Complexity analysis and performance evaluation 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 RRLIP2009-09

Abstract

The multicore revolution is underway, bringing new chips introducing more complex memory architectures. Classical algorithms must be revisited in order to take the 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 tradeoff between shared and distributed caches. Comprehensive simulation results confirm the analytical performance predictions and fully establish the practical significance of our new algorithms.

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-one 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 cache to distributed caches. To increase performance, 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 [8] (the shared cache being the disk) and that of master-slave implementations with limited memory [7] (the shared cache being the master's memory). The latter paper [7] presents the Maximum Reuse 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.

2 Problem statement

2.1 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

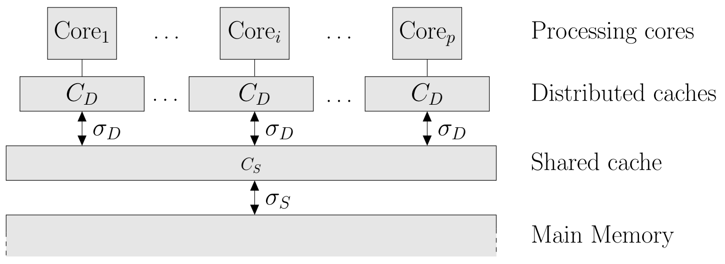


Figure 1: Multicore architecture model.

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 are supposed to be *inclusive*, which 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 assume in the following 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 $2L$ implementing a classical *LRU* replacement policy.

In the following, our objective is twofold: (i) minimize the number of cache misses during the computation of 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, and that coherency mechanisms' impact on performance can be neglected.

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 ranges from 32 to 100 on most platforms.

2.2 Communication volume

The key point to performance in a multicore architecture is efficient data reuse. A simple way to assess data locality is to count and 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 [8], Irony, Toledo and Tiskin propose a lower bound on the number of communications needed to perform a matrix product. We have extended this study to our hierarchical cache architecture. In all the following, 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 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 when these Z cache misses occurs.
- Let $comp(c)$ be the amount of computation done by core c

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

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

2.3.1 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:

$$CCR \geq \frac{Z}{k \times Z\sqrt{Z}} = \sqrt{\frac{27}{8Z}}$$

2.3.2 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 $Z = C_S$ and $K = \sum_c comp(c)$, we have a lower bound on the communication-to-computation ratio for shared-cache misses:

$$CCR_S = \frac{M_S}{K} = \frac{M_S}{\sum_c comp(c)} \geq \sqrt{\frac{27}{8C_S}}$$

2.3.3 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

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

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

$$CCR_D = \frac{1}{p}\sum_{c=1}^{p}\left(\frac{M_{Dc}}{comp(c)}\right) \geq \sqrt{\frac{27}{8C_D}}$$

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

2.3.4 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:

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

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

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

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

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

3 Algorithms

In the out-of-core algorithm of [8], the three matrices A, B and C are equally accessed throughout time. This naturally leads to allocating one third of the available memory to each matrix. This algorithm has a communication-to-computation ratio of $O\left(\frac{mnz}{\sqrt{M}}\right)$ for a memory of size M but it does not use the memory optimally. The Maximum Reuse Algorithm [7] proposes a more efficient memory allocation: it splits the available memory into $1 + \mu + \mu^2$ blocks, storing 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 compute $C_{i_1 \ldots i_2, j_1 \ldots j_2} += A_{i,j} \times B_{i_1 \ldots i_2, j}$. 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, the communication-to-computation ratio is $\frac{2}{\sqrt{M}}$ for large matrices.

To adapt the Maximum Reuse 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. The main idea is to design a "data-thrifty" algorithm that reuses matrix elements as much as possible and loads each required data only once in a given loop. Since the outermost loop is prevalent, we load the largest possible square block of data in this loop, and adjust the size of the other blocks for the inner loops, according to the objective (shared-cache, distributed-cache, tradeoff) of the algorithm. We define two parameters 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 shared-cache misses

Algorithm 1: Adaptation of the Maximum Reuse Algorithm minimizing the number of shared misses M_S

```
for Step = 1 to $\frac{m \times n}{\lambda^2}$ do
    Load a new block $C[i, \dots, i+\lambda;\ j, \dots, j+\lambda]$ of $C$ in the shared cache
    for $k = 1$ to $z$ do
        Load a row $B[k;\ j, \dots, j+\lambda]$ of $B$ in the shared cache
        for $i' = i$ to $i + \lambda$ do
            Load the element $a = A[k;\ i']$ in the shared cache
            foreach core $c = 1 \dots p$ in parallel do
                Load the element $a = A[k;\ i']$ in the distributed cache of core $c$
                for $j' = j + (c-1) \times \frac{\lambda}{p}$ to $j + c \times \frac{\lambda}{p}$ do
                    Load $B_c = B[k;\ j']$ in the distributed cache of core $c$
                    Load $C_c = C[i';\ j']$ 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
```

To minimize the number of shared-cache misses, we adapt the Maximum Reuse Algorithm with parameter λ. 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, each row of C_{block} is divided into sub-rows of $\frac{\lambda}{p}$ elements, which are then distributed element per element together with corresponding element of B and one element of A, updated by the different cores, and written back in shared cache.

As soon as C_{block} is completely updated, it is written back in main memory and a new C_{block} is loaded. This is described in details in Algorithm 1, and the memory layout is depicted in Figure 2.

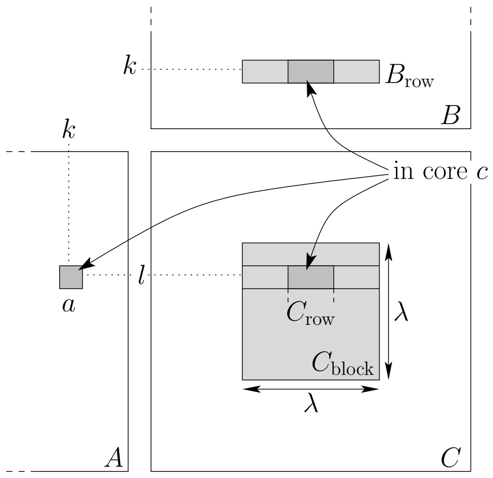


Figure 2: Data layout for Algorithm 1.

Note that the space required in each distributed-cache to process 1 bloc of q^2 elements of C is $1 + 1 + 1$. 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 . Let , S_D be the size (in matrix coefficient) of each distributed cache (remember that C_D is expressed in blocks) Distributed cache must obviously be large enough to perform the matrix product, that is to say that:

$$3 \leq C_D$$
$$3q^2 \leq S_D$$

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

- *Shared-cache misses*

  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 number of shared cache misses of:

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

  The communication-to-computation ratio is therefore:

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

  For large matrices, this leads to a shared-cache CCR of $2/\lambda$, which is close to the lower bound derived earlier.

- *Distributed-caches misses*

  In this algorithm, each block of C is sequentially updated z times by rows of λ elements, each row is distributed elements per elements, thus requiring $\frac{\lambda}{p}$ steps. Therefore, each distributed cache holds one element of C. This therefore results in $\frac{mnz}{p}$ cache misses. For the computation of each block of size λ^2, $z \times \frac{\lambda}{p}$ elements are loaded from B in each distributed cache, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total number of shared cache misses of:

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

  The communication-to-computation ratio is therefore:

  $$\begin{aligned} CCR_D &= \frac{\frac{2mnz}{p} + \frac{mnz}{\lambda}}{\frac{mnz}{p}} \\ &= 2 + \frac{p}{\lambda} \end{aligned}$$

  This communication-to-computation does not depend upon the dimensions of the matrices, and is the same for large matrices. Moreover, it is far from the lower bound on distributed cache misses.

3.2 Minimizing 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 $p \times C_D \leq C_S$, we know that the shared cache has the capacity to store all necessary 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 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 2: Adaptation of the Maximum Reuse Algorithm minimizing the number of distributed misses M_D

```
$\mathit{offset}_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$\mathit{offset}_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{\sqrt{p}} \rfloor$
for Step = 1 to $\frac{m \times n}{p\mu^2}$ do
    Load a new block $C[i,\dots,i+\sqrt{p}\mu;\ j,\dots,j+\sqrt{p}\mu]$ of $C$ in the shared cache
    foreach core $c = 1\dots p$ in parallel do
        Load
        $C_c = C[i+\mathit{offset}_i\times\mu,\dots,i+(\mathit{offset}_i+1)\times\mu;\ j+\mathit{offset}_j\times\mu,\dots,j+(\mathit{offset}_j+1)\times\mu]$
        in the distributed cache of core $c$
    for $k = 1$ to $z$ do
        Load a row $B[k;\ j,\dots,j+\sqrt{p}\mu]$ of $B$ in the shared cache
        foreach core $c = 1\dots p$ in parallel do
            Load $B_c = B[k;\ j+\mathit{offset}_j\times\mu,\dots,j+(\mathit{offset}_j+1)\times\mu]$ in the distributed
            cache of core $c$
            for $i' = i+\mathit{offset}_i\times\mu$ to $i+(\mathit{offset}_i+1)\times\mu$ 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 $c = 1\dots p$ in parallel do
        Update block $C_c$ in the shared cache
    Write back the block of $C$ to the main memory
```

- *Shared-cache misses*

  The number of shared-cache misses is:

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

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 large matrices, the communication-to-computation is $\frac{2}{\mu\sqrt{p}} = \sqrt{\frac{32}{8\times\sqrt{p}C_D}}$. This is far from the lower bound since $\sqrt{p}C_D \leq pC_D \leq C_S$.

- *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 large matrices, the communication-to-computation is asymptotically close to the value $\frac{2}{\mu} = \sqrt{\frac{32}{8C_D}}$, which is close to the lower bound $\sqrt{\frac{27}{8C_D}}$.

3.3 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 3 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.

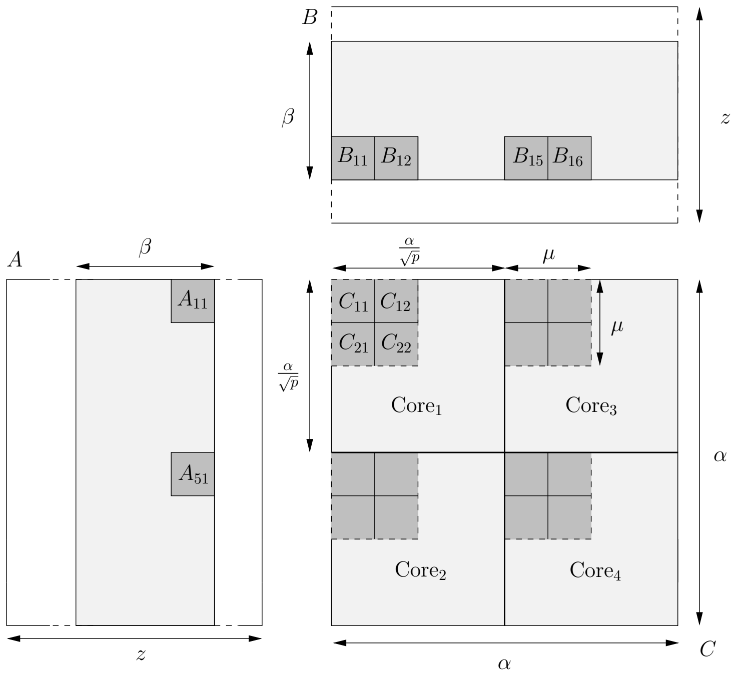


Figure 3: Data distribution of matrices A, B and C: light gray block resides in shared-cache, dark gray blocks are distributed among distributed-caches ($\alpha = 8, \mu = 2,\ p = 4$)

- *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:

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

Algorithm 3: Adaptation of the Maximum Reuse Algorithm minimizing the overall time T_{data}

```
$\mathit{offset}_i = (\text{My\_Core\_Num}() - 1) \pmod{\sqrt{p}}$
$\mathit{offset}_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{\sqrt{p}} \rfloor$
for Step $= 1$ to $\frac{m \times n}{\alpha^2}$ do
    Load a new block $C[i, \dots, i+\alpha;\ j, \dots, j+\alpha]$ of $C$ in the shared cache
    for Substep $= 1$ to $\frac{z}{\beta}$ do
        $k = 1 + (\mathit{Substep} - 1) \times \beta$
        Load a new block row $B[k, \dots,\ k+\beta;\ j, \dots, j+\alpha]$ of $B$ in the shared cache
        Load a new block column $A[i, \dots, i+\alpha;\ 1+(k-1)\times\beta, \dots,\ 1+k\times\beta]$ of $A$ in
        the shared cache
        foreach core $c = 1 \dots p$ in parallel do
            for $\mathit{subi} = 1$ to $\mathit{subi} = \frac{\alpha}{\sqrt{p}\mu}$ do
                for $\mathit{subj} = 1$ to $\mathit{subj} = \frac{\alpha}{\sqrt{p}\mu}$ do
                    Load
                    $C_{\mu c} = C[i + \mathit{offset}_i \times \frac{\alpha}{\sqrt{p}} + (\mathit{subi}-1)\times\mu, \dots, i + \mathit{offset}_i \times \frac{\alpha}{\sqrt{p}} + (\mathit{subi}) \times$
                    $\mu;\ j + \mathit{offset}_j \times \frac{\alpha}{\sqrt{p}} + (\mathit{subj}-1)\times\mu, \dots, j + \mathit{offset}_j) \times \frac{\alpha}{\sqrt{p}} + (\mathit{subj}) \times \mu]$
                    in the distributed cache of core $c$
                    for $k' = k$ to $k' = k + \beta$ do
                        Load $B_c =$
                        $B[k';\ j + \mathit{offset}_j \times \frac{\alpha}{\sqrt{p}} + (\mathit{subj}-1)\times\mu, \dots, j + \mathit{offset}_j \times \frac{\alpha}{\sqrt{p}} + (\mathit{subj})\times\mu]$
                        in the distributed cache of core $c$
                        for $i' = i + \mathit{offset}_i \times \frac{\alpha}{\sqrt{p}} + (\mathit{subi}-1)\times\mu$ to
                        $i + \mathit{offset}_i \times \frac{\alpha}{\sqrt{p}} + (\mathit{subi}) \times \mu$ 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
```

For large matrices, the communication-to-computation is asymptotically close to $\sqrt{\frac{32}{8}}\frac{1}{\alpha}$ which is farther from the lower bound than the shared-cache optimized version since $\alpha \le \lambda \approx \sqrt{C_S}$.

- *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:

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

  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:

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

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

- *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 $2\alpha \times \beta + \alpha^2 \le C_D$, 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 $2\beta\alpha + \alpha^2 \le C_S$ enables us to express β as a function of α and C_S. As a matter of a fact, we have:

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

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

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

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

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

where:

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

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 tradeoff algorithm will follow the sketch of either shared-cache optimized version or distributed-caches one:

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

4 Simulation results

We have presented three algorithms minimizing different objectives (shared cache misses, distributed cache misses and overall time spent in data movement) and provided a theoretical analysis of their performance. However, our simplified multicore model makes some assumptions that are not realistic on a real hardware platform. In particular it uses an ideal and omniscient data replacement policy instead of a classical *LRU* policy. This led us to design a multicore cache simulator and implement all our algorithms, as well as the outer-product [2] and Toledo [8] algorithms, using different cache policies. The goal is to experimentally assess the impact of the policies on the actual performance of the algorithms, and to measure the gap between the theoretical prediction and the observed behavior. The main motivation behind the choice of a simulator instead of a real hardware platform resides in commodity reasons: simulation enables to obtain desired results faster and allows to easily modify multicore processor parameters (cache sizes, number of cores, bandwidths, . . .).

4.1 Settings

The driving feature of our simulator was simplicity. It implements the cache hierarchy of our model, and basically counts the number of cache misses in each cache level. It offers two data replacement policies, *LRU* (Least Recently Used) and *IDEAL*. In the *LRU* mode, read and write operations are made at the distributed cache level (top of hierarchy); if a miss occurs, operations are propagated throughout the hierarchy until a cache hit happens. In the *IDEAL* mode, the user manually decides which data needs to be loaded/unloaded in a given cache; I/O operations are not propagated throughout the hierarchy in case of a cache miss: it is the user responsibility to guarantee that a given data is present in every caches below the target cache.

We have implemented two reference algorithms: (i) *Outer Product*, the algorithm in [2], for which we organize cores as a (virtual) processor torus and distribute square blocks of data elements to be updated among them; and (ii) *Equal*, an algorithm inspired by [8], which uses a simple equal-size memory scheme: one third of distributed caches is equally allocated to each loaded matrix sub-block. In fact, the algorithm in [8] deals with a single cache level, hence we decline it in two versions, *Shared Equal* for shared cache optimization, and *Distributed Equal* for distributed cache optimization. We have also implemented the three versions of the Multicore Maximum Reuse Algorithm:

- *Shared Opt.*, the version to minimize the number of shared caches misses M_S
- *Distributed Opt.*, the version to minimize the number of distributed cache misses M_D
- *Tradeoff*, the version to minimize the data access time T_{data}.

In the experiments, we simulated a "realistic" quad-core processor with 8MB of shared cache and four distributed caches of size 256KB dedicated to both data and instruction. We assume that two-thirds of the distributed caches are dedicated to data, and one-third for the instructions. Results using a more pessimistic repartition of one half for data and one-half for instructions are also given. Recall that square blocks of matrix coefficients have size $q \times q$. Depending on the chosen unit block size, caches sizes communicated to our algorithms will thus be the following:

- For $q = 32$: we derive $C_S = 977$ and $C_D = 21$ (or $C_D = 16$ for the pessimistic assumption)

- For $q = 64$: $C_S = 245$ & $C_D = 6$ (or $C_D = 4$)
- Finally, for $q = 80$: $C_S = 157$ & $C_D = 4$ (or $C_D = 3$).

4.2 *LRU* vs *IDEAL*

Here we assess the impact of the data replacement policy on the number of shared cache misses and on the performance achieved by the algorithm. Figure 4 shows the total number of shared cache misses for Shared Opt., in function of the matrix dimension. Figure 5 is the counterpart of Figure 4 but for Distributed Opt. . The same holds for Figure 6 and Tradeoff.

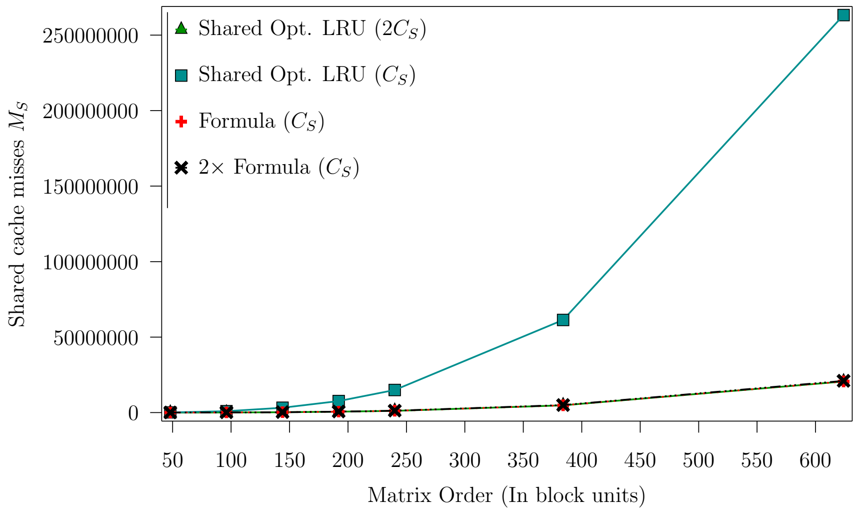


Figure 4: Impact of *LRU* policy on the number of shared cache misses M_S of Shared Opt. with $C_S = 977$

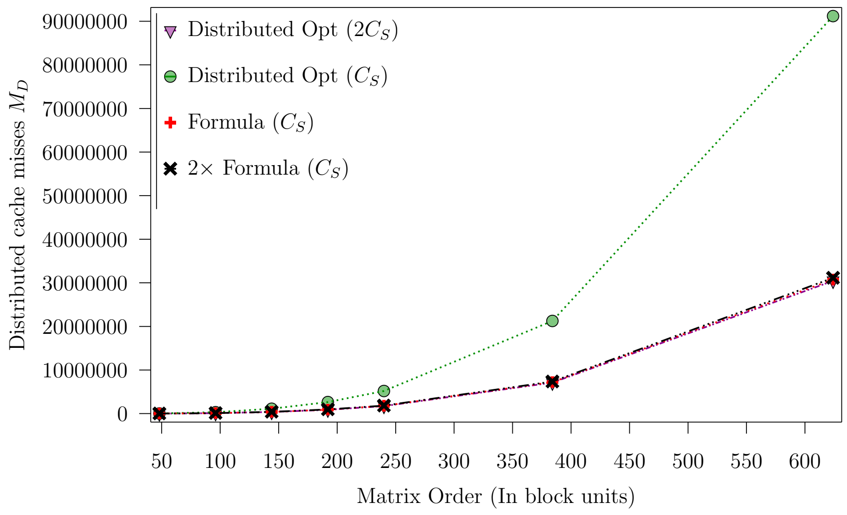


Figure 5: Impact of *LRU* policy on the number of distributed cache misses M_D of Distributed Opt. with $C_D = 21$

While *LRU* (C_S) (the *LRU* policy with a cache of size C_S) achieves significantly more cache-misses than predicted by the theoretical formula, *LRU* ($2C_S$) is quite close and always achieve less than twice the number of cache misses predicted by the theoretical formula, thereby

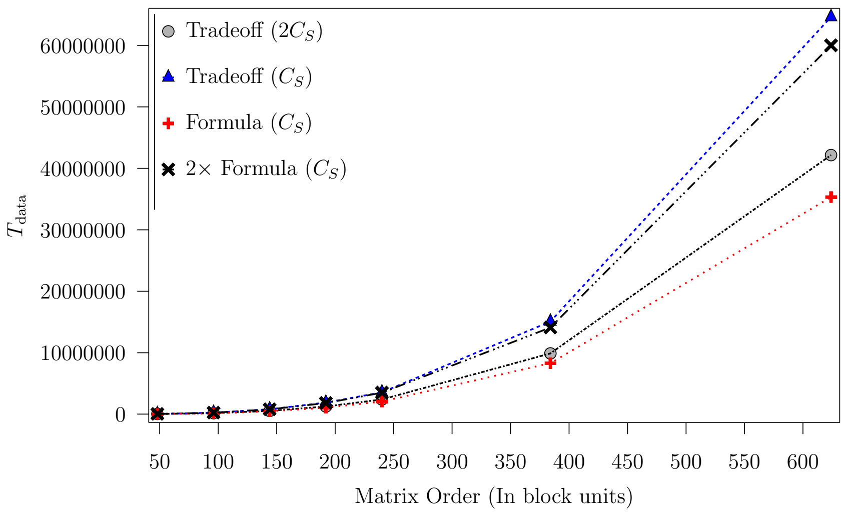


Figure 6: Impact of *LRU* policy on T_{data} of Tradeoff with $C_S = 977$ and $C_D = 21$

experimentally validating the prediction of [5]. Furthermore, similar results are obtained for Shared Equal. and Distributed Equal. Note that Outer Product is insensitive to cache policies, since it is not focusing on cache usage.

This leads us to run our tests using the following two simulation settings:

- The **IDEAL** setting, which corresponds to the use of the omniscient *IDEAL* data replacement policy assumed in the theoretical model. It relies on the *IDEAL* mode of the simulator and deckares entire cache sizes (C_S and/or C_D) to the algorithms

- The **LRU-50** setting, which relies on a *LRU* cache data replacement policy, but declares only one half of cache sizes to the algorithms. The other half is thus used by the *LRU* policy as kind of an automatic prefetching buffer.

4.3 Performance

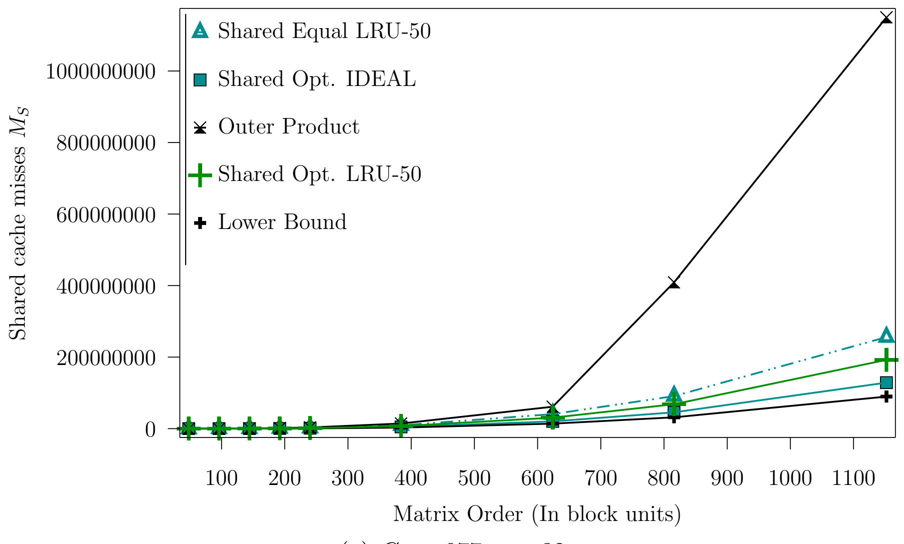
4.3.1 Shared Opt.

Figure 7 depicts the number of shared cache misses achieved by Shared Opt versions **LRU-50** and **IDEAL**, in comparison with Outer Product, Shared Equal and the lower bound $m^3\sqrt{\frac{27}{8C_S}}$, according to the matrix dimension m. For every block sizes q, we see that Shared Opt. performs significantly better than Outer Product and Shared Equal for the **LRU-50** setting. Under the **IDEAL** setting, it is closer to the lower bound, but this latter setting is not realistic.

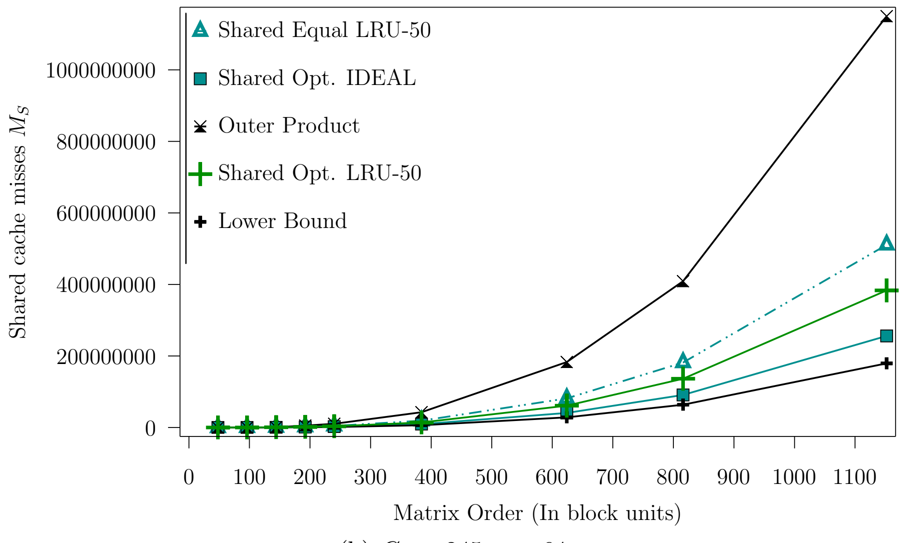
4.3.2 Distributed Opt.

Figure 8 is the counterpart of Figure 7 for distributed caches. On Figure 8(a) and Figure 8(b), we similarly see that Distributed Opt. performs significantly better than Outer Product and Distributed Equal for the **LRU-50** setting. Under the **IDEAL** setting, it is very close to the lower bound $\frac{m^3}{p}\sqrt{\frac{27}{8C_D}}$.

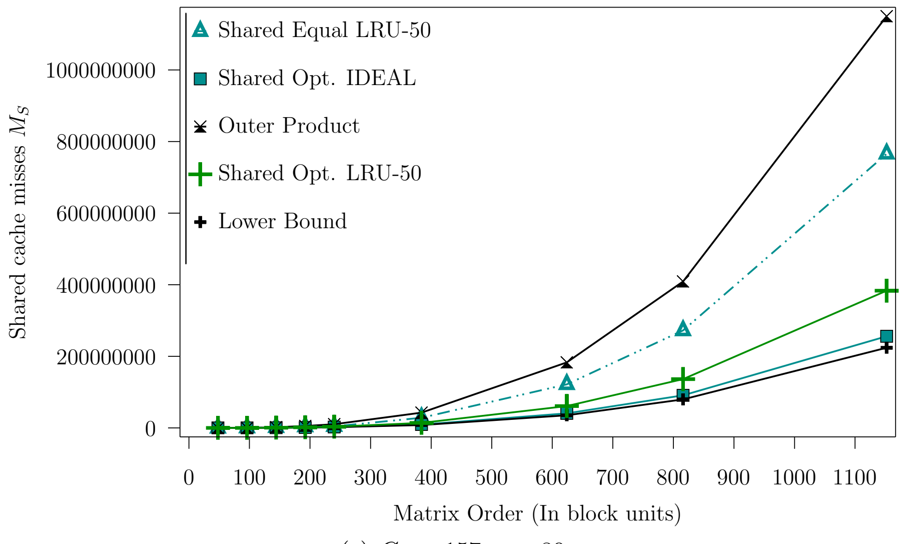
However, if we increase the size of the unit block, say $q = 64$ instead of 32, Distributed Opt. has not enough space in memory to perform better than Outer Product and Distributed Equal: as a matter of a fact, in that case $\mu = 1$. This is shown on Figure 8(c).



(a) $C_S = 977$, $q = 32$

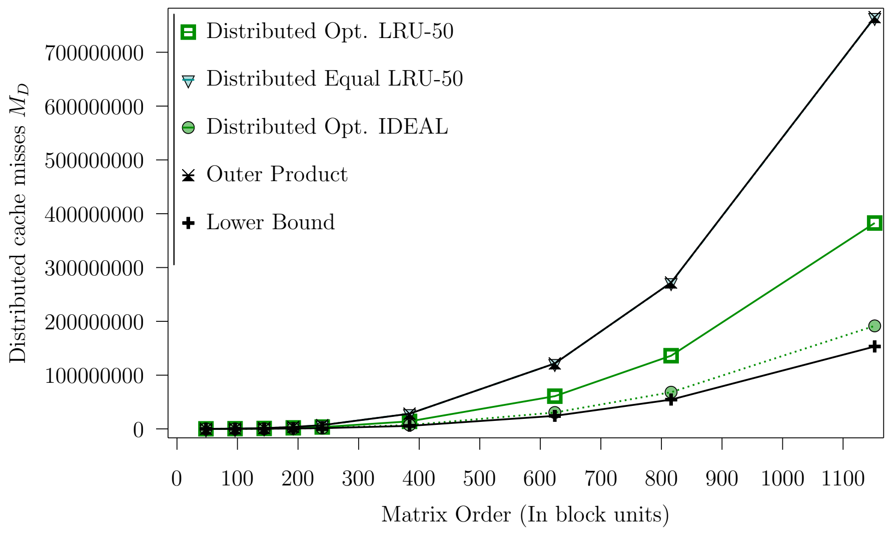


(b) $C_S = 245$, $q = 64$

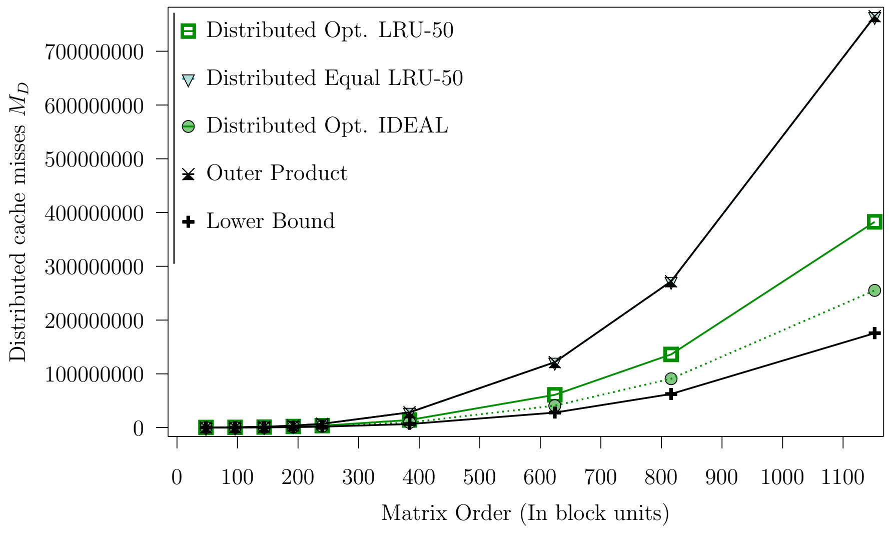


(c) $C_S = 157$, $q = 80$

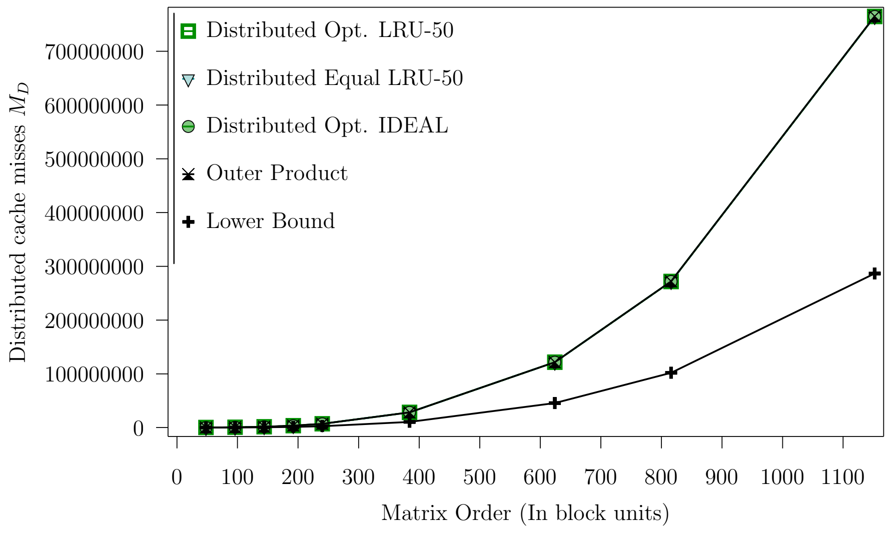
Figure 7: Shared cache misses M_S in function of matrix order.



(a) $C_D = 21$: $q = 32$, data occupy one half of distributed cache



(b) $C_D = 16$: $q = 32$, data occupy two thirds of distributed cache



(c) $C_D = 6$: $q = 64$, data occupy one half of distributed cache

Figure 8: Distributed cache misses M_D in function of matrix order.

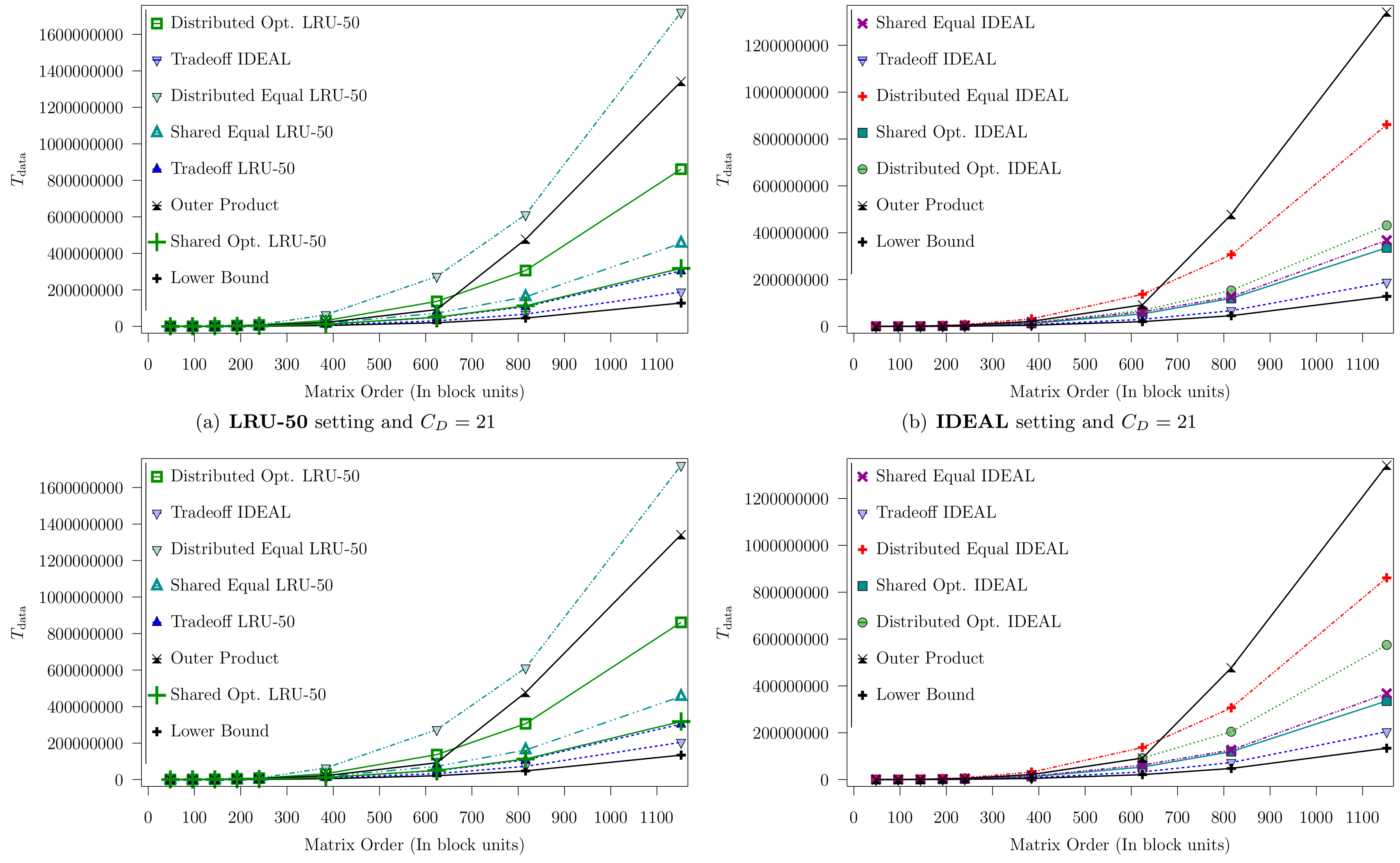
Altogether, unit block of size $q = 64$ or larger is not a relevant choice for Distributed Opt., but with $q = 32$, it will always outperform other algorithms in terms of distributed cache misses.

4.3.3 Tradeoff

The overall time spent in data movement T_{data} of all six **LRU-50** algorithms is shown for each cache configuration on Figure 9, Figure 10 and Figure 11. Using $q = 32$ and **LRU-50** setting, as shown on Figures 9(a) and 9(c) , Tradeoff offers the best overall performance, although Shared Opt. is very close. Moreover, looking at Figures 9(b) and 9(d), we see that Tradeoff outperforms even more significantly other algorithms with the **IDEAL** setting.

However, using $q = 64$ and **LRU-50** setting, as shown on Figures 10(a) and 10(c) , Tradeoff only offers the best overall performance under the pessimistic distributed-cache usage assumption (data can only occupy one half of distributed caches), although Shared Opt. is once again very close. If we use the loosened cache usage constraint, Shared Opt. takes the lead over Tradeoff. This is due to the fact that the LRU policy with a larger distributed-cache benefits more to Shared Opt. (which is unaware of distributed caches) than to Tradeoff. This is confirmed by Figures 10(b) and 10(d), on which we see that Tradeoff and Shared Opt are tie with the **IDEAL** setting.

Finally, using $q = 80$ with **LRU-50** setting, as depicted on Figures 11(a) and 11(c) , Tradeoff never ranks better than Shared Opt under both cache usage assumptions. This trend is probably due to the artificial constraints set on cache-related parameters: for instance we require that α divides m and is a multiple of both $\sqrt{p}$ and μ. In the implementations, parameters λ and α can be significantly lower than their optimal numerical value. This is confirmed by Figures 11(b) and 11(d), on which we see that Shared Opt. significantly outperforms Tradeoff even using the **IDEAL** setting.



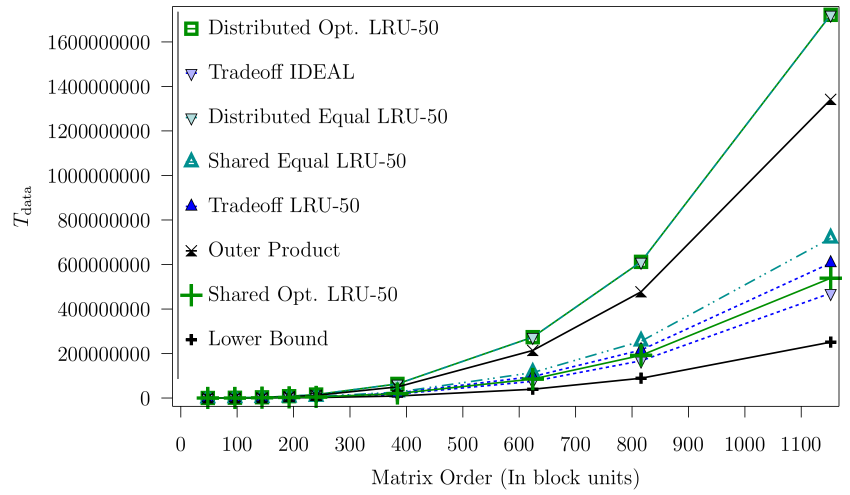
(a) **LRU-50** setting and $C_D = 21$

(b) **IDEAL** setting and $C_D = 21$

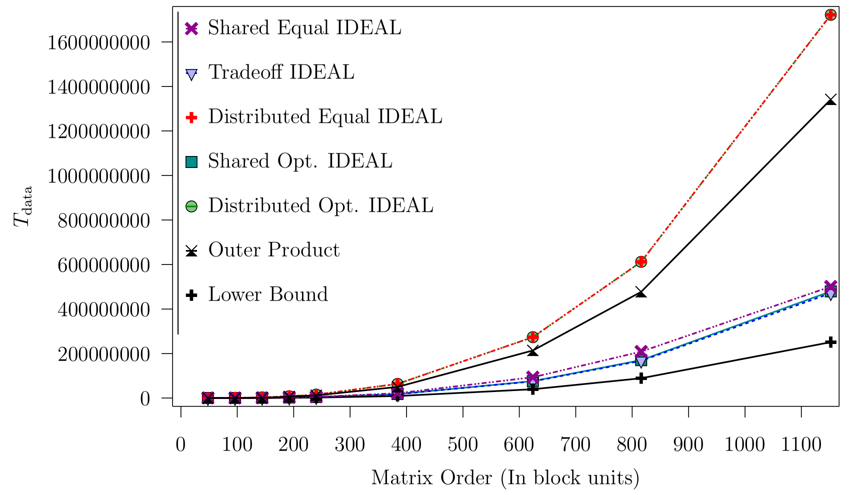
(c) **LRU-50** setting and $C_D = 16$

(d) **IDEAL** setting and $C_D = 16$

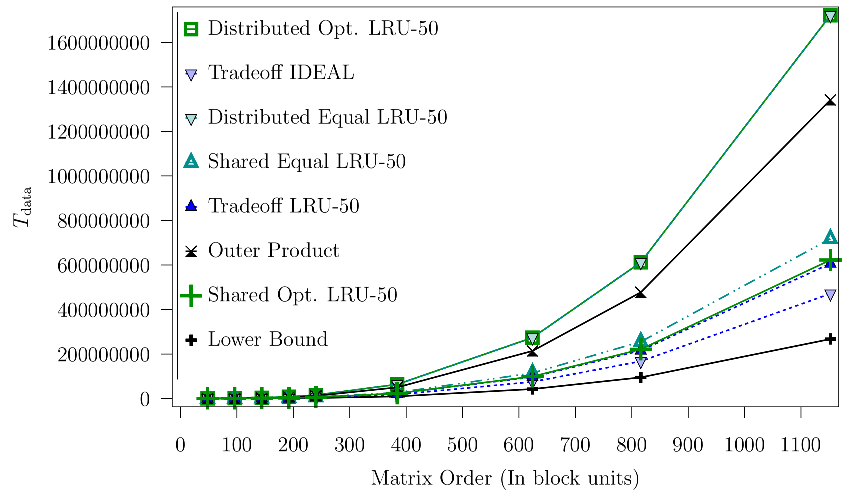
Figure 9: Overall data time T_{data}, $C_S = 977$.



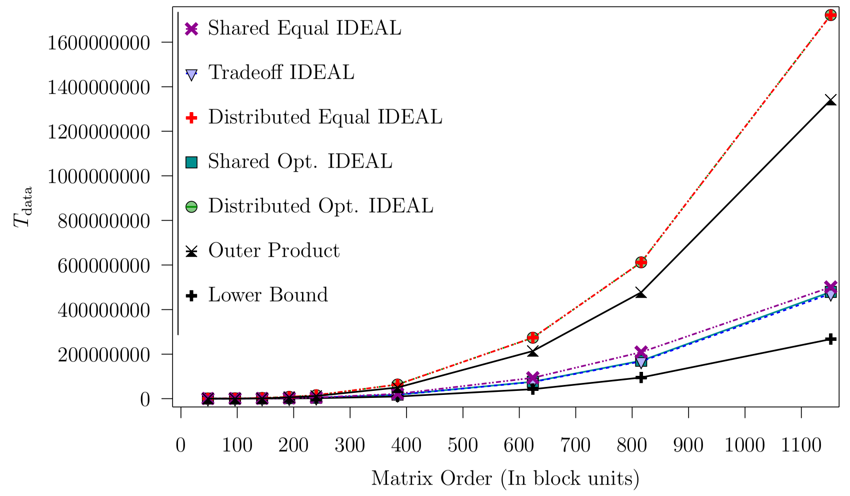
(a) **LRU-50** setting and $C_D = 6$



(b) **IDEAL** setting and $C_D = 6$

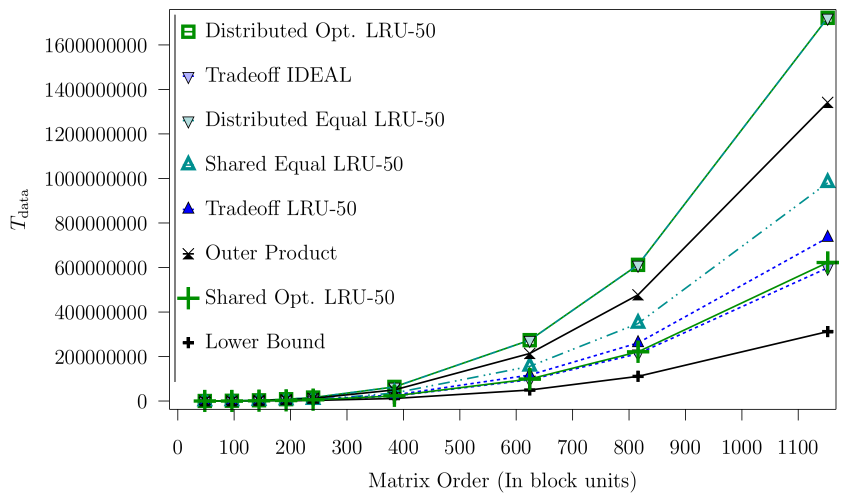


(c) **LRU-50** setting and $C_D = 4$

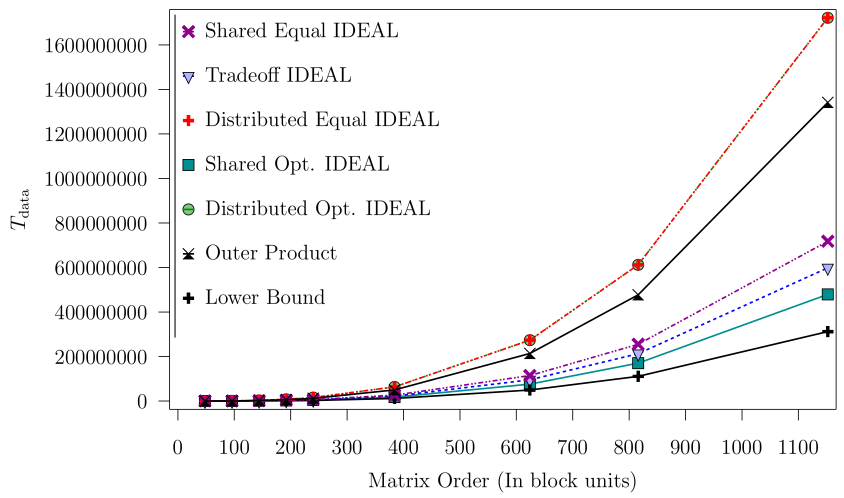


(d) **IDEAL** setting and $C_D = 4$

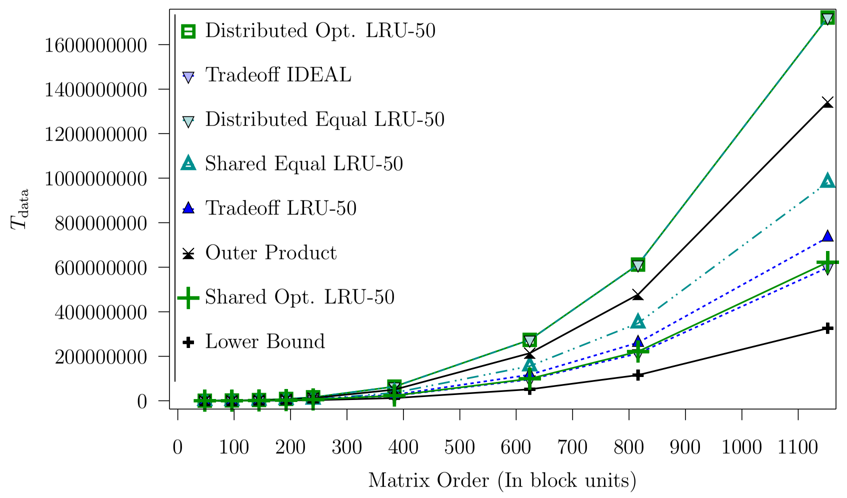
Figure 10: Overall data time T_{data}, $C_S = 245$.



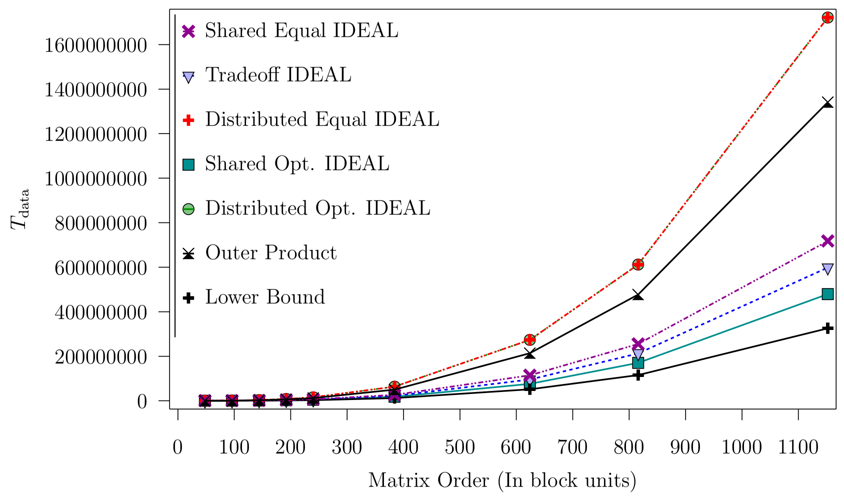
(a) **LRU-50** setting and $C_D = 4$



(b) **IDEAL** setting and $C_D = 4$



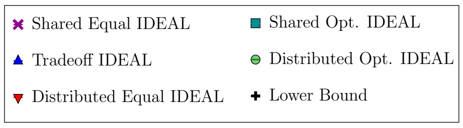
(c) **LRU-50** setting and $C_D = 3$

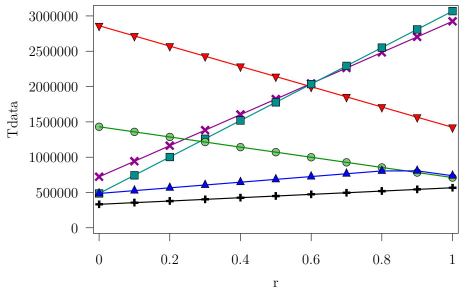


(d) **IDEAL** setting and $C_D = 3$

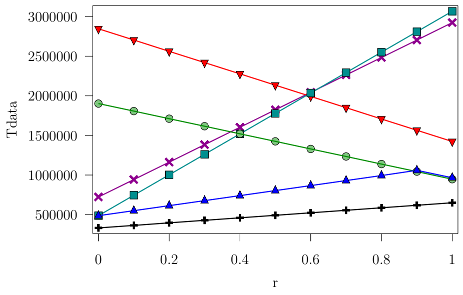
Figure 11: Overall data time T_{data}, $C_S = 157$.

We also run another experiment to assess the impact of cache bandwidths on T_{data}. We introduce the parameter r defined as: $r = \frac{\sigma_S}{\sigma_D + \sigma_S}$. Figure 12 reports results for square matrices of size $m = 384$ and the several caches configurations used before.

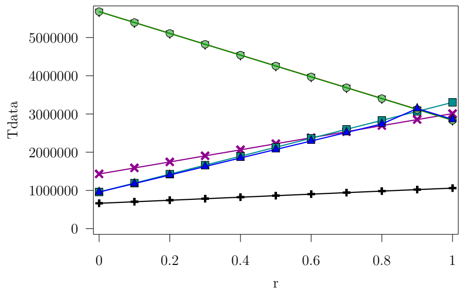




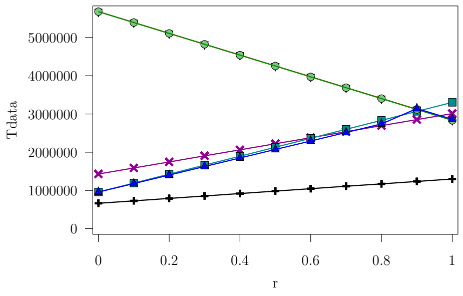
(a) $C_S = 977$ and $C_D = 21$



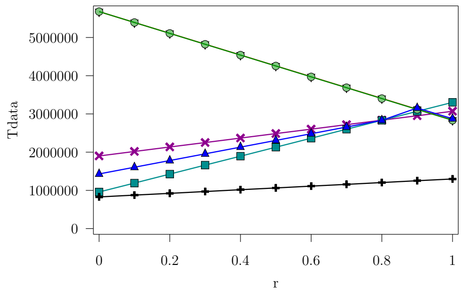
(b) $C_S = 977$ and $C_D = 16$



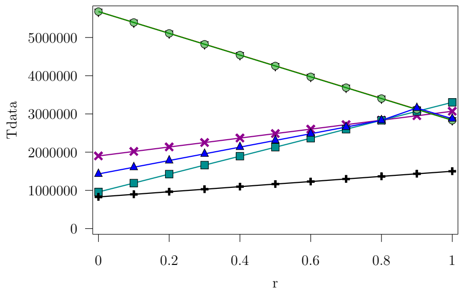
(c) $C_S = 245$ and $C_D = 6$



(d) $C_S = 245$ and $C_D = 4$



(e) $C_S = 157$ and $C_D = 4$



(f) $C_S = 157$ and $C_D = 3$

Figure 12: Cache bandwidth impact on T_{data} in function of r, the ratio between σ_S and σ_D. Results are given for a square matrix of dimension $m = 384$.

With $q = 32$ we see that Tradeoff performs best, and still offers the best performance even after distributed misses have become predominant under both optimistic and pessimistic distributed-caches usage assumptions. When the latter event occurs, plots cross over: Shared Opt. and Distributed Opt. achieve the same T_{data}. We also point out that when $r = 0$, Tradeoff achieves almost the same T_{data} than Shared Opt., while when $r = 1$, it ties Distributed Opt.

However, with $q = 64$ and $q = 80$ Tradeoff does not offer the lowest T_{data}, Shared Opt. ties or outperforms while shared cache misses are predominant under both distributed-caches assumptions. This is probably because our implementation suffers from the rounding of cache parameters: parameters α and β cannot be adjusted adequately.

5 Related work

Algorithms– In [8], the authors introduce several lower bounds on the communication volume for standard matrix multiplication algorithms. 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. In [7], the authors introduce the Maximum Reuse Algorithm, a matrix product algorithm for master-slave platforms. They improve the lower bound introduced in [8], and show that their algorithm is close to this bound for large matrices. However, neither [8] nor [7] deals with multicore processors and the additional level of cache hierarchy that they imply.

Models– The *ideal-cache* model is presented in [5], together with the cache-oblivious paradigm, which aims at provide asymptotically optimal "cache-unaware" algorithms. A key contribution is the proof that the ideal cache-model can efficiently be simulated with a LRU replacement policy. However, the model only focuses on single-core processors. It is extended in [3] for multicore processors: the authors of [3] study divide-and-conquer cache-oblivious algorithms for several problems,

and they design algorithms that are asymptotically optimal for both shared and distributed caches misses. The emphasis is on models and asymptotic performance rather than on algorithms for fixed-size matrices.

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 architectures. However, they do not explicitly use the notion of cache misses, but instead focus on the number of blocks transferred between shared cache and main memory; distributed caches are not considered in their analysis.

In [10], the authors study the behavior of DGEMM kernels and introduce a fine-tuning version leading to better performance than Intel's parallel DGEMM in the MKL library. This paper provides a fine-grain analysis in terms of cache related notions, as for instance cache misses or false sharing, on a real hardware platform. Our analysis is at a higher level (our algorithms call sequential DGEMM kernels) and is not associated to any particular hardware architecture.

Experiments– In [9], the authors present performance results for dense linear algebra using recent NVIDIA GPUs, and analyse some factors impacting performance on these particular multicore processors through the performance evaluation of their matrix-matrix multiplication kernel. This work also is at a lower level since it focuses on implementing a better DGEMM routine for GPUs.

6 Conclusion

In this report, 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 corresponding 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. Every algorithm introduced in this paper has been tested, and is behavior validated, on the simulator that we have designed, using realistic parameters.

Future work will be twofold. On the algorithmic side, we will tackle more complex operations, such as LU factorization or path problems. On the more practical side, we will implement all algorithms on state-of-the-art multicore machines, being classical multicore CPUS as well as more exotic chips like GPUS. This will be a first step towards the (more ambitious) task of designing efficient algorithms for clusters of multicores: we expect yet another level of hierarchy (or tiling) in the algorithmic specification to be required in order to match the additional complexity of such platforms.

References

[1] D. A. Bader, V. Kanade, and K. Madduri. SWARM: A parallel programming framework for multicore processors. In *IPDPS'07, the 21st IEEE Int. Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

[3] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA'08, the 19th ACM-SIAM symposium on Discrete algorithms*, pages 501–510. SIAM Press, 2008.

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

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS'99, the 40th IEEE Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society Press, 1999.

[6] D. Ironya, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Computing*, 64(9):1017–1026, 2004.

[7] J.-F. Pineau, Y. Robert, F. Vivien, and J. Dongarra. Matrix product on heterogeneous master-worker platforms. In *PPoPP'2008, the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 53–62. ACM Press, 2008.

[8] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[9] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08, the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2008.

[10] S. Zuckerman, M. Pérache, and W. Jalby. Fine tuning matrix multiplications on multicore. In *High Pefroamnce Computing HiPC'08*, pages 30–41. Springer Verlag LNCS 5374, 2008.