Exploiting the Locality Properties of Peano Curves for Parallel Matrix Multiplication

Michael Bader

Institut für Informatik, Technische Universität München, Germany
bader@in.tum.de

Abstract. The present work studies an approach to exploit the locality properties of an inherently cache-efficient algorithm for matrix multiplication in a parallel implementation. The algorithm is based on a blockwise element layout and an execution order that are derived from a Peano space-filling curve. The strong locality properties induced in the resulting algorithm motivate a parallel algorithm that replicates matrix blocks in local caches that will prefetch remote blocks before they are used. As a consequence, the block size for matrix multiplication and the cache sizes, and hence the granularity of communication, can be chosen independently. The influence of these parameters on parallel efficiency is studied on a compute cluster with 128 processors. Performance studies show that the largest influence on performance stems from the size of the local caches, which makes the algorithm an interesting option for all situations where memory is scarce, or where existing cache hierarchies can be exploited (as in future manycore environments, e.g.).

1 Introduction

Space-filling curves have become a quite well-established tool for parallelisation in scientific computing, which is mainly a result of their strong locality properties. In matrix computations, recursive and block-recursive approaches – which includes approaches based on space-filling curves, Morton order, and similar – can exploit such locality properties to obtain cache-efficient algorithms (see [4] for an overview). In [1,2], we introduced a block-recursive algorithm for matrix multiplication based on Peano space-filling curves, where the Peano curve's locality properties lead to an inherently cache-efficient multiplication scheme with a highly local access pattern to memory. The present paper addresses the question about how these locality properties can be used to obtain an efficient parallel implementation of matrix multiplication. The key idea of the presented approach is to add local software caches to replicate remote matrix blocks on the local processors, and thus turn an inherently cache-efficient algorithm into one that also scales well in a parallel implementation.

Existing parallel algorithms for matrix multiplication, such as PUMMA[3], SUMMA[5], or SRUMMA[7], are typically based on substructuring the involved matrices into smaller blocks. These blocks not only define the distribution of the matrices to several processing units, they also determine the data units

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 801–810, 2008.

that need to be transfered between the processors and, together with the parallel block layout, the resulting communication pattern. Prefetching of remote blocks, however, and overlapping communication with computation is restricted to double-buffering techniques in those approaches.

In contrast, the presented cache-oriented approach exploits the algorithm's strong locality properties for efficient prefetching and communication hiding. The size of the prefetched matrix blocks can be chosen independent of the amount of additional local memory to hold copies of remote matrix blocks, and independent of the dimension of the sequentially executed block multiplications (which also has a strong influence on achievable performance). In addition, the resulting algorithm can be tuned for specific applications – for example when memory is short and the amount of additional local memory is therefore the limiting factor – or for specific hardware, such as for latency and bandwidth of the communication. In a parallel work[6], we showed that a hardware-oriented implementation of our Peano algorithm achieves excellent performance on multicore platforms. Hence, the present study also aims at estimating the capability of the Peano multiplication for future manycore processors with 10–100 cores.

2 Matrix Multiplication Using Peano Curves

To compute the product of two $n \times n$-matrices, as in $C = C + AB$, we need to perform the update $c_{ij} = c_{ij} + a_{ik}b_{kj}$ for all triples $(i, j, k) \in \{1, \ldots, n\}^3$. Due to commutativity, we can execute the updates in any sequence we find appropriate; i.e. we may choose any 3D-traversal of the index space $\{1, \ldots, n\}^3$. Similarly, we may choose any suitable 2D-traversal of the index spaces of the matrices, $\{1, \ldots, n\}^2$, as a storage scheme to map the matrix elements to a contiguous sequence of memory addresses. In [1], we have shown that using a Peano curve for both the 3D- and the 2D-traversal, i.e. the sequence of element updates and the order of the matrix elements, leads to an inherently local scheme for matrix multiplication. For the simple example of multiplying two 3×3-matrices,

$$\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix} \begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix} = \begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}, \tag{1}$$

where the element indices indicate the order in which the elements are stored in memory, this leads to the execution order given in figure 1. Note that the update operations on the elements c_r are computed in an inherently local order – from each operation to the next, the involved matrix elements are either reused or one of their direct neighbours in memory is accessed.

Figure 2 illustrates how the Peano element order is extended to store larger matrices. We use 2D *iterations* of a Peano curve, which is described by a nested-recursive scheme of four block-numbering patterns: P, Q, R, and S. Starting from the initial pattern P, the four block patterns are recursively combined and lead to a contiguous storage scheme of matrix blocks. The recursion is stopped once the matrix blocks become smaller than a given block size. On these *atomic*

$$
\begin{array}{cccccccc}
c_0 \mathrel{+}= a_0b_0 & & c_3 \mathrel{+}= a_8b_3 & \rightarrow & c_4 \mathrel{+}= a_7b_3 & & c_7 \mathrel{+}= a_1b_6 & \rightarrow & c_8 \mathrel{+}= a_2b_6 \\
\downarrow & & \uparrow & & \downarrow & & \uparrow & & \downarrow \\
c_1 \mathrel{+}= a_1b_0 & & c_2 \mathrel{+}= a_8b_2 & & c_5 \mathrel{+}= a_6b_3 & & c_6 \mathrel{+}= a_0b_6 & & c_8 \mathrel{+}= a_3b_7 \\
\downarrow & & \uparrow & & \downarrow & & \uparrow & & \downarrow \\
c_2 \mathrel{+}= a_2b_0 & & c_1 \mathrel{+}= a_7b_2 & & c_5 \mathrel{+}= a_5b_4 & & c_5 \mathrel{+}= a_0b_5 & & c_7 \mathrel{+}= a_4b_7 \\
\downarrow & & \uparrow & & \downarrow & & \uparrow & & \downarrow \\
c_2 \mathrel{+}= a_3b_1 & & c_0 \mathrel{+}= a_6b_2 & & c_4 \mathrel{+}= a_4b_4 & & c_4 \mathrel{+}= a_1b_5 & & c_6 \mathrel{+}= a_5b_7 & & c_8 \mathrel{+}= a_8b_8 \\
\downarrow & & \uparrow & & \downarrow & & \uparrow & & \downarrow & & \uparrow \\
c_1 \mathrel{+}= a_4b_1 & \rightarrow & c_0 \mathrel{+}= a_5b_1 & & c_3 \mathrel{+}= a_3b_4 & \rightarrow & c_3 \mathrel{+}= a_2b_5 & & c_6 \mathrel{+}= a_6b_8 & \rightarrow & c_7 \mathrel{+}= a_7b_8
\end{array}
$$

Fig. 1. Optimal execution order for the 3×3-multiplication given in equation (1)

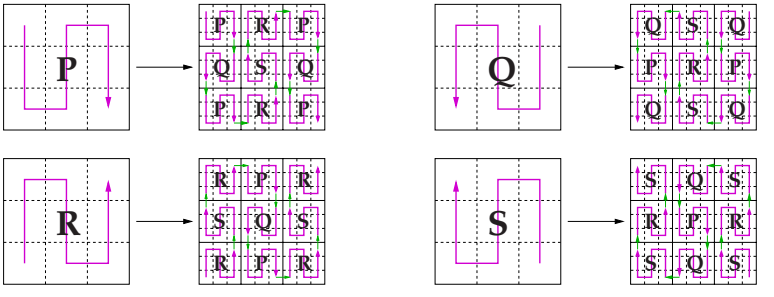


Fig. 2. Recursive construction of the Peano element order

blocks, standard column-major order is used, such that standard library routines (`dgemm`, e.g.) can be used for the sequential atomic block multiplications. Hence, the resulting hybrid numbering scheme is applicable to matrices of arbitrary size, including non-square matrices, if non-square atomic blocks are allowed.

The basic multiplication scheme for 3×3-matrices, as given in figure 1, extends to a block-recursive scheme for larger matrices, if we replace the matrix elements in (1) by matrix blocks numbered according to the Peano order. Equation (2) shows such a blockwise matrix multiplication. Each matrix block is named with respect to its numbering pattern and indexed with the name of the global matrix and the position within the storage scheme:

$$
\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}
\begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix}
=
\begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}. \tag{2}
$$

The block operations are executed following the scheme given in figure 1, starting with $P_{C0} \mathrel{+}= P_{A0}P_{B0}, Q_{C1} \mathrel{+}= Q_{A1}P_{B0}, P_{C2} \mathrel{+}= P_{A2}P_{B0}$, etc. For block multiplications such as $Q_{C1} \mathrel{+}= Q_{A1}P_{B0}$, where matrices are numbered according to alternate numbering patterns, schemes analogous to that in figure 1 are derived, where one, two, or all three of the indices of the three involved matrices are traversed in inverse order. The resulting eight recursive multiplication schemes can thus be combined into a single recursive procedure, where the three index traversal directions are given as parameters – cf. figure 5 in section 4 for a rough sketch or [1] for the full algorithm.

3 Exploiting the Peano Algorithm's Locality Properties

The resulting Peano algorithm for matrix multiplication has excellent locality properties, which are illustrated by its memory access pattern plotted in figure 3. In [1], we quantified these locality features by proving the following properties:

P1. The element traversal of all three involved matrices can be achieved entirely by index increments and decrements: after an element is accessed, the next access will be either to itself or to its direct left or right neighbour.

P2. Any sequence of k^3 floating point operations is executed on only $\mathcal{O}(k^2)$ *contiguous* elements in each matrix. Vice versa, on any block of k^2 contiguous elements, at least $\mathcal{O}(k^3)$ operations are performed. Hence, we can precisely predict how much computing time is spent on any given block of memory.

P3. As a result, a machine that only operates on a working memory consisting of M lines of L elements each, such as a cache memory or a replicated block of memory within a parallel computer, will require only $\mathcal{O}\big(n^3/(L\sqrt{M})\big)$ transfer operations to load matrix elements into the working memory – which is asymptotically optimal.

Property P1 motivates to use the model of a parallel, multi-tape Turing machine to describe the adopted approach to efficiently parallelise the Peano algorithm. Hence, let's consider the model of a parallel Turing machine with three tapes to store the matrices A, B, and C, and with several processing units that simultaneously access the shared Turing tapes via their respective read-write-heads (as illustrated in figure 4). Property P1 then guarantees that all read-write-heads will only move to directly neighbouring elements on the tapes.

To let our Turing machine more closely resemble real-world parallel computers, we allow each Turing unit to replicate a section of each matrix tape in some kind of local memory. Property P2 then guarantees that each Turing unit will spend a guaranteed amount of computing time within these replicated sections of memory. As at least $\mathcal{O}(k^3)$ operations will be executed, the units can precisely estimate when the end of the replicated section will be reached, and can thus issue a timely relocation of the replicated section (i.e. a prefetch of elements).

Finally, property P3 gives an estimate on how often the local copies of the Turing tapes have to be updated. Hence, if the tapes are stored in the distributed memory of a parallel computer, property P3 is a precise estimate of the required number of communication operations.

4 Parallelisation and Implementation

The parallel implementation of the Peano algorithm closely follows this idea of a multi-tape Turing machine. The Global Arrays toolkit [9] and the underlying ARMCI library [8] are used for distributed storage of the matrices and for communication. Each involved matrix is stored as a *global array*, which is evenly distributed to all available processes and takes the role of a Turing tape. Each processor holds a *tape cache* (as in figure 4) that is implemented to replicate a

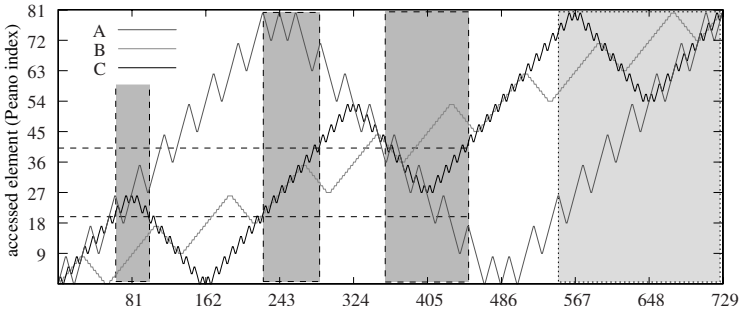


Fig. 3. Locality of data access during the Peano multiplication; the diagram shows the accessed memory locations within matrices A, B, and C throughout the 729 subsequent element operations of a 9×9 matrix multiplication. The grey boxes indicate one parallel partition when using the work-oriented partitioning (dotted box) or the *owner-computes* partitioning (dashed boxes) – see also section 4.

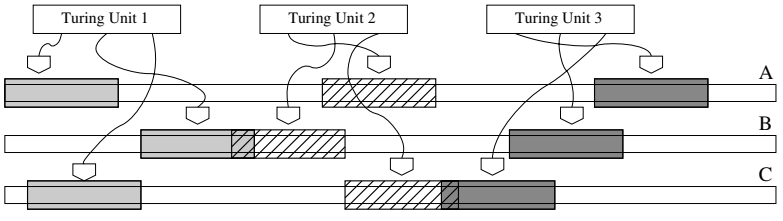


Fig. 4. Parallel Turing machine with several control units; each unit controls three heads that write to the three tapes jointly used to store the matrices. The highlighted parts of the matrix tapes are replicated in some local memory of the Turing units.

part of the global array in local memory. The tape caches are organised into four cache lines that hold a given number of atomic matrix blocks.

Two *read caches* replicate matrix blocks of A and B. The currently accessed cache line and its two (cyclic) neighbours always hold a contiguous section of the respective global array. The fourth block, in the meantime, prefetches one of the adjacent cache lines, using the `phsA` and `phsB` parameters to anticipate the next accessed block. As in the SRUMMA approach[7], non-blocking communication ensures that explicit prefetching into the tape cache and block multiplications on other tape cache lines are performed in parallel.

In addition, a *write cache* accumulates block products that have to be added to the result matrix C. A tape cache line that is accessed for the first time is initialised with zeros. At the same time, the least recently used cache line is written back to distributed memory by initiating a non-blocking operation that accumulates the intermediate result to the respective block of the result matrix (using ARMCI's non-blocking accumulate call `NGA_NbAcc`).

The tape cache mechanism is not only responsible for saving communication operations; it also encapsulates all communication operations and hides them

```
peanomult(int phsA, int phsB, int phsC, int dim) {
  if ((dim <= BLOCKSIZE) && /* block mult. in local task list */ ){
    // manage read and write access to matrix blocks in tape caches:
    Abuf = Acache.readAccess(a, phsA);
    Bbuf = Bcache.readAccess(b, phsB);
    Cbuf = Ccache.writeAccess(c);
    // call BLAS-dgemm for block matrix multiplication:
    dgemm ('n', 'n', dim, dim, dim, 1.0,
           Abuf, dim, Bbuf, dim, 1.0, Cbuf, dim);
  } else {
    /* 27 recursive calls: */
    peanomult( phsA, phsB, phsC, dim/3); a += phsA; c += phsC;
    peanomult( phsA,-phsB, phsC, dim/3); a += phsA; c += phsC;
    peanomult( phsA, phsB, phsC, dim/3); a += phsA; b += phsB;
    /* ... */
  } }
```

Fig. 5. Sketch of the parallelised Peano multiplication: the parameters `phsA`, `phsB`, and `phsC` (values ± 1) control which of the eight recursive multiplication schemes is used. `Acache`, `Bcache`, and `Ccache` are the local tape caches for matrices A, B, and C. `a`, `b`, and `c` (here as global variables) are the starting indices of the current matrix blocks. Each processor performs the entire recursion, and decides for each atomic block multiplication whether this is part of its own task list.

from the rest of the implementation. The block recursive algorithm only requires one call for each matrix that requests the next accessed block from the respective tape cache – see figure 5 for the general structure of this algorithm.

For load distribution, the linear sequence of block multiplications generated by the Peano recursion is split into equally sized parts, which are distributed onto the available processors. For sufficiently small atomic blocks, this *task-list oriented* strategy leads to an excellent load balance. For example, three levels of recursion will lead to 27^3 atomic block operations – distributing these to 128 processors results in a load imbalance of less than 1 %. As an alternative approach, an *owner computes* scheme can be used, where each atomic block operation is performed by the processor that owns the respective atomic block of the result matrix C (see illustration in figure 3). This requires considerably smaller atomic block sizes to avoid load imbalances. However, this strategy completely avoids write access conflicts to the matrix C, and is therefore especially suited for multi- and manycore environments with shared cache memories[6].

5 Performance Results

The parallel implementation of the Peano algorithm was tested on an *Infiniband cluster* with 32 Opteron nodes; each node contains four AMD Opteron 850 processors (2.4 GHz) connected to 8 GB of shared memory, and is equipped with one MT23108 InfiniBand Host Channel Adapter card for communication. The atomic block multiplications were executed by the `dgemm` implementation of

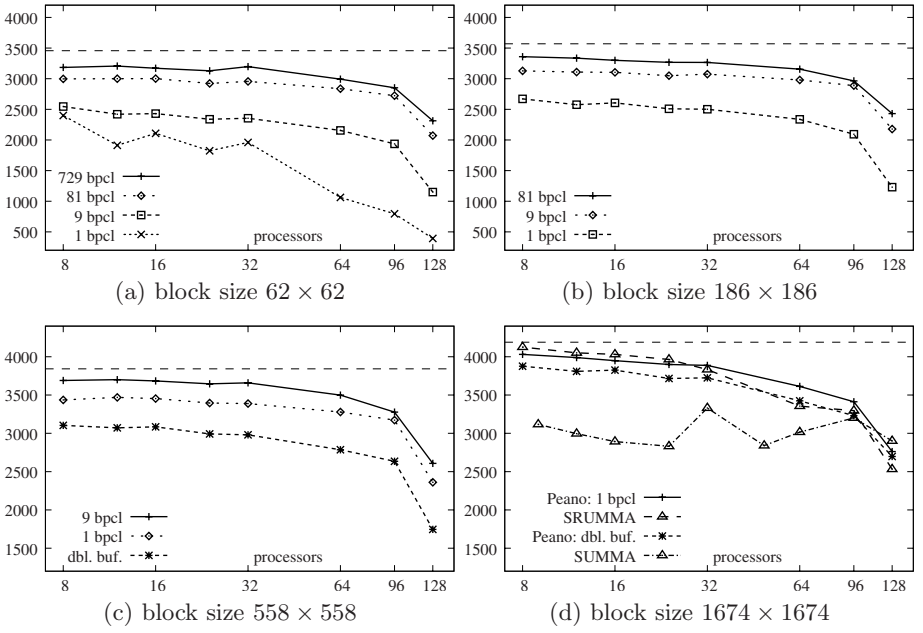


Fig. 6. Parallel efficiency of the parallel Peano algorithm: diagrams (a)–(d) show the achieved MFlop/s per processor for various atomic block sizes. Results are given for different sizes of the tape caches ('bpcl' = 'blocks per cache line') and, in (c) and (d) for double buffering ('dbl. buf.'). The dashed horizontal lines mark the achievable sequential performances for the respective block size. Diagram (d) gives the performance of SRUMMA[7] and ScaLAPACK's SUMMA implementation[5] for comparison.

ACML (AMD Core Math Library, v. 3.6.0). Performance was always evaluated in terms of achieved MFlop/s *per processor* to show the parallel efficiency of the method. The matrix size for all tests was 15066×15066. We used Peano layouts of 243×243, 81×81, 27×27, and 9×9 atomic blocks, starting with 62×62 as the smallest atomic block size (the best size for the level 1 hardware cache[6]).

Figure 6 shows the MFlop/s rates measured for increasing number of processors, and expresses how the parallel efficiency of the Peano algorithm depends on the size of the atomic blocks and of that of the tape caches. At least for up to 32 processors, the Peano implementation scales well for all atomic block sizes. For more processors, efficiency deteriorates, as several processors have to share only one InfiniBand adapter for communication, which noticeably reduces bandwidth and latency, and hence also parallel performance. We also observe a general increase of MFlop/s with growing size of the tape caches. By increasing the cache sizes, the performance can be driven close to the achievable sequential performance, which is determined by ACML's performance for the given block size (dashed lines). Diagram (d) also includes the performance of the SRUMMA implementation in Global Arrays [7,9] and that of ScaLAPACK's implementation of SUMMA[5] to show that the Peano implementation is well competitive

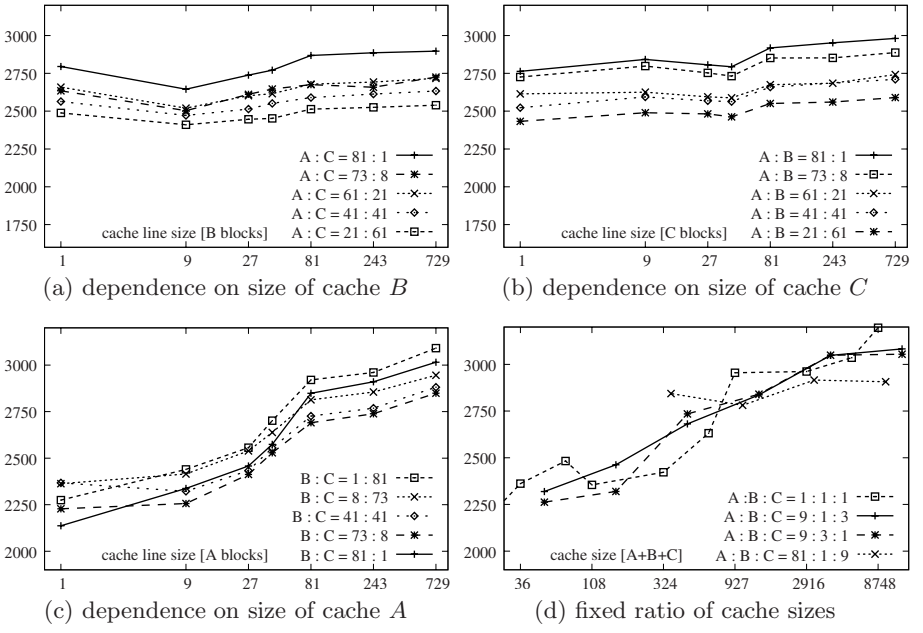
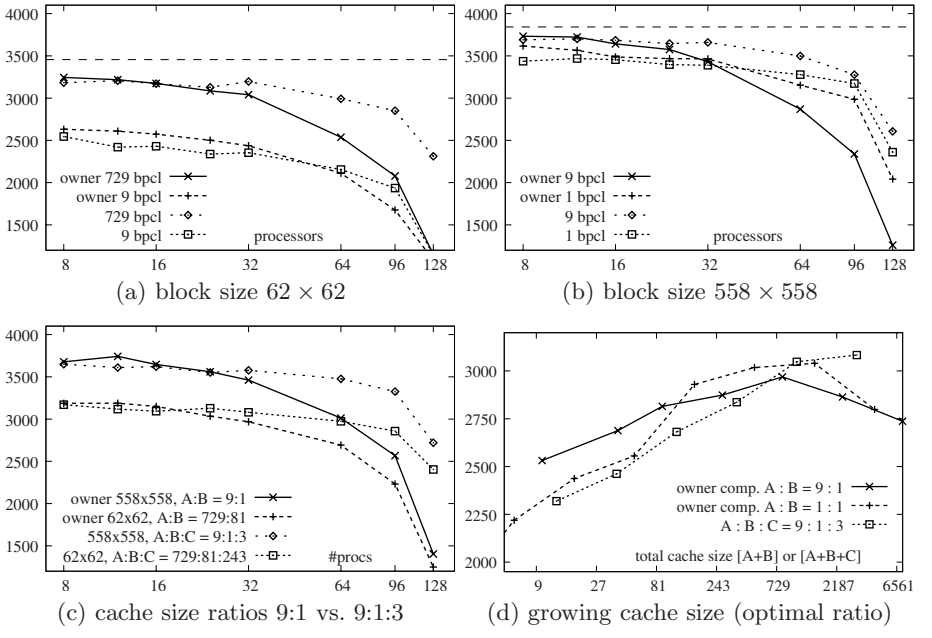


Fig. 7. Parallel efficiency (on 32 processors with uniform block size 62×62) for varying sizes of the tape caches for matrices A, B, and C. In plot (d), the given cache size indicates the total number of 62×62-blocks cached in all three tape caches.

compared to these established approaches. Apparently, SUMMA falls behind SRUMMA and the Peano algorithm, as long as these can overlap communication and computation. In contrast, SUMMA's minimisation of communication makes it less vulnerable to slow communication when all 128 processors are used.

From the memory access pattern given in figure 3, we can expect that choosing caches of different size for the three involved matrices should be advantageous. For example, the access pattern to B is much more local than that to A, which suggests a smaller cache for B. Judging from the access patterns an optimal cache size ratio of $A : B : C = 9 : 1 : 3$ is to be expected. Performance tests to study this aspect in detail are illustrated in figure 7: diagrams (a), (b), and (c), plot the performance when only one of the three cache sizes is increased while the respective other two cache sizes are kept constant. In addition, different size ratios between the other two caches were tested. Diagrams 7(a) and (b) indeed show a slight increase of performance with growing sizes of B and C; however, the more substantial performance gain seems to result from increasing the cache for A, which is supported by the results in figure 7(c), where an increase of the cache size for A, while keeping the cache sizes for B and C constant, leads to a much stronger performance gain. Figure 7(c) also indicates best performance, when the cache for C is chosen comparably large. Hence, figure 7(d) compares the performance when using different fixed ratios of the cache sizes. As expected, the ratio $9 : 1 : 3$ leads to the smoothest increase of performance with growing total



(a) block size 62×62 (b) block size 558×558

(c) cache size ratios 9:1 vs. 9:1:3 (d) growing cache size (optimal ratio)

Fig. 8. Performance of the owner-computes scheme ('owner') compared to work-oriented distribution: In (a) and (b), results (in MFlop/s per processor) are given for different sizes of the tape caches (analogous to Fig. 6). In (c), the optimal cache size ratios where used (total cache size identical; two different block sizes). Plot (d) shows the impact of growing cache size (optimal vs. uniform cache size ratio).

cache size, and seems to be the best overall choice. However, the performance differences are small and are overlapped by local performance maxima that occur when a cache line size matches the block size of the Peano layout.

Figure 8 compares the performance when using the *owner computes* approach for load distribution instead of the regular work-oriented approach. The *owner computes* strategy proved to be faster for up to 16 or even 32 processors. For more processors, performance quickly deteriorates, and the work-oriented distribution is clearly the better choice. This slow-down is due to the additional cache misses that occur for the owner computes scheme (note the "splitted" owner-computes partition in figure 3). At such tape cache misses, prefetching of the operand matrices fails, and a blocking call to obtain the required matrix blocks is necessary. This leads to performance penalties for large blocks and especially for large tape cache lines, but also for short task lists (if comparably many processes are used), as then the number or the cost of the tape cache misses grows.

The results for the *owner computes* strategy are especially important for using the Peano algorithm on multi- and manycore platforms[6], because in such a setting the owner computes strategy avoids costly synchronisation of the write accesses to matrix C, and especially of resulting cache coherence conflicts.

6 Conclusion

The present study shows that the inherently cache efficient Peano algorithm can be turned into a competitive parallel implementation of matrix multiplication. The key idea is to include an additional cache level, the *tape caches*, to store and prefetch remote matrix blocks. The performance results show that the size of these caches is the key parameter to achieve optimal parallel efficiency. Thus, the Peano algorithm is especially suitable for situations where memory is scarce and can not easily be invested for replicating large remote matrix blocks.

Our primary aim, however, is to combine the hardware-oriented multicore implementation of our Peano algorithm[6] with the parallelisation approach presented in this paper, which will require the combination of the owner-computes approach with the work-oriented load distribution. In such an algorithm, only two components would need to be hardware-aware: the multiplication kernel for the atomic block multiplications has to be tuned to the specific CPU; and the size of the tape caches has to be adopted to the communication parameters (latency and bandwidth) of the parallel platform. The goal is a both parallel and cache oblivious algorithm that consequently exploits the Peano curve's locality properties on all memory levels, and therefore works well on parallel platforms of all kind – from multi- and manycore CPUs up to parallel compute clusters.

References

1. Bader, M., Zenger, C.: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra Appl. 417(2–3) (2006)
2. Bader, M., Franz, R., Guenther, S., Heinecke, A.: Hardware-oriented Implementation of Cache Oblivious Matrix Operations Based on Space-filling Curves. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967. Springer, Heidelberg (2008)
3. Choi, J., Dongarra, J.J., Walker, D.W.: PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. Concurrency: Practice and Experience 6(7) (1994)
4. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1) (2004)
5. van de Geijn, R., Watts, J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm. Concurrency: Practice and Experience 9(4) (1997)
6. Heinecke, A., Bader, M.: Parallel Matrix Multiplication based on Space-filling Curves on Shared Memory Multicore Platforms. In: Proc. 2008 Computing Frontiers Conf. and co-located workshops: MAW 2008 & WREFT 2008, Ischia (2008)
7. Krishnan, M., Nieplocha, J.: SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems. In: Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS 2004) (2004)
8. Nieplocha, J., Carpenter, B.: ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In: Proc. of RTSPP IPPS/SDP (1999)
9. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Apra, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. Int. J. of High Perf. Comp. Appl. 20(2) (2006)