Complexity analysis of matrix product on multicore architectures

Mathias Jacquelin, Loris Marchal and Yves Robert
École Normale Supérieure de Lyon, France
{Mathias.Jacquelin|Loris.Marchal|Yves.Robert}@ens-lyon.fr

LIP Research Report RR2008-41

Abstract

The multicore revolution is underway. Classical algorithms have to be revisited in order to take hierarchical memory layout into account. In this paper, we aim at minimizing the number of cache misses paid during the execution of the matrix product kernel on a multicore processor, and we show how to achieve the best possible trade-off between shared and distributed caches.

1 Introduction

Dense linear algebra kernels are the key to performance for many scientific applications. Some of these kernels, like matrix multiplication, have extensively been studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [4] and the ScaLAPACK outer product algorithm [2]. Typically, parallel implementations work well on 2D processor grids: input matrices are sliced horizontally and vertically into square blocks; there is a one-to-mapping of blocks onto physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor grids.

However, algorithms based on a 2D grid (virtual) topology are not well suited for multicore architectures. In particular, in a multicore architecture, memory is shared, and data accesses are performed through a hierarchy of caches, from shared caches to distributed caches. We need to take further advantage of data locality, in order to minimize data movement. This hierarchical framework resembles that of out-of-core algorithms [6] (the shared cache being the disk) and that of master-slave implementations with limited memory [8] (the shared cache being the master's memory). The latter paper presents the Maximum Re-use Algorithm which aims at minimizing the communication volume from the master to the slaves. Here, we adapt this study to multicore architectures, by taking both cache levels into account. Due to lack of space, the review of related work is provided in the companion research report [7].

2 Problem statement

2.1 Modeling multicore architectures

A major difficulty of this study is to come up with a realistic but still tractable model of a multicore processor. We assume that such a processor is composed of p cores, and that each core has the same computing speed. The processor is connected to a memory, which is supposed to be large enough to contain all necessary data (we do not deal with out-of-core execution here). The data path from the memory to a computing core goes through two levels of caches. The first level of cache is shared among all cores, and has size C_S, while the second level of cache is distributed: each core has its own private cache, of size C_D. Caches

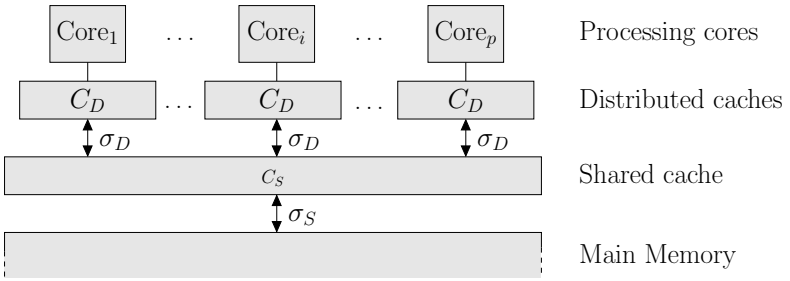

Figure 1: Multicore architecture model.

are supposed to be *inclusive*, which in our case means that the shared cache contains *at least* all the data stored in every distributed cache. Therefore, this cache must be larger than the union of all distributed caches: $C_S \geq p \times C_D$. Our caches are also "fully associative", and can therefore store any data from main memory. Figure 1 depicts the multicore architecture model.

The hierarchy of caches is used as follows. When a data is needed in a computing core, it is first sought in the distributed cache of this core. If the data is not present in this cache, a *distributed-cache miss* occurs, and the data is then sought in the shared cache. If it is not present in the shared cache either, then a *shared-cache miss* occurs, and the data is loaded from the memory in the shared cache and afterward in the distributed cache. When a core tries to write to an address that is not in the caches, the same mechanism applies. Rather than trying to model this complex behavior, we will assume in the following that we have an *ideal cache model* [5]: we suppose that we are able to totally control the behavior of each cache, and that we can load any data into any cache (shared of distributed), with the constraint that a data has to be first loaded in the shared cache before it could be loaded in the distributed cache. Although somewhat unrealistic, this simplified model has been proven not too far from reality: it is shown in [5] that an algorithm causing N cache misses with an ideal cache of size L will not cause more than $2N$ cache misses with a cache of size $2L$ and implementing a classical LRU replacement policy.

In the following, our objective is twofold: (i) minimize the number of cache misses during the computation a matrix product, and (ii) minimize the predicted data access time of the algorithm. To this end, we need to model the time needed for a data to be loaded in both caches. To get a simple and yet tractable model, we consider that cache speed is characterized by its bandwidth. The shared cache has bandwidth σ_S, thus a block of size S needs S/σ_S time-unit to be loaded from the memory in the shared cache, while each distributed cache has bandwidth σ_D. Moreover, we assume that concurrent loads to several distributed caches are possible without contention.

Finally, the purpose of the algorithms described below is to compute the classical matrix product $C = A \times B$. In the following, we assume that A has size $m \times z$, B has size $z \times n$, and C has size $m \times n$. We use a block-oriented approach, to harness the power of BLAS routines [2]. Thus, the atomic elements that we manipulate are not matrix coefficients but rather square blocks of coefficients of size $q \times q$. Typically, $q = 80$ or 100 on most platforms.

2.2 Minimizing communication volume

The key point to performance in a multicore architecture is an efficient use of the data. A simple way to assess data locality is to count and to minimize the number of cache misses, that is the number of times each data has to be loaded in a cache. Since we have two types of caches in our model, we try to minimize both the number of misses in the shared cache and the number of misses in the distributed caches.

We denote by M_S the number of cache misses in the shared cache. As for distributed caches, since accesses from different caches are concurrent, we denote by M_D the maximum of all distributed caches misses: if $M_D^{(c)}$ is the number of cache misses for the distributed cache of core c, $M_D = \max_c M_D^{(c)}$.

In a second step, since the former two objectives are conflicting, we aim at minimizing the overall time T_{data} required for data movement. With the previously introduced bandwidth, it can be expressed as:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

Depending on the ratio between cache speeds, this objective provides a tradeoff between both cache miss quantities.

2.3 Lower bound on communication

In [6], Irony, Toledo and Tiskin propose a lower bound on the number of communications needed to perform a matrix product. Their study focuses on a system with a memory of size M and concludes that the communication-to-computation ratio of a matrix product is lower-bounded by $\sqrt{\frac{27}{8M}}$. We have extended this study to our hierarchical cache architecture (see details in [7]). In our case, the communication-to-computation is the ratio between the number of cache misses and the number of operations. With $comp(c)$ being the amount of computation done by core c, we can define this ratio for both type of caches: $CCR_S = M_S/(\sum_c comp(c))$ is the CCR for the shared cache, and $CCR_D = \frac{1}{p}\sum_{c=1}^{p}(M_D/comp(c))$ is the CCR for the distributed cache (averaged over all cores c).

The overall amount of computation for the matrix product is mnz, and in all our algorithms, this amount is equally balanced among cores, so that $comp(c) = mnz/p$ for all cores. In both case, we have been able to extend the previous bound:

$$CCR_S \geq \sqrt{\frac{27}{8C_S}} \quad \text{and} \quad CCR_D \geq \sqrt{\frac{27}{8C_D}}.$$

3 Maximum re-use algorithm for multicore architectures

In the analysis of [6], lower bounds are obtained when the three matrices A, B and C are equally accessed throughout time. This naturally leads to allocating one third of the available memory space to each matrix. Indeed, the authors of [6] proves that, under some conditions, this framework leads to a communication-to-computation ratio of $O\left(\frac{mnz}{\sqrt{M}}\right)$, of the same order as the lower bound.

Check this part :
However, our goal is to create a data-thrifty algorithm, re-using matrix elements as much as possible and loading required data only one time in a given loop. Since the outermost loop is prevalent in order to reduce the total amount of loaded data, we must load the largest square block of data in this loop. Therefore, in the inner loops, we must load the smallest blocks in respect with the two main objective of our algorithm. This is the basis of the algorithm presented in [8].

It proposes to split the available memory into $1 + \mu + \mu^2$ blocks, so as to store a square block $C_{i_1 \ldots i_2, j_1 \ldots j_2}$ of size μ^2 of matrix C, a row $B_{i_1 \ldots i_2, j}$ of size μ of matrix B and one element $A_{i,j}$ of matrix A (with $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$). This allows to computes $C_{i_1 \ldots i_2, j_1 \ldots j_2} += A_{i,j} \times B_{i_1 \ldots i_2, j}$. Then, with the same block of C, other computations can be accumulated by considering other elements of A and B. The block of C is stored back only when it has been processed entirely, thus avoiding any future need of reading this block to accumulate other contributions. Using this framework, we can reach a communication-to-computation ratio of $\frac{2}{\sqrt{M}}$ for a memory of size M, for large matrices.

To adapt the maximum re-use algorithm to multicore architectures, we must take into account both cache levels. Depending on our objective, we adapt the previous data allocation scheme so as to fit with the shared cache, with the distributed caches, or with both. We define two parameters that are

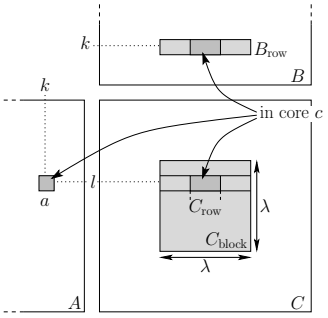

Figure 2: Data layout for Algorithm 1.

derived from the maximum re-use algorithm, and that will prove helpful to compute the size of the block of C that should be loaded in the shared cache or in a distributed cache:

- λ is the largest integer with $1+\lambda+\lambda^2 \leq C_S$;
- μ is the largest integer with $1+\mu+\mu^2 \leq C_D$.

In the following, we assume that λ is a multiple of μ, so that a block of size λ^2 that fits in the shared cache can be easily divided in blocks of size μ^2 that fit in the distributed caches.

3.1 Minimizing the number of shared-cache misses

In this section, we study the first objective presented in Section 2.2, namely the minimization of the number of shared-cache misses. This can be directly done by adapting the maximum re-use algorithm with parameter λ. As in the maximum re-use algorithm, a square block C_{block} of size λ^2 of C is allocated in the shared cache, together with a row of λ elements of B and one element of A. Then, the row of C_{block} is distributed and computed by the different cores. This is described in details in Algorithm 1, and the memory layout is depicted in Figure 2.

In this algorithm, the whole matrix C is loaded in the shared cache, thus resulting in mn cache misses. For the computation of each block of size λ^2, z rows of size λ are loaded from B, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total of $M_S = mn + 2mnz/\lambda$ shared-cache misses. For large matrices, this leads to a shared-cache CCR of $2/\lambda$, which is close to the lower bound.

3.2 Minimizing the number of distributed-cache misses

Our next objective is to minimize the number of distributed-cache misses. To this end, we use the parameter μ defined earlier to store in each distributed cache a square block of size μ^2 of C, a fraction of row (of size μ) of B and one element of A. Contrarily to the previous algorithm, the block of C will be totally computed before being written back to the shared cache. All p cores work on different blocks of C. Thanks to the constraint on the size of caches ($p \times C_D \leq C_S$), we know that the shared cache has the capacity to store all necessary data. The overall number of distributed cache misses on a core will then be $M_D = \frac{1}{p}(mn + 2mnz/\mu)$ (see [7] for details). For large matrices, this leads to a distributed-cache CCR of $2/\mu$, which is close to the lower bound.

3.3 Minimizing data access time

To get a tradeoff between minimizing the number of shared-cache and distributed-cache misses, we now aim at minimizing $T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$. In this case, the sketch of the algorithm, detailed in [7], is the following:

1. A block from C of size $\alpha \times \alpha$ is loaded in the shared cache. Its size satisfies $p \times \mu^2 \leq \alpha^2 \leq \lambda^2$. Both extreme cases are obtained when one of σ_D and σ_S is negligible in front of the other.
2. In the shared cache, we also load a block from B, of size $\beta \times \alpha$, and a block from A of size $\alpha \times \beta$. Thus, we have $2\alpha \times \beta + \alpha^2 \leq C_D$.
3. The $\alpha \times \alpha$ block of C is split into sub-blocks of size $\mu \times \mu$ which are processed by the different cores. These sub-blocks of C are cyclicly distributed among every distributed-caches. The same holds for the block-row of B which is split into $\beta \times \mu$ block-rows and cyclically distributed, row by row (i.e. by blocks of size $1 \times \mu$), among every distributed-cache.
4. The contribution of the corresponding β (fractions of) columns of A and β (fractions of) lines of B es added to the block of C. Then, another $\mu \times \mu$ block of C residing in shared cache is distributed among every

```
for Step = 1 to (m×n)/λ² do
    Load a new block C_block (of size λ × λ) from C in the shared cache
    for k = 1 to z do
        Load a row B_row (of size λ) from row z of B in the shared cache
        Distribute B_row to the distributed caches
        for l = 1 to λ do
            foreach core c in parallel do
                Load the element a = A[l, k] in the shared and distributed cache
                Load a row C_row (of size λ/p) from C_block in the distributed cache
                Compute the new contribution: C_row ← C_row + a × B_row
                Write back C_row to the shared cache
    Write back the block C_block to main memory
```

Algorithm 1: Adaptation of the maximum re-use algorithm.

distributed-cache, going back to step 3.

5. As soon as all elements of A and B have contributed to the $\alpha \times \alpha$ block of C, another β columns/lines from A/B are loaded in shared cache, going back to step 2.
6. Once the $\alpha \times \alpha$ block of C in shared cache is totally computed, a new one is loaded, going back to step 1.

With this algorithm, we get: $T_{\text{data}} = \frac{1}{\sigma_S}(mn + \frac{2mnz}{\alpha}) + \frac{1}{\sigma_D}(\frac{mnz}{p\beta} + \frac{2mnz}{p\mu})$. Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, it allows to compute the best value for parameters α and β, depending on the ratio σ_S/σ_D (again, see details in [7]).

4 Conclusion

In this short paper, we proposed cache-aware algorithms for multicore processors. We have proposed a model for multicore memory layout. Using this model, we have extended a lower bound on cache misses, and proposed cache-aware algorithms. For both types of caches, our algorithms reach a CCR which is close to the lower bound for large matrices. We also propose an algorithm for minimizing the overall data access time, which realizes a tradeoff between shared and distributed cache misses. This work will be extended to more complex operations, like LU factorization, and is to be validated through simulations or real experiments.

References

[1] David A. Bader, Varun Kanade, and Kamesh Madduri. Swarm: A parallel programming framework for multicore processors. In *IPDPS*, pages 1–8, 2007.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[3] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[4] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[5] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.

[6] Dror Irony, Sivan Toledo, and Alexandre Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[7] Mathias Jacquelin, Loris Marchal, and Yves Robert. Complexity analysis of matrix product on multicore architectures. Research report, LIP, December 2008.

[8] Jean-François Pineau, Yves Robert, Frédéric Vivien, and Jack Dongarra. Matrix product on heterogeneous master-worker platforms. *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 20-23 2008.

A Toledo's lower bound on communications

In this section, we derive a lower bound on the number of cache misses. Our analysis is based on the work in [6]. We consider a computing system (which consists of one or several computing cores) using a cache of size Z. We estimate the number of computations that can be performed owing to Z consecutive cache misses, that is owing to Z consecutive load operations. We recall that each matrix element is in fact a matrix block of size $q \times q$. We use the following notations :

- Let η_{old}, ν_{old}, and ξ_{old} be the number of blocks in the cache used by blocks of A, B and C just before the beginning of the Z cache misses.
- Let η_{read}, ν_{read}, and ξ_{read} be the number of blocks of A, B and C read in the main memory during these Z cache misses.

Before the beginning of the Z cache misses, the cache holds at most Z blocks of data, therefore, after Z cache misses, we have:

$$\begin{cases} \eta_{old} + \nu_{old} + \xi_{old} \leq Z \\ \eta_{read} + \nu_{read} + \xi_{read} = Z \end{cases} \tag{1}$$

A.1 Loomis-Whitney's inequality

The following lemma, given in [6] and based on Loomis-Whitney inequality, is valid for any conventional matrix multiplication algorithm $C = AB$, where A is $m \times z$, B is $z \times n$ and C is $m \times n$. A processor that contributes to N_C elements of C and accesses N_A elements of A and N_B elements of B can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications. According to this lemma, if we let K denote the number of elementary multiplications performed by the computing system, we have:

$$K \leq \sqrt{N_A N_B N_C}$$

No more than $(\eta_{old} + \eta_{read})q^2$ elements of A are accessed, hence $N_A = (\eta_{old} + \eta_{read})q^2$. The same holds for B and C: $N_B = (\nu_{old} + \nu_{read})q^2$ and $N_C = (\xi_{old} + \xi_{read})q^2$. Let us simplify the notations using the following variables:

$$\begin{cases} \eta_{old} + \eta_{read} = \eta \times Z \\ \nu_{old} + \nu_{read} = \nu \times Z \\ \xi_{old} + \xi_{read} = \xi \times Z \end{cases} \tag{2}$$

Then the following equation is obtained:

$$K = \sqrt{\eta\nu\xi} \times Z\sqrt{Z} \times q^3 \tag{3}$$

Writing $K = km\sqrt{m}q^3$, we obtain the following system of equation:

$$\text{MAXIMIZE } k \text{ SUCH THAT}$$
$$\begin{cases} k \leq \sqrt{\eta\nu\xi} \\ \eta + \nu + \xi \leq 2 \end{cases}$$

Note that the second inequality comes from Equation (1). This system admits a solution which is $\eta = \nu = \xi = \dfrac{2}{3}$ and $k = \sqrt{\frac{8}{27}}$. This gives us a lower bound on the communication-to-computation ratio (in terms of blocks) for any matrix multiplication algorithm:

$$CCR_{opt} = \frac{Z}{k \times Z\sqrt{Z}} = \sqrt{\frac{27}{8Z}}$$

A.2 Bound on shared-cache misses

We will first use the previously obtained lower bound to the study of shared-cache misses, considering everything above this cache level as a single processor and the main memory as a master which sends and receives data. Therefore, with $Z = C_S$, we have a lower bound on the communication-to-computation ratio for shared-cache misses:

$$CCR_{S_{opt}} = \frac{C_S}{K} = \sqrt{\frac{27}{8C_S}}$$

A.3 Bound on distributed-caches misses

In the case of the distributed caches, we first apply the previous result, on a single core c, with cache size C_D. We thus have

$$CCR_c \leq \sqrt{\frac{27}{8C_D}}$$

We have define the overall distributed CCR as the average of all CCR_c, so this result also holds for the CCR_D:

$$CCR_D \leq \sqrt{\frac{27}{8C_D}}$$

Indeed, we could even have a stronger result, on the minimum of all CCR_c.

A.4 Bound on overall data access time

The previous bound on the CCR can be extended to the data access time, as it is defined as a linear combination of both M_S and M_D:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

We can bound M_S using the bound on the CCR:

$$M_S = CCR_S \times mnz \leq mnz \times \sqrt{\frac{27}{8C_S}}$$

As for distributed-cache misses, it is more complex, since M_D is the maximum of all misses on distributed caches, and CCR_D is the average CCR among all cores. In order to get a tight bound, we consider only algorithms where both computation and cache misses are equally distributed among all cores (this applies to all algorithms developed in this paper). In this case, we have

$$M_D = \max_c M_D^{(c)} = M_D^{(0)} = \frac{mnz}{p} \times CCR_0 \leq \frac{mnz}{p} \times \sqrt{\frac{27}{8C_D}}$$

Thus, we get the following bound on the overall data access time:

$$T_{\text{data}} \leq mnz \times \left(\frac{1}{\sigma_S} \times \sqrt{\frac{27}{8C_S}} + \frac{1}{p\sigma_D} \times \sqrt{\frac{27}{8C_D}} \right).$$

B Minimizing the number of shared-cache misses

In this section, we study the first objective presented in section 2.2, namely the minimization of the number of shared-cache misses. Indeed, since there are two types of cache misses with different costs, giving priority to the minimization of the mose expensive kind of cache miss is a potential way to improve performances. We therefore introduce an algorithm which aims at minimizing M_S and then optimizes M_D accordingly.

This can be directly done by adapting the maximum re-use algorithm with parameter λ. Remember that λ is the largest possible integer such that $1 + \lambda + \lambda^2 \leq C_S$.

As in the maximum re-use algorithm, a block of size λ^2 of C is allocated in the shared cache, together with a row of λ elements of B and one element of A. Then, the block of C is divided into sub-blocks of size $1 \times \frac{\lambda}{p}$ broadcast among every distributed-caches. A row of $\frac{\lambda}{p}$ of B and one element of A are also loaded. The small rows of C are then processed by the cores and written back in shared-cache. Once this is done, another element of A is loaded in shared cache and another row of C could be updated following the same process. This is described in details in Algorithm 2.

Note that the space required in each distributed-cache to process $\frac{\lambda}{p}$ elements of C is $1+\frac{\lambda}{p}+\frac{\lambda}{p}$. For now, we have no assumption on the minimal size of distributed caches. We thus need to make an additional assumption on the distributed cache sizes, $C_D \geq \phi + \sqrt{C_S}$ where ϕ is a constant satisfying $\phi \geq 1$, to ensure that data fits into distributed-caches, that is to enforce that $1 + \frac{2\lambda}{p} \leq C_D$.

As a matter of a fact, we have $p \geq 2$ (since we work on multi-cores), and $\lambda \geq 1$ (at least one block of each matrices fits in shared cache). From this we notice that:

$$1 + \frac{2\lambda}{p} \leq 1 + \lambda \leq \phi + \lambda \leq \phi + \sqrt{1 + \lambda + \lambda^2} \leq \phi + \sqrt{C_S} \leq C_D$$

Therefore with this assumption on cache sizes, we are sure that distributed caches are large enough to perform the required computations.

Algorithm 2: Adaptation of the maximum re-use algorithm.

for Step $= 1$ *to* $\frac{m \times n}{\lambda^2}$ **do**
- Load a new block $C[i, \ldots, i+\lambda;\ j, \ldots, j+\lambda]$ of C in the shared cache
- **for** $k = 1$ *to* z **do**
  - Load a row $B[k;\ j, \ldots, j+\lambda]$ of B in the shared cache
  - **foreach** *core* $c = 1 \ldots p$ *in parallel* **do**
    - Load $B_c = B\left[k;\ j + \frac{c-1}{p} \times \lambda, \ldots, j + \frac{c}{p} \times \lambda\right]$ in the distributed cache of core c
  - **for** $i' = i$ *to* $i + \lambda$ **do**
    - Load the element $a = A[k;\ i']$ in the shared cache
    - **foreach** *core* $c = 1 \ldots p$ *in parallel* **do**
      - Load the element $a = A[k;\ i']$ in the distributed cache of core c
      - Load $C_c = C\left[i';\ j + (c-1) \times \frac{\lambda}{p}, \ldots, j + c \times \frac{\lambda}{p}\right]$ in the distributed cache of core c
      - Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
      - Update block C_c in the shared cache
- Write back the block of C to the main memory

B.1 Shared-cache misses

The number of shared-cache misses is:

$$M_S = \frac{mn}{\lambda^2} \times z \times \left(\lambda + \lambda + \lambda^2\right) = mn + \frac{2mnz}{\lambda}$$

In terms of block operations, the communication-to-computation ratio is therefore:

$$CCR_S = \frac{mn + \frac{2mnz}{\lambda}}{mnz} = \frac{1}{z} + \frac{2}{\lambda}$$

For very large matrices, where m, n and z are large, the communication-to-computation is asymptotically close to the value $\frac{2}{\sqrt{C_S}} = \sqrt{\frac{32}{8C_S}}$, which is close from the lower bound $CCR_{S_{opt}} = \sqrt{\frac{27}{8C_S}}$ derived earlier.

B.2 Distributed-caches misses

The number of distributed-cache misses achieved by our algorithm is:

$$M_D = \frac{\frac{mn}{\lambda^2} \times z \times \lambda \times p \times \left(1 + \frac{1}{p} + \frac{\lambda}{p}\right)}{p} = \frac{mnz}{p} + \frac{mnz}{p\lambda} + \frac{mnz}{\lambda}$$

In terms of block operations, the communication-to-computation ratio is therefore:

$$CCR_D = \frac{\frac{mnz}{p} + \frac{mnz}{p\lambda} + \frac{mnz}{\lambda}}{\frac{mnz}{p}} = 1 + \frac{1}{\lambda} + \frac{p}{\lambda}$$

This communication-to-computation does not depend upon the dimensions of the matrices, and therefore is the same for very large matrices. Hence, we can see that this is far from the lower bound derived earlier, which is $CCR_{D_{opt}} = \sqrt{\frac{27}{8C_D}}$.

C Minimizing the number of distributed-cache misses

In this section, we will focus on our second objective which consists to minimize the number of distributed-cache misses. In order to do so, we use the parameter μ defined earlier for the distributed-cache to store in each distributed cache a block of size $\mu \times \mu$ of C, a fraction of line (of size μ) of B and one element of A. Contrarily to the previous algorithm, the block of C will be totally computed before being written back to the shared cache and the distributed memory.

All p cores will work on different blocks of C. Thanks to constraint on the size of caches ($p \times C_D \leq C_S$), we know that the shared cache has the capacity to store all temporary data.

The $\mu \times \mu$ blocks of C are distributed among the distributed-caches in a 2-D cyclic way, because it helps reduce (and balance between A and B) the number of shared-cache misses: in this case, assuming that $\sqrt{p}$ is an integer, we load a $\sqrt{p}\mu \times \sqrt{p}\mu$ block of C in shared cache, together with a row of $\sqrt{p}\mu$ elements of B. Then, $\sqrt{p} \times \mu$ elements from a column of A are sequentially loaded in the shared cache ($\sqrt{p}$ non contiguous elements are loaded at each step), then distributed among the distributed-caches (cores in the same "row" (resp. "column") accumulate the contribution of the same (resp. different) element of A but of different (resp. the same) $\sqrt{p} \times \mu$ fraction of row from B).

Algorithm 3: Adaptation of the maximum re-use algorithm.

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$offset_j = \lfloor \frac{c-1}{\sqrt{p}} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{p\mu^2}$ **do**
  Load a new block $C[i, \ldots, i + \sqrt{p}\mu;\ j, \ldots, j + \sqrt{p}\mu]$ of C in the shared cache
  foreach *core* $c = 1 \ldots p$ *in parallel* **do**
    Load
    $C_c = C[i + offset_i \times \mu, \ldots, i + (offset_i + 1) \times \mu;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$
    in the distributed cache of core c
  for $k = 1$ *to* z **do**
    Load a row $B[k;\ j, \ldots, j + \sqrt{p}\mu]$ of B in the shared cache
    foreach *core* $c = 1 \ldots p$ *in parallel* **do**
      Load $B_c = B[k;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$ in the distributed cache of core c
      for $i' = i + offset_i \times \mu$ *to* $i + (offset_i + 1) \times \mu$ **do**
        Load the element $a = A[k;\ i']$ in the shared cache
        Load the element $a = A[k;\ i']$ in the distributed cache of core c
        Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
  foreach *core* $c = 1 \ldots p$ *in parallel* **do**
    Update block C_c in the shared cache
  Write back the block of C to the main memory

C.1 Shared-cache misses

The number of shared-cache misses is:

$$M_S = \frac{mn}{p\mu^2} \times \left(p\mu^2 + z \times 2\sqrt{p}\mu\right) = mn + \frac{2mnz}{\mu\sqrt{p}}$$

Hence, the communication-to-computation ratio is:

$$CCR_S = \frac{mn + \frac{2mnz}{\mu\sqrt{p}}}{mnz} = \frac{1}{z} + \frac{2}{\mu\sqrt{p}}$$

For very large matrices, where m, n and z are large, the communication-to-computation is asymptotically close to the value: $\frac{2}{\mu\sqrt{p}} = \sqrt{\frac{32}{8 \times \sqrt{p} C_D}}$. This is far from the lower bound $CCR_{S_{opt}} = \sqrt{\frac{27}{8C_S}}$ derived earlier since $\sqrt{p}C_D \leq pC_D \leq C_S$.

C.2 Distributed-caches misses

The number of distributed-cache misses achieved by our algorithm is:

$$M_D = \frac{\frac{mn}{p\mu^2} \times p \times \left(\mu^2 + z \times 2\mu\right)}{p} = \frac{mn}{p} + \frac{2mnz}{\mu \times p}$$

Therefore, the communication-to-computation ratio is:

$$CCR_D = \frac{\frac{mn}{p} + \frac{2mnz}{\mu \times p}}{\frac{mnz}{p}} = \frac{1}{z} + \frac{2}{\mu}$$

For very large matrices, where m, n and z are large, the communication-to-computation is asymptotically close to the value $\frac{2}{\mu} = \sqrt{\frac{32}{8C_D}}$. We can see that this is close from the lower bound $CCR_{D_{opt}} = \sqrt{\frac{27}{8C_D}}$ derived earlier.

D Minimizing data access time

Our two objectives are antagonistic, and in both previous approaches, optimizing the number of cache misses of one type leads to a large number of cache misses of the other type. Indeed, minimizing M_S ends up with a number of distributed-cache misses proportional to the common dimension of matrices A and B, and in the case of large matrices this is clearly problematic. On the other hand, focusing on M_D only is not efficient since we dramatically under-use the shared cache: a large part of it is not utilized.

This motivates us to look for a tradeoff between the latter two solutions. However, both kinds of cache misses have different costs, since bandwidths between each level of our memory architecture are different. Hence we have introduced the overall time for data movement, defined as:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

Depending on the ratio between cache speeds, this objective provides a tradeoff between both cache miss numbers.

To derive an algorithm optimizing this tradeoff, we start from algorithm presented for optimizing the shared-cache misses. Looking closer to the downside of this algorithm, which is the fact that the part of M_D due to the elements of C is proportional to the common dimension z of matrices A and B, we can see that we can reduce this amount by loading blocks of β columns (resp. of rows) of A (resp. B). This way, square blocks of C could be processed longer by the cores before being unloaded and written back in shared-cache instead of being unloaded after that every element of the column of A residing in shared-cache has been used. However, blocks of C must be smaller than before, and instead of being λ^2 blocks, they are now of size α^2 where α and β are defined under the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$.

In this case, the sketch of Algorithm 4 is the following:

1. A block of size $\alpha \times \alpha$ of C is loaded in the shared cache. Its size satisfies $p \times \mu^2 \leq \alpha^2 \leq \lambda^2$. Both extreme cases are obtained when one of σ_D and σ_S is negligible in front of the other.
2. In the shared cache, we also load a block from B, of size $\beta \times \alpha$, and a block from A of size $\alpha \times \beta$. Thus, we have $2\alpha \times \beta + \alpha^2 \leq C_D$.
3. The $\alpha \times \alpha$ block of C is split into sub-blocks of size $\mu \times \mu$ which are processed by the different cores. These sub-blocks of C are cyclicly distributed among every distributed-caches. The same holds for the block-row of B which is split into $\beta \times \mu$ block-rows and cyclicly distributed, row by row (i.e. by blocks of size $1 \times \mu$), among every distributed-caches.
4. The contribution of the corresponding β (fractions of) columns of A and β (fractions of) lines of B es added to the block of C. Then, another $\mu \times \mu$ block of C residing in shared cache is distributed among every distributed-caches, going back to step 3.
5. As soon as all elements of A and B have contributed to the $\alpha \times \alpha$ block of C, another β columns/lines from A/B are loaded in shared cache, going back to step 2.
6. Once the $\alpha \times \alpha$ block of C in shared cache is totally computed, a new one is loaded, going back to step 1.

Algorithm 4: Adaptation of the maximum re-use algorithm.

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$offset_j = \lfloor \frac{c-1}{\sqrt{p}} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{\alpha^2}$ **do**
  Load a new block $C[i, \ldots, i+\alpha;\ j, \ldots, j+\alpha]$ of C in the shared cache
  for Substep $= 1$ *to* $\frac{z}{\beta}$ **do**
    $k = 1 + (Substep - 1) \times \beta$
    Load a new block row $B[k, \ldots,\ k+\beta;\ j, \ldots, j+\alpha]$ of B in the shared cache
    Load a new block column $A[i, \ldots, i+\alpha;\ 1+(k-1)\times\beta, \ldots,\ 1+k\times\beta]$ of A in the shared cache
    foreach *core* $c = 1 \ldots p$ *in parallel* **do**
      for $subi = 1$ *to* $subi = \frac{\alpha}{\sqrt{p}\mu}$ **do**
        for $subj = 1$ *to* $subj = \frac{\alpha}{\sqrt{p}\mu}$ **do**
          Load
          $C_{\mu c} = C[i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi - 1) \times \mu, \ldots, i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi) \times \mu;\ j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj - 1) \times \mu, \ldots, j + offset_j) \times \frac{\alpha}{\sqrt{p}} + (subj) \times \mu]$ in the distributed cache of core c
          for $k' = k$ *to* $k' = k + \beta$ **do**
            Load $B_c =$
            $B[k';\ j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj-1) \times \mu, \ldots, j + offset_j \times \frac{\alpha}{\sqrt{p}} + (subj) \times \mu]$
            in the distributed cache of core c
            for $i' = i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi - 1) \times \mu$ *to*
            $i + offset_i \times \frac{\alpha}{\sqrt{p}} + (subi) \times \mu$ **do**
              Load the element $a = A[i',\ k']$ in the distributed cache of core c
              Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
          Update block $C_{\mu c}$ in the shared cache
  Write back the block of C to the main memory

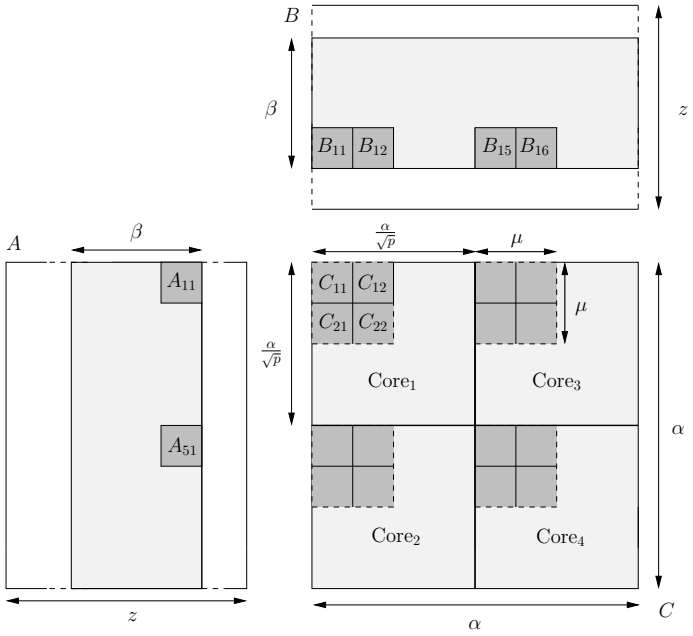

Figure 3: Data distribution of matrices A, B and C: light gray block resides in shared-cache, dark gray blocks are distributed among distributed-caches ($\alpha = 8, \mu = 2,\ p = 4$)

D.1 Shared-cache misses

The number of shared-cache misses is given by:

$$M_S = \frac{mn}{\alpha^2}\left(\alpha^2 + \frac{z}{\beta} \times 2\alpha\beta\right) = mn + \frac{2mnz}{\alpha}$$

The communication-to-computation ratio is therefore:

$$CCR_S = \frac{1}{z} + \frac{2}{\alpha}$$

For very large matrices, where m, n and z are large, the communication-to-computation is asymptotically close to the value $\sqrt{\frac{32}{8}}\frac{1}{\alpha}$ which is less close to our previous shared-cache optimized version since $\alpha \leq \lambda \approx \sqrt{C_S}$.

D.2 Distributed-caches misses

In the general case (i.e. $\alpha > \sqrt{p}\mu$), the number of distributed-cache misses achieved by our new algorithm is:

$$M_D = \frac{1}{p} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times \frac{z}{\beta} + \frac{z}{\beta} \times 2\beta\mu\right)\right] = \frac{mn}{p} \times \frac{z}{\beta} + \frac{2mnz}{p\mu}$$

The communication-to-computation ratio is therefore $\frac{1}{\beta}+\frac{2}{\mu}$, which for large matrices is asymptotically close to $\frac{1}{\beta}+\sqrt{\frac{32}{8C_D}}$. This is far from the lower bound $\sqrt{\frac{27}{8C_D}}$ derived earlier. To optimize this CCR, we could try to increase the value of β. However, increasing the parameter β implies a lower value of α, resulting in more shared-cache misses.

Remark. Note that if we are in the special case $\alpha = \sqrt{p}\mu$, we only need to load each $\mu \times \mu$ sub-block of C once, since a core is only in charge of one sub-block of C, therefore the number of distributed-caches misses becomes:

$$M_D = \frac{1}{p} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times 1 + \frac{z}{\beta} \times 2\beta\mu \right) \right] = \frac{mn}{p} + \frac{2mnz}{p\mu}$$

In this case, we come back to the distributed-cache optimized case, and the distributed CCR is close to the bound.

D.3 Data access time

With this algorithm, we get an overall data access time of:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D} = \frac{mn + \frac{2mnz}{\alpha}}{\sigma_S} + \frac{\frac{mnz}{p\beta} + \frac{2mnz}{p\mu}}{\sigma_D}$$

Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, it allows us to compute the best value for parameters α and β, depending on the ratio σ_S/σ_D. Since we work under the assumption of large matrices, the first term in mn can be neglected in front of the other terms, so basically, our problem reduces to minimizing the following expression:

$$\frac{2}{\sigma_S \alpha} + \frac{1}{p\sigma_D \beta} + \frac{2}{p\sigma_D \mu}$$

The constraint $2\beta\alpha + \alpha^2 \leq C_S$ enables us to express β as a function of α and C_S. As a matter of a fact, we have:

$$\beta \leq \frac{C_S - \alpha^2}{2\alpha}$$

Hence, the objective function becomes:

$$F(\alpha) = \frac{2}{\sigma_S \alpha} + \frac{2\alpha}{p\sigma_D (C_S - \alpha^2)}$$

Note that we have removed the term $\frac{2}{p\sigma_D \mu}$ because it only depends on μ and therefore is minimal when $\mu = \lfloor \sqrt{C_S - 3/4} - 1/2 \rfloor$, i.e. its largest possible value.

The derivative $F'(\alpha)$ is:

$$F'(\alpha) = \frac{2(C_S + \alpha^2)}{p\sigma_D (C_S - \alpha^2)^2} - \frac{2}{\sigma_S \alpha^2}$$

And therefore, the root is

$$\alpha_{\text{num}} = \sqrt{C_S \frac{2p\sigma_D + \sigma_S \left(1 - \sqrt{1 + \frac{8p\sigma_D}{\sigma_S}} \right)}{(2p\sigma_D - \sigma_S)}}.$$

Altogether, the best parameters values in order to minimize the total data access time in the case of square blocks are:

$$\begin{cases} \alpha = \min\left(\alpha_{\max}, \max\left(\sqrt{p}\mu, \alpha_{\text{num}}\right)\right) \\ \\ \beta = \max\left(\left\lfloor \frac{C_S - \alpha^2}{2\alpha} \right\rfloor, 1\right) \end{cases}$$

where:

$$\alpha_{\max} = \sqrt{C_S + 1} - 1$$

The parameter α depends on the values of bandwidths σ_S and σ_D. In both extreme cases, it will take a particular value indicating that the tradeoff algorithm will follow the sketch of either the shared-cache optimized version or the distributed-caches one:

- When the bandwidth σ_D is significantly higher than σ_S, the parameter α becomes:

  $$\alpha_{\text{num}} \approx \sqrt{C_S} \quad \Longrightarrow \quad \alpha = \alpha_{\max}, \beta = 1$$

  which means that the tradeoff algorithm chooses a shared-cache optimized version of the Multicore Maximum Re-use Algorithm whenever distributed caches are significantly faster than the shared cache.

- On the contrary, when the bandwidth σ_S is significantly higher than σ_D, α becomes:

  $$\alpha_{\text{num}} \approx \sqrt{C_S \frac{2p\sigma_D}{-\sigma_S}} \quad \Longrightarrow \quad \alpha = \sqrt{p}\mu, \beta = 1$$

  which means that the tradeoff algorithm chooses the distributed optimized version of the Multicore Maximum Re-use Algorithm whenever distributed caches are significantly slower than the shared cache, although it does not seem to be a realistic case.

E Related work

In [5], the authors introduce the *ideal-cache model* which is a fundamental building block of our study. They also present the cache-oblivious paradigm, which aims to provide asymptotically optimal "cache-unaware" algorithms, and introduce some of them. However, the introduced model is not adapted to multicore processors, but instead focuses only on single-core processors. They show that in that case, a hierarchy or cache could be could be "clustered" and thus, the memory architecture could be restricted to the ideal cache-model. Nevertheless, this results does not hold for multicore processors since caches are now heterogeneous (private and shared). Another interesting contribution is the proof that the ideal cache-model could be simulated in an efficient way with a LRU replacement policy.

In [3], the authors present a general *multicore-cache model* aiming at modeling the cache architecture of multicore processors; they study divide-and-conquer algorithms for several problems on this model, and they introduce an online scheduler asymptotically matching the sequential cache complexity for both shared and private (or distributed) caches misses, while offering full parallel speedup. Their asymptotic counting of cache misses proves their algorithms to be very efficient. Contrarily to our study, their assume that algorithms are oblivious of cache sizes. Moreover, they only focus on a particular set of algorithms that are not in the scope of our study.

In [1], the authors introduce a theoretical model for multicore processors intended to be used to analyze the complexity of algorithms on these new platforms. They also describe a framework called SWARM that aims at providing an open-source library for developing software on multicore architecture. They intend to minimize the dominant term of their model, which is either the time complexity, or the time spent to load data from main memory to shared cache, or the time spent into synchronization between cores. However, they do not explicitly use the notion of cache misses; instead, they focus on the number of blocks transferred between shared cache and main memory to express the memory complexity of studied algorithms: distributed-caches are not considered at all in their analysis.

In [6], the authors introduce many lower bounds on communication volume for the standard algorithms for matrix multiplication. The scope of their work ranges from one processor and its main memory to several distributed memory processors. They also provide a lower bound for a processor having a fast cache and a large slow memory. However, this work does not provide any results for multicore processors and their peculiar cache hierarchy.

In [8], the authors introduce the Maximum Re-use Algorithm, which is a matrix product algorithm for a master-slave platform. They extend the lower bound introduced in [6] to their context, and show that their Maximum Re-use Algorithm is close from this bound for large matrices. However, they do not consider caches nor multicore processors; instead they focus only on single-core processors communicating with their own memory.