Complexity analysis of matrix product on multicore architectures

Mathias Jacquelin, Loris Marchal and Yves Robert

Ecole Normale Supérieure de Lyon
Mathias.Jacquelin@ens-lyon.fr
http://graal.ens-lyon.fr/~mjacquel

Rocquencourt, February 4, 2009

Recent evolution of processors

From simple single core architectures . . .

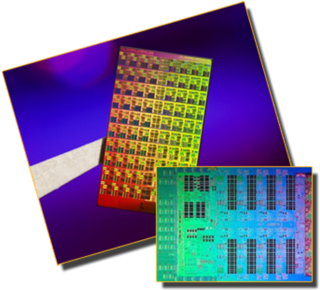
Recent evolution of processors

From simple single core architectures . . .

Speed used to be obtained through ILP

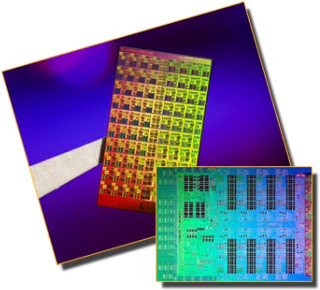
Recent evolution of processors

. . . to multi-core and upcoming many-core processors



Recent evolution of processors

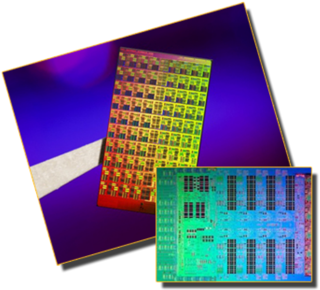
. . . to multi-core and upcoming many-core processors



Now, algorithms need to explicitly exploit TLP,
similar to classical parallel programming

Recent evolution of processors

. . . to multi-core and upcoming many-core processors



Now, algorithms need to explicitly exploit TLP,
similar to classical parallel programming

More important: must efficiently use memory, especially caches

New architectures, new problems to tackle

Target algorithms: Dense linear algebra kernels
(key to performance for many scientific applications)

Calls for revisiting old problems

- Algorithms based on a 2D grid topology are not well suited for multicore architectures
- Hierarchy of cache memories
- Need to take further advantage of data locality

Need to adapt algorithms: new objective functions, new models

New architectures, new problems to tackle

Target algorithms: Dense linear algebra kernels
(key to performance for many scientific applications)

Calls for revisiting old problems

- Algorithms based on a 2D grid topology are not well suited for multicore architectures
- Hierarchy of cache memories
- Need to take further advantage of data locality

Need to adapt algorithms: new objective functions, new models

New architectures, new problems to tackle

Target algorithms: Dense linear algebra kernels
(key to performance for many scientific applications)

Calls for revisiting old problems

- Algorithms based on a 2D grid topology are not well suited for multicore architectures
- Hierarchy of cache memories
- Need to take further advantage of data locality

Need to adapt algorithms: new objective functions, new models

New architectures, new problems to tackle

Target algorithms: Dense linear algebra kernels
(key to performance for many scientific applications)

Calls for revisiting old problems

- Algorithms based on a 2D grid topology are not well suited for multicore architectures
- Hierarchy of cache memories
- Need to take further advantage of data locality

Need to adapt algorithms: new objective functions, new models

New architectures, new problems to tackle

Target algorithms: Dense linear algebra kernels
(key to performance for many scientific applications)

Calls for revisiting old problems

- Algorithms based on a 2D grid topology are not well suited for multicore architectures
- Hierarchy of cache memories
- Need to take further advantage of data locality

Need to adapt algorithms: new objective functions, new models

Outline

1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication
2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results
3. Conclusion

Outline

1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication
2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results
3. Conclusion

Outline

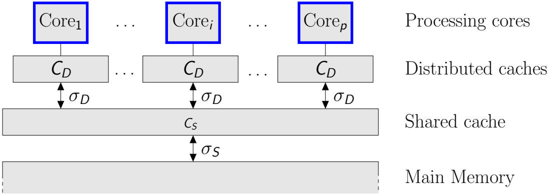
1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication

2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results

3. Conclusion

Modeling multicore architectures

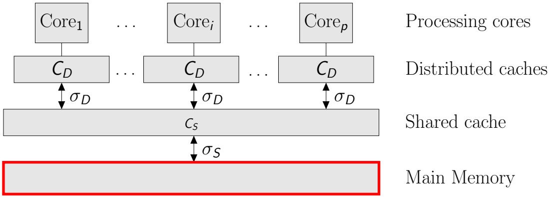
Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Modeling multicore architectures

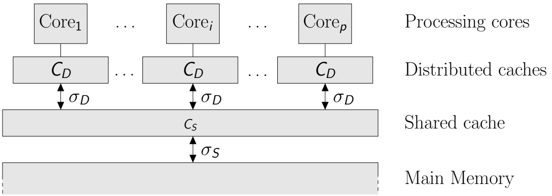
Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Modeling multicore architectures

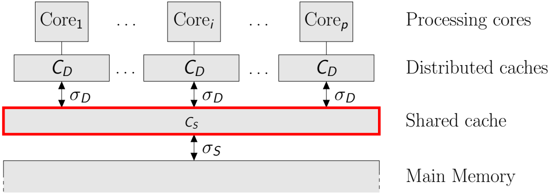
Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Modeling multicore architectures

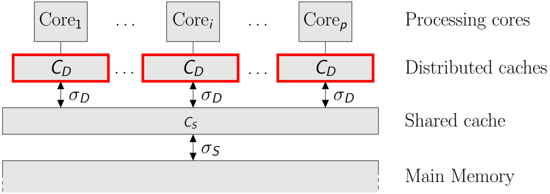
Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Modeling multicore architectures

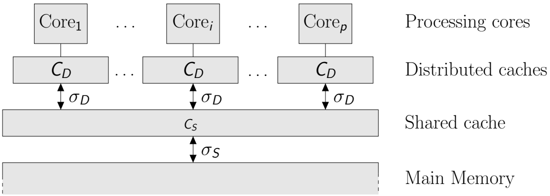
Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Modeling multicore architectures

Difficulty: Come up with a realistic but still tractable model



- p identical cores, computing speed w
- Large main memory
- Two levels of caches:
  - a first level shared by all cores of size C_S and bandwidth σ_S
  - a second level of cache distributed, each of size C_D and bandwidth σ_D
  - *Caches are* **inclusive** *and* **fully associative**

Outline

1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication
2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results
3. Conclusion

Studied case and objectives

Target: Compute the matrix product $C = A \times B$.

- A is $m \times z$, B is $z \times n$ and C has size $m \times n$

We use a block-oriented approach, thus, manipulate square blocks of coefficients.

First objective: Communication volume of shared cache.

- M_S is the number of cache misses in the shared cache

Second objective: Communication volume of distributed caches.

- M_D is the maximum of all distributed caches misses

Third objective: Overall time T_{data} required for data movement.

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

Provides a tradeoff between both cache miss quantities

Studied case and objectives

Target: Compute the matrix product $C = A \times B$.

- A is $m \times z$, B is $z \times n$ and C has size $m \times n$

We use a block-oriented approach, thus, manipulate square blocks of coefficients.

First objective: Communication volume of shared cache.

- M_S is the number of cache misses in the shared cache

Second objective: Communication volume of distributed caches.

- M_D is the maximum of all distributed caches misses

Third objective: Overall time T_{data} required for data movement.

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

Provides a tradeoff between both cache miss quantities

Studied case and objectives

Target: Compute the matrix product $C = A \times B$.

- A is $m \times z$, B is $z \times n$ and C has size $m \times n$

We use a block-oriented approach, thus, manipulate square blocks of coefficients.

First objective: Communication volume of shared cache.

- M_S is the number of cache misses in the shared cache

Second objective: Communication volume of distributed caches.

- M_D is the maximum of all distributed caches misses

Third objective: Overall time T_{data} required for data movement.

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

Provides a tradeoff between both cache miss quantities

Studied case and objectives

Target: Compute the matrix product $C = A \times B$.

- A is $m \times z$, B is $z \times n$ and C has size $m \times n$

We use a block-oriented approach, thus, manipulate square blocks of coefficients.

First objective: Communication volume of shared cache.

- M_S is the number of cache misses in the shared cache

Second objective: Communication volume of distributed caches.

- M_D is the maximum of all distributed caches misses

Third objective: Overall time T_{data} required for data movement.

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

Provides a tradeoff between both cache miss quantities

Outline

1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication

2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results

3. Conclusion

Lower bound on communication

- Irony, Toledo and Tiskin show that on a system with a memory of size M, the communication-to-computation ratio of matrix product is lower-bounded by: $\sqrt{\frac{27}{8M}}$.
- In our case, with $comp(c)$ being the amount of computation done by core c, we have:
  - $CCR_S = M_S/(\sum_c comp(c))$ for the shared cache
  - $CCR_D = \frac{1}{p}\sum_{c=1}^{p}(M_D/comp(c))$ for the distributed cache.
- In all our algorithms, the amount of computation is equally balanced among cores, so that $comp(c) = mnz/p$ for all cores. Therefore:

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

Outline

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

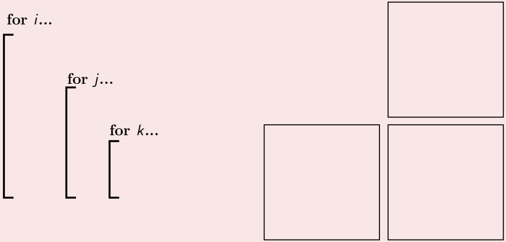
Rule 2 In a given loop, required data must be loaded once

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once

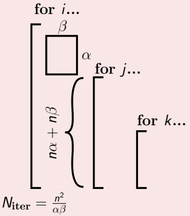


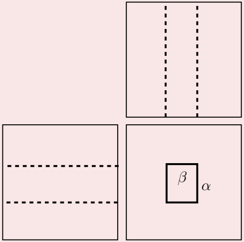
General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



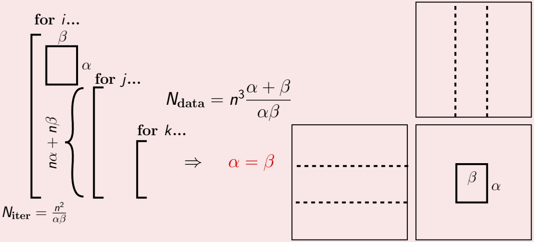


General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



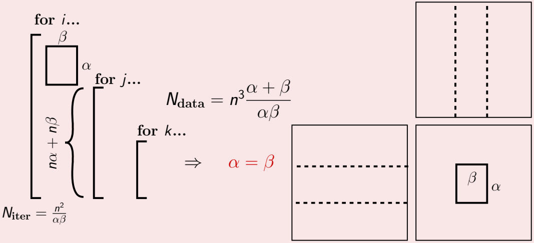
Observation Outermost loop is prevalent in order to minimize loaded data

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



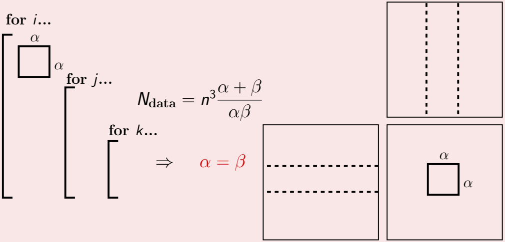
Corollary 1 In **outermost loop**, load the **largest square** blocks

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



$$N_{\text{data}} = n^3 \frac{\alpha + \beta}{\alpha\beta}$$

$$\alpha = \beta$$

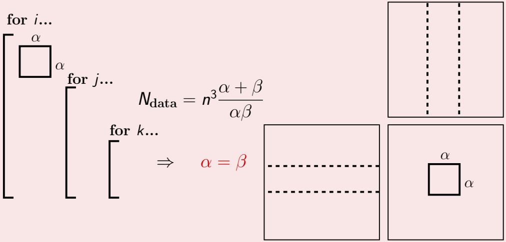
Corollary 1 In **outermost loop**, load the **largest square** blocks

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



Corollary 1 In **outermost loop**, load the **largest square** blocks

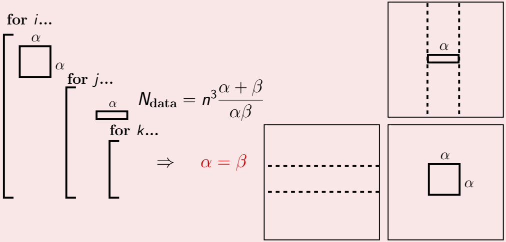
Corollary 2 In **inner loops**, load the **smallest block** allowing to respect Rules 1 & 2.

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



$$N_{\text{data}} = n^3 \frac{\alpha + \beta}{\alpha\beta}$$

$\Rightarrow \quad \alpha = \beta$

Corollary 1 In **outermost loop**, load the **largest square** blocks

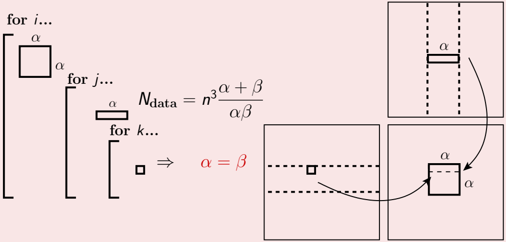
Corollary 2 In **inner loops**, load the **smallest block** allowing to respect Rules 1 & 2.

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

Rule 1 Loaded data must be re-used as much as possible

Rule 2 In a given loop, required data must be loaded once



Corollary 1 In **outermost loop**, load the **largest square** blocks

Corollary 2 In **inner loops**, load the **smallest block** allowing to respect Rules 1 & 2.

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

- **Rule 1** Loaded data must be re-used as much as possible
- **Rule 2** In a given loop, required data must be loaded once
- **Corollary 1** In **outermost loop**, load the **largest square** blocks
- **Corollary 2** In **inner loops**, load the **smallest block** allowing to respect Rules 1 & 2.

How ?

- Split the available memory into $1 + \alpha + \alpha^2$ blocks

General principle and Maximum Re-Use Algorithm

Main Objective: Create a data-thrifty algorithm, memory of size M

- Rule 1: Loaded data must be re-used as much as possible
- Rule 2: In a given loop, required data must be loaded once
- Corollary 1: In **outermost loop**, load the **largest square** blocks
- Corollary 2: In **inner loops**, load the **smallest block** allowing to respect Rules 1 & 2.

How ?

- Split the available memory into $1 + \alpha + \alpha^2$ blocks

Result

- A CCR of $\frac{2}{\sqrt{M}}$ for a memory of size M for large matrices

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

Adaptation to multicore

- Must take into account both cache levels:
  - Previous data allocation scheme adapted so as to fit caches.
- Two parameters depending on cache sizes:
  - λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$
  - μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
  - These parameters will be used separately
  - For the sake of simplicity, we assume that λ is a multiple of μ

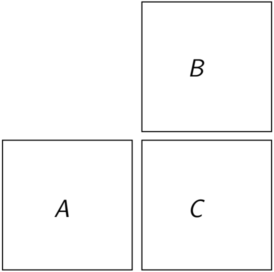
Outline

Minimizing the number of shared-cache misses

for $step = 1$ to $\frac{n^2}{\lambda^2}$

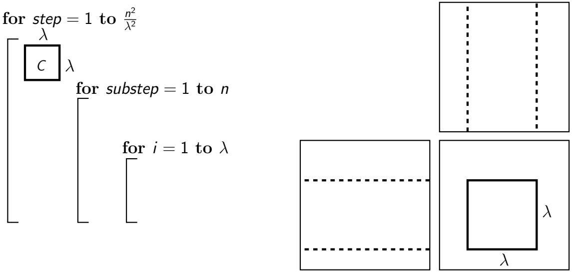
    for $substep = 1$ to n

        for $i = 1$ to λ



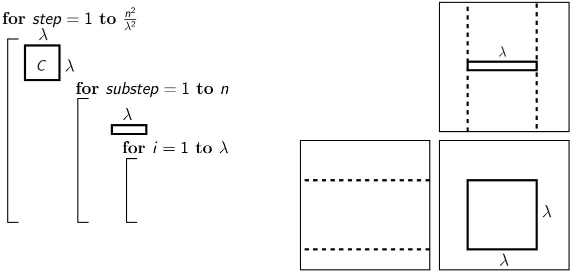
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



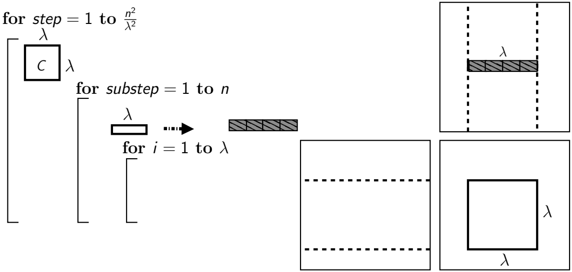
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



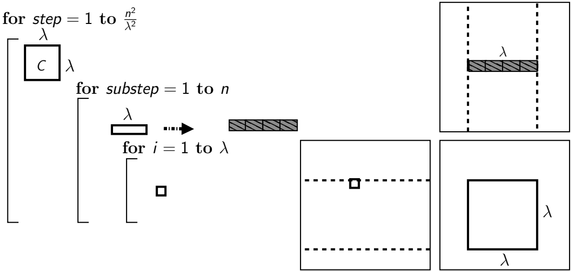
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



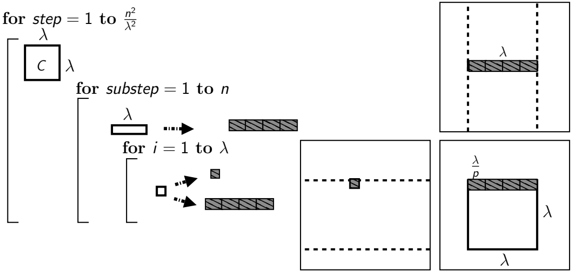
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



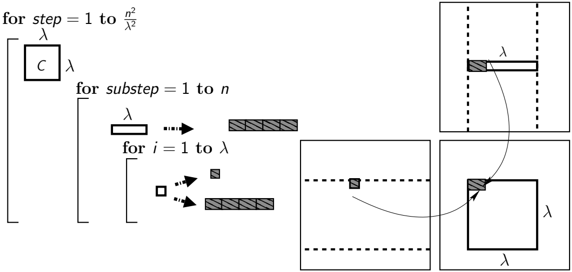
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



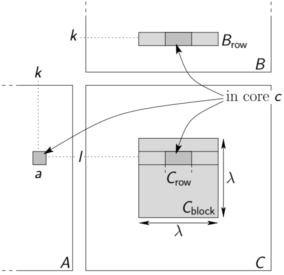
- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses



- We load in shared cache:
  - A square block of size λ^2 of C
  - A row of λ elements of B
  - One element of A
- Then, rows of C_{block} and elements of A are distributed and computed.
- Repeat until the block of C had been fully updated.

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Minimizing the number of shared-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size λ^2, z rows of size λ are loaded from B as well as $z \times \lambda$ elements of A.

- $M_S = mn + 2mnz/\lambda$
- For large matrices, CCR is $2/\lambda$ ☺

Distributed-cache misses

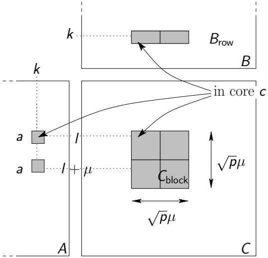
At each step, we:

- load z times λ elements of A one by one
- load z rows of size λ/p of B
- update $\lambda \times z$ times rows of size λ/p of C

- $M_D = \frac{mnz}{\lambda} \times (1 + 1/p + \lambda/p)$
- CCR is $(p+1)/\lambda + 1$ ☹

Outline

Minimizing the number of distributed-cache misses



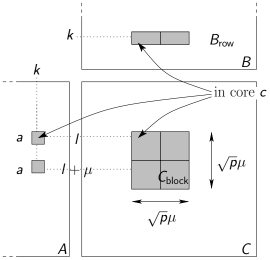
for *step* $= 1$ **to** $\frac{n^2}{p\mu^2}$

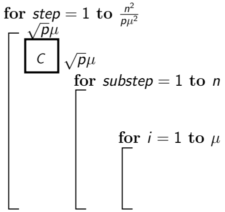
  for *substep* $= 1$ **to** n

    for $i = 1$ **to** μ

- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

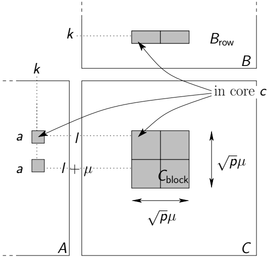
Minimizing the number of distributed-cache misses

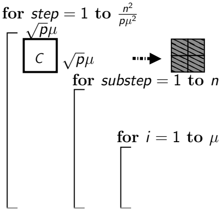




- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

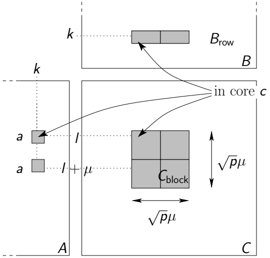
Minimizing the number of distributed-cache misses

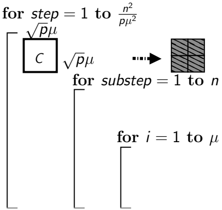




- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

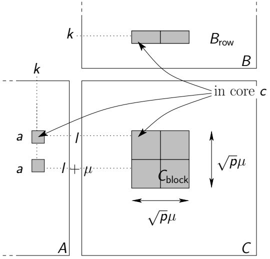
Minimizing the number of distributed-cache misses

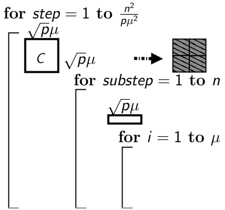




- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

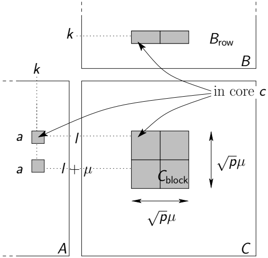
Minimizing the number of distributed-cache misses

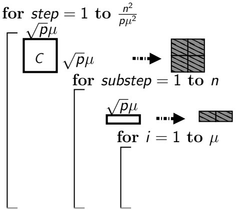




- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

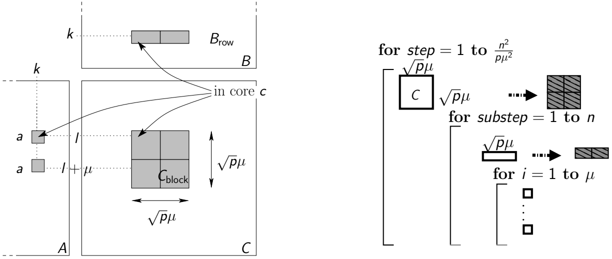
Minimizing the number of distributed-cache misses





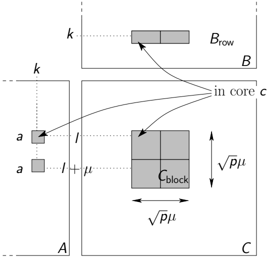
- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

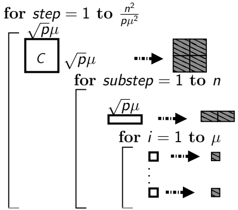
Minimizing the number of distributed-cache misses



- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

Minimizing the number of distributed-cache misses





- μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$
- A square block of size $(\sqrt{p}\mu)^2$ of C is loaded in the shared cache and, subblocks of size μ^2 are distributed to every cores
- Then, repeatedly, z times:
  - a row of $\sqrt{p}\mu$ elements of B is loaded in the shared cache and distributed
  - $\sqrt{p}$ elements of A are sequentially read μ times in shared cache and distributed in order to contribute to current sub-blocks of C

Minimizing the number of distributed-cache misses

Shared-cache misses

- Elements of C are loaded once in shared cache
- For each block of size $(\sqrt{p}\mu)^2$ of C, we load:
  - z rows of size $\sqrt{p}\mu$ of B
  - $z \times \sqrt{p}\mu$ elements of A.
- $M_S = mn + 2mnz/\sqrt{p}\mu$ ☹
- For large matrices, CCR is $2/\sqrt{p}\mu$

Distributed-cache misses

- mn/p elements of C are loaded once in each distributed cache
- Then, at each step, we load z times:
  - A row of μ elements of B
  - μ sequential elements of A
- $M_D = mn/p + 2mnz/p\mu$
- For large matrices CCR is $2/\mu$ ☺

Outline

Minimizing data access time

Why do we need a tradeoff ?

- Previous objectives were antagonistic
- Bandwidths not taken into account.

New objective: overall time spent in data movement

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

Minimizing data access time

Why do we need a tradeoff ?

- Previous objectives were antagonistic
- Bandwidths not taken into account.

New objective: overall time spent in data movement

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

Minimizing data access time

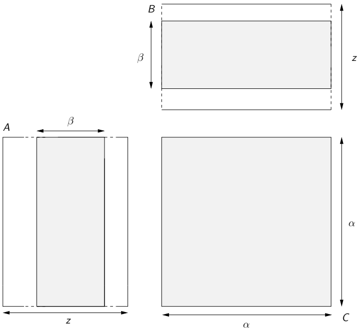
Why do we need a tradeoff ?

- Previous objectives were antagonistic
- Bandwidths not taken into account.

New objective: overall time spent in data movement

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

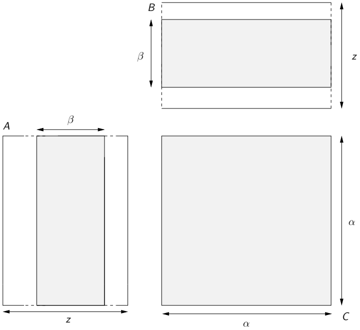
Minimizing data access time



- We load in shared cache:
  - a square block of size α^2 of C
  - a block column of size $\alpha \times \beta$ of A
  - a block row of the same size of B

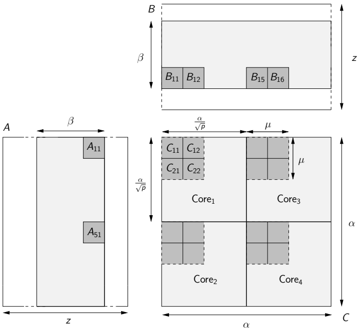
  $\Rightarrow$ only z/β iterations
- μ^2 blocks of C are distributed, with proper rows of B and element of A

Minimizing data access time



- We load in shared cache:
  - a square block of size α^2 of C
  - a block column of size $\alpha \times \beta$ of A
  - a block row of the same size of B
  - $\Rightarrow$ only z/β iterations
- μ^2 blocks of C are distributed, with proper rows of B and element of A

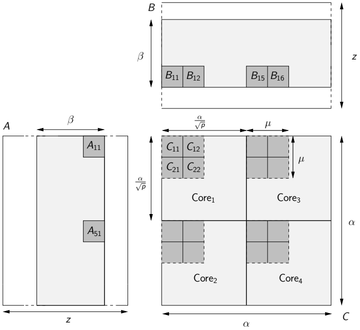
Minimizing data access time



- We load in shared cache:
  - a square block of size α^2 of C
  - a block column of size $\alpha \times \beta$ of A
  - a block row of the same size of B

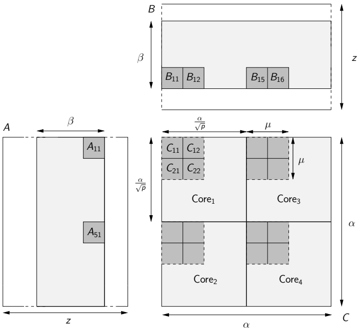
  $\Rightarrow$ only z/β iterations
- μ^2 blocks of C are distributed, with proper rows of B and element of A

Minimizing data access time



- Depending on β, we cannot load as many elements of C as before
  - We need to find the best tradeoff between β and α

Minimizing data access time



- Depending on β, we cannot load as many elements of C as before
  - We need to find the best tradeoff between β and α

Minimizing data access time

- New constraint on shared cache: $2\beta\alpha + \alpha^2 \leq C_S$
- Our new tradeoff algorithm has an overall data access time:

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

- The objective function is:

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

- We can now compute the best numerical values of parameters α and β

Minimizing data access time

- New constraint on shared cache: $2\beta\alpha + \alpha^2 \leq C_S$
- Our new tradeoff algorithm has an overall data access time:

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

- The objective function is:

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

- We can now compute the best numerical values of parameters α and β

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  $\Rightarrow$ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  $\Rightarrow$ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  $\Rightarrow$ Distributed version.

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  $\Rightarrow$ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  $\Rightarrow$ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  $\Rightarrow$ Distributed version.

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  $\Rightarrow$ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  $\Rightarrow$ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  $\Rightarrow$ Distributed version.

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  ⇒ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  ⇒ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  ⇒ Distributed version.

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  $\Rightarrow$ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  $\Rightarrow$ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  $\Rightarrow$ Distributed version.

Minimizing data access time

- α depends on the values of bandwidths σ_S and σ_D.

  ⇒ In both extreme cases, the algorithm will follow the sketch of either shared or distributed cache optimized version:

  - When $\sigma_D \gg \sigma_S$:

  ⇒ Shared version.

  - On the contrary, when $\sigma_S \gg \sigma_D$ (not realistic):

  ⇒ Distributed version.

Outline

Hardware platforms & Implementation details

Replacement policy:

- Model: Caches use an **ideal data replacement policy**
- On most current hardware platforms: **LRU data replacement policy**

Hardware platforms & Implementation details

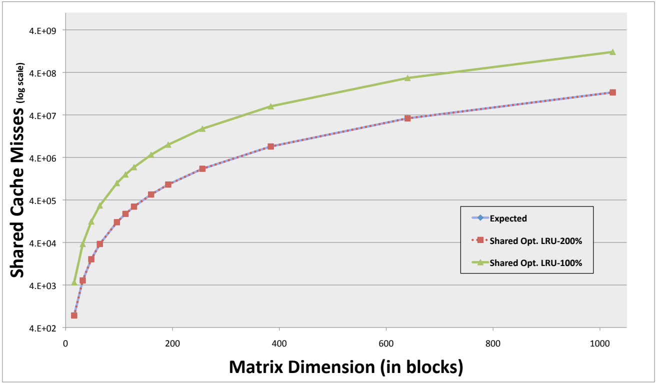
Replacement policy:

- Model: Caches use an **ideal data replacement policy**
- On most current hardware platforms: **LRU data replacement policy**

LRU vs. Ideal: In *Cache Oblivious Algorithms*, the authors stated that the number of cache misses obtained using an ideal data replacement policy on a cache of size M could be obtained using a LRU policy on a cache of size $2M$

Hardware platforms & Implementation details

LRU vs. Ideal: In *Cache Oblivious Algorithms*, the authors stated that the number of cache misses obtained using an ideal data replacement policy on a cache of size M could be obtained using a LRU policy on a cache of size $2M$

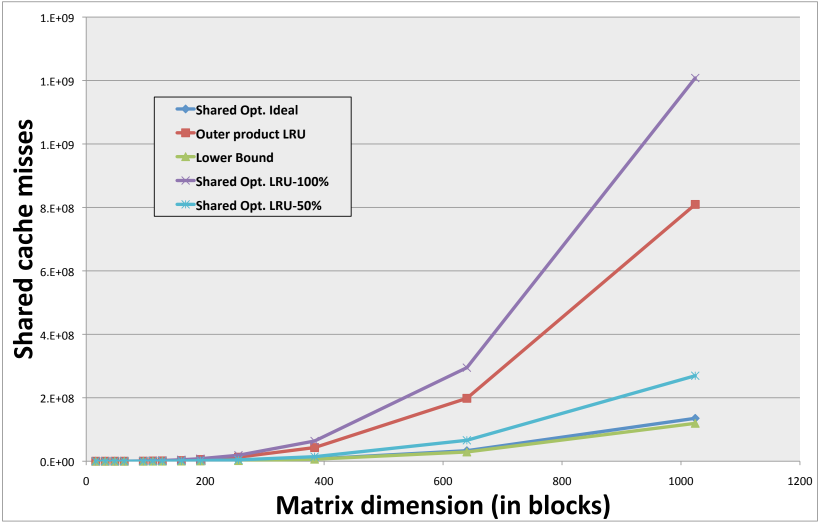


Hardware platforms & Implementation details

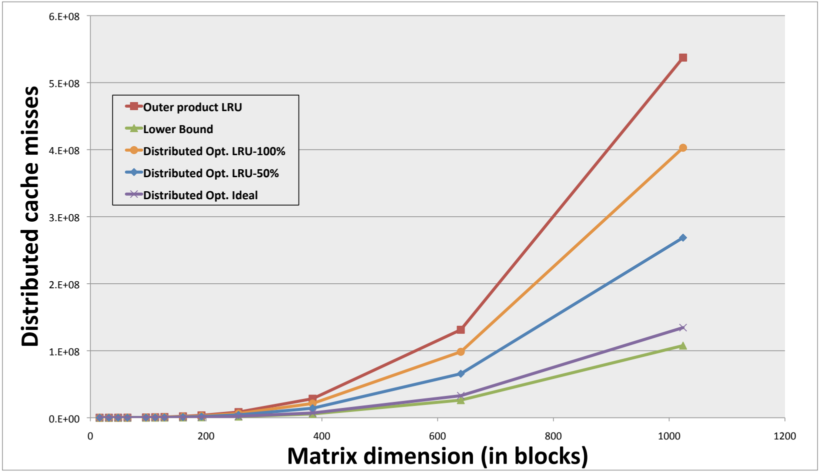
Benchmarked algorithms:

- Outer Product
- Multicore Maximum Re-use Algorithm:
  - 3 versions:
    - Shared Opt.
    - Distributed Opt.
    - Tradeoff
  - 3 simulation settings:
    - IDEAL: *explicit loads in every cache, no propagation*
    - LRU-100%: *LRU policy, using entire cache*
    - LRU-50%: *LRU policy, half-cache for automatic prefetching*

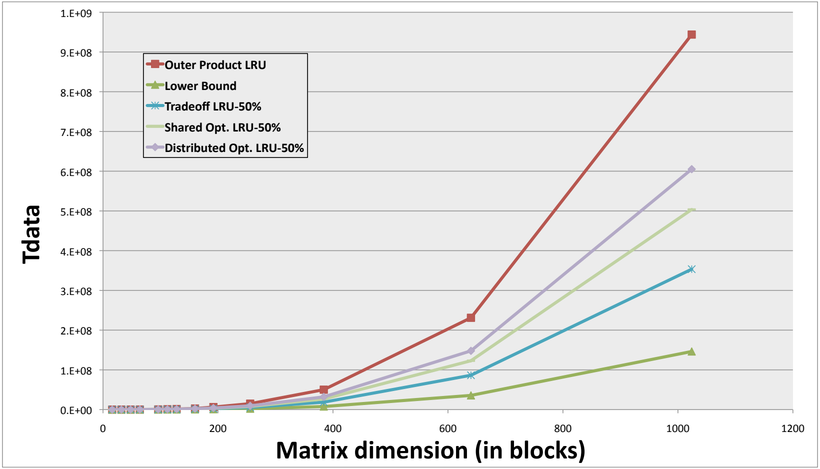
Experimental results obtained on our cache simulator



Experimental results obtained on our cache simulator



Experimental results obtained on our cache simulator



Outline

1. Problem statement
   - Modeling multicore architectures
   - Studied case and objectives
   - Lower bound on communication
2. Maximum re-use algorithm for multicore architectures
   - Minimizing the number of shared-cache misses
   - Minimizing the number of distributed-cache misses
   - Minimizing data access time
   - Experimental results
3. Conclusion

Complexity analysis of matrix product

- Model for multicore memory layout.
- For large matrices, our cache aware algorithms are close to the lower bounds.
- New algorithm realizing a tradeoff between both cache misses types.
- Our three algorithms were implemented, simulated and their behavior validated.

We now plan to extend our work to more complex kernels, like LU factorization

☺ Promising algorithmic research directions to explore !