Performance of CAP-specified linear algebra algorithms

Marc Mazzariol, Benoit A. Gennart, Vincent Messerli, Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne, EPFL
gigaview@di.epfl.ch

Abstract. The traditional approach to the parallelization of linear algebra algorithms such as matrix multiplication and LU factorization calls for static allocation of matrix blocks to processing elements (PEs). Such algorithms suffer from two drawbacks : they are very sensitive to load imbalances between PEs and they make it difficult to take advantage of pipelining opportunities. This paper describes dynamic versions of linear algebra algorithms, where subtasks (matrix block multiplication, matrix block LU factorization) are dynamically allocated to PEs. It analyses theoretically the performance of the dynamic algorithms. This paper's contribution is to show that the dynamic-pipelined linear-algebra algorithms can be specified compactly in CAP and yet achieve good performance. CAP is a C++ language extension for the specification of parallel applications based on macro-dataflow graphs. The CAP model, based on macro-dataflow graphs, is general and supports pipelining.

1 Introduction

The traditional approach to the parallelization of linear algebra algorithms such as matrix multiplication and LU factorization calls for static allocation of matrix blocks to processing elements (PEs). Such algorithms suffer from two drawbacks : they are very sensitive to load imbalances between PEs and they make it difficult to take advantage of pipelining opportunities. Such load imbalances do not occur on dedicated parallel hardware, but are common on network of workstations. On the other hand, pipelined applications are more difficult to write, and tend to overload the process in charge of allocating sequential operations to PEs. This paper describes dynamic pipelined algorithms for parallel matrix multiplication algorithms and LU factorization, and analyzes their performance. This paper contribution is to show that dynamic-pipelined algorithms can be compactly specified in CAP and achieve good performance.

CAP (Computer-Aided Parallelization [3]) is a C++ language extension which supports the specification of pipelined concurrent programs. CAP's framework is based on decomposing high-level operations such as 2-D and 3-D image reconstruction, optimization problems or mathematical computations into a set of sequential suboperations with data dependencies. The application programmer uses the CAP language (1) to specify data dependencies between sequential suboperations, and (2) to assign each suboperation to an execution thread. The CAP data dependency model is similar to the macro dataflow model used successfully by the designers of the Mentat language [5]. The CAP preprocessor translates the CAP specification into a set of concurrent programs communicating through communication libraries such as MPI, PVM, TCP/IP, or through shared memory. Due to its support for pipelining, CAP generated programs achieve the performance of custom-made parallel programs. CAP is compositional, i.e. it is possible to reuse a parallel routine such as matrix multiplication without modification inside another parallel algorithm such as LU factorization.

Section 2 explains the static and dynamic algorithms for matrix multiplications, analyzes theoretically their performance, and lists the CAP specification for the dynamic matrix-multiplication algorithm. Section 3 describes the dynamic algorithm for LU factorization and analyzes theoretically its performance. Section 4 lists experimental

EuroPVM-MPI'97 , Krakow, Poland, November 97, LNCS 1332, 351-358

performance results. In the experimental setup, the hardware consists of a network of biprocessor SPARC 20 workstations connected through FDDI. Sequential routines are performed by the BLAS software package [1].

2 Matrix Multiplication

2.1 Notations

Lowercase letters represent matrix terms. Uppercase letters represent matrices. Subscripted uppercase letters represent matrix blocks. The matrix size is N^2. The number of blocks in the matrix is p^2, and the size of the blocks is $n^2=(N/p)^2$. A row of blocks is called a horizontal matrix slice. A column of blocks is called a vertical matrix slice. For simplicity we assume that N modulo p= 0. Using these conventions, the matrix multiplication is written as $c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$, and the block matrix multiplication is written as $C_{mn} = \sum_{l=1}^{p} A_{ml} \cdot B_{ln}$. We consider a physical machine consisting of P^2 processing elements (PEs) connected to a single client requesting the computation. We assume that the hardware supports direct memory access, i.e. that it is possible to perform data transfers between PEs without interrupting the PEs involved.

2.2 Dynamic parallel algorithm

The dynamic version of the algorithm assumes that initially both input matrices are located in the client address space. The client divides both matrices in p^3 matrix block pairs, and sends the matrix block pairs to the PEs for partial matrix computation. The partial results are returned to the client for merging in the final resulting matrix.

The dynamic algorithm requires p times the transfer of both input matrices and p times the transfer of the output matrix. The total transfer requirement is $3pN^2$. The transfers are well distributed among the PEs, but the client, receiving and transferring all messages is clearly a potential bottleneck. No synchronization is required between the PEs and it is possible to keep the PEs busy during the execution of the algorithm, assuming several matrix block pairs are queued waiting to be executed by each PEs. It is easy to perform load balancing, so as to make the algorithm insensitive to PEs load imbalances. Besides being a communication bottleneck, the client is also a memory bottleneck as it is required to store both input matrices and the output matrix in the client memory. There are two standards solution to reduce the client bottleneck : (1) out-of-core programming, where matrix blocks are store on disks and prefetched as they are required and (2) use multiple clients, each owning part of the matrix data. This paper analyses theoretically and experimentally the performance of the dynamic algorithm with a single client, and shows possibilities and limits of the dynamic algorithm.

2.3 Dynamic parallel algorithm : performance analysis

The timing diagram for the dynamic matrix multiplication is presented in Fig. 1. In Fig. 1, there are 5 PEs, one for the client and 4 for the computation threads. Each PE features one additional thread for communication. The time line flows from left to right. The thick arrows represent thread activity, i.e. time during which the communication or computation occurs. We have assumed DMA, i.e. that communication can overlap computation on a given PE. The shaded boxes indicate the data transfers. The thin arrows represent data dependencies. The final timing diagram section (rightmost part of Fig. 1) represents a minimal length section for 4 matrix-block-pair transfers (client to computation threads), 4 sequential matrix multiplications, and 4 matrix-block transfers (computation threads to client). The other sections of Fig. 1 are a stretched version of the rightmost timing diagram, which allows to overlap two timing diagram sections without resource conflict. The critical path through the graph consists of P^2

matrix-block-pair transfers (to the threads, when initializing the pipeline), p^3/P^2 sequential matrix multiplications, and one matrix-block transfer (to the client, for the last partial result).

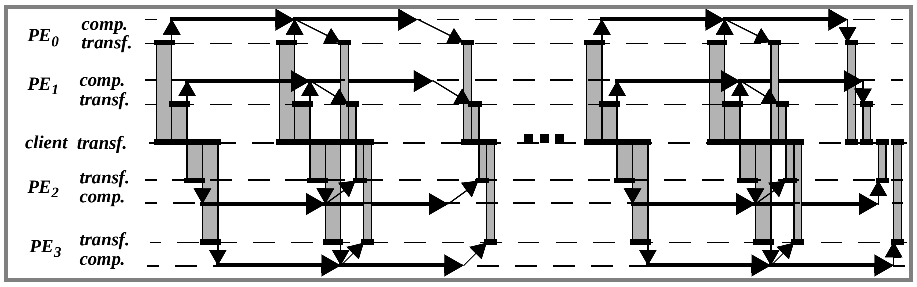

Fig. 1. Dynamic matrix multiplication : timing diagram

Assuming a network latency of l_t, a network throughput of $8/\tau_t$ (τ_t is the transfer time for 1 matrix term, and 8 is the number of Bytes per double), and a computation throughput of $1/\tau_c$ (τ_c is the nominal computation time for 1 term of the resulting matrix), the formula for the delay is :

$$t = P^2(l_t + 2\tau_t n^2) + \frac{\tau_c N^3}{P^2} + (l_t + \tau_t n^2) \qquad (1)$$

The condition on the network bandwidth is that the transfer time during one section of the timing diagram be less than the computation time during the same time :

$$P^2((l_t + 2\tau_t n^2) + (l_t + \tau_t n^2)) < \tau_c n^3 \qquad (2)$$

If we assume reasonable values for the parameters (l_t = 1ms, τ_t=1.6µs/elem (5MB/s throughput), τ_c=875ns/elem, all numbers resulting from experimental measurements on the FDDI network and the BLAS routine dgemm), we find for a matrices of 1024x1024 terms, for a block of 128x128 terms and the number of computation processes P^2 ranging from 1 to 20, speedups ranging from 1 to 19.5, showing that the dynamic approach is definitely valid. The left and right part of Equation (2) in the worst case (P^2 = 20) are 1.61s and 1.83s, ensuring that the condition holds.

2.4 CAP specification of the matrix multiplication

Program 1 is the textual specification of the dynamic matrix multiplication in CAP. The indexed parallel construct in Program 1 (line 5 to 9) features 3 range indices. The CAP runtime iteratively calls the SplitInput routine (line 9) on the TwoMatricesT Input token, and generates matrix block pairs. As soon as a matrix-block-pair is generated it is sent to the appropriate thread for a sequential matrix computation (line 10). When all matrix block pairs are sent, the CAP run time initializes in the client address space (called Main, line 9) the output matrix (MatrixT Output, line 9). When a computation thread completes a sequential matrix multiplication, it returns the partial result immediately to the client thread, and gets a new matrix block pair from its input queue. The client thread, as soon as it receives a partial result from a computation thread, merges the partial result into the output matrix Result.

Program 1 specifies all communication and synchronization requirements of the parallel matrix multiplication program. The rest is sequential C++ code required to specify how to split the input matrices in matrix block pairs, to merge partial results into the output matrix, to describe the TwoMatricesT and MatrixT tokens, to specify the Sequential matrix multiplication.

.

```
operation CompositeThreadT::ParallelMatrixMultiplaction (int p)
  in TwoMatricesT Input
  out MatrixT Output
{
  indexed
    ( int i = 0 ; i < p; i++ ) // first construct range
    ( int j = 0 ; j < p; j++ )// second construct range
    ( int k = 0 ; k < p; k++ ) // third construct range
  parallel (SplitInput(p,i,j,k), MergeOutput(p,i,j,k), Main, MatrixT Output)
    ( Thread[((i*p+j)*p+k)%4].SequentialMatrixMultiplication ) ;
}
```

Prog. 1. CAP specification of the ParallelMatrixMultiplication operation

The issue of whether matrix blocks are copied by the SplitInput routine is left to the implementation of the TwoMatricesT token and the SplitInput routine. A simple implementation will implement TwoMatricesT tokens as actual matrices and implement the SplitInput routine as memory copies. A sophisticated implementation will implement TwoMatricesT tokens as matrix references and make sure that copies occur only when data is transferred from one address space to the other. Our implementation uses the sophisticated BLAS array data referencing mechanism which avoids unnecessary data copies. We ensure that data is copied only when data is transferred between address spaces. The actual ParallelMatrixMultiplication operation we used implements load balancing and consists of 24 lines of CAP specification.

3 LU factorization

The LU factorization is interesting in two respects. It features more data dependencies than the matrix multiplication, and uses the matrix multiplication. This example is ideally suited to show two important features of CAP, its compositionality and its support for pipelining. We reuse without modification the CAP parallel matrix multiplication specification in the LU factorization algorithm. We pipeline triangular system resolution with matrix multiplication and permutations, to achieve higher speedups. The complex data dependencies of the algorithm also shows the generality of the CAP approach.

3.1 Problem description

Golub [4, p.100] describes the block-based LU factorization (Fig. 2). We summarize it here. Consider a matrix A divided in 4*4 blocks. Step 0 of the block based LU factorization algorithm consists of (1) performing the LU factorization on the leftmost vertical slice (dgetf2_ routine in LaPack, LU in Fig. 2) ; (2) permuting all other vertical slices using the permutation vector produced by the LU factorization ; (3) solving the triangularized system consisting of the top left block and the rest of the top horizontal slice (trsm_ routine in BLAS, T in Fig. 2) ; (4) multiplying the leftmost vertical slice (but the top block) and the topmost horizontal slice (but the left block) (dgemm_ routine in BLAS, M in Fig. 2) and accumulate the result of the multiplication in the lower-right 3*3 blocks. Step 1 repeats the process on the lower-right 3*3-block matrix. Step 2 on the lower-right 2*2-block matrix, and the last step works only on the lower-right matrix block. In all steps, permutations are performed on complete horizontal slices.

3.2 Parallelization

The block-based matrix multiplication algorithm is easy to parallelize because the size of each transfer is O(n^2) and the number of scalar multiplications per matrix block multiplication is O(n^3). It is therefore always possible to select n so that communication time is small compared to computation time. On the other hand, it very difficult to parallelize triangular system resolution, since the number of scalar operations in triangular system resolution is $O(n^2)$, of the same order as a matrix block transfer. This is especially true in the case of a network of workstation, where network latency is high.

However we cannot afford to perform at each step the block LU factorization followed by the block triangular system resolutions sequentially, before performing in parallel the matrix multiplication. The sequential part of the algorithm would be much too long. We resort to pipelining to achieve good performance. We perform the LU factorization by block, sequentially on the client thread, but start the triangular system solution and the matrix multiplication as soon as the required data is available. This leads to the following timing diagram (Fig. 2).

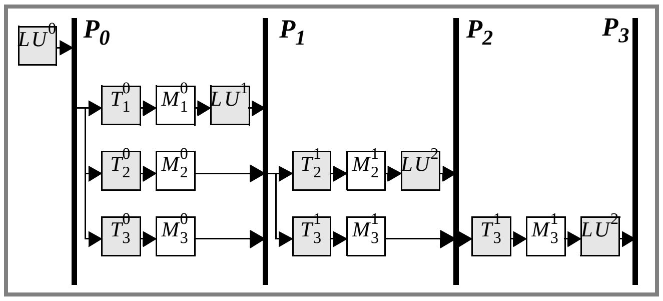

Fig. 2. Timing diagram for the 4*4-block LU factorization

In Fig. 2, the gray blocks represent the operations performed by the client thread called Main. We let the Main thread perform the LU factorization on the first matrix vertical slice (LU^0_0), and get the permutation vector P_0, which we apply to the matrix. The permutation represents a global synchronization barrier, as we have to wait until all the permutations have been performed until we can start the triangular system resolution and the matrix multiplications. So far all steps have been performed sequentially. Then we start working in pipeline fashion : (1) Main solves the first triangular system (T^0_1, involving matrix blocks A_{00} and A_{01}) ; (2) as soon as the result is available, Main launches on available computation threads matrix multiplication M^0_1, involving matrix blocks A_{01}, A_{10}, A_{20}, and A_{30} ; (3) Main immediately starts with a new triangular system resolution (T^0_2, involving matrix blocks A_{00} and A_{02}). This continues until there are no more triangular systems to solve. Additionally, as soon as the results of matrix multiplication M^0_1 is available in the Main thread address space, the Main thread can compute the LU factorization for the next algorithm step (LU^1_1), and get the permutation vector P_1. When all the matrix multiplication results have been returned to the Main thread and the accumulation has been performed, the permutation is performed sequentially on the whole matrix (thick vertical line labelled P_i in Fig. 2). When the permutation is complete, it is possible to start a new parallel LU factorization step.

Fig. 2 is a simplified timing diagram where all permutations are performed serially after all matrix blocks are available. In fact, the permutation vector can be applied to the matrix vertical slices as soon as the accumulation step is performed for that vertical slice. The program on which performance measurements have been performed takes advantage of pipelined permutations. The CAP specification is a 24-line textual version of Fig. 2.

The dominant part of the algorithm is the matrix multiplication, and that it is performed in parallel by the computation threads. The only part of this algorithm during which the computation threads cannot work is the triangular system resolution of the first block of each step. During the rest of the time, all computation threads can perform matrix multiplications. The duration of step i t_i is given by equation (3). Equation (5) is the sum of two terms : the sequential triangular system resolution time on one block t_{res} and the multiplication of the two matrices (see equation (1)). The matrices to be multiplied at each step have sizes (N-(i+1)n)*n and n*(N-(i+1)n).

$$t_i = t_{res} + t_* = \tau_{res} n^2 + P^2(l_t + 2\tau_t n^2) + \lceil (p - i - 1)^2 / P^2 \rceil \cdot \tau_c n^3 + (l_t + \tau_t n^2) \tag{3}$$

The total time to required to perform the parallel LU factorization is the sum of the t_i for all steps, plus the LU factorization on the first vertical slice.

$$t = t_{lu} + \sum_{i=0}^{p-2} t_i \approx \tau_{lu} n^2 N + (p-1)(\tau_{res} n^2 + P^2(l_t + 2\tau_t n^2) + (l_t + \tau_t n^2)) + (\tau_c N^3 / 3P^2) \tag{4}$$

The approximated formula of equation (4) is true provided the multiplication takes longer than the triangular system resolution and the LU factorization together. This condition is usually true for the early steps of the parallel LU factorization, but becomes false at the end of the algorithm. We express in equation (5) the condition that the client thread computation time is less that the PEs multiplication time for step i and check that it is true for most of the steps of the algorithm.

$$\tau_{res} n^2 (p - i - 1) + \tau_{lu} n^3 (p - i - 1) < P^2(l_t + 2\tau_t n^2) + \lceil (p - i - 1)^2 / P^2 \rceil \cdot \tau_c n^3 + (l_t + \tau_t n^2) \tag{5}$$

We analyze the values of equation 4 as well as speedups as a function of the number of processors, for a 2048*2048 matrix (N=2048), block size of 128 (n = 128, p = 16), and for unitary values of the LU factorization τ_{lu}, triangular system resolution τ_{res} and matrix multiplication τ_c of 260ns, 700ns and 875ns. The communication latency and throughput are 1ms and 5MB/s respectively. For 10 computation threads the formula gives speed-ups of 9.43 and for 20 computation threads, the speed-up reaches 17.1.

4 Performance measurements

We have run our performance measurements on a network of Sun Sparc20 with two processors, connected through FDDI. We run a single computation thread and a single communication thread per workstation, thus allowing for overlapped communications and computations. No single thread can exceed 100% utilization per processor.

We integrated LaPack into CAP. Sequential routines are BLAS routines. The implementation effort consisted of wrapping BLAS routines in CAP tokens, and providing serialization routines (PVM Pack and Unpack) for the tokens, so that BLAS structures can be transferred from one address space to the other. The communication and synchronization between sequential operations are specified in CAP. We use a PVM-style library called MPS. MPS is thread based whereas PVM is process based. The library was developed at the EPFL and runs under Solaris and WindowsNT.

4.1 Dynamic matrix multiplication

Fig. 3 displays the speedup-results of the matrix multiplication on 1000x1000 matrices. We compare the speed of the parallel program run with 1 to 20 computation threads (i.e. 21 workstations involved) to the sequential program performance (single LaPack call). Two threads run on each workstation, one for computation and one for communication.

The left part of Fig. 3 shows a near linear speedup as a function of the number of computation threads. The single thread matrix multiplication consisting of a single call to LaPack is performed in 875s. The 20-computation-PEs matrix multiplication with a 125x125 block size is performed in 46.6s, for 18.8 speed-up, close to the theoretically predicted speedup. The non-linearity is mostly due to pipeline startup and shutdown times. Larger matrix sizes would improve the situation. This is impossible without out-of-core programming, i.e. storing matrix data on disks, and prefetching the data in memory as required.

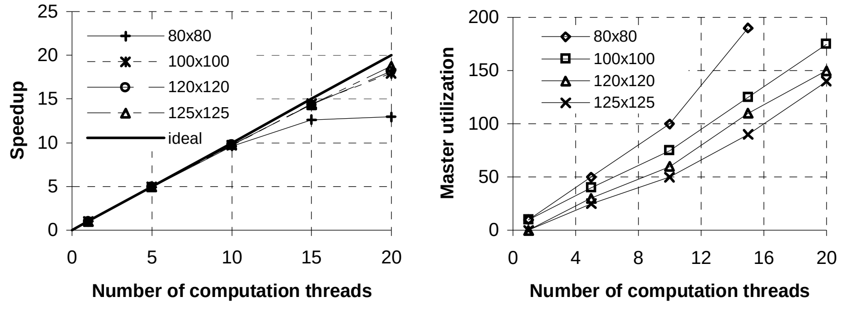

Fig. 3. Matrix multiplication speedup and client thread utilization

The right part of Fig. 2 shows the load on the client PE. The client PE utilization is affected by the block size. As is expected the smaller the block size, the more data needs to be transferred between client and computation threads. The utilization ranges between 0 and 200%, showing that both computation and communication threads are fully utilized. For the matrix multiplication, the client computation thread handles matrix-block-pair serialization and partial result accumulation. The client communication thread handles the matrix deserialization, as well as serialized data emission and reception. The computation threads have a 100% utilization for the complete duration of the algorithm, except pipeline startup and shutdown times.

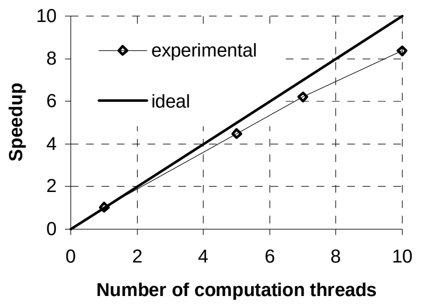

Fig. 4. .LU factorization performance

4.2 LU factorization

Fig. 4 shows the performance results of the LU factorization for a 2000-by-2000 matrix, for a number of processor varying from 1 to 10. The single process sequential computation time for the LU factorization of the matrix is 2095s. The fastest parallel time we achieved is with 10 processors is 250s, for an 8.38 speedup. The difference with the theoretical model is due to the fact that for later steps of the algorithm, condition (5) becomes false.

4.3 Result analysis

Client overload. The client load is much higher than the theoretical analysis suggests. This is due to the fact that data structure serialization is a time consuming process requiring many data copies, in particular in the current version of the LSP library.

Moreover the use of the TCP/IP protocol between workstations consumes large CPU resources. Typically a 5MB/s throughput at the client leads to a 100% CPU utilization.

It is not possible to handle matrices much larger than 1000x1000 without the client thread starting to swap. The three matrices required for a 1000x1000 multiplication represent 24 MBs of data. Out of core programming is required, where the client thread prefetches the matrix blocks from the disks as they are required. As pipelining is a CAP feature, disk prefetching is not difficult to integrate in the CAP implementation of the LaPack routines. The disk prefetching mechanism has been successfully used in the CAP implementation of a 3-D visualization package.

The CAP environment. The good performance of CAP generated programs can be attributed to the following factors : (1) threads are allocated statically to processing elements ; (2) operation overhead is small. The input data of each operation is tagged with a 20-byte header, which typically represents a overhead of less than 1% ; (3) data can be passed by reference between threads sharing a common address space ; (4) operations are routed dynamically to threads, allowing to adapt the load of each processing element ; (5) pipelining is a part of the CAP semantics, allowing to remove unnecessary synchronization barrier. Current difficulties in the CAP implementation of the LaPack routines are : (1) the Solaris version of the MPS communication library performs 3 copies for each transfer ; (2) serialization of sophisticated data structures is a time-consuming effort ; (3) some data dependencies are impossible to specify in the current version of the language ; (4) pipeline control requires the programmer's attention.

5 Conclusion

This contribution has presented pipelined-parallel algorithms for matrix multiplication and LU decomposition. It has analyzed the theoretical performance of the algorithms, shown their CAP implementation, and presented performance results. The results show that the dynamic pipelined approach is viable for matrix multiplication and LU decomposition for 10 to 20 PEs. Large matrix sizes require out of core programming. CAP support for pipelining is effective to overcome the large network latencies. CAP specifications are short, and yet the parallel programs achieve good performance, demonstrating the validity of the CAP model.

Future work involves evaluating theoretically and experimentally the performance of a CAP-specified static algorithm ([4], p.299), and of the combined static and dynamic algorithms. Future work will also address the issue of out-of-core programming.

References

[1] E. Anderson et al. LAPack User's Guide, 2nd edition. Society for Industrial and Applied Mathematics. Philadelphia, 1995.

[2] A. Geist et al. PVM 3 User's Guide and Reference Manual. September 94. URL : http://www.epm.ornl.gov/pvm/.

[3] B. A. Gennart et al.. Computer-Assisted Generation of PVM/C++ Programs Using CAP. In Proc. EuroPVM'96, p. 259- 269. LNCS 1156. Münich, Oct. 1996.

[4] G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins University Press. ISBN 0-8018-5414-8. Third edition, 1996.

[5] A. S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. IEEE Computer, Vol. 26, No. 5, May 1993, pp 39-51.