Matrix factorizations on multicores with OpenMP
(Calcul Réparti et Grid Computing)

alfredo.buttari@enseeiht.fr

for an up-to-date version of the slides:
http://buttari.perso.enseeiht.fr

Introduction

Objective of these lectures

Present the main issues and challenges we have to face when programming shared memory (multicore) computers. Understand the limits of an algorithm and show techniques to achieve better parallelism and performance.

Why

Because multicore processors are ubiquitous

How

We will take a reference algorithm from a book, we will show why it is not suited for parallelization and we will modify it producing different variants that achieve better and better parallelism and performance

Introduction

Which algorithm?

The Householder QR factorization

- because you have studied it (Algèbre Linéaire Numérique, first year)
- because there's a lot of literature about it and its parallelization

Which tools for the parallelization?

OpenMP

- because you have studied it (Systèmes Concurrents, second year)
- because it's simple

(Very) Brief OpenMP resume

Section 1

OpenMP

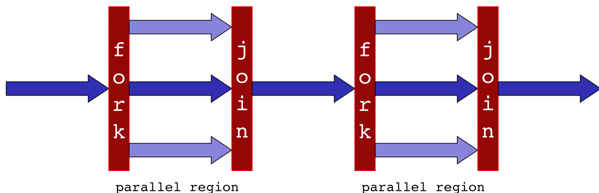
How to program multicores: OpenMP

OpenMP (Open specifications for MultiProcessing) is an Application Program Interface (API) to explicitly direct multi-threaded, shared memory parallelism.

- Comprised of three primary API components:
  - Compiler directives (OpenMP is a compiler technology)
  - Runtime library routines
  - Environment variables
- Portable:
  - Specifications for C/C++ and Fortran
  - Already available on many systems (including Linux, Win, IBM, SGI etc.)
- Full specs
  `http://openmp.org`
- Tutorial
  `https://computing.llnl.gov/tutorials/openMP/`

How to program multicores: OpenMP

OpenMP is based on a fork-join execution model:



- Execution is started by a single thread called master thread
- when a parallel region is encountered, the master thread spawns a set of threads
- the set of instructions enclosed in a parallel region is executed
- at the end of the parallel region all the threads synchronize and terminate leaving only the master

Parallel regions and other OpenMP constructs are defined by means of compiler directives:

```c
#include <omp.h>

main ()  {

  int var1, var2, var3;

  /* Serial code  */

#pragma omp parallel private(var1, var2)
                     shared(var3)
  {

    /* Parallel section executed
       by all threads  */

  }

  /* Resume serial code  */

}
```

```fortran
program hello

  integer :: var1, var2, var3

!  Serial code

!$omp parallel private(var1, var2)
!$omp& shared(var3)

!  Parallel section executed by all threads

!$omp end parallel

!  Resume serial code

end program hello
```

OpenMP: worksharing constructs

The DO/for directive:

```
program do_example

  integer   :: i, chunk
  integer, parameter :: n=1000, chunksize=100
  real(kind(1.d0)) :: a(n), b(n), c(n)

  ! Some sequential code...
  chunk = chunksize

  !$omp parallel shared(a,b,c) private(i)

  !$omp do
  do i = 1, n
     c(i) = a(i) + b(i)
  end do
  !$omp end do

  !$omp end parallel

end program do_example
```

OpenMP: the task construct

Example of the TASK construct:

```
program example_task

  integer :: i, n
  n = 10

!$omp parallel
!$omp master
  do i=1, n
!$omp task firstprivate(i)
     call tsub(i)
!$omp end task
  end do
!$omp end master
!$omp end parallel

  stop
end program example_task

subroutine tsub(i)
  integer :: i
  integer :: iam, nt, omp_get_num_threads, &
       &omp_get_thread_num

  iam = omp_get_thread_num()
  nt  = omp_get_num_threads()

  write(*,'("iam:",i2,"   nt:",i2,"   i:",i4)')iam,nt,i

  return
end subroutine tsub
```

result

```
iam: 3   nt: 4   i:   3
iam: 2   nt: 4   i:   2
iam: 0   nt: 4   i:   4
iam: 1   nt: 4   i:   1
iam: 3   nt: 4   i:   5
iam: 0   nt: 4   i:   7
iam: 2   nt: 4   i:   6
iam: 1   nt: 4   i:   8
iam: 3   nt: 4   i:   9
iam: 0   nt: 4   i:  10
```

Data scoping in tasks

The data scoping clauses `shared`, `private` and `firstprivate`, when used with the `task` construct are not related to the threads but to the tasks.

- `shared(x)` means that when the task is executed `x` is the same variable (the same memory location) as when the task was created
- `private(x)` means that `x` is private to the task, i.e., when the task is created, a brand new variable `x` is created as well. This new copy is destroyed when the task is finished
- `firstprivate(x)` means that `x` is private to the task, i.e., when the task is created, a brand new variable `x` is created as well and its value is set to be the same as the value of `x` in the enclosing context at the moment when the task is created. This new copy is destroyed when the task is finished

If a variable is private in the parallel region it is implicitly firstprivate in the included tasks

Data scoping in tasks

```
program example_task

  integer :: x, y, z, j


...


j = 2

x = func1(j)


j = 4

y = func2(j)



z = x+y


...


end program example_task
```

Data scoping in tasks

```
program example_task

  integer :: x, y, z, j

!$omp parallel private(x,y)
...
!$omp master

j = 2
!$omp task ! x is implicitly private, j shared
x = func1(j)
!$omp end task

j = 4
!$omp task ! y is implicitly private, j shared
y = func2(j)
!$omp end task

!$omp taskwait

z = x+y

!$omp end master
...
!$omp end parallel

end program example_task
```

Data scoping in tasks

```
program example_task

  integer :: x, y, z, j, xc, yc

!$omp parallel private(x,y)
...
!$omp master

j = 2
!$omp task shared(xc) firstprivate(j)
xc = func1(j)
!$omp end task

j = 4
!$omp task shared(yc) firstprivate(j)
yc = func2(j)
!$omp end task

!$omp taskwait

z = xc+yc

!$omp end master
...
!$omp end parallel

end program example_task
```

OpenMP: the `task` construct

The depend clause enforces additional constraints on the scheduling of tasks.

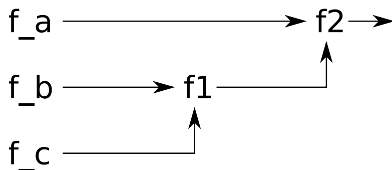
Task dependences are derived from the dependence-type of a depend clause and its list items, where dependence-type is one of the following:

- The **in** dependence-type. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout dependence-type list.
- The **out** and **inout** dependence-types. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out, or inout dependence-type list.

OMP tasks: example

Thanks to the specified dependencies the OpenMP runtime can build a graph of dependencies and schedule the tasks accordingly

```
!$omp parallel
!$omp single
!$omp task depend(out:a)
  a = f_a()
!$omp end task

!$omp task depend(out:b)
  b = f_b()
!$omp end task

!$omp task depend(out:c)
  c = f_c()
!$omp end task

!$omp task depend(in:b,c) depend(out:x)
  x = f1(b, c)
!$omp end task

!$omp task depend(in:a,x) depend(out:y)
  y = f2(a, x)
!$omp end task

!$omp end single
!$omp end parallel
```



OMP tasks: pointers

When declaring dependencies using pointers, pay attention to the difference between the pointer object and the pointed object

```
void main(){

  int i;
  int *pnt;

  ...
#pragma omp parallel
  {
#pragma omp single
    {
      for(i=0; i<n; i++)
        {
#pragma omp task firstprivate(i,pnt) depend(inout:pnt)
          printf("Hello! I am %2d in iteration %2d\n",omp_get_thread_num(),i);
          *pnt = i;
          pnt++;
        }
    }
  }
}
```

OMP tasks: pointers

When declaring dependencies using pointers, pay attention to the difference between the pointer object and the pointed object

```
void main(){

  int i;
  int *pnt;

  ...
#pragma omp parallel
  {
#pragma omp single
    {
      for(i=0; i<n; i++)
        {
#pragma omp task firstprivate(i,pnt) depend(inout:pnt)
          printf("Hello! I am %2d in iteration %2d\n",omp_get_thread_num(),i);
          *pnt = i;
          pnt++;
        }
    }
  }
}
```

Task at iteration `i` depends on task at iteration `i-1` because dependencies are computed using the pointer object

OMP tasks: pointers

When declaring dependencies using pointers, pay attention to the difference between the pointer object and the pointed object

```
void main(){

  int i;
  int *pnt;

  ...
#pragma omp parallel
  {
#pragma omp single
    {
      for(i=0; i<n; i++)
        {
#pragma omp task firstprivate(i,pnt) depend(inout:*pnt)
          printf("Hello! I am %2d in iteration %2d\n",omp_get_thread_num(),i);
          *pnt = i;
          pnt++;
        }
    }
  }
}
```

OMP tasks: pointers

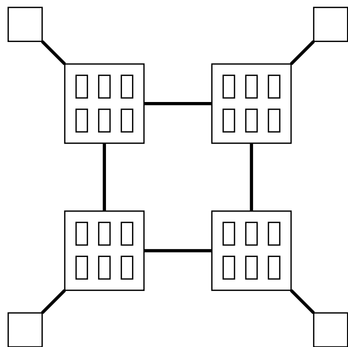
When declaring dependencies using pointers, pay attention to the difference between the pointer object and the pointed object

```
void main(){

  int i;
  int *pnt;

  ...
#pragma omp parallel
  {
#pragma omp single
    {
      for(i=0; i<n; i++)
        {
#pragma omp task firstprivate(i,pnt) depend(inout:*pnt)
          printf("Hello! I am %2d in iteration %2d\n",omp_get_thread_num(),i);
          *pnt = i;
          pnt++;
        }
    }
  }
}
```

All task are independent because dependencies are computed using the pointed object

Target architecture

Our target architecture has four, hexa-core AMD Opteron 8431 processors connected through HyperTransport links in a ring topology with a memory module attached to each of them



- clock frequency 2.4GHz → peak performance in DP $s = 2.4 * 2 * 2 = 9.6$ Gflop/s (the first 2 is for SSE units, the second is for dual-issue);
- memory frequency = 667 MHz → peak memory bandwidth $b = 0.667 * 2 * 8 = 10.6$ GB/s (the 2 is for dual-channel and the 8 is for the bus width).

Roofline model and BLAS routines

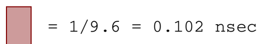
Performance modeling and evaluation

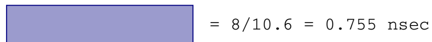
How to model and evaluate the performance of a sequential and/or multithreaded code?

Assumptions:

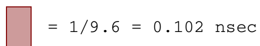
- An operation is a succession of data transfers and computations.
- Computations and data transfers can be done at the same time: we will assume that while doing some computations, we can prefetch data for the computations step.
- Once in the cache, the access to data costs nothing.

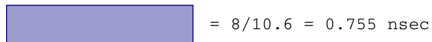
Performance modeling and evaluation





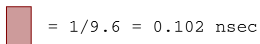
Performance modeling and evaluation

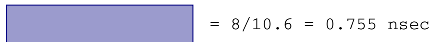


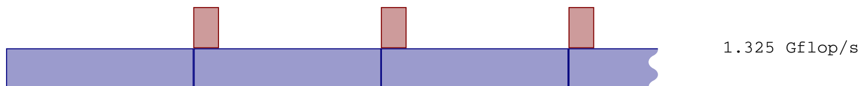


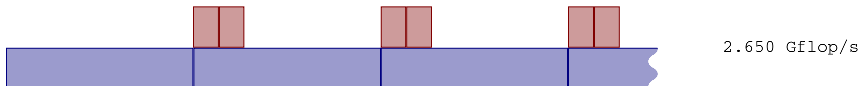
1.325 Gflop/s

Performance modeling and evaluation

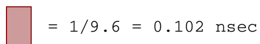


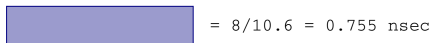


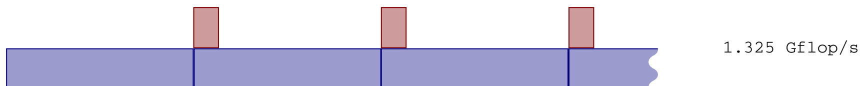


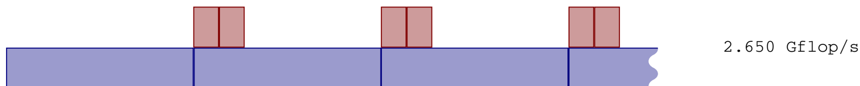


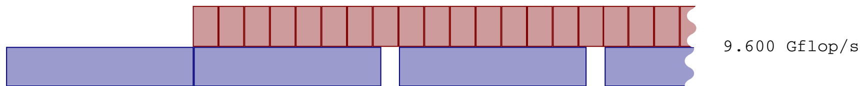
Performance modeling and evaluation











The roofline model

For a given computer, let

- s be the peak performance (speed) of one core in number of floating point operations (flops) per second flop/s
- b the memory bandwidth of one socket in GB/s

For a given algorithm, let

- W be the total workload in number of flops
- D be the volume of data, in bytes, transferred to/from the memory

Under the assumption that computation and memory transfers are done concurrently, the time to execute this algorithm on one core is

$$t = max\left(\frac{W}{s}, \frac{D}{b}\right)$$

The roofline model

What is the maximum speed (in flop/s) at which the algorithm can be executed?

$$\text{max speed} = \frac{W}{t} = min \begin{cases} s \\ \\ b \times \frac{W}{D} \end{cases}$$

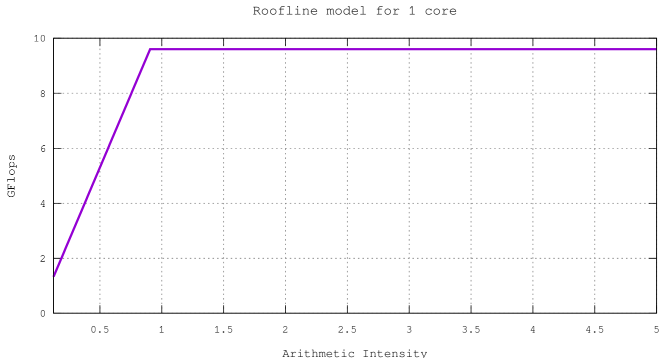
This is called the Roofline model [1]. The quantity W/D is called arithmetic intensity and denotes the amount of work done per byte transferred to/from memory.

The roofline model

What is the maximum speed (in flop/s) at which the algorithm can be executed?

$$\text{max speed} = \frac{W}{t} = min \begin{cases} 9.6 \\ 10.6 \times \frac{W}{D} \end{cases}$$

This is called the Roofline model [1]. The quantity W/D is called arithmetic intensity and denotes the amount of work done per byte transferred to/from memory.



Arithmetic intensity

The arithmetic intensity is defined as the number of operations per data transfer.

How is this quantity important? A modern processor (single-core for the moment) has a typical peak performance around 10 Gflop/s for double-precision computations and a typical memory bandwidth of 10 GB/s = 10/8 GDW/s. This means that a single core can process data much faster than the memory can stream it. This is why cache memories exist.

Caches

Data that is reused over time (temporal locality) or which is used right after some other contiguous data (spatial locality) can be kept in cache memories and accessed much faster.

Therefore, if we can reuse a data multiple times once we have brought it into cache, performance improves.

BLAS routines

The BLAS [2, 3, 4] library offers subroutines for basic linear algebra operations (it's called Basic Linear Algebra Subroutines for a reason...). These routines are grouped in three levels

`Level-1`

vector or vector-vector operations such as

- `_axpy`: $y = \alpha x + y$
- `_dot`: $dot = x^T y$
- `_nrm2`: $nrm2 = \|x\|$

BLAS routines

Level-2

matrix-vector operations such as

- `_gemv`: $y = \alpha A x + \beta y$
- `_ger`: $A = \alpha x y^T + A$
- `_trsv`: $x = T^{-1} x \quad (T \text{ triangular})$

Level-3

matrix-matrix operations such as

- `_gemm`: $C = \alpha A B + \beta C$
- `_trsm`: $\alpha X = T^{-1} X \quad (T \text{ triangular})$

BLAS routines

From the point of view of performance, there is a considerable difference among the three levels and it is due to the different ratios between floating-point operations and memory-to-cpu data transfers:

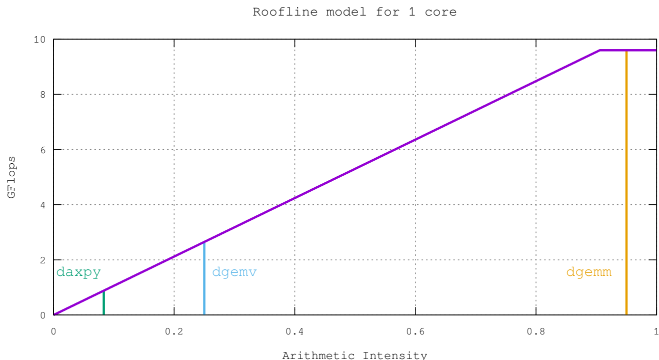
- `Level-1` routines typically perform $\mathcal{O}(n)$ floating-point operations on $\mathcal{O}(n)$ values
- `Level-2` routines typically perform $\mathcal{O}(n^2)$ floating-point operations on $\mathcal{O}(n^2)$ values
- `Level-3` routines typically perform $\mathcal{O}(n^3)$ floating-point operations on $\mathcal{O}(n^2)$ values

This means that, unlike in `Level-1` and `2` where each coefficient is used only once (no locality, operational intensity $\mathcal{O}(1)$), in `Level-3` routines each coefficient is used n times (lots of locality, operational intensity $\mathcal{O}(n)$).

BLAS routines

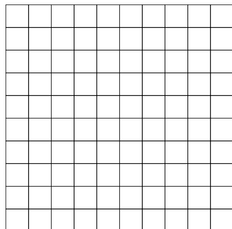
Here is how `Level-1`, `2` and `3` BLAS perform on our reference architecture:

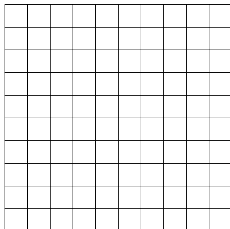
routine	#ops	#data	Gflop/s	RM
daxpy	$2n$	$3n$	0.46	0.88
dgemv	$2n^2$	$n^2 + 2n$	1.18	2.65
dgemm	$2n^3$	$4n^2$	8.95	9.60

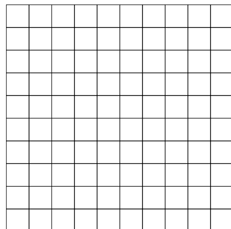


BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

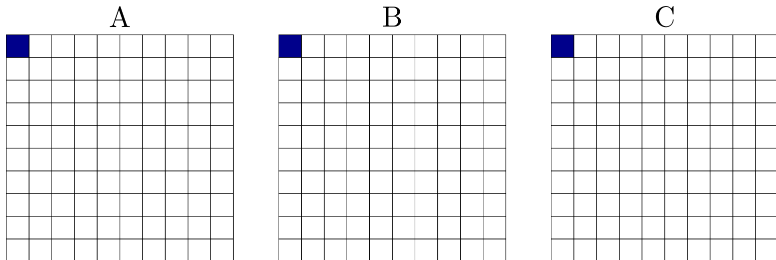






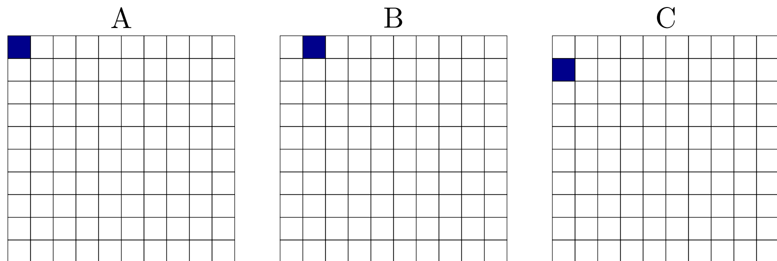
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



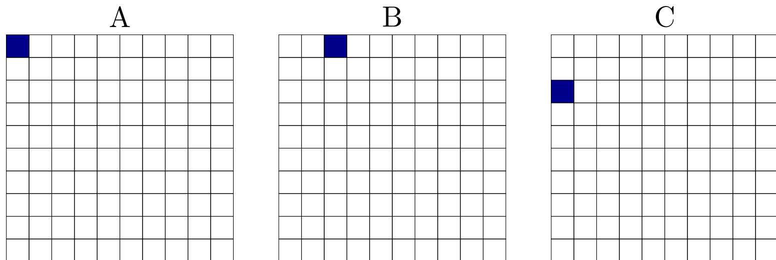
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



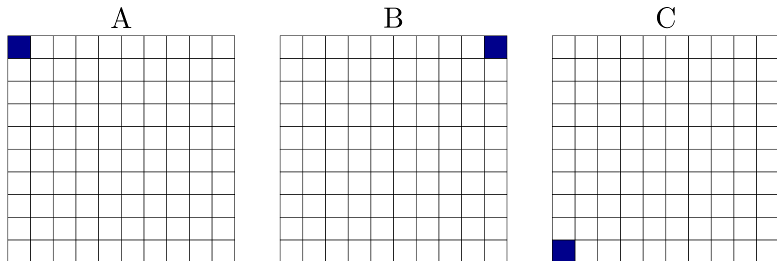
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

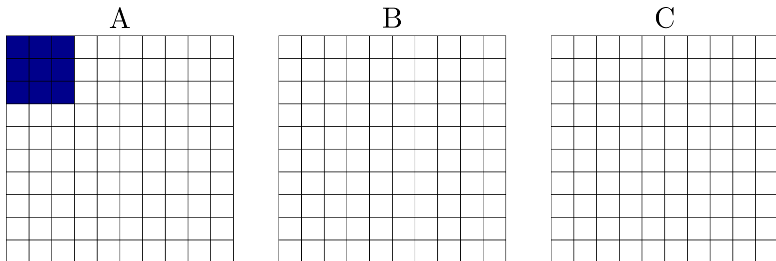


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

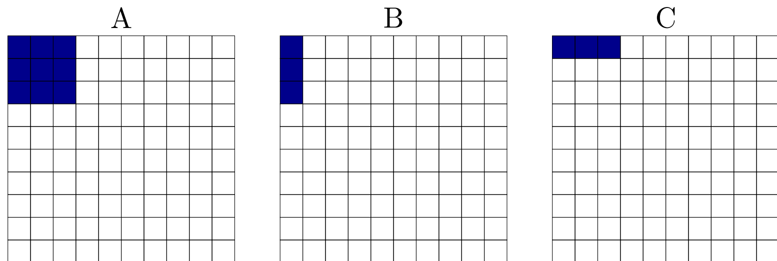


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

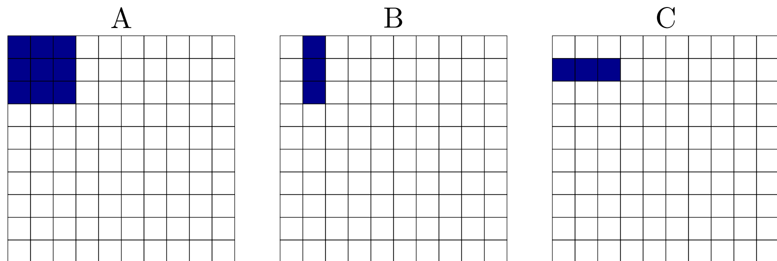


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

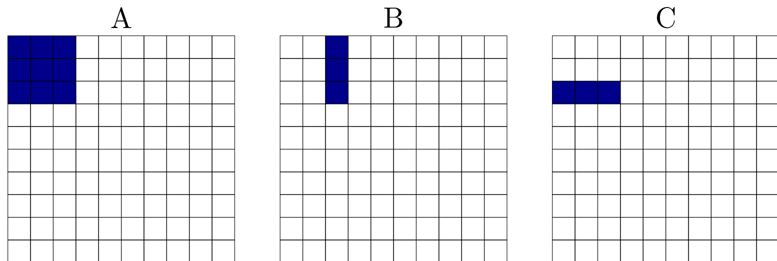


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

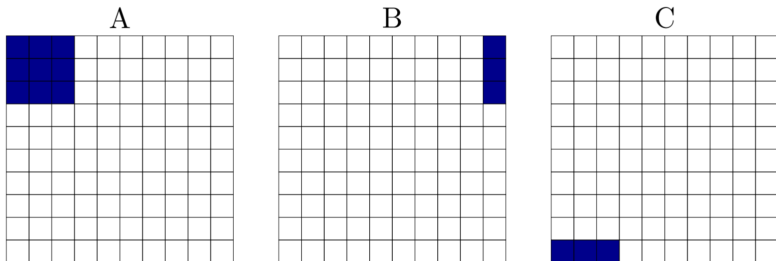


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

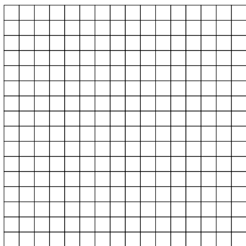
Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



number of cache misses:

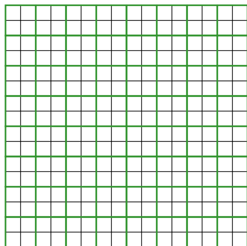
- unblocked $= 2n^3 = 2G$
- blocked $= (b^2 + 2bn) * ((n/b)^2) = 67M \quad (b = 30)$

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:

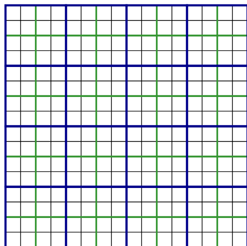


BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:

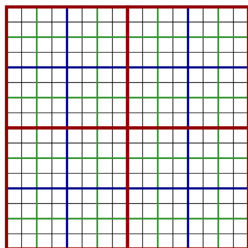


Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



More about BLAS optimization techniques can be found in ATLAS [5], or GotoBLAS [6, 7].

The roofline model: multicores

Assume the algorithm is executed in parallel by p processes, each running on a different core:

$$t = \max_{i=0}^{p-1} \left(t^i\right) = \max_{i=0}^{p-1} \left(\max \left(\frac{W^i}{s^i}, \frac{D^i}{b^i} \right) \right)$$

i.e., the total execution time corresponds to the time taken by the slowest and/or most loaded process.

Assume the algorithm is embarrassingly parallel: each processes

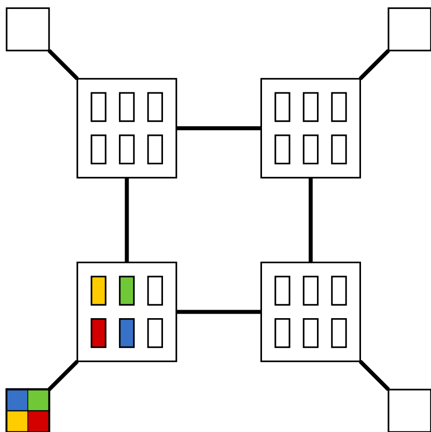
- executes $W^i = W/p$ operations
- and uses $D^i = D/p$ data

$$t = \max_{i=0}^{p-1} \left(t^i\right) = \max_{i=0}^{p-1} \left(\max \left(\frac{W}{ps^i}, \frac{D}{pb^i} \right) \right)$$

The roofline model: multicores

Case 1: bad data placement

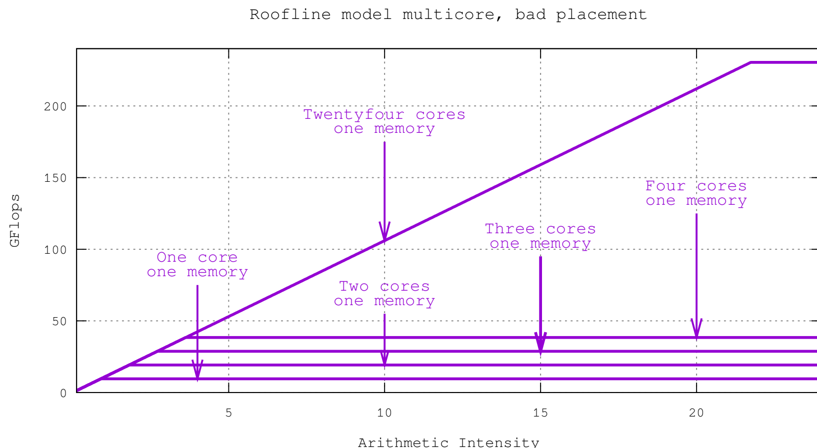
In this case all the processes read/write data from/to the same memory and, thus, the bandwidth is shared among all the processes. Therefore for each process the bandwidth is $b^i = b/p$



$$t = \max_{i=0}^{p-1}\left(\max\left(\frac{W}{ps^i}, \frac{D}{pb^i}\right)\right) = \max\left(\frac{W}{ps}, \frac{Dp}{pb}\right)$$

The roofline model: multicores

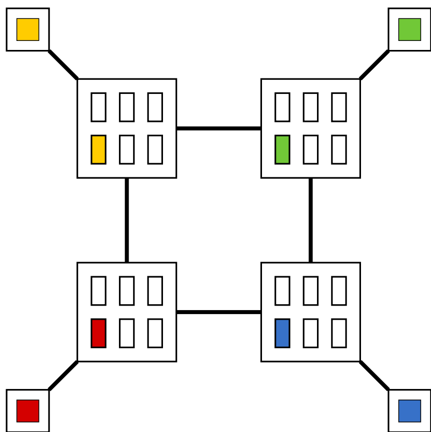
Case 1: bad data placement



The roofline model: multicores

Case 2: good data placement

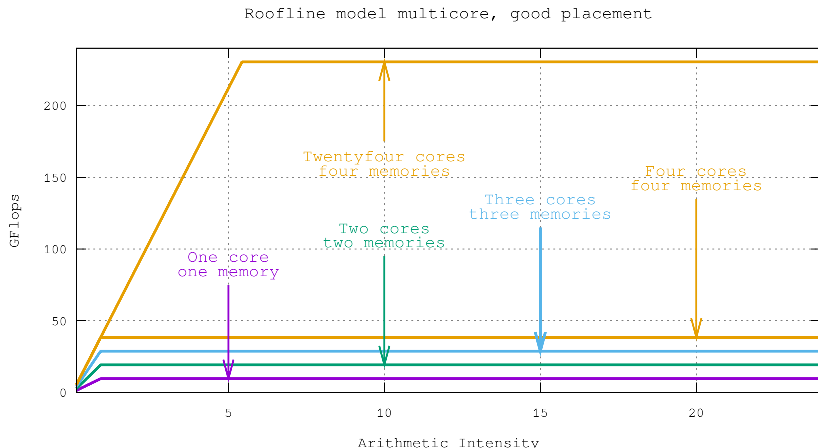
In this case each process reads/writes data from a different memory and, thus, has bandwidth $b^i = b$. If p is greater than the number of sockets, the bandwidth will be somehow shared depending on the placement of data and processes.



$$t = \max_{i=0}^{p-1} \left(\max \left(\frac{W}{ps^i}, \frac{D}{pb^i} \right) \right) = \max \left(\frac{W}{ps}, \frac{D}{pb} \right)$$

The roofline model: multicores

Case 2: good data placement



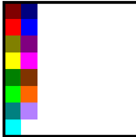
Threads and data placement

- threads: controlling the placement of threads depends on the threading technology in use: for example, with OpenMP the `OMP_PROC_BIND` and `OMP_PLACES` can be used.
- data: the data placement can be controlled in (at least) three different ways:
  - implicit: through the first touch rule (e.g., in linux). Data is allocated in the memory module closer to the first thread that touches it
  - explicit: using libraries such as `numactl`
  - interleaved: allocated memory is spread over multiple memory modules in pages in a round-robin fashion

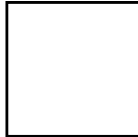
Interleaved memory allocation

Matrix

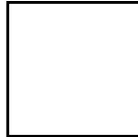
Normal allocation



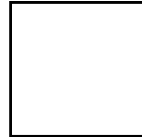
NUMA-0



NUMA-1



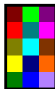
NUMA-2



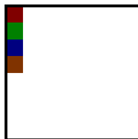
NUMA-3

Interleaved memory allocation

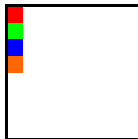
Matrix



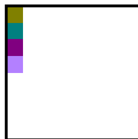
Interleaved allocation
(`numactl -i all`)



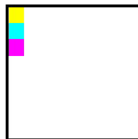
NUMA-0



NUMA-1



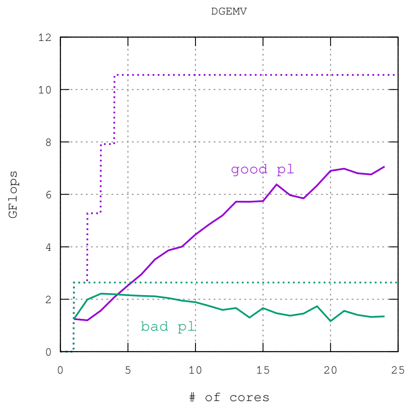
NUMA-2

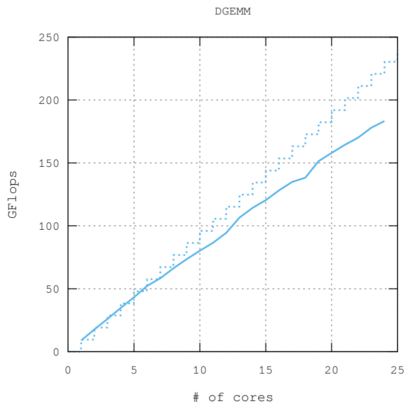


NUMA-3

Memory interleaving should provide a more uniform distribution of data that (supposedly) reduces conflicts and increases the average memory bandwidth

Multithreaded BLAS routines

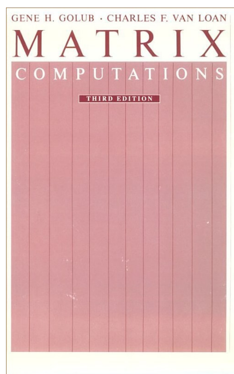




If the data is not correctly placed, it doesn't help much to use more cores for memory bound operations like `Level-2` BLAS. In any case performance is limited by the (aggregated) bandwidth for memory-bound operations.

Dense factorizations on multicores

Point QR factorization



The “Matrix Computations” book by Golub and Van Loan [8] presents the Householder algorithm for the QR factorization of a matrix A of size $m \times n$ as

for $k = 1 \ldots n$
  $[v_k, \tau_k] =$ **house**$(A(k:m, k))$
  $A(k:m, k:n) = (I_{m-k+1} - \tau_k v_k v_k^T) A(k:m, k:n)$
  if $k < m$
    $A(k+1:m, k) = v_k(2:m-k+1)$
  end
end

This algorithm decomposes the matrix A into the product of an orthogonal matrix Q and a triangular one R and can be used to solve linear systems of any shape. Its cost is (assuming $m \geq n$)

$$\sum_{k=1}^{n} 4(m-k+1)(n-k) = 2n^2(m - \frac{n}{3})$$

Householder QR

The V and R matrices replace the A matrix:

LAPACK point QR: dgeqr2

```
do  k = min(m,n)
   ! Generate elementary reflector H(k)
   ! to annihilate A(k+1:m,k)
   call _larfp(m-k+1, a( k, k ), a(min(k+1,m),k), &
             & 1, tau( k ) )
   ! Apply H(i) to A(k:m,k+1:n) from the left
   akk = a(k,k)
   a(k,k) = 1.d0
   call _larf('left', m-k+1, n-k, a(k,k), 1, tau(k), &
            & a(k,k+1), m, work )
   a(k,k) = ak
end do
```

Where the _larfp routine computes the Householder reflection and zeroes out the subdiagonal coefficients of the k-th column $A(k:m,j)$ and the _larf one applies the Householder reflection to the trailing submatrix $A(k:m,k:n)$.

The `_larfp` routine computes a Householder reflection $H = (I - \tau v v^T)$ with

$$v = \frac{x + sign(x(1))\|x\|_2 e_1}{x(1) + sign(x(1))\|x\|_2}, \quad \tau = \frac{sign(x(1))(x(1) - \|x\|_2)}{\|x\|_2}.$$

and, thus, **it is exclusively based on Level-1 BLAS operations**.

The `_larf` routine **does not** explicitly compute the Householder matrix $H = (I - \tau v v^T)$ but rather it applies it as follows

$$(I - \tau v v^T)A = (A - \tau(v(v^T A)))$$

and, therefore, **it is exclusively based on Level-2 BLAS operations**.

Block algorithms

The point QR algorithm we have seen so far only use `Level-1` and `2` routines: `_nrm2` (inside `_larfp`), `_gemv` (inside `_larf`) etc. As a results it cannot achieve a reasonable performance on modern processors with (deep) cache hierarchies.

Considerable performance improvements can be obtained if operations are rearranged in such a way that the factorization is achieved through the following steps:

1. factorize a small portion of the matrix using unblocked, `Level-2` BLAS code and accumulate the computed transformations. This small portion is commonly referred to as panel;
2. apply the accumulated transformations to the trailing submatrix at once using `Level-3` routines;
3. repeat steps 1 and 2 on the trailing submatrix.

These algorithms are commonly called block (or blocked) and are at the base of the LAPACK library.

Theorem (Compact WY representation [14])

Let $Q = H_1 ... H_{k-1} H_k$, with $H_i \in \mathbb{R}^{m \times m}$ an Householder transformation

$$H_k = (I - \tau_k v_k v_k^T)$$

and $k \leq m$. Then, there exist an upper triangular matrix $T \in \mathbb{R}^{k \times k}$ and a matrix $V \in \mathbb{R}^{m \times k}$ such that

$$Q = I + V_k T_k V_k^T.$$

Where:

$$T_k = \begin{bmatrix} T_{k-1} & -\tau_k T_{k-1} V_{k-1}^T v_k \\ 0 & -\tau_k \end{bmatrix}, \qquad V_k = \begin{bmatrix} V_{k-1} & v_k \end{bmatrix}$$

Block Householder QR

Take a matrix A of size $m \times n$ and partition it as below with A_{11} of size $b \times b$, $b \ll m, n$ and A_{22} of size $(m-b) \times (n-b)$
The basic step of the block QR factorization is

$$
{Q_1}^T \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] = \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & \tilde{A}_{22} \end{array} \right]
$$

achieved through the following operations [14]

$$
\left[\begin{array}{c} A_{11} \\ A_{21} \end{array} \right] \rightarrow \left[\begin{array}{c} V_{11} \\ V_{21} \end{array} \right], \ [R_{11}], \ [T_1]
$$

$$
\left[\begin{array}{c} R_{12} \\ \tilde{A}_{22} \end{array} \right] = \left(I - \left[\begin{array}{c} V_{11} \\ V_{21} \end{array} \right] \cdot \left[T_{11}^T \right] \cdot \left[V_{11}^T \, V_{21}^T \right] \right) \left[\begin{array}{c} A_{12} \\ A_{22} \end{array} \right]
$$

Block Householder QR

LAPACK block-column QR: _geqrf (more or less)

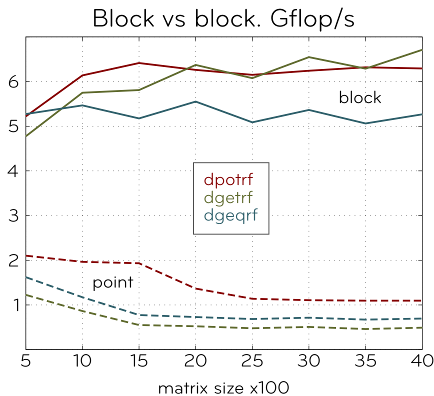
```
subroutine _geqrf(A, T, b)
   input      : A(m,n), b
   output     : A(m,n), T(b,n)

   do k=1, n, b
      call _geqrt (A(k:m,k:k+b-1), T(1:b,k:k+b-1),
                 & b)
      call _gemqrt(A(k:m,k:k+b-1), T(1:b,k:k+b-1),
                 & A(k:m,k+b:n), b)
   end do
```

The _gemqrt routine uses Level-3 operations such as _gemm and _trmm. The _geqrt is commonly referred to as **panel reduction** and the _gemqrt as **trailing submatrix update** (or, simply, update).

Block factorizations

Experimental results on AMD Opteron 2.4GHz:

- reference LAPACK from Netlib
- ACML BLAS



Fork-join multithreading

LAPACK block factorizations are essentially an interleaved sequence of `Level-2` (the panels reductions) and `Level-3` (the trailing submatrix updates) operations. The first are not parallelizable (to some extent), the second yes.

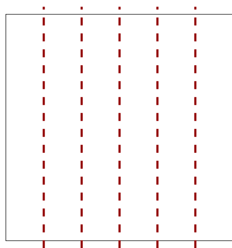
The naïve way of speeding up LAPACK factorizations on multicore systems is simply to link with a multithreaded BLAS. This is commonly referred to as fork-join parallelism.

Amdahl's law says this is not a good choice.

In the following we will focus on the QR factorization (without the hassle of pivoting) but the same holds for LU and Cholesky.

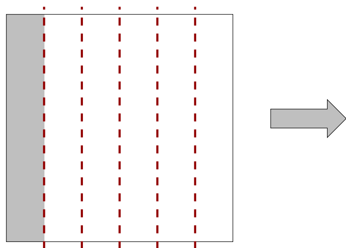
Fork-join multithreading

The forks and joins are more evident if the trailing submatrix update is explicitly parallelized with a 1d block-column partitioning



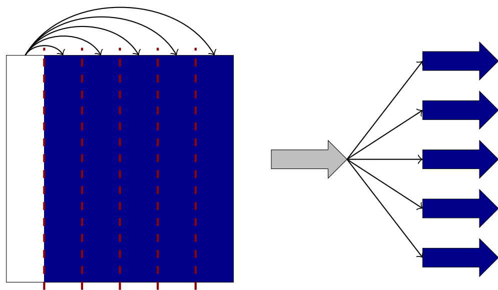
Fork-join multithreading

The forks and joins are more evident if the trailing submatrix update is explicitly parallelized with a 1d block-column partitioning



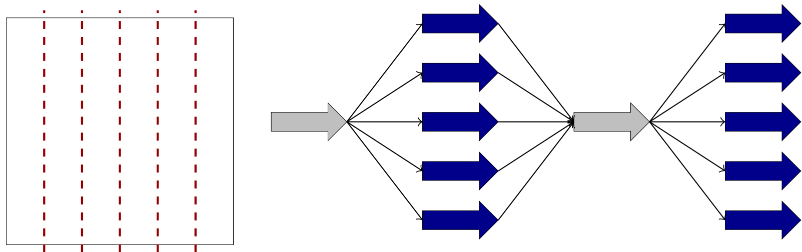
Fork-join multithreading

The forks and joins are more evident if the trailing submatrix update is explicitly parallelized with a 1d block-column partitioning



Fork-join multithreading

The forks and joins are more evident if the trailing submatrix update is explicitly parallelized with a 1d block-column partitioning



Fork-join multithreading

Here is a simple fork-join parallelization for the case where the trailing submatrix is also split into block-columns of size b (with $q = n/b$ and $p = m/b$):

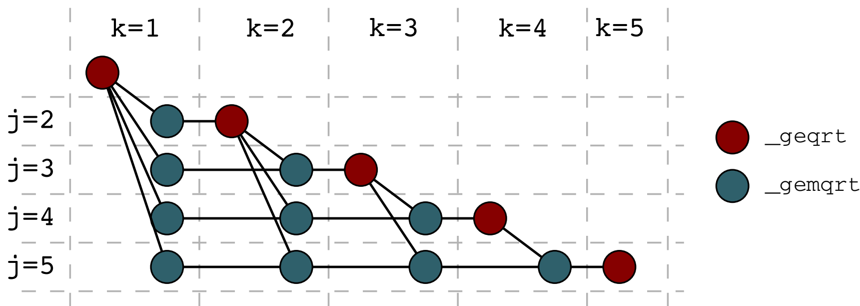
Fork-join block QR

```
subroutine V,R = block_col_qr(A)
   input: A(m,n)      output: V(m,n), R(n,n)

   do k=1, q
      V[1,k], R[1,k] = _geqrt(A[1,k])
      !$omp parallel do
      do j=k+1,q
         A[1,j] = _gemqrt(V[1,k], A[1,j])
      end do
   end do
```

Simplified notation: `A[1,k]` denotes the k-th block column. Also, note that on output V and R overwrite A. The T matrix is omitted for clarity.

DAG and critical path

The dependency graph for the previous algorithm (assuming 5 block-columns) is

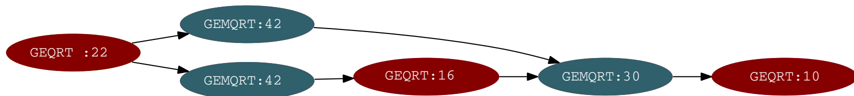


- Critical Path: the longest directed path between input and output nodes
- Critical Path Length: the sum of weights along the critical path
- Average Degree of Concurrency: $\frac{\sum_{i\in DG} w_i}{\sum_{i\in CP} w_i}$

Fork-join multithreading: performance model

- A panel of size $m \times b$ takes $2b^2(m - \frac{b}{3})$ flops (or $2(3p-1)$ time units of $b^3/3$)
- An update of size $m \times b$ takes $3b^3 + 4(m-b)b^2$ flops (or $3(4p-2)$ time units)

Assume a matrix of size $m \times n = p * b \times q * b$ with $p = 4,\ q = 3$



Best possible speedup is

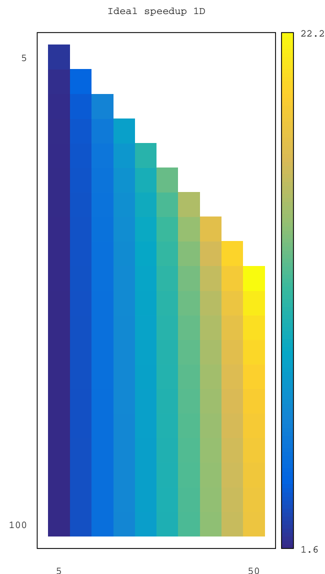
$$\frac{\sum_{i \in DG} w_i}{\sum_{i \in CP} w_i} = \frac{162}{120} = 1.35$$

Fork-join multithreading: performance model

For a matrix of size $p \times q$ (remember, $p = m/b$ and $q = n/p$ with b being the block-column size), the best possible speedup with the fork-join approach is

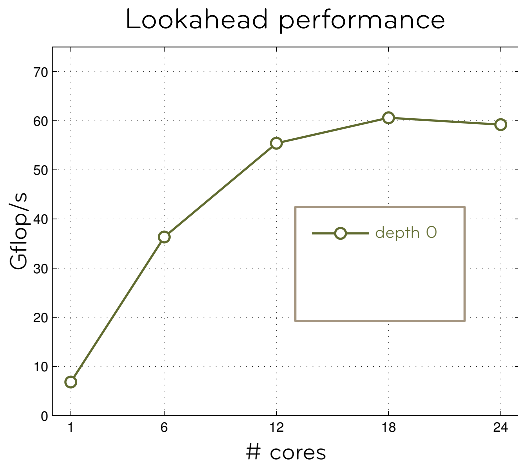
$$\frac{\sum_{k=0}^{q-1} 2\left(3\left(p-k\right)-1\right)+\left(q-k-1\right)\left(3\left(4\left(p-k\right)-1\right)\right)}{\sum_{k=0}^{q-1} 2\left(3\left(p-k\right)-1\right)+\left(3\left(4\left(p-k\right)-1\right)\right)}$$

$$= \frac{6pq^2 - 2q^3}{18pq - 9q^2}$$



Fork-join multithreading

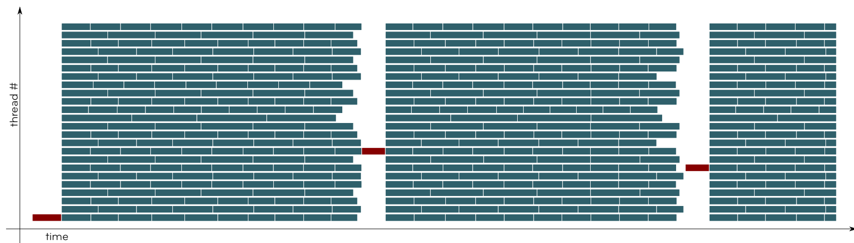
As expected, this actually does not lead to great performance.



The achieved speedup is less than 9 on 24 cores.

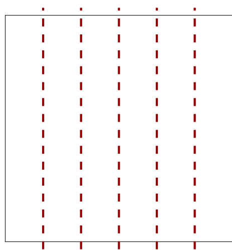
Fork-join multithreading

The poor scalability is due to the fact that all the threads but one are idle when the panel reduction is performed.



Lookahead

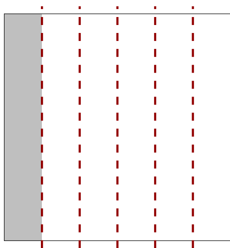
It has to be noted that the reduction of panel $k+1$ can be done independently of the update of all the columns $k+2, ..., n$ with respect to panels $1, ..., k$.
Therefore, as soon as the update of column $k+1$ with respect to panel k is done, the reduction of panel $k+1$ can be executed and, possibly, overlapped with other update operations.



The panel reduction is not parallelized but rather hidden behind update operations

Lookahead

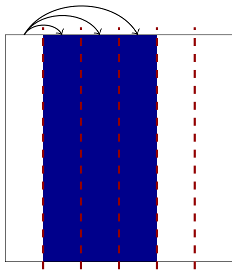
It has to be noted that the reduction of panel $k+1$ can be done independently of the update of all the columns $k+2, ..., n$ with respect to panels $1, ..., k$.
Therefore, as soon as the update of column $k+1$ with respect to panel k is done, the reduction of panel $k+1$ can be executed and, possibly, overlapped with other update operations.



The panel reduction is not parallelized but rather hidden behind update operations

Lookahead

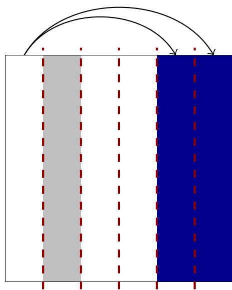
It has to be noted that the reduction of panel $k+1$ can be done independently of the update of all the columns $k+2, ..., n$ with respect to panels $1, ..., k$.
Therefore, as soon as the update of column $k+1$ with respect to panel k is done, the reduction of panel $k+1$ can be executed and, possibly, overlapped with other update operations.



The panel reduction is not parallelized but rather hidden behind update operations

Lookahead

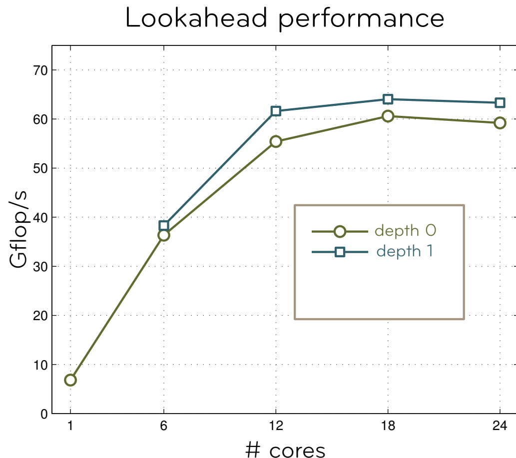
It has to be noted that the reduction of panel $k+1$ can be done independently of the update of all the columns $k+2, ..., n$ with respect to panels $1, ..., k$.
Therefore, as soon as the update of column $k+1$ with respect to panel k is done, the reduction of panel $k+1$ can be executed and, possibly, overlapped with other update operations.



The panel reduction is not parallelized but rather hidden behind update operations

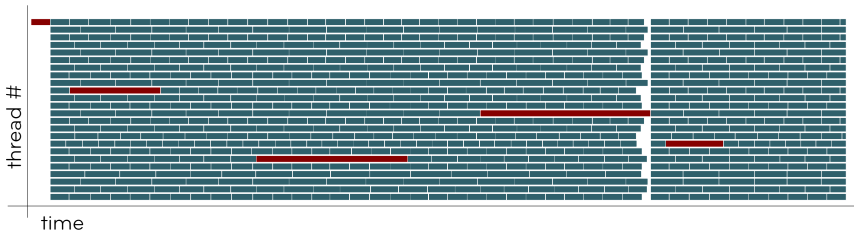
1D multithreading with lookahead

This technique is called lookahead [15] and actually gives better performance



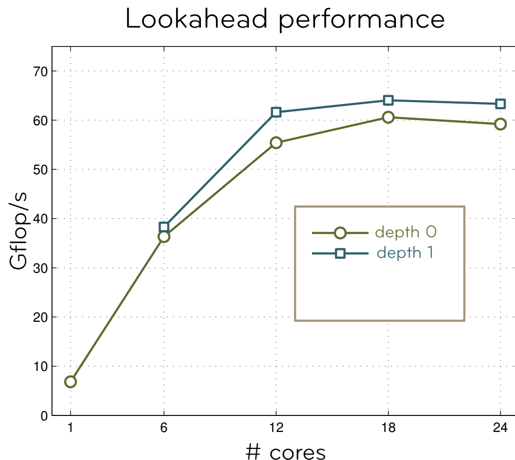
1D multithreading with lookahead

The overlap between panel reductions and updates is clear in the execution traces



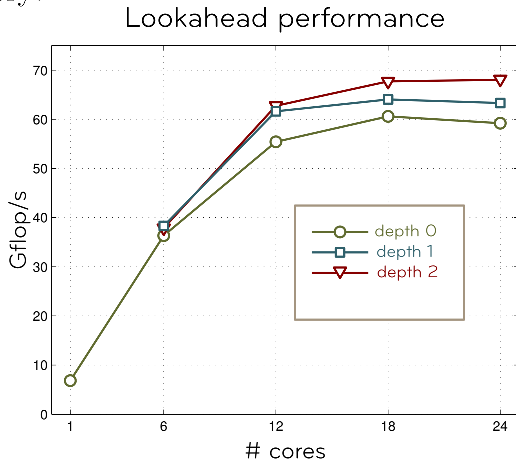
1D multithreading with lookahead

Lookahead can be seen as a way to pipeline different iterations of the outer loop of a factorization. Multiple iterations can be pipelined and their number is commonly referred to as lookahead depth.



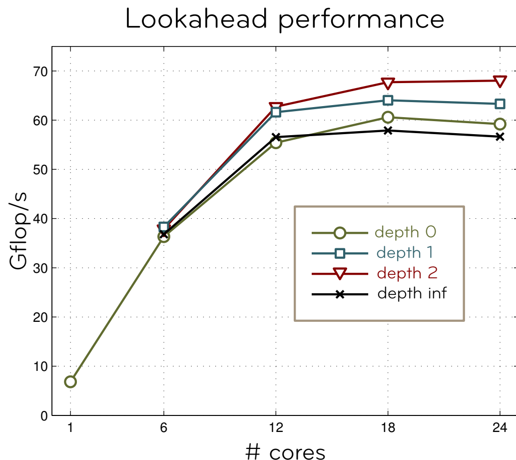
1D multithreading with lookahead

A depth 2 gives even slightly better performance. Can we have more? In distributed memory the lookahead depth had to be limited to avoid excessive memory consumption. What about shared memory?



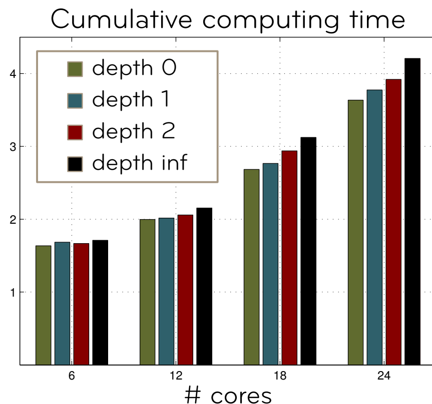
1D multithreading with lookahead

An infinite depth lookahead does not yield good results. why does this happen?



1D multithreading with lookahead

The reason for the poor performance with infinite lookahead lies in cache locality. When many iterations are pipelined, panels are moved in and out of cache memories depending on what update is performed.



The cumulative computing time is the total time spent in all tasks (i.e., `_geqrt` and `_gemqrt`).

1D multithreading with lookahead

The previous graph shows two important facts

1. The more cores are used, the higher is the cumulative times, i.e., the slower each single operation is. This is, again, due to data locality (either in caches or in NUMA memory modules). This mostly happens because cores operate on data that do not necessarily reside in their own memory hierarchy.
2. The deeper the lookahead is, the higher is the cumulative times, i.e., the slower each single operation is. This is due to the fact that with shallow lookaheads panels has a higher chance of staying in cache memories and of being reused for multiple update operations.

1D block-column with infinite lookahead

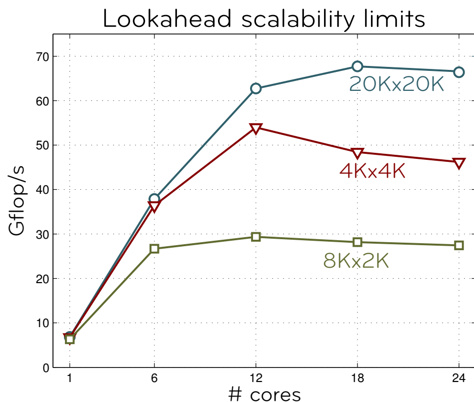
Here is a parallel QR factorization by block-columns with infinite lookahead based on the OpenMP `task` construct:

block-column QR with ∞ lookahead

```
subroutine V,R = block_col_qr(A)
   input: A(m,n)      output: V(m,n), R(n,n)

   !$omp parallel private(j,k)
   !$omp master
   do k=1, q
      !$omp task depend(in:A[1,k])
      !$omp&     depend(out:V[1,k],R[1,k])
      V[1,k], R[1,k] = _geqrt(A[1,k])
      do j=k+1,q
         !$omp task depend(in:V[1,k]) &
         !$omp&     depend(inout:A[1,j])
         A[1,j] = _gemqrt(V[1,k], A[1,j])
      end do
   end do
   !$omp end master
   !$omp end parallel
```

The limits of 1D parallelism

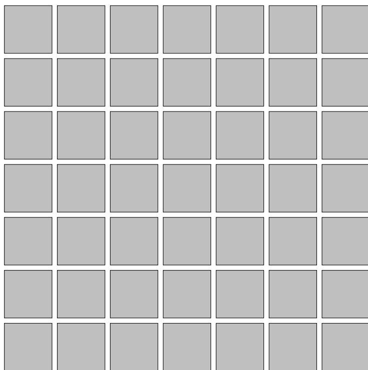
1D parallelization does not work well on small matrices or on *tall-and-skinny* (i.e., strongly overdetermined) ones



This is due to the fact that a 1D partitioning does not deliver enough concurrency in these cases.

The limits of 1D parallelism

This can be overcome by applying to the matrix a 2D partitioning, commonly referred to as 2D block partitioning or, less frequently, tile partitioning



Tile QR

The tile QR algorithm based on the 2D block decomposition [16] can be roughly described like this:

1. At step k the QR factorization of the diagonal tile is done.
2. The previous transformation modifies all the other tiles along the k-th row.
3. then the diagonal tile (which is now an upper triangle) is used to annihilate all the subdiagonal tiles i one at a time.
4. Each of the previous operations also modifies the tiles along rows k and i

Tile QR

Assume the previously described _geqrt and _gemqrt Define the following two new kernels

$$R_{12},\ V_{12} = \texttt{_tsqrt}(R_1,\ A_2), \quad where \quad Q_{12}R_{12} = \begin{pmatrix} R_1 \\ A_2 \end{pmatrix}$$

Computes the QR factorization of a matrix formed by a triangle on top of a square. V_{12} is a $b \times b$ square block containing the Householder vectors that form Q_{12}.

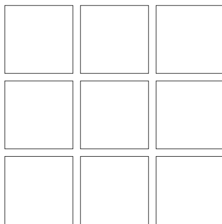
$$\tilde{B}_1,\ \tilde{B}_2 = \texttt{_tsmqrt}(B_1,\ B_2,\ V_{12}), \quad where \quad \begin{pmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{pmatrix} = Q_{12}^T \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

Applies the transformation computed by _tsqrt to a matrix formed by two squares, one on top of the other.

Tile QR

The (sequential) tile QR factorization proceeds like this

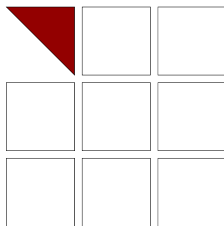
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

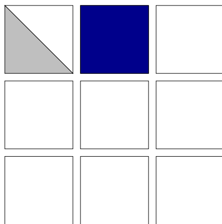
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

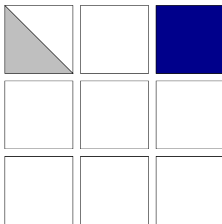
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

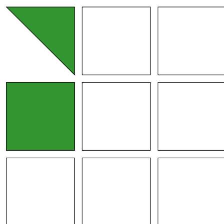
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

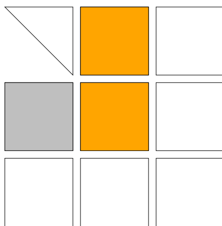
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

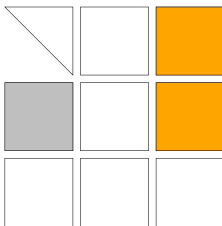
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

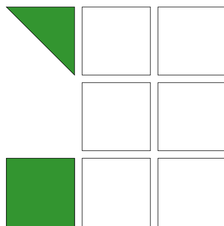
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

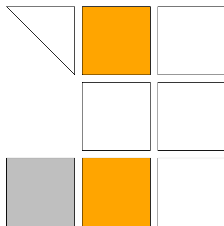
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

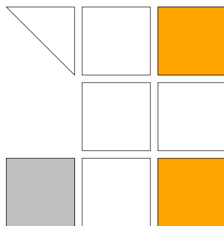
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

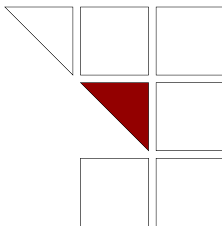
The (sequential) tile QR factorization proceeds like this

1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt`: update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt`: update rows k and i wrt `_tsqrt`



The (sequential) tile QR factorization proceeds like this

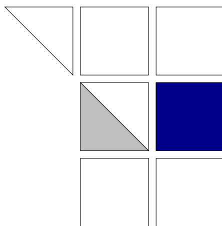
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

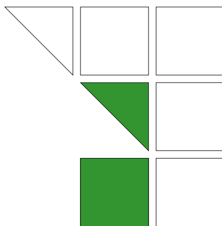
1. `_geqrt` : factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

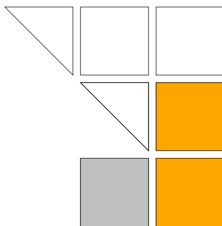
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

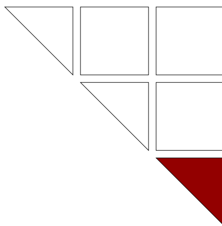
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

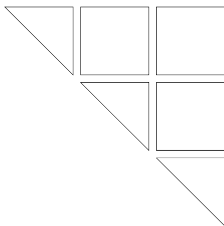
1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

1. `_geqrt :` factorize the k-th diagonal tile
2. `_gemqrt:` update the k-th row wrt the `_geqrt`
3. `_tsqrt :` use the diagonal tile to kill a subdiagonal one on row i, column k
4. `_tsmqrt:` update rows k and i wrt `_tsqrt`



Tile QR

Sequential tile QR with flat tree

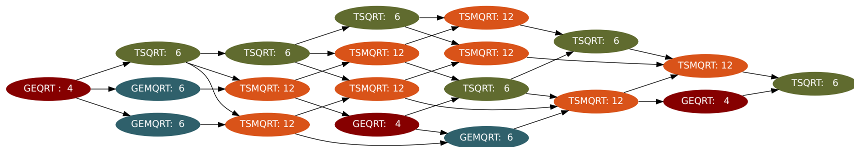
```
subroutine tiled_qr_flattree(A)
   input: A(m,n)      output: V(m,n), R(n,n)

   do k=1, q
      V[k,k], R[k,k] = _gemqrt(A[k,k])

      do j=k+1, q
         A[k,j] = _gemqrt(A[k,j], V[k,k])
      end do

      do i=k+1, p
         V[i,k], R[k,k] = _tsqrt(R[k,k], A[i,k])
         do j=k+1, q
            A[k,j], A[i,j] = _tsmqrt(A[k,j], A[i,j], V[i,k])
         end do
      end do
   end do
```

Tiled QR: critical path analysis

Tasks costs in $b^3/3$:

- `_geqrt` : 4
- `_gemqrt`: 6
- `_tsqrt` : 6
- `_tsmqrt`: 12

For a matrix of size $m \times n = p * b \times q * b$ with $p = 4,\ q = 3$ the dependency graph is

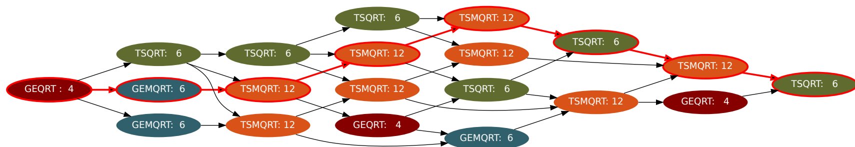


Tiled QR: critical path analysis

Tasks costs in $b^3/3$:

- `_geqrt :` 4
- `_gemqrt:` 6
- `_tsqrt :` 6
- `_tsmqrt:` 12

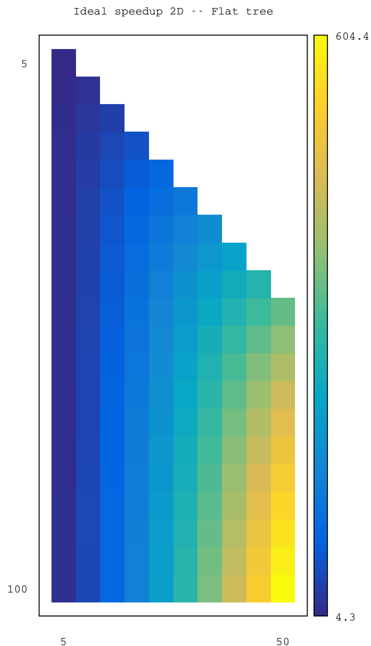
For a matrix of size $m \times n = p * b \times q * b$ with $p = 4, \ q = 3$ the dependency graph is



and the best achievable speedup is

$$\frac{\sum_{i \in DG} w_i}{\sum_{i \in CP} w_i} = \frac{162}{70} = 2.31$$

Tiled QR: critical path analysis



Assuming $q \leq p$

$$\frac{\sum_{k=1}^{q} 4 + 6(q-k) + 6(p-k) + 12(p-k-1)(q-k)}{4 + 6 + 12(p-2) + (12+6)(q-1)}$$

Much better theoretical speedups with respect to the 1D parallelization.

Tile QR

The tile algorithm exposes much more concurrency than the block-column one thanks to the 2D data (and workload) partitioning.

However, its execution pattern is much more complex which renders the parallelization more difficult. One idea, that has recently received great interest, consists in defining a task for each of the elementary operations calls, and then arranging all these tasks into a Directed Acyclic Graph (DAG) which is nothing more than the graph of dependencies.

Modern tools exists that automatically build the DAG and use it for dynamically scheduling its tasks to the available processing units. These tools are also commonly called runtime execution engines and are based on what we normally refer to as a Data-Flow or Task-Flow programming models. Examples of such tools are QUARK (used in the PLASMA [17] project), Parsec, StarPU or even OpenMP 4.0.

Parallel tile QR with flat tree

```
subroutine tiled_qr_flattree(A)
   input: A(m,n)      output: V(m,n), R(n,n)

   !$omp parallel private(i,j,k)
   !$omp master
   do k=1, q
      !$omp task depend(inout:A[k,k])
      V[k,k], R[k,k] = _gemqrt(A[k,k])

      do j=k+1, q
         !$omp task depend(in:V[k,k]) depend(inout:A[k,j])
         A[k,j] = _gemqrt(A[k,j], V[k,k])
      end do

      do i=k+1, p
         !$omp task depend(in:R[k,k], A[i,k])
         !$omp&     depend(out:R[k,k], V[i,k])
         V[i,k], R[k,k] = _tsqrt(R[k,k], A[i,k])
         do j=k+1, q
            !$omp task depend(in:V[i,k])
            !$omp&     depend(inout:A[k,j],A[i,j])
            A[k,j], A[i,j] = _tsmqrt(A[k,j], A[i,j], V[i,k])
         end do
      end do
   end do
   !$omp end master
   !$omp end parallel
```

Tile QR

The DAG expresses the dependencies between the tasks:
a task can be executed only when all the other tasks to which it is connected by an incoming edge are completed

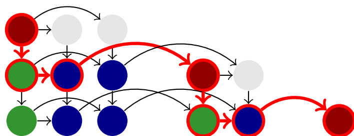
Based on this principle, a task whose dependencies are satisfied can be dynamically assigned to a running thread which then takes care of executing the corresponding job.

Threads work asynchronously and no synchronizations (other than those in the DAG) are imposed by the execution model. This minimizes the threads idle time.

Tile QR

All the classical methods for scheduling the tasks in a dependency graph can be used for minimizing the makespan (execution time) based on a number of criteria; e.g.

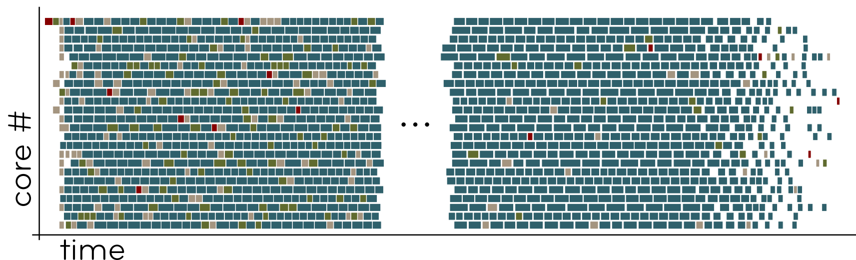
- assign higher priority to tasks which lie along the critical path



- define a loose coupling between threads and tasks in order to improve data locality
- assign tasks to preferred threads and employ work stealing techniques in case of starvation

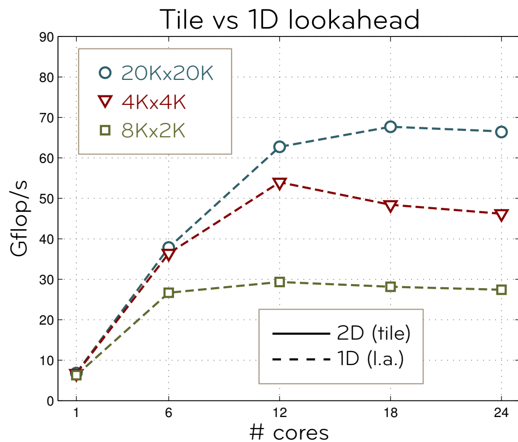
Tile QR

As shown by the execution traces, the threads idle time is reduced (apart from the tail)



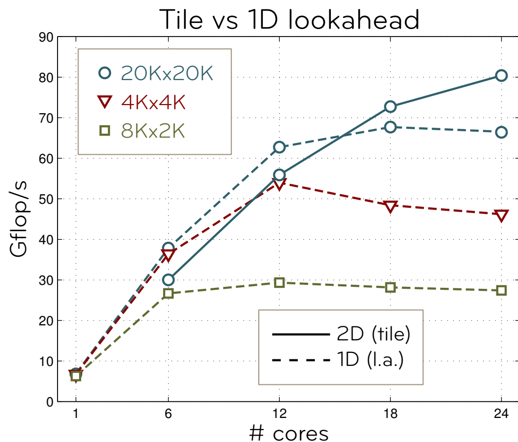
The execution trace is much more dense (almost no idle time). An extra overhead is added to handle the increased number of tasks.

Tile QR



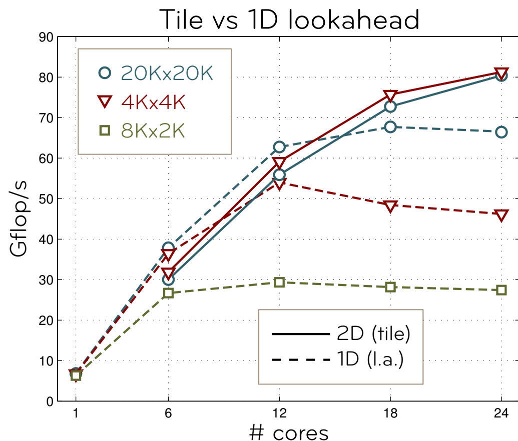
Tile QR

- on the big, square matrix tile QR starts off slower but then scales better



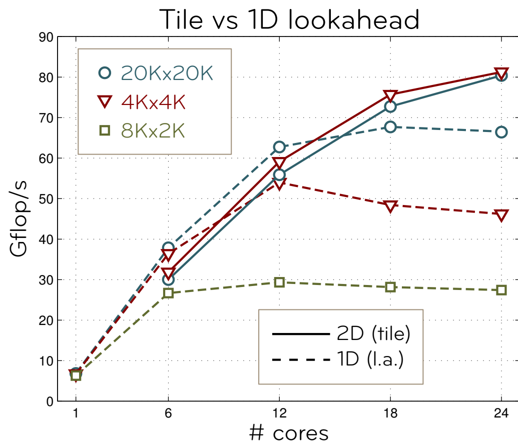
Tile QR

- on the big, square matrix tile QR starts off slower but then scales better
- tile QR has the same (better?) performance and scalability on the smaller matrix



Tile QR

- on the big, square matrix tile QR starts off slower but then scales better
- tile QR has the same (better?) performance and scalability on the smaller matrix



In both cases the better performance/scalability can be explained with the increased concurrency exposed by the tile algorithm

Tile QR

You may have noticed hat all the tasks related to tiles along a column form a chain in the DAG. This means that their execution is serialized. This is only slightly better than what we had with a 1D partitioning mostly because the reduction/update on a column is not parallelized but only decomposed in multiple tasks that can be interleaved with others.

Therefore, we can still expect that this version of the tile algorithm suffers on tall and skinny matrices.

Thanks to the great flexibility of Householder transformations, it is possible to devise different algorithms that are more suited to this kind of matrices [18]

Tile QR

Introduce the two following kernels:

$$R_{12},\ \ V_{12}, = \texttt{_ttqrt}(R_1,\ R_2), \quad where \quad Q_{12}R_{12} = \left(\begin{array}{c} R_1 \\ R_2 \end{array} \right)$$

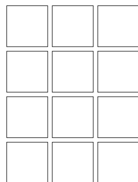
Computes the QR factorization of a matrix formed by a triangle on top of another triangle. V_{12} is a $b \times b$ triangular block containing the Householder vectors that form Q_{12}.

$$\tilde{B}_1,\ \tilde{B}_2 = \texttt{_ttmqrt}(B_1,\ B_2,\ V_{12}), \quad where \quad \left(\begin{array}{c} \tilde{B}_1 \\ \tilde{B}_2 \end{array} \right) = Q_{12}^T \left(\begin{array}{c} B_1 \\ B_2 \end{array} \right)$$

Applies the transformation computed by `_ttqrt` to a matrix formed by two squares, one on top of the other.

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



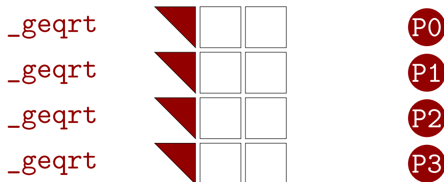
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



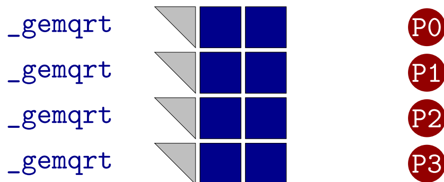
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



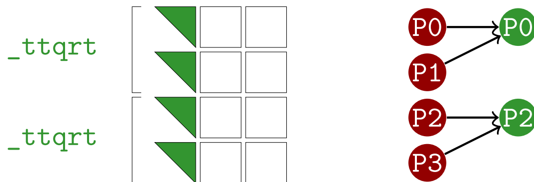
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



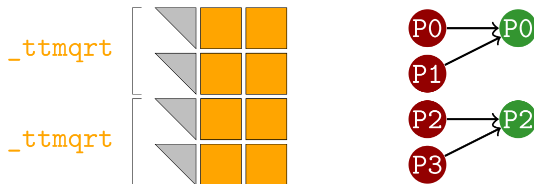
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



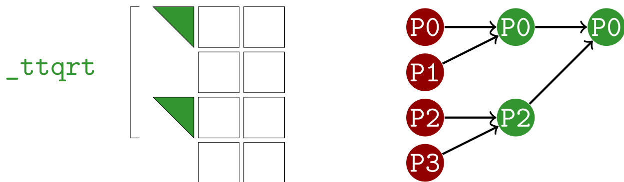
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



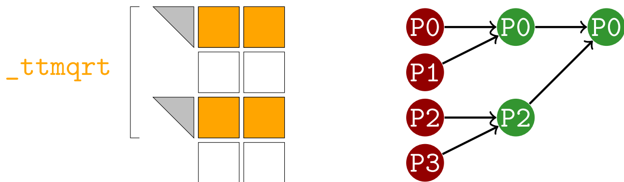
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



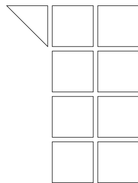
This allows for :

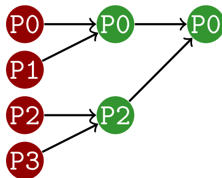
1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

Tile QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently





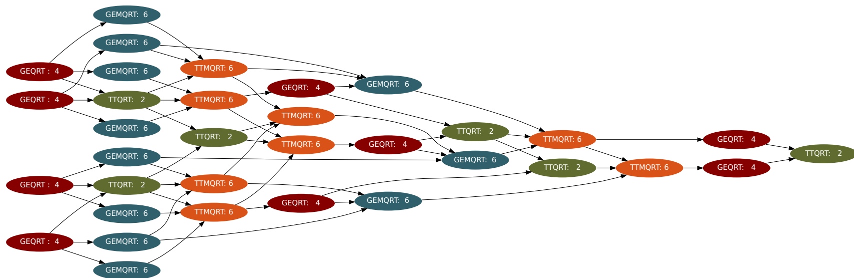
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `_ttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `_ttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $p \gg q$)

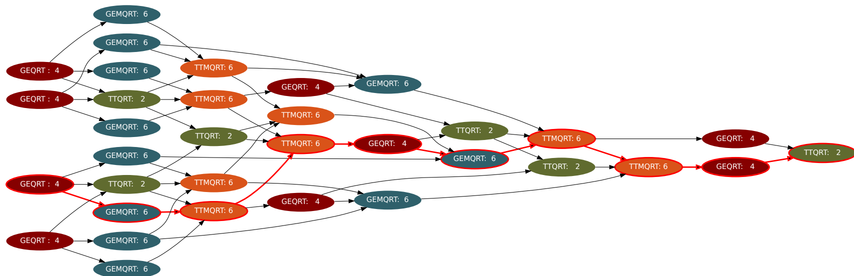
Critical path analysis

`_ttqrt` and `_ttmqrt` take, respectively 2 and 6 time units
For a matrix of size $m \times n = p * b \times q * b$ with $p = 4, \; q = 3$ the dependency graph is



Critical path analysis

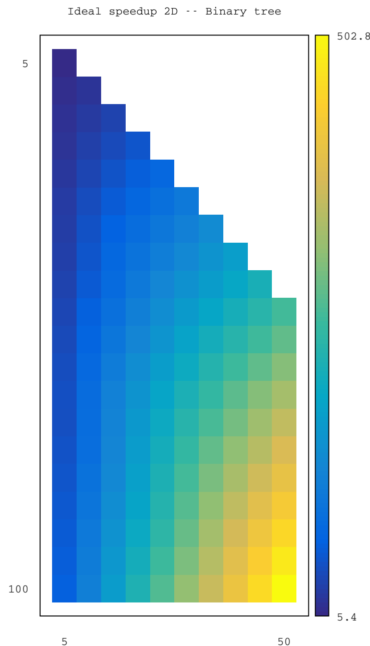
`_ttqrt` and `_ttmqrt` take, respectively 2 and 6 time units
For a matrix of size $m \times n = p * b \times q * b$ with $p = 4,\ q = 3$ the dependency graph is



and the best achievable speedup is

$$\frac{\sum_{i \in DG} w_i}{\sum_{i \in CP} w_i} = \frac{162}{50} = 3.24$$

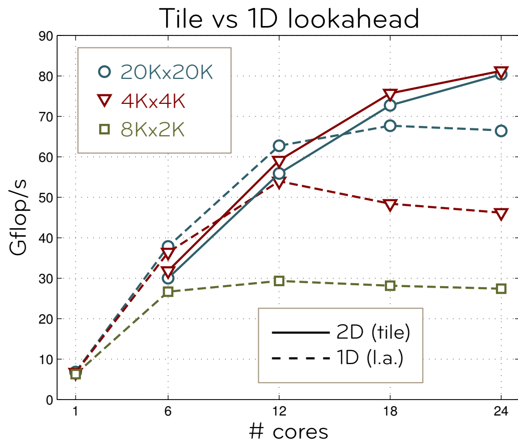
Critical path analysis



The binary panel reduction provides better parallelism for extremely overdetermined matrices but suffer from poor pipelining of successive panel stages on less overdetermined ones. Moreover, the `_ttqrt` and `_ttmqrt` are less efficient than the `_tsqrt` and `_tsmqrt` ones because they involve two triangular blocks.

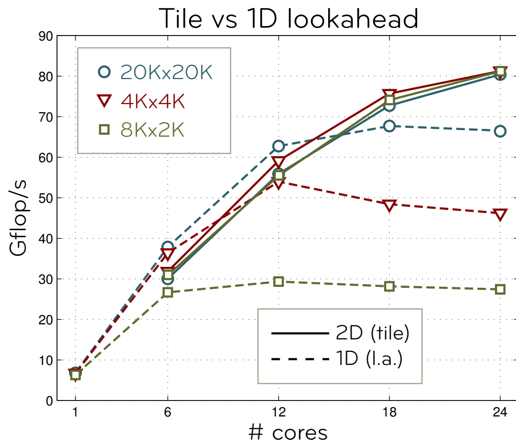
Tile QR

Note that different approaches are possible and that the reduction tree can have different shapes.
Using these algorithms the tile QR algorithm scales on tall and skinny matrices as well as on square ones



Tile QR

Note that different approaches are possible and that the reduction tree can have different shapes.
Using these algorithms the tile QR algorithm scales on tall and skinny matrices as well as on square ones



Performance Analysis

Performance evaluation

The performance of a complex, parallel algorithm is hard to evaluate, especially on a complex architecture.

The classical way to evaluate the performance and scalability of a parallel code are the speedup and the parallel efficiency:

$$S = \frac{t^{-}(1)}{t^{=}(p)}, \qquad e = \frac{t^{-}(1)}{t^{=}(p) \times p}$$

where $t^{-}(1)$ is the time to run the best sequential algorithm on 1 process and $t^{=}(p)$ is the time to execute the parallel algorithm on p processes.

This metrics, however do not contain much information and are not useful to identify the factors that limit the performance and scalability of the code.

Performance evaluation

In order to achieve a more detailed performance analysis, we can observe that (independently of the algorithm)

$$t(p) = \frac{t_t(p) + t_i(p) + t_c(p) + t_o(p)}{p}$$

where

- $t_t(p)$: The total time spent in tasks which represent the workload of the application.
- $t_i(p)$: The total idle time spent waiting for dependencies between tasks to be satisfied. $t_i(1) := 0$
- $t_c(p)$: The total time spent performing explicit communications that are not overlapped by computations. This corresponds to the time spent by processes waiting for data to be transferred on their associated memory node before being able to execute a task. $t_c(1) := 0$
- $t_o(p)$: this is the total overhead for handling parallelism (e.g., create and schedule tasks). $t_o(1) := 0$

Performance evaluation

$$e(p) = \frac{t^{-}(1)}{t^{=}(p) \times p} = \frac{t^{-}(1)}{t_t^{=}(p) + t_o^{=}(p) + t_c^{=}(p) + t_i^{=}(p)} =$$

$$\underbrace{\frac{t^{-}(1)}{t_t^{=}(1)}}_{e_a} \cdot \underbrace{\frac{t_t^{=}(1)}{t_t^{=}(p)}}_{e_t} \cdot \underbrace{\frac{t_t^{=}(p)}{t_t^{=}(p) + t_o^{=}(p)}}_{e_o} \cdot \underbrace{\frac{t_t^{=}(p) + t_o^{=}(p)}{t_t^{=}(p) + t_o^{=}(p) + t_c^{=}(p)}}_{e_c} \cdot \underbrace{\frac{t_t^{=}(p) + t_o^{=}(p) + t_c^{=}(p)}{t_t^{=}(p) + t_o^{=}(p) + t_c^{=}(p) + t_i^{=}(p)}}_{e_p}$$

where:

- e_a: the **algorithm efficiency**, which measures how the overall efficiency is reduced by the data partitioning and the use of parallel algorithms.
- e_t: the **task efficiency**, measures the loss of tasks efficiency due to parallelism (memory contention, cache, NUMA etc.).
- e_o: the **overhead efficiency**, which measures the cost of the overhead with respect to the actual work done.
- e_c: the **communication efficiency**, which measures the cost of communications with respect to the actual work done due to data transfers between workers.
- e_p: the **pipeline efficiency**, which measures how much concurrency is available and how well it is used.

Performance evaluation: observations

$$e_a = \frac{t^-(1)}{t_t^=(1)}$$

The algorithm used for a sequential execution is commonly different than the one used for a parallel execution. Also, no data and/or operations partitioning is commonly applied in a sequential execution, whereas in parallel data and/or operations are partitioned to achieve better concurrency. Therefore $t_p^-(1) \leq t_t^=(1)$ because:

- the parallel algorithm may perform more operations if it is beneficial for parallelism
- in the parallel algorithm operations are done at a smaller granularity which implies lower tasks efficiency

Performance evaluation: observations

$$e_t = \frac{t_t^{=}(1)}{t_t^{=}(p)}$$

When an algorithm is executed sequentially, all the data are stored in its own memory (or NUMA module) and, therefore, can be accessed fast. When the algorithm is executed in parallel on a shared memory machine, the time of each single task may increase because of poorer cache behavior, remote accesses in the NUMA system (i.e., implicit communications) and contentions on the shared memory.

$$e_o = \frac{t_t^{=}(p)}{t_t^{=}(p) + t_r^{=}(p)}$$

Commonly the overhead is relatively small but, for example, if tasks are of very small granularity the relative cost for handling them grows and thus this efficiency is lesser.

Performance evaluation: observations

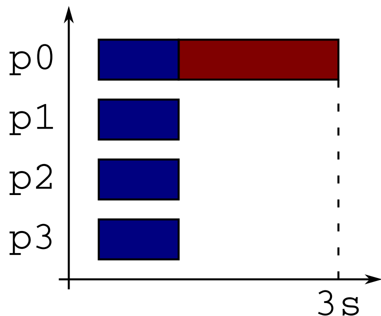
$$e_c = \frac{t_t^{=}(p) + t_r^{=}(p)}{t_t^{=}(p) + t_r^{=}(p) + t_c^{=}(p)}$$

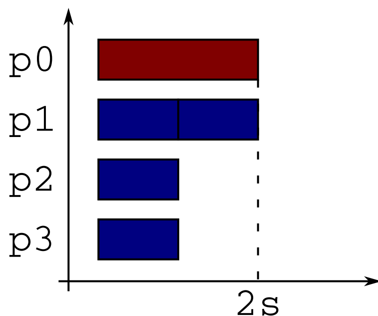
Explicit communications obviously reduce the performance and scalability of a code wrt the sequential execution. If the data distribution is bad, this cost may become dominant. In shared memory, multithreaded parallelism, there are no explicit communications (i.e., $t_c(p) = 0$) and thus we will ignore this metric here. Note that there are implicit communications in the NUMA system which decrease the tasks efficiency e_t.

Performance evaluation: observations

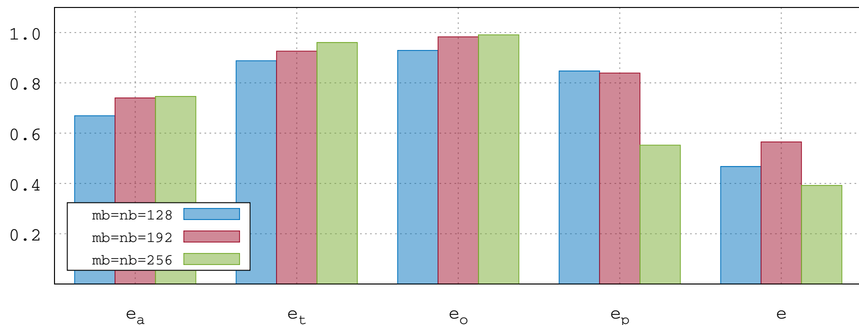
$$e_p = \frac{t_t^{=}(p) + t_r^{=}(p) + t_c^{=}(p)}{t_t^{=}(p) + t_r^{=}(p) + t_c^{=}(p) + t_i^{=}(p)}$$

If, for example, we have a DAG which is a chain, this efficiency will be very small no matter what, because only one process will be working at any time and all the others will be idle. If, however, concurrency is available, this efficiency can still be low because of a bad scheduling policy.





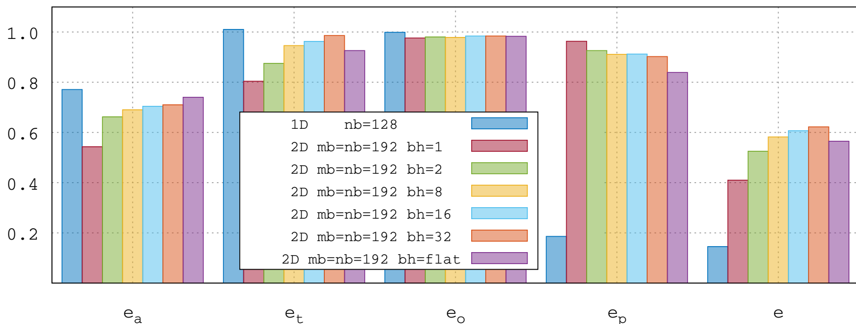
Performance evaluation: experiments



Increasing the block size

- improves e_a and e_t because of the large granularity
- reduces the overhead because there are fewer tasks and of bigger size
- reduces e_p because of lower concurrency available

Performance evaluation: experiments



- 1D has better e_a, e_t and e_o because of the larger granularity
- 2D has much much better e_p because of the higher concurrency available
- a smaller **bh** gives more concurrency (better e_p) but lower e_a and e_t

[1] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: Communications of the ACM 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[2] C. L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: ACM Transactions on Mathematical Software 5 (1979), pp. 308–323.

[3] J. J. Dongarra et al. “An extented set of Fortran Basic Linear Algebra Subprograms”. In: ACM Transactions on Mathematical Software 14 (1988), 17 and 18–32.

[4] J. J. Dongarra et al. “A set of level 3 basic linear algebra subprograms”. In: ACM Trans. Math. Softw. 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: http://doi.acm.org/10.1145/77626.79170.

[5] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: Parallel Computing 27.1–2 (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps), pp. 3–35.

[6] Kazushige Goto and Robert A. van de Geijn. “Anatomy of high-performance matrix multiplication”. In: ACM Trans. Math. Softw. 34.3 (May 2008), 12:1–12:25. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: http://doi.acm.org/10.1145/1356052.1356053.

References II

[7] K. Goto and R. van de Geijn.
On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note 9. 2002. URL: citeseer.ist.psu.edu/goto02reducing.html.

[8] G. H. Golub and C. F. Van Loan. Matrix Computations. 3rd ed. Baltimore and London: The Johns Hopkins University Press, 1996.

[9] N. J. Higham. Accuracy and Stabilty of Numerical Algorithms. 2. Philadelphia: SIAM, 2002.

[10] L.N. Trefethen and D.I. Bau. Numerical linear algebra. Society for Industrial and Applied Mathematics, 1997. ISBN: 9780898714876.

[11] G. W. Stewart. Matrix Algorithms: Volume 1, Basic Decompositions. Society for Industrial and Applied Mathematics, 1998. ISBN: 0898714141.

[12] Jack J. Dongarra et al.
Numerical Linear Algebra for High-Performance Computers. Philadelphia: SIAM Press, 1998.

[13] James W. Demmel. Applied numerical linear algebra. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-389-7.

References III

[14] Christian Bischof and Charles van Loan. “The WY representation for products of householder matrices”. In: SIAM J. Sci. Stat. Comput. 8.1 (Jan. 1987), pp. 2–13. ISSN: 0196-5204. DOI: 10.1137/0908009. URL: http://dx.doi.org/10.1137/0908009.

[15] Peter. Strazdins, Australian National University., and Australian National University. A comparison of lookahead and algorithmic blocking techniques for parallel matrix fa English. Australian National University Canberra, 1998, 15 p. :

[16] A. Buttari et al. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In: Parallel Computing 35.1 (2009), pp. 38–53. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2008.10.002.

[17] E. Agullo et al. “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects”. In: Journal of Physics: Conference Series 180.1 (2009), p. 012037.

[18] B. Hadri et al. “Tile QR factorization with parallel panel processing for multicore architectures”. In: IPDPS. IEEE, 2010, pp. 1–10. URL: http://dx.doi.org/10.1109/IPDPS.2010.5470443.