Hierarchical Matrix-Matrix Multiplication Based on Multiprocessor Tasks

Sascha Hunold[1], Thomas Rauber[1], and Gudula Rünger[2]

[1] Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth, Germany
[2] Fakultät für Informatik, Technische Universität Chemnitz, Germany

Abstract. We consider the realization of matrix-matrix multiplication and propose a hierarchical algorithm implemented in a task-parallel way using multiprocessor tasks on distributed memory. The algorithm has been designed to minimize the communication overhead while showing large locality of memory references. The task-parallel realization makes the algorithm especially suited for cluster of SMPs since tasks can then be mapped to the different cluster nodes in order to efficiently exploit the cluster architecture. Experiments on current cluster machines show that the resulting execution times are competitive with state-of-the-art methods like *PDGEMM*.

1 Introduction

Matrix multiplication is one of the core computations in many algorithms of scientific computing and numerical analysis. Many different implementations have been realized over the years, including parallel ones. On a single processor *ATLAS* [7] or PHiPAC [1] create efficient implementations by exploiting the specific memory hierarchy and its properties. Parallel approaches are often based on decomposition, like Cannon's algorithm or the algorithm of Fox. Efficient implementation variants of the latter are *SUMMA* or *PUMMA*, see also [3] for more references. Matrix-matrix multiplication by Strassen or Strassen-Winograd benefits from a reduced number of operations but require a special schedule for a parallel implementation. Several parallel implementations have been proposed in [2,5,4].

Most clusters use two or more processors per node so that the data transfer between the local processors of a node is much faster than the data transfer between processors of different nodes. It is therefore often beneficial to exploit this property when designing parallel algorithms. A task parallel realization based on multiprocessor tasks (M-tasks) is often suited, as the M-tasks can be mapped to the nodes of the system such that the intra-task communication is performed within the single nodes. This can lead to a significant reduction of the communication overhead and can also lead to an efficient use of the local memory hierarchy. Based on this observation, we propose an algorithm for matrix multiplication which is hierarchically organized and implemented with multiprocessor tasks. At each hierarchy level recursive calls are responsible for the computation of different blocks with hierarchically increasing size of the result matrix. The

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 1–8, 2004.

processors are split into subgroups according to the hierarchical organization which leads to a minimization of data transfer required. Moreover, only parts of one input matrix are moved to other processors during the execution of the algorithm, i.e., the local parts of the other matrix can be kept permanently in the local cache of the processors.

We have performed experiments on three different platforms, an IBM Regatta p690, a dual Xeon cluster with an SCI interconnection network, and a Pentium III cluster with a fast Ethernet interconnect. For up to 16 processors, the algorithm is competitive with the *PDGEMM* method from *ScaLAPACK* and outperforms this method in many situations. Thus the algorithm is well-suited to be used as a building block for other task parallel algorithms.

The rest of the paper is organized as follows. Section 2 describes the hierarchical algorithm. The implementation of the algorithm is presented in Section 3. Section 4 presents experimental results and Section 5 concludes the paper.

2 Hierarchical Matrix Multiplication

The hierarchical matrix multiplication performs a matrix multiplication $A \cdot B = C$ of an $m \times n$ matrix A and an $n \times k$ matrix B in a recursively blockwise manner on p processors. We assume that $p = 2^i, i \in \mathbb{N}$, and that p divides m and k without remainder. During the entire algorithm the input matrix A is distributed in a row blockwise manner, i.e. processor q stores the rows with indices $((q-1)\cdot s+1, \ldots, q\cdot s), s = m/p, q = 1, \ldots, p$. Input matrix B is distributed columnwise with varying mappings in the computation phases. Initially the distribution is column blockwise, i.e. processor q stores the columns with indices $((q-1)\cdot s'+1, \ldots, q\cdot s')$. The columns are exchanged in later steps, see Figure 1.

The hierarchical matrix multiplication computes the result matrix C in $\log p + 1$ steps and processor q is responsible for the computation of the s rows with indices $((q-1)\cdot s+1, \ldots, q\cdot s)$ of C. The computation is organized so that disjoint processor groups compute the diagonal blocks C_{lk} in parallel, which contain the $(s \cdot 2^{l-1})^2$ entries c_{ij} with $2^{l-1} \cdot (k-1) \cdot s + 1 \leq i, j \leq 2^{l-1} \cdot k \cdot s$. The coarse computational structure is the following:

```
Hierarchical Matrix Multiplication(n,p) =
  for (l = 1 to log p + 1)
    for k = 1, ..., p/2^(l-1) compute in parallel
      compute_block (C_lk);
```

Figure 1, bottom row, illustrates the computation of blocks C_{lk}. A diagonal block C_{lk} is computed by calling compute_block(C_{lk}) which is performed in parallel by all processors of a group. If in one group only a single processor q performs compute_block, i.e. $l = 1$, this processor computes one initial diagonal block C_{lk} by using its local entries of A and B. Otherwise, the computation of the two diagonal sub-blocks $C_{l-1,2k-1}$ and $C_{l-1,2k}$ of C_{lk} have already been completed in the preceding step by two other processor groups and the computation of C_{lk} is completed by computing the remaining sub-blocks $C'_{l-1,2k-1}$ and $C'_{l-1,2k}$ in the following way:

The initial column blocks of B are virtually grouped into larger column blocks according to the hierarchical binary clustering: for $1 \leq l \leq \log p + 1$ and $1 \leq k \leq p/2^{l-1}$, column block B_{lk} contains $s' \cdot 2^{l-1}$ columns of B; these columns have the indices $(2^{l-1}(k-1) \cdot s' + 1, \ldots, 2^{l-1}k \cdot s')$. The first index l of B_{lk} determines the size of the column block, the second index k numbers the column blocks of the same size.

The function compute_block() first exchanges the column blocks $B_{l-1,2k-1}$ and $B_{l-1,2k}$ of matrix B that are needed for the computation of $C'_{l-1,2k-1}$ and $C'_{l-1,2k}$, respectively, between the processors of the corresponding groups. This can be done in parallel since the processors of the group can be grouped into pairs which exchange their data. After the transfer operations the sub-blocks $C'_{l-1,2k-1}$ and $C'_{l-1,2k}$, respectively, are computed in parallel by recursive calls. At any point in time, each local memory needs to store at most s rows of A and s' columns of B and only columns of B are exchanged between the local memories.

3 Task Parallel Implementation

The realization of the task parallel matrix multiplication (*tpMM*) is based on a hierarchy of multiprocessor groups. The resulting implementation uses the runtime library *Tlib* which supports the programming with hierarchically structured M-tasks and provides a tool to handle multiprocessor groups built on top of MPI communicators [6].

The program realizes the recursive structure of the algorithm and uses a description of the block of the result matrix C that is computed in the current recursion step. This description contains the start column and the extent of the sub-block. The implementation exploits that the algorithm fills the basic blocks of C by alternating between basic blocks in the diagonal and the anti-diagonal position, see Figure 1. More precisely, the recursion in each phase subdivides the current block of C into sub-blocks containing 2×2 basic blocks, which are then filled in the diagonal and anti-diagonal direction. The program of *tpMM* uses the functions below. The variables $A, B, C, m, n, k, m_A = m/p$ and $k_B = k/p$ are declared and defined globally.

compute_block(*comm***,** *lcc***,** *cc***,** *type***)** is the recursive function for computing $C = A \cdot B$. `comm` is the current communicator of the recursion step. `lcc` denotes the leftmost column of C and `cc` specifies the number of columns of C for the next recursion step. `type` $\in$ {DIAGONAL, ANTIDIAGONAL} indicates if compute_block updates a diagonal or anti-diagonal block of C.

multiply(*cc***,** *lcc***)** performs the actual work of multiplying two sub-matrices and computes one basic block of C. The function is performed on a single processor and is realized by using fast one-processor implementations such as *BLAS* or *ATLAS*.

exchange_columns(*comm***)** performs the data exchange between pairs of processors in the current communicator. For each call of the function, each processor participates in exactly one data exchange. The function ensures that

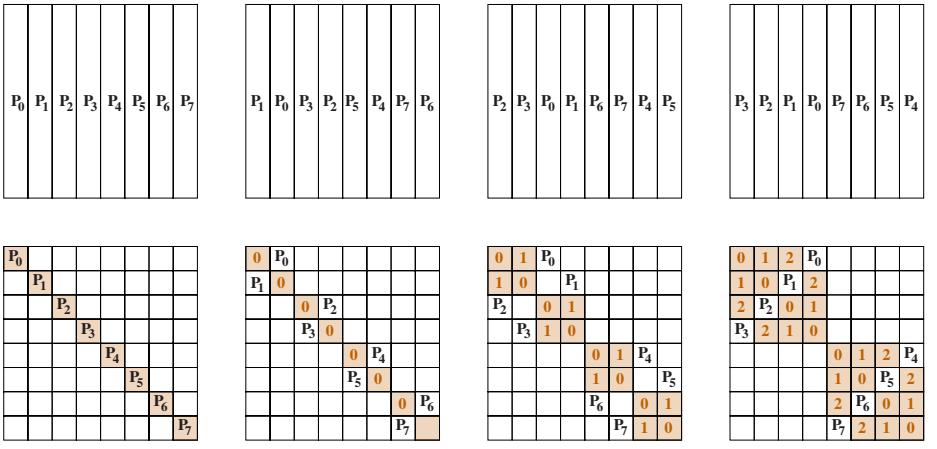

Fig. 1. Data distribution of matrix B (top row) and computation order of the result matrix (bottom row) for processors $P_0, \ldots, P_7$ for the first half of the steps. Each block is labeled with the owning processor. The numbers 0,1,2 denote the phase in which the blocks of C are computed.

processor P_i sends/receives a block of B to/from P_j, $j = (i+\text{sizeof}(comm)/2) \bmod \text{sizeof}(comm)$.

The pseudo-code of `compute_block` is given below. To perform a multiplication the programmer just needs to call `compute_block` and pass the corresponding parameters. The computation phases of *tpMM* reuse the communicators several times according to the recursive structure of the algorithm. Figure 2 illustrates the recursive splitting and the communicator reuse for $p = 8$ processors.

```
function compute_block(comm, lcc, cc, type)
  if ( Comm_size(comm) == 1 ) { multiply(cc, lcc) }
  else {
    subcomm_i = split(comm); /* splitting into subcommunicator i={0,1} */
    if ( type == DIAGONAL )
      { lcc_0 = lcc; lcc_1 = lcc+cc/2 }
    else
      { lcc_0 = lcc+cc/2; lcc_1 = lcc }
    compute_block(subcomm_i, lcc_i, cc/2, DIAGONAL); /* task parallel */
    exchange_columns(comm);
    if ( type == DIAGONAL )
      { lcc_0 = lcc+cc/2; lcc_1 = lcc }
    else
      { lcc_0 = lcc; lcc_1 = lcc+cc/2 }
    compute_block(subcomm_i, lcc_i, cc/2, ANTIDIAGONAL); /* task parallel */
  }
```

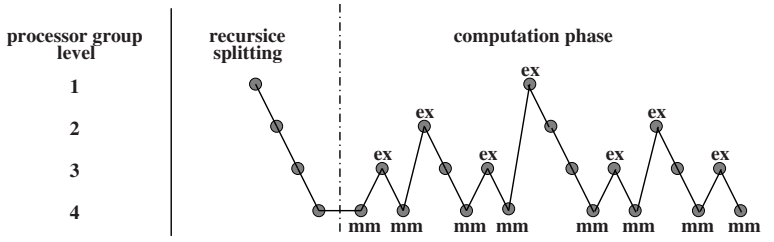

Fig. 2. Usage of processor groups during the computation of *tpMM* for three recursive splittings into sub-groups and four hierarchical levels. The matrices to be multiplied are decomposed into eight blocks of rows and columns, respectively. *mm* denotes the matrix multiplication for a single block, *ex* denotes the exchange of data at the corresponding communicator level.

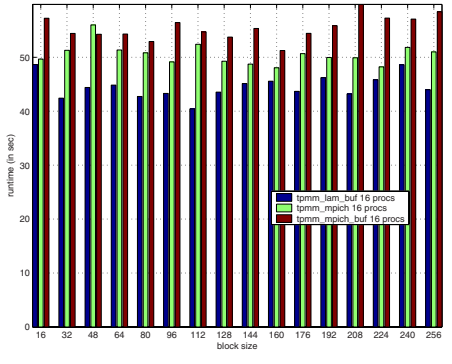

Fig. 3. *tpMM* overlapping tests on *CLiC* for matrix dimension $n = 4096$ and 16 processors.

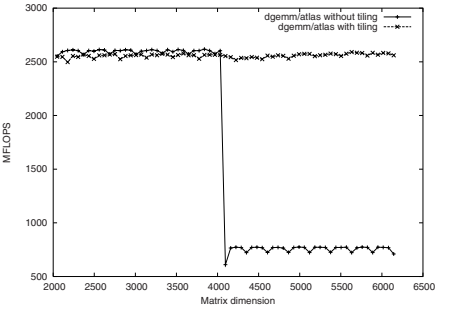

Fig. 4. comparison of *DGEMM* from *ATLAS* with and without tiling enabled (on *dual Beowulf cluster*; m, k=2048; n is varying).

4 Experimental Results

The runtime tests of *tpMM* were performed on a *IBM Regatta p690* (AIX, 6 x 32 processors, Power4+ 1.7GHz, Gigabit Ethernet) operated by the Research Centre Jülich, on a Linux *dual Beowulf cluster* (16 x 2 procs., Xeon 2.0 GHz, SCI network) and the *CLiC* (Chemnitzer Linux Cluster, 528 procs., P3 800 MHz, Fast-Ethernet) at TU Chemnitz.

Minimizing communication costs. The communication overhead of many applications can be reduced by overlapping communication with computation. To apply overlapping to *tpMM*, the block of B that each processor holds is not transfered entirely in one block. The blocks are rather send simultaneously in multiple smaller sub-blocks while performing local updates of matrix C. This requires non-blocking send and recv operations. Figure 3 shows runtime tests on *CLiC* using mpich and lam. The suffix "buf" refers to `MPI_Ibsend`, the

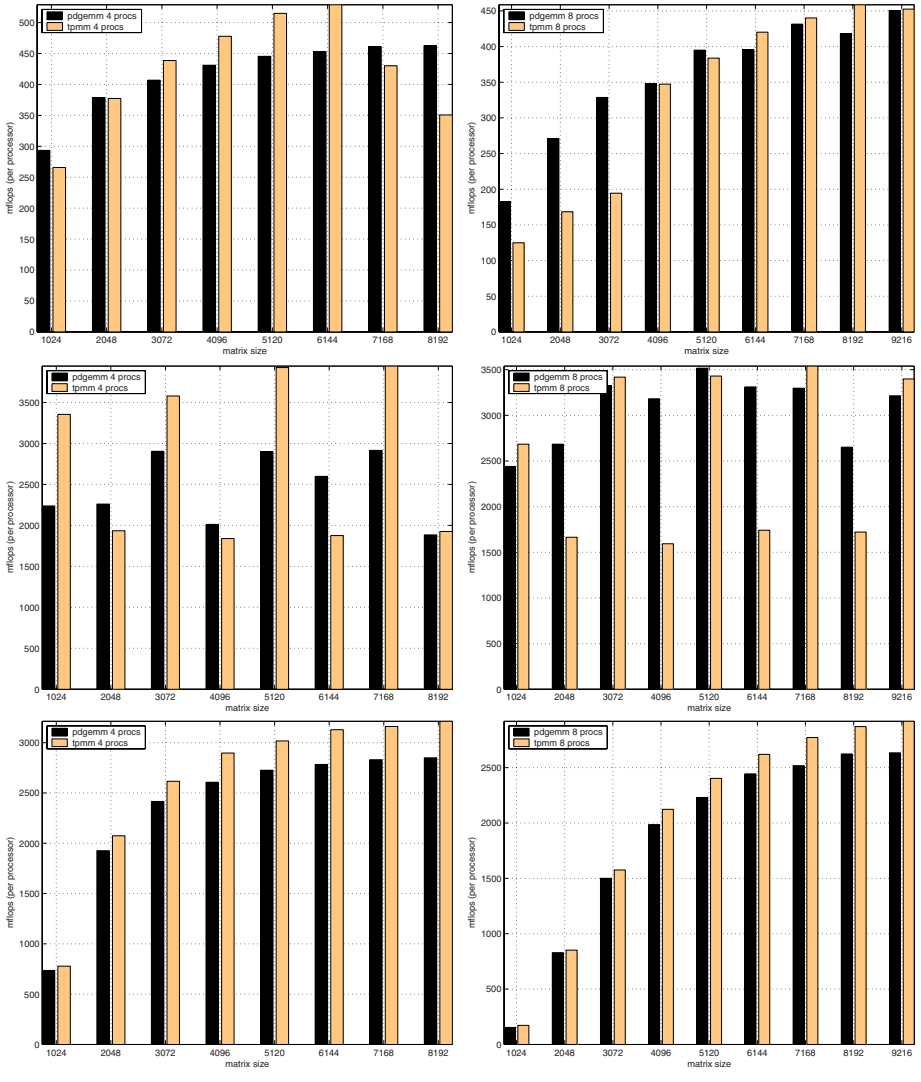

Fig. 5. MFLOPS per node reached by *PDGEMM* and *tpMM* on *CLiC*, *IBM Regatta p690* and *dual Beowulf cluster* (top to bottom).

buffered version of `MPI_Isend`. For these tests matrices A, B, and C of dimension 4096×4096 and 16 processors are used, so that each processor holds 256 columns of B. In the experiments local updates with block sizes (matrix B) of $4 \leq \text{blocksize} \leq 256$ are performed. For the full block size of 256, no overlapping is achieved and this result can be used for comparison. The experiments show that neither non-blocking nor non-blocking buffered communication leads to a significant and predictable improvement.

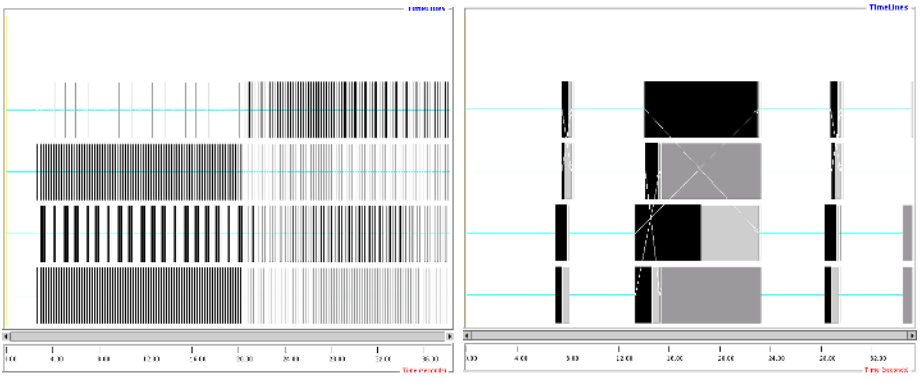

Fig. 6. JUMPSHOT-4 profiles of *PDGEMM* (upper) and *tpMM* (below) recorded on *dual Beowulf cluster* using 4 processors. Darker boxes represent Send-operations and segments in light grey denote either calls to MPI_Recv or MPI_Wait in case of non-blocking communication.

Underlying libraries. Low level matrix-matrix multiplications on one processor (*BLAS* level 3) are performed by *ATLAS* [7] which optimizes itself at compile time to gain maximum performance for a given architecture. Runtime experiments of *tpMM* on *dual Beowulf cluster* with more than 8 processors show a dramatic drop of the MFLOPS rate per node when using larger matrices (> 4096). According to a detailed profiling analysis the performance loss is caused by an internal call to *DGEMM*. Tests with a series of *DGEMM* matrix-matrix multiplications with fixed dimensions of m_A and k_B and variable n are presented in Figure 4. It turned out that when there are more than twice as many rows of B as columns *ATLAS* internally calls a different function which results in poor performance. This situation is likely to happen when executing *tpMM* with large input matrices. One possible work-around is a tiling approach of the original multiplication by dividing the problem into multiple sub-problems. The tiling of the local matrices A and B must ensure that each tile is as big as possible and two tiles must fulfill the requirements to perform a matrix-matrix multiplication (columns of tile t_i^A = rows of t_j^B). With tiling the local matrix-matrix multiplication achieves a similar MFLOPS-rate for all inputs (see Figure 4).

Overall performance evaluation of tpMM. Figures 5 shows the MFLOPS reached by *DGEMM* and *tpMM* on the three test systems considered. Since both methods perform the same number of operations (in different order), a larger MFLOPS rate corresponds to a smaller execution time. The figures show that for 4 processors, *tpMM* leads to larger MFLOPS rates on all three machines for most matrix sizes. For 8 processors, *tpMM* is usually slightly faster than *DGEMM*. For 16 processors, *tpMM* is faster only for the IBM Regatta system. The most significant advantages of *tpMM* can be seen for the IBM Regatta system. For 32 and more processors, *DGEMM* outperforms *tpMM* in most cases.

Figure 6 presents trace profiles of *PDGEMM* and *tpMM*. The profile of *PDGEMM* contains a huge number of communications even though only 4 processors were involved. In contrast, the pattern of *tpMM* shows only a small number of required communication calls. *PDGEMM* is superior if there are many processors involved and the matrix is sufficiently large. In these cases overlapping of computation with communication can be achieved and the block size remains suitable to avoid cache effects and communication overhead. On the other hand, *tpMM* decreases the communication overhead (e.g. numerous startup times) what makes it faster for a smaller group of nodes. Thus, *tpMM* is a good choice for parallel systems of up to 16 processors. For larger parallel systems, *tpMM* can be used as a building block in parallel algorithms with a task parallel structure of coarser granularity.

5 Conclusions

We have proposed a hierarchical algorithm for matrix multiplication which shows good performance for smaller numbers of processors. Our implementation outperforms *PDGEMM* for up to 16 processors on recent machines. Due to the good locality behavior, *tpMM* is well suited as building block in hierarchical matrix multiplication algorithms in which *tpMM* is called on smaller sub-clusters. Experiments have shown that *tpMM* can be combined with one-processor implementations which have been designed carefully to achieve a good overall performance.

References

1. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI c coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.
2. Frédéric Desprez and Frédéric Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. Technical Report RR2002-24, Laboratoire de l'Informatique du Parallélisme (LIP), June 2002. Also INRIA Research Report RR-4482.
3. R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.
4. Brian Grayson, Ajay Shah, and Robert van de Geijn. A High Performance Parallel Strassen Implementation. Technical Report CS-TR-95-24, Department of Computer Sciences, The Unversity of Texas, 1, 1995.
5. Qingshan Luo and John B. Drake. A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed-Memory Computers. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 221–226. ACM Press, 1995.
6. T. Rauber and G. Rünger. Library Support for Hierarchical Multi-Processor Tasks. In *Proc. of the Supercomputing 2002*, Baltimore, USA, 2002.
7. R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, 1997.