the performance results obtained satisfying since we have achieved a large fraction of a distributed memory machine's peak performance on a moderate size problem with a modest programming effort.

6 Analysis of Performance Results

In this section we analyze the performance results we have obtained and explore the potential for improved performance. We also analyze how performance varies with parameters like block size and features of machine architectures.

To be concrete, we select matrix order 2000 for most of our discussions. We expect to reach close to the best possible performance at this matrix size, that is, we expect the performance vs. problem size curve to become nearly flat at this problem size. Currently the best performance is 375 MFLOPS, but we expect to get much better performance with system upgrades and software improvements in the near future.

We discuss specific reasons for expecting better performance, as well as reasons for limitations on maximum expected performance.

6.1 Limits on Performance

While the 8×8 Warp array configuration we use has a peak performance around 1.28 GFLOPS, we do not expect the performance to be very close to the peak for an order 2000 matrix. Some reasons are listed as follows:

(1) **Pivoting Overhead:** Row pivoting is called once for every column in the matrix and requires significant data movement. It also involves comparisons of the order of matrix size, using at most P processors in a $P \times P$ processor array. Although this overhead grows slowly and is alleviated by low latency, we still expect this phase to take about 10-15% of computation time.

(2) **Communication and software overhead:** Communication overhead is roughly related to O (N^2P) since volume of data is related to O (N^2), and in almost all cases, data movement is restricted to a single processor row or column. Also the software overhead due to function calls is significant since we use fairly small block sizes. For matrices of order 2000, we expect these factors to account for 10-15% of computation time.

(3) **Non Parallel Operations:** While matrix multiplication, the operation used for a major portion of the computation is highly parallel, other operations are not completely parallel. Solution of a triangular system of equations uses only half the processors on the average. Updating in the LU decomposition of a single column block involves a large fraction of the processors, (depending on the block size) but is not fully load balanced. We expect a performance degradation of up to 10% of execution time due to these factors.

(4) **Non Optimal Arithmetic:** Each iWarp processor can sustain one Floating Add/Subtract and one Floating Multiply every 2 cycles, which would give it a performance of 20 MFLOPS. This peak performance can be achieved for matrix multiplication, where typically two operands are in memory, and the result is added to a register. However, the peak performance cannot be achieved for an operation like SAXPY, where two words need to be fetched from memory, and the result has to be stored back in memory. These operations can be minimized with algorithm/block size selection but cannot be eliminated completely. Also compiler usage of registers for array values would alleviate the problem, but it is a difficult optimization that not many compilers perform. We expect some performance degradation due to slower arithmetic in some situations.

Because of these reasons we expect to reach up to 60 to 70% of the peak performance for matrix of order 2000. Performance would keep increasing and slowly approach the peak performance for larger matrix sizes.

6.2 Potential for Improved Performance

We discuss methods to remove some of the reasons that have caused inefficiency as described in the last section.

System Upgrades

The iWarp system that we used for our experiments has some temporary limitations that are expected to go away with the next system upgrade. We outline the main system related factors that cause performance degradation.

- To ensure correctness in certain situations, the assembler inserts additional delays in loops. While we have overcome this problem to some extent by unrolling loops in frequently called code sections, we expect a significant speedup when the problem is solved by new hardware.

- The compiler available to us does not use the "Long Instruction Word" feature of the iWarp machine, by which multiple instructions can be scheduled in the same cycle. A new optimizing compiler with this feature is being tested and expected to be available soon. Again, the problem was somewhat alleviated by hand coding parts of BLAS routines in assembly language, but rest of the code should execute faster with the new compiler.

Communication Reduction

Eliminating unnecessary communication is an important optimization for our approach in general. It is possible to eliminate a large portion of communication related to parallel matrix multiplication called from LU decomposition.

In the present implementation, every time matrix multiplication is invoked, the blocks involved in the matrix multiplication are aligned in the processor array before the computation phase of matrix multiplication can begin. At the end of the computation phase, all sub-blocks in a block are moved back to their original location. This introduces a significant overhead for starting up and finishing the matrix multiplication routine. We noticed that in the course of the algorithm, a particular sub-block of an operand matrix is always aligned in the same way for matrix multiplication. Moreover any matrix block that is an operand matrix for a block matrix multiplication never participates in any other operation except matrix multiplication, after the first matrix multiplication. Thus a block needs to be aligned at most once for matrix multiplication, while currently it may be realigned up to N/B times, where N/B is the number of block rows. We plan to incorporate this improvement in our program and it should virtually eliminate this overhead for matrix multiplication.

Program Optimization

We are planning to improve our implementation in several other minor ways. We outline some of these steps.

- **Using faster mathematical operations**: In the current implementation, we do a lot of computation using SAXPY routines. As discussed earlier, SAXPY arithmetic can be done only at half of an iWarp processor's peak floating point performance. Also, the SAXPY routine is normally called with a vector size equal to sub-block size. This is very inefficient since the function call overhead is relatively high as only block size (16 in our examples) operations are done per function call. We plan to replace SAXPY operations with matrix-matrix and matrix-vector operations whenever possible.

- **Efficient communication library:** We are building a library that efficiently implements frequently called communication operations. We expect to reduce the communication overhead significantly with the use of optimized communication routines.

- **Selecting block size**: Optimal block size depends on the problem size. By choosing near optimal block sizes for different problem sizes, we should be able to improve performance significantly. This is discussed in more detail in the next section.

6.3 Block Size and Performance

An important parameter that influences performance in a blocked algorithm is the size of individual blocks. A smaller block size means better parallelism and load balancing, but it also implies larger communication and software overhead. Essentially a larger block size increases the granularity of computations, thus saving on overhead, but a larger portion of the computation is done local to a block, which is not as efficient as operations between blocks.

We ran our LU decomposition program for different block sizes and the results are plotted in Figure 7. These measurements were taken without any optimization for any specific block size, hence the absolute results are not plotted. Instead we plot the results to show the best possible performance we expect the iWarp machine to achieve relative to the theoretical peak performance.We have extrapolated the graphs to show expected performance over problem sizes that we cannot currently run due to memory limitations.

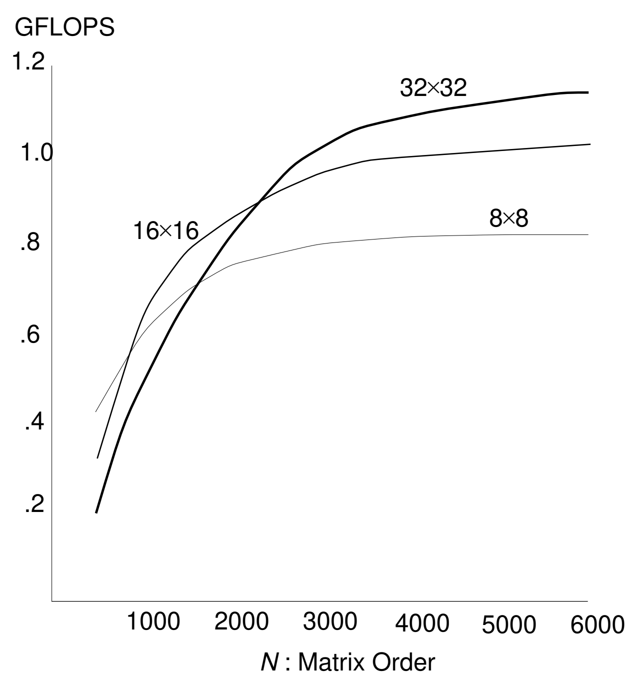


Figure 7 Block size and performance

The optimal block size depends on the problem size and increases with the problem size. A relatively small sub-block size of 8 gives the best performance for an order 500 matrix, but its performance curve begins to flatten after a matrix order of 1500 or so. sub-block size of 16 does not give as good performance for an order 500 matrix as a size 8 sub-block, but is slightly better for a problem size of 1000, and the performance continues to increase appreciably with problem size, up to a problem size of around 2000. This trend continues with block size 32. We plan to do such experimentation with other algorithms and develop heuristics for automatic block size selection.

6.4 Comparison with Conventional Distributed Memory Machines

The data distribution and the methods presented in this paper can be implemented on any distributed memory machine, but iWarp has some unique features that are useful for our approach. Low-latency communication and support for systolic communication are very beneficial for small to moderate problem sizes. However, for large problem sizes, the computation to communication ratio is much higher, and performance is mainly determined by the peak floating point performance of the processors. For instance, comparing iWarp to a conventional hypercube with the same number of processors, we expect performance characteristics plotted in Figure 4 [9]. We see that iWarp is expected to perform much better for small matrix sizes.

7 Automatic Parallelization

The key to achieving good performance using our methodology is to make calls to parallel block routines whenever possible. If we start with a sequential program that makes calls to sequential block routines, the process of replacing those calls by calls to corresponding parallel routines is mechanical, and can be done automatically. This is the basis of the tool that we are developing to automate parallelization of programs using a fixed set of block routines, an example being the LAPACK library.

Of course, not all of any program is composed of calls to block routines, even though for many linear algebra applications, most of the computation can be delegated to matrix-matrix block routines. We have to use conventional compiler technology to parallelize program parts outside of block routines, but efficiency of compiler parallelization is less critical since a relatively small part of the computation is dependent on it. We plan to develop an automatic parallelizer in two stages. We discuss these in the next two sections.

7.1 iWarp as Block Routine Coprocessor

The first version of out parallelization tool would execute all code outside of calls to block routines sequentially on a single processor, and use the whole iWarp array as a coprocessor for block routines. The data would have to be moved from a single processor to the whole array or in reverse, every time the execution phase moves from executing block routines to executing other code. This creates a significant overhead, but it is a simple and practical approach for large block sizes, since data movement involved is typically of order B^2 for block size B, while the number of arithmetic operations in a block routine is of order B^3. We plan to optimize this approach and eliminate redundant data movement, that is data that is not involved in the next sequential or parallel operation should not be moved. Work is already underway for developing this tool.

7.2 Complete Automatic System

The design discussed above has the disadvantage that significant data movement is involved for using block operations. Moreover computations outside of block operations must be done sequentially, even though limited parallelism may be available. To remedy these drawbacks, we have to be able to execute program segments outside of block routines in parallel whenever possible, and execute sequential code on the processor that owns the corresponding data elements, thus eliminating unnecessary data movement. This requires developing a conventional parallelizing compiler to handle code outside of block routines.

We already have a parallelizing compiler available to us for experimentation [17] and it has been successful in getting good speedups on many linear algebra programs. We are currently in the process of transferring and enhancing those ideas in a parallelizing Fortran compiler. This system would be the other main component of a fully automatic parallelizing system.

Some additional issues that the automatic tool would address are as follows:

- **Choosing block routines:** All the matrix-matrix block routines will be separately optimized, so this is not an issue for the compiler. However, the compiler is required to select between different available implementations of block routines. For instance, matrix multiplication operation A = B x C can be implemented in several different ways. Any one of the matrices A, B and C can be left in place, and the other two matrices move across the processor array. Depending on factors like the sizes and shapes of the three matrices, a particular routine may be more efficient than the others.
- **Optimizing data placement and movement:** For simplicity it is desirable that all routines expect input data to be distributed in some canonical form, and exit with output data in the same form. However, optimizing data placement and movement is possible between calls to routines. As discussed in an earlier section, operands of block matrix multiplication in LU decomposition need not be restored to their assigned location between block matrix multiplications. This saves significant data movement. In general, sophisticated data flow and data dependence information is needed to optimize data movement at compile time.
- **Overlapping I/O with computation:** iWarp supports spooling operations that can transfer data from memory of one processor to that of another, while the processors involved are doing other computations. Thus it is possible to do data movement completely in par-

allel with computation. This optimization can be done only when permissible under data dependency constraints, but has significant potential for blocked linear algebra codes.

8 Summary and Conclusions

We believe that this paper has described a promising direction in automatic generation of efficient parallel implementations of linear algebra programs. This new direction is made possible by recent advances in several research areas. Blocked programs in LAPACK, systolic algorithms for frequently used block operations, and the fine-grain parallel architecture of iWarp are the important advances that paved the way for this research.

We have presented a data distribution scheme and discussed how it can be used to effectively implement blocked linear algebra algorithms on distributed memory machines. We have discussed several details of implementation and demonstrated that good performance can be achieved on iWarp using our methodology.

This is an ongoing research project and we are currently working on two main aspects of it. We are developing parallel implementations of several routines from LAPACK to demonstrate that the approach works for a variety of algorithms. We are also in the process of developing a tool for automatic parallelization of LAPACK routines on iWarp. We expect to report further on the project in future publications.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, C-36 (12): 1523-1538, December 1987.

[2] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J.Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of the Supercomputing Conference*, pages 330--339, November 1988.

[3] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J.Webb, "Supporting systolic and memory communication in iWarp, in" *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70--81, Seattle, WA, May 1990.

[4] M. Dayde and I. Duff. Level 3 BLAS in LU Factorization on the CRAY-2, ETA-10P, and IBM3090-200/VF. *The International Journal of Supercomputer Applications,* 3(2):40--70, 1989.

[5] M. Dayde and I. Duff. Use of parallel level 3 BLAS in LU Factorization on three vector multiprocessors the Alliant FX/80, the CRAY2, and the IBM 3090 VF. In *Proceedings of the 1990 International Conference on SUPERCOMPUTING,* pages 82--95, Amsterdam, The Netherlands, June 1990.

[6] J. Dongarra, J. Du Croz, I. Duff, and S.Hammarling,. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16 (1):1-17, March 1990.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14 (1):1-17, March 1988.

[8] J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Technical Report ANL-MCS-TM-97, Argonne National Lab, September 1987.

[9] J. Dongarra and S. Ostrouchov. Lapack block factorization algorithms on the Intel IPSC/860. Technical Report CS-90-115, Computer Science Department, University of Tennessee, October 1990.

[10] W. Gentleman and H. T. Kung. Matrix triangularization by systolic arrays. In *Proceedings of SPIE Symposium*, Vol. 298, Real-Time Signal Processing IV, pages 19-26, August 1981.

[11] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. Impact of hierarchical memory systems on linear algebra algorithm design. *The International Journal of Supercomputer Applications*,3(2):40--70, 1989.

[12] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, February 1991.

[13] H. T. Kung,. Why Systolic Architectures?. *Computer Magazine*, 15 (1): 37-46, January 1982.

[14] H.T. Kung and C. Leiserson. Systolic Arrays (for VLSI). In *Sparse Matrix Proceedings* 1978, edited by I. S. Duff and G. W. Stewart. A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Section 8.3, pp. 37-46.

[15] C. Lawson, R. Hanson, R. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 16: 308--323, 1979.

[16] H. Ribas. *Automatic Generation of Systolic Programs from Nested Loops.* Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, June 1990.

[17] P.S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers,* Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, May 1989.