Parallel programming with OpenMP

A. Buttari (CNRS, IRIT)

January 31, 2022

Introduction to multicores

Outline

Why multicores? the three walls

What is the reason for the introduction of multicores?
Uniprocessors performance is leveling off due to the *"three walls"*:

- ILP wall: Instruction Level Parallelism is near its limits
- Memory wall: caches show diminishing returns
- Power wall: power per chip is getting painfully high

The ILP wall

There are two common approaches to exploit ILP:

- Vector instructions (SSE, AltiVec etc.)
- Out-of-order issue with in-order retirement, speculation, register renaming, branch prediction etc.

Neither of these can generate much concurrency because:

- irregular memory access patterns
- control dependent computations
- data dependent memory access

Multicore processors, on the other side, exploit Thread Level Parallelism (TLP) which can virtually achieve any degree of concurrency

The Memory wall

The gap between processors and memory speed has increased dramatically. Caches are used to improve memory performance provided that data locality can be exploited.
To deliver twice the performance with the same bandwidth, the cache miss rate must be cut in half; this means:

- For dense matrix-matrix multiply or dense LU, 4x bigger cache
- For sorting or FFTs, the square of its former size
- For sparse or dense matrix-vector multiply, forget it

What is the cost of complicated memory hierarchies?

LATENCY

TLP (that is, multicores) can help overcome this inefficiency by means of multiple streams of execution where memory access latency can be hidden.

The Power wall

ILP techniques are based on the exploitation of higher clock frequencies.
Processors performance can be improved by a factor k by increasing frequency by the same factor.
Is this a problem? yes, it is.

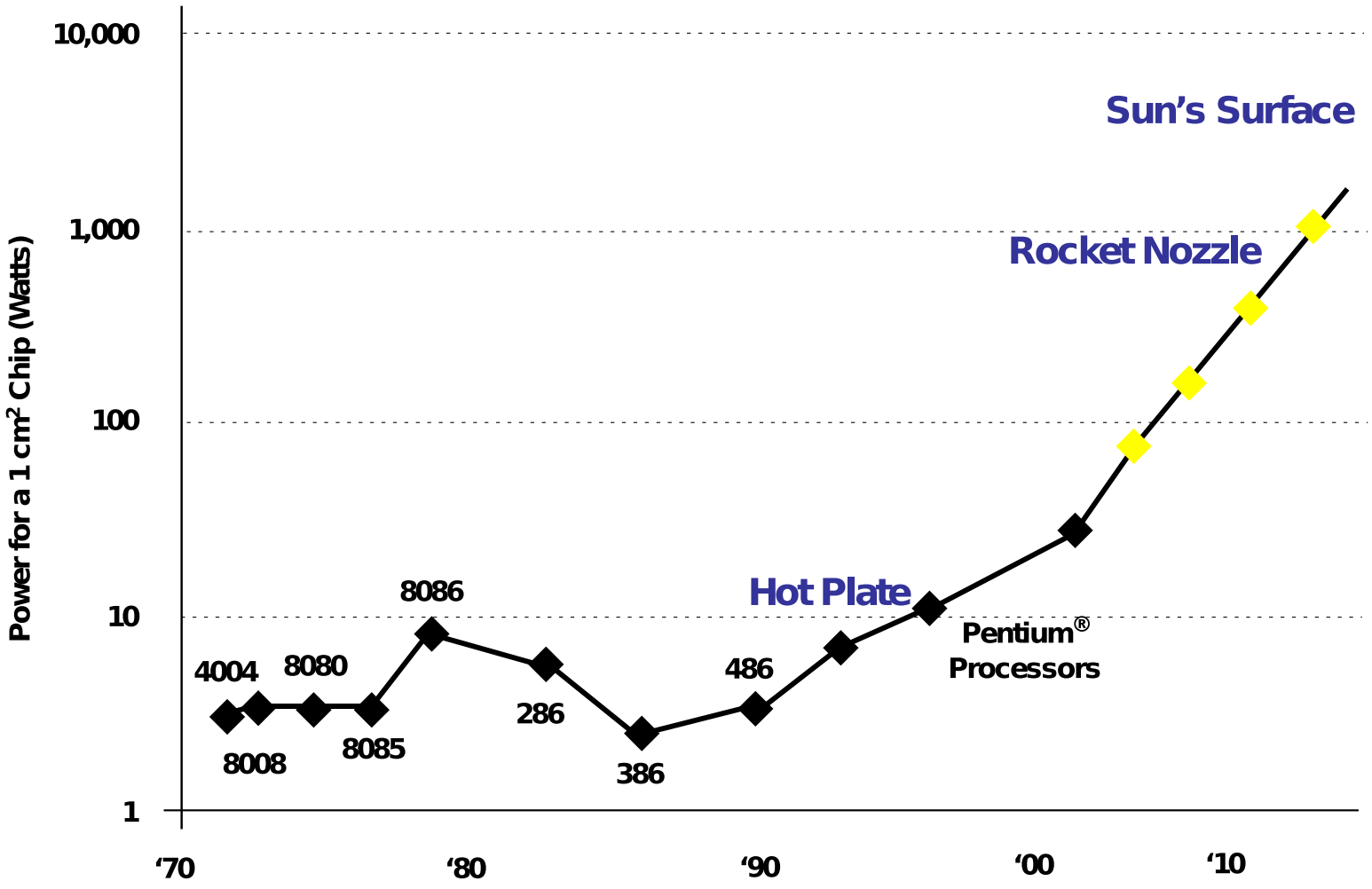
$$P \simeq P_{dynamic} = CV^2 f$$
$$P_{dynamic} = dynamic \quad power$$
$$C = capacitance$$
$$V = voltage$$
$$f = frequency$$

but

$$f_{max} \sim V$$

Power consumption and heat dissipation grow as f^3!

The Power wall



Source: Pat Gelsinger, Intel, ISSCC 2001

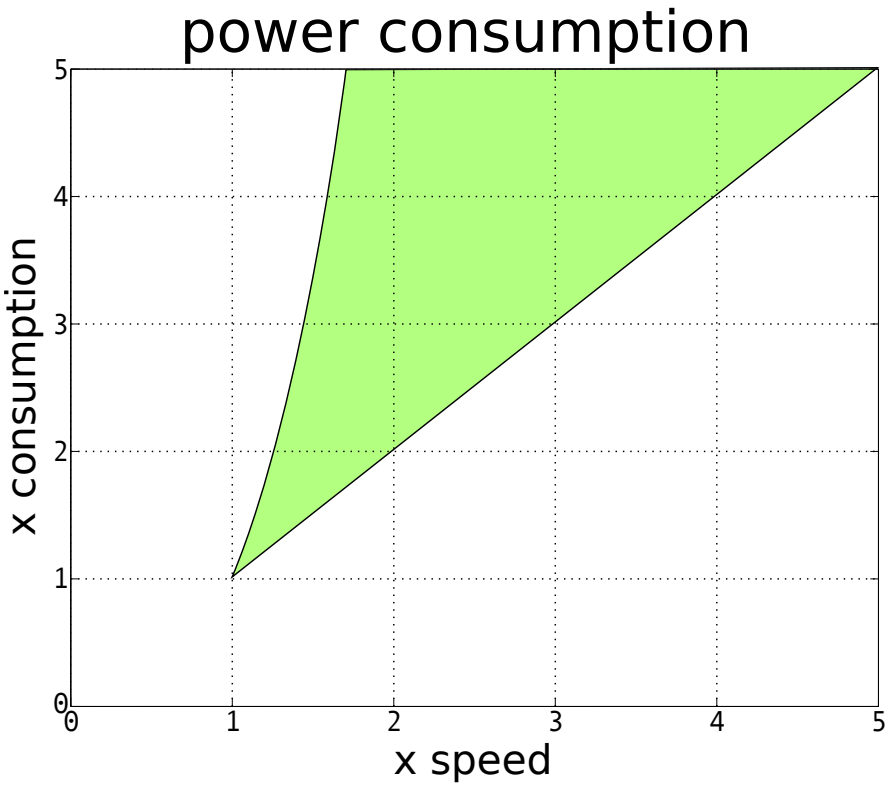
The Power wall

Is there any other way to increase performance without consuming too much power?
Yes, with multicores:
a k-way multicore is k times faster than an unicore and consumes only k times as much power.

$$P_{dynamic} \propto C$$

Thus power consumption and heat dissipation grow linearly with the number of cores (i.e., chip complexity or number of transistors).

The Power wall



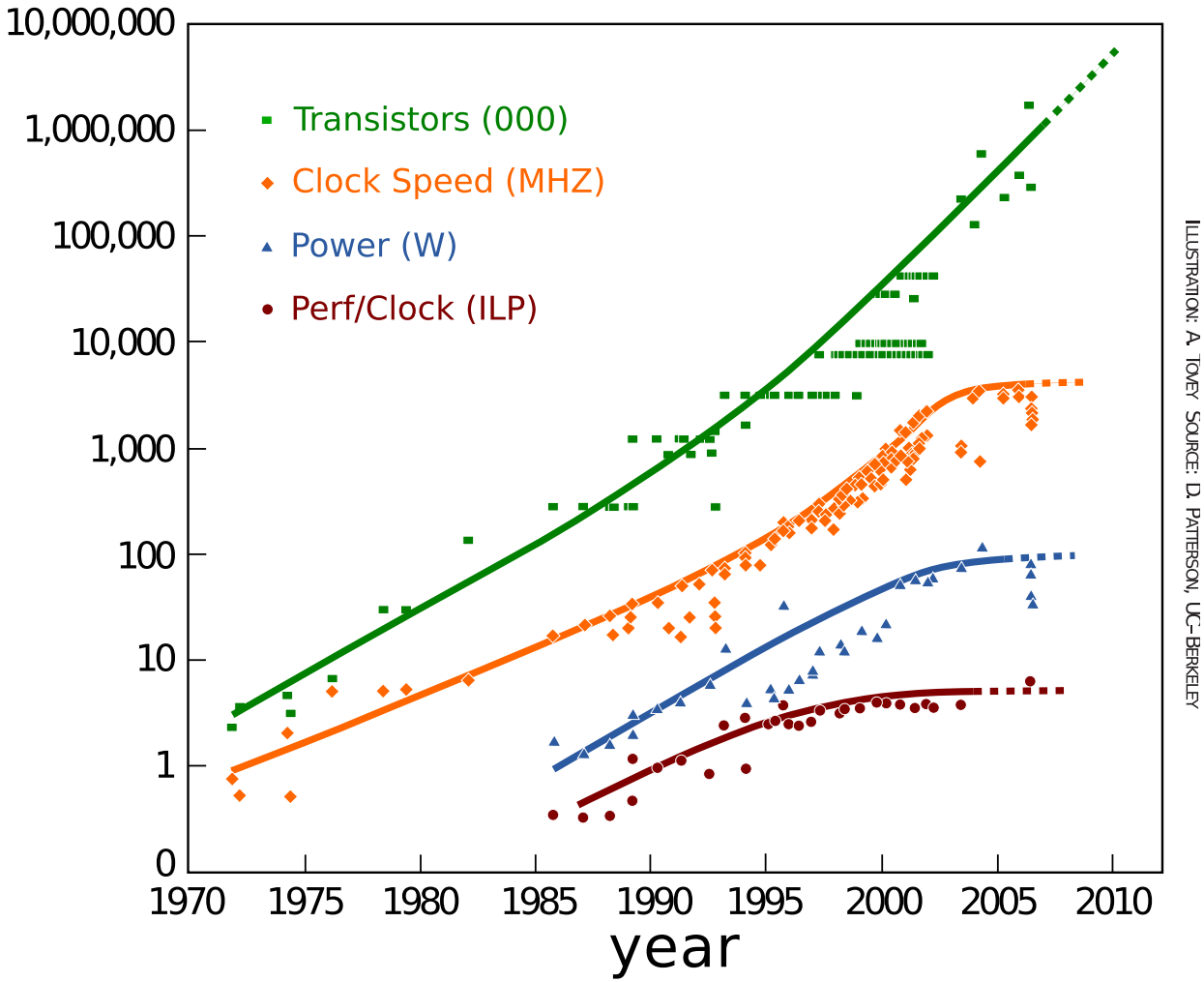
It is even possible to reduce power consumption while still increasing performance.

Assume a single-core processor with frequency f and capacitance C.

A quad-core with frequency $0.6 \times f$ will consume 15% less power while delivering 2.4 higher performance.

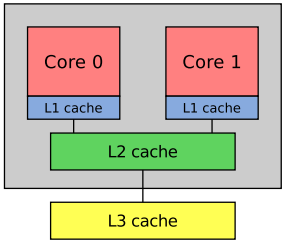
The Moore's Law

The Moore's law: the number of transistors in microprocessors doubles every two years.
The Moore's law, take 2: the performance of microprocessors doubles every 18 months.

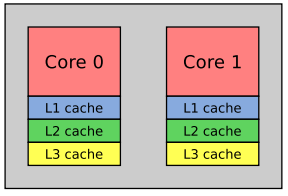


Examples of multicore architectures

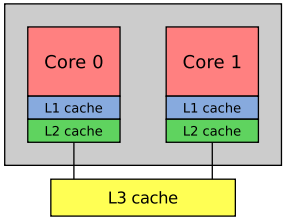
Power 4 (2001)



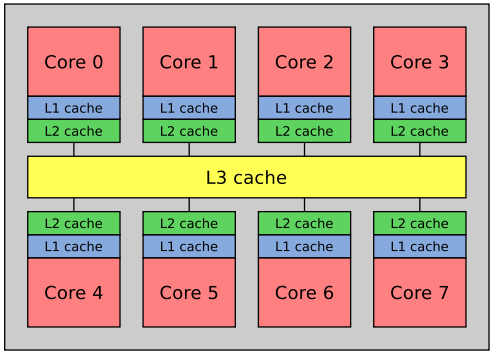
Intel Itanium2 (2004)



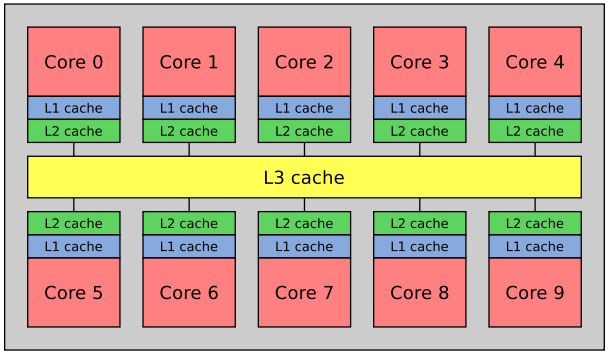
AMD Opteron (2005)



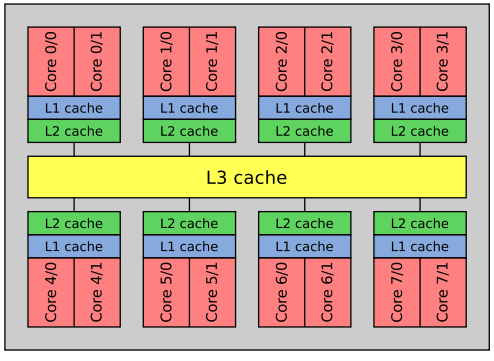
Power 7 (2010)



Intel Xeon-E7 (2011)



AMD Bulldozer (2011)



Conventional Multicores

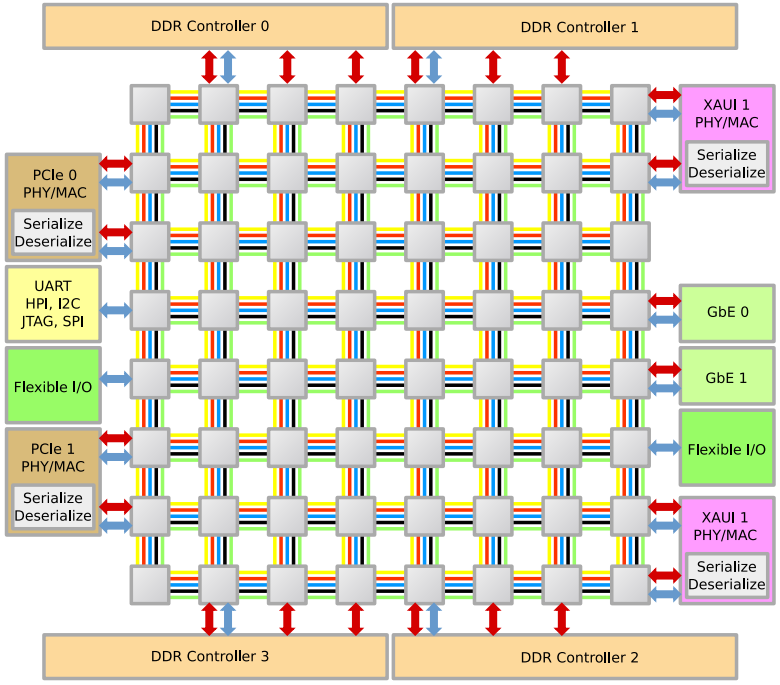
What are the problems with all these designs?

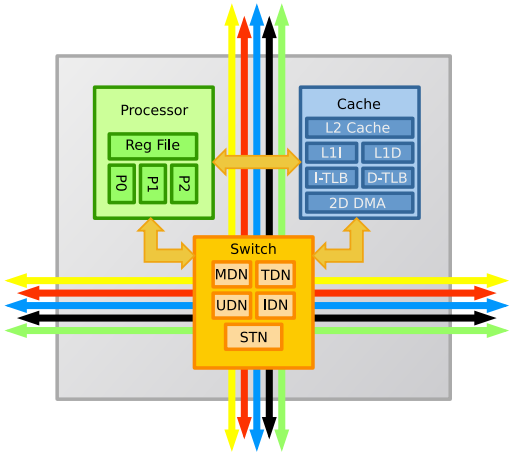
- Core-to-core communication. Although cores lie on the same piece of silicon, there is no direct communication channel between them. The only option is to communicate through main memory.
- Shared memory bus. On modern systems, processors are much faster than memory; example:
  Intel Woodcrest:
  - at 3.0 GHz each core can process $3 \times 4(SSE) \times 2(dualissue) = 24$ single-precision floating-point values in a nanosecond.
  - at 10.5 GB/s the memory can provide $10.5/4 \simeq 2.6$ single-precision floating-point values in a nanosecond.

  One core is 9 times as fast as the memory!
  Attaching more cores to the same bus only makes the problem worse unless heavy data reuse is possible.

The future of multicores

TILE64 is a microcontroller manufactured by Tilera. It consists of a mesh network of 64 "tiles", where each tile houses a general purpose processor, cache, and a non-blocking router, which the tile uses to communicate with the other tiles on the processor.



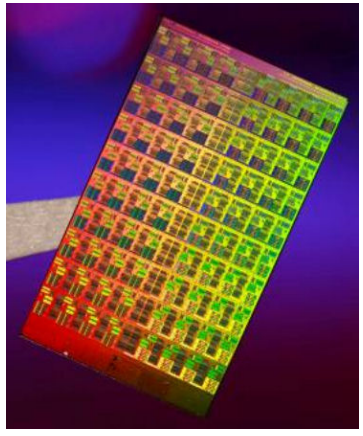


- 4.5 TB/s on-chip mesh interconnect
- 25 GB/s towards main memory
- no floating-point

Intel Polaris

Intel Polaris 80 cores prototype:

- 80 tiles arranged in a 8×10 grid
- on-chip mesh interconnect with 1.62 Tb/s bisection bandwidth
- 3-D stacked memory (future)
- consumes only 62 Watts and is 275 square millimeters
- each tile has:
  - a router
  - 3 KB instruction memory
  - 2 KB data memory
  - 2 SP FMAC units
  - 32 SP registers

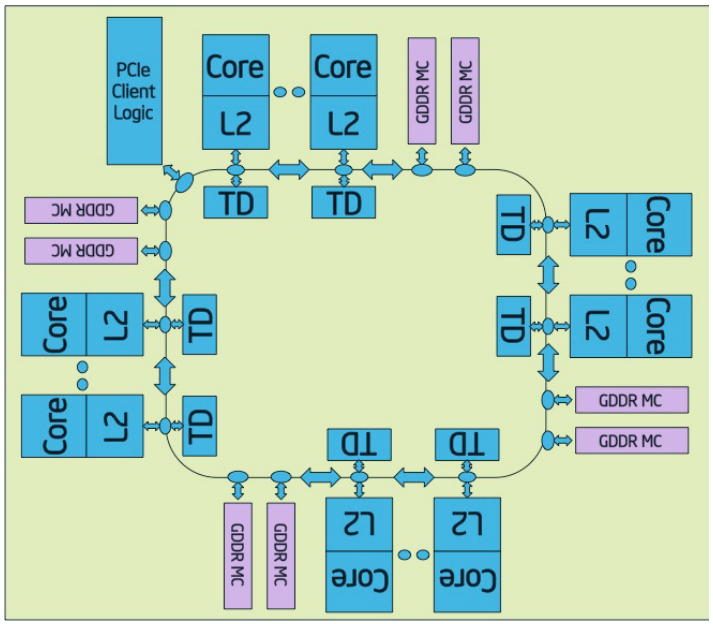


That makes $4(FLOPS) \times 80(tiles) \times 3.16GHz \simeq 1TFlop/s$. The first TFlop machine was the ASCII Red made up of 10000 Pentium Pro, taking 250 mq and 500 KW...

Intel Xeon Phi

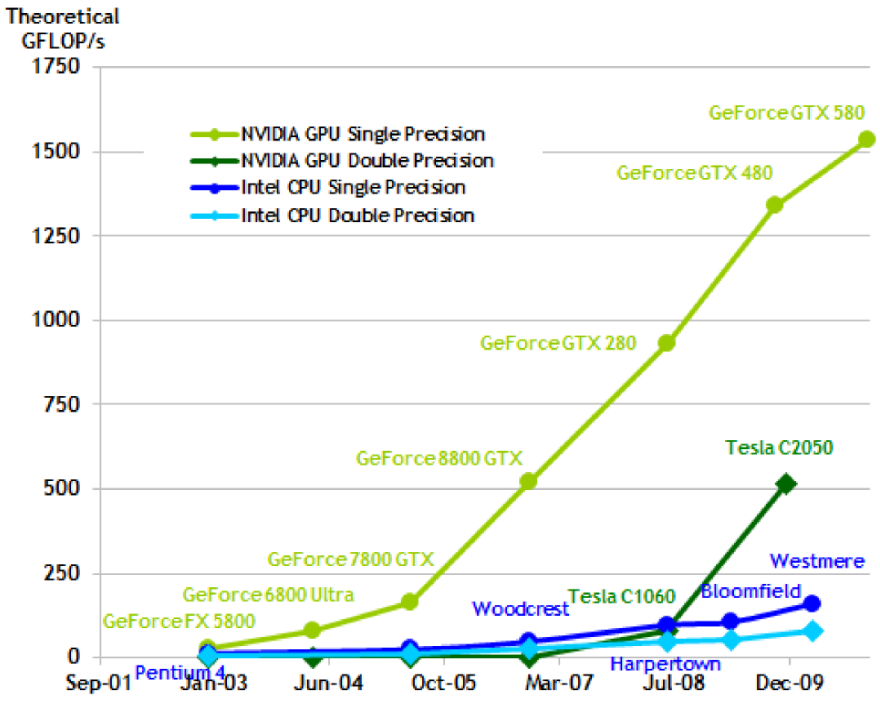
In 2012 Intel released the Xeon Phi boards based on the MIC (Many Integrated Cores) architecture

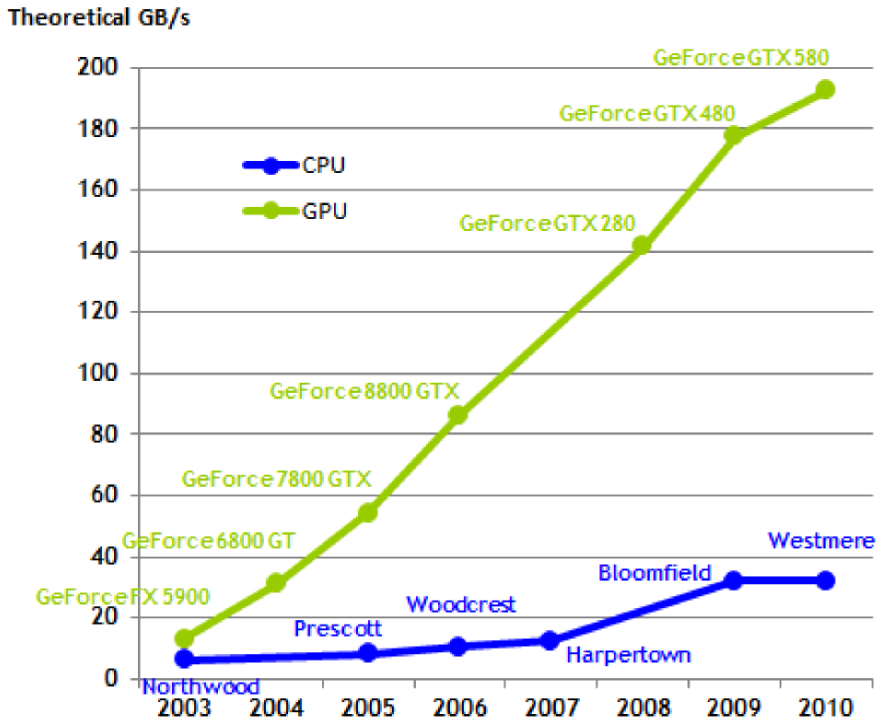
- connected to the main CPU (host) through PCI
- up to 61 cores @ 1.238 GHz
- 512-bit vector instructions (AVX) including FMA
- 1.208 Tflop/s
- 4 threads per core
- 352 GB/s memory bandwidth
- 16 GB memory
- on board high speed ring interconnect
- 300 WATTS
- fully x86 compliant



Other computing devices: GPUs

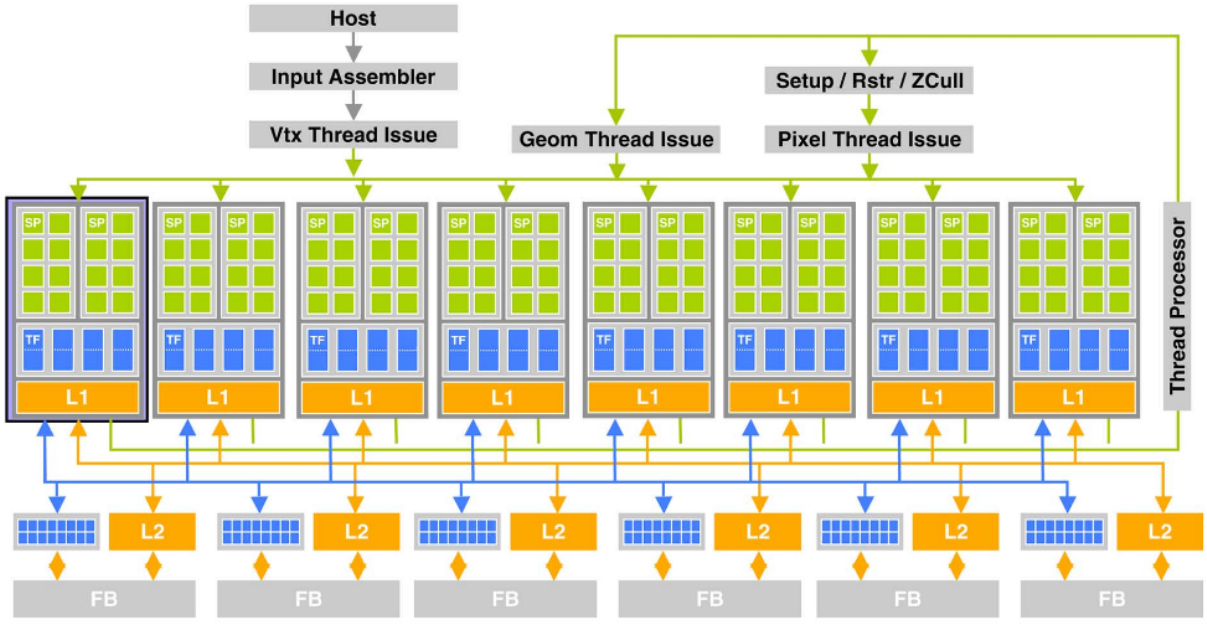
NVIDIA GPUs vs Intel processors: performance

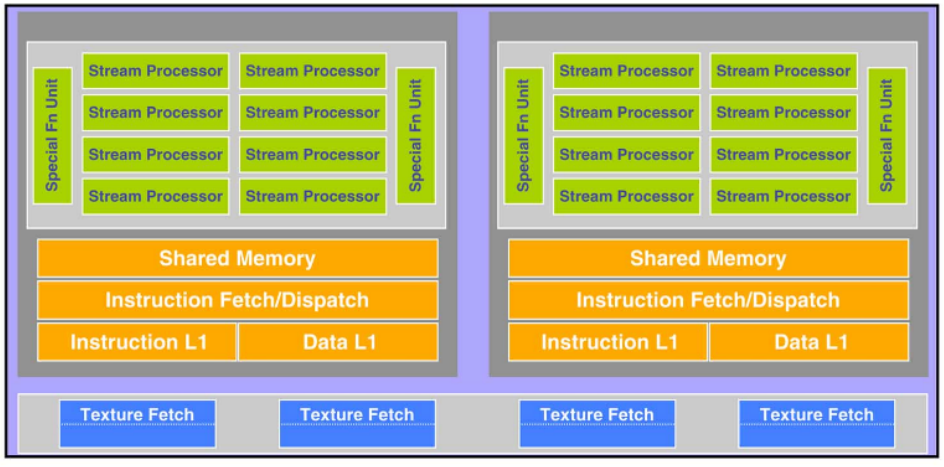




Other computing devices: GPUs

NVIDIA GeForce 8800 GTX:





16 streaming multiprocessors of 8 thread processors each.

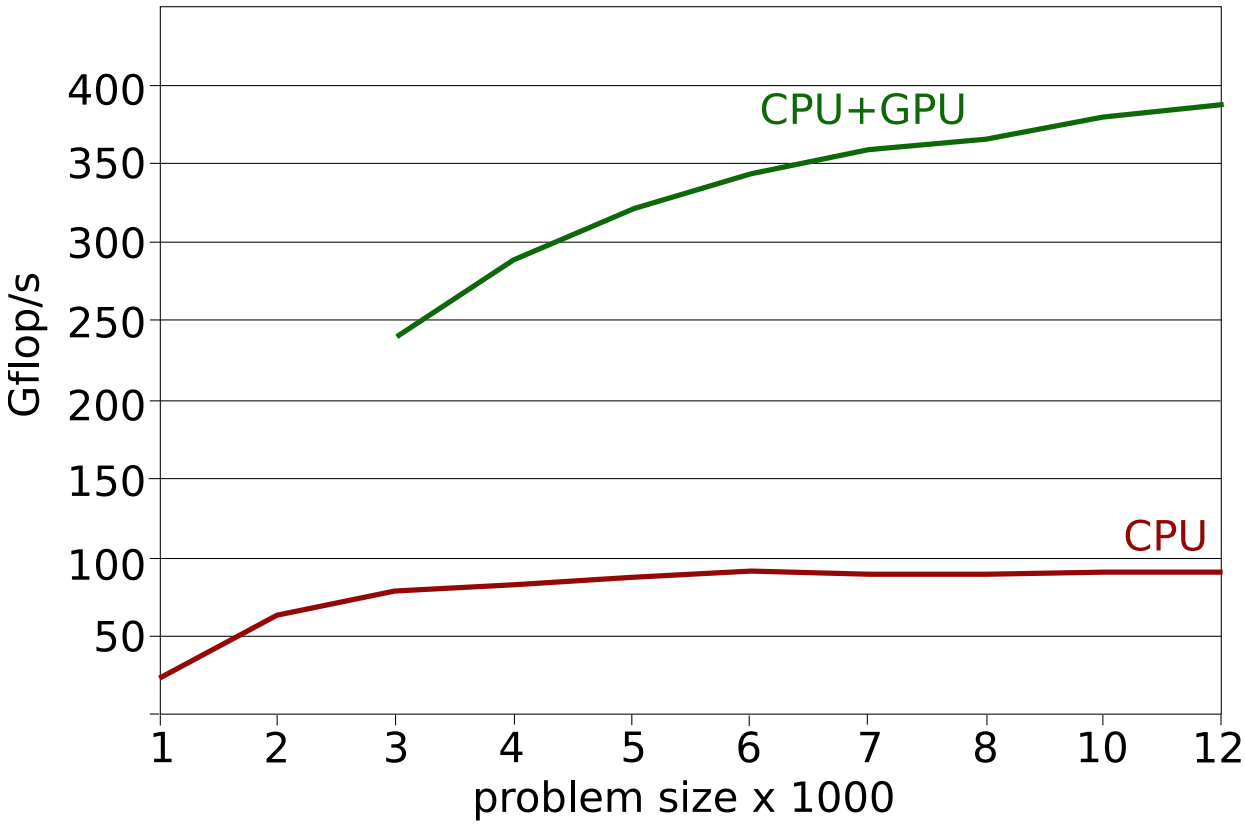
Other computing devices: GPUs

How to program GPUs?

- SPMD programming model
  - coherent branches (i.e. SIMD style) preferred
  - penalty for non-coherent branches (i.e., when different processes take different paths)
- directly with OpenGL/DirectX: not suited for general purpose computing
- with higher level GPGPU APIs:
  - AMD/ATI HAL-CAL (Hardware Abstraction Level - Compute Abstraction Level)
  - NVIDIA CUDA: C-like syntax with pointers etc.
  - RapidMind
  - PeakStream

Other computing devices: GPUs

LU on 8-cores Xeon + GeForce GTX 280:



OpenMP programming

Outline

Section 2

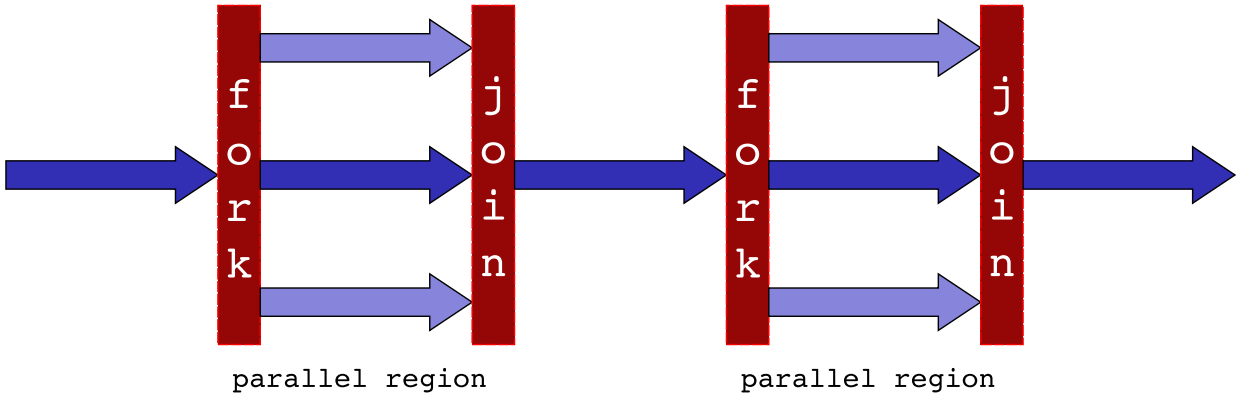
OpenMP

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/`

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

How to program multicores: OpenMP

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: the PARALLEL construct

The PARALLEL one is the main OpenMP construct and identifies a block of code that will be executed by multiple threads:

```
!$OMP PARALLEL [clause ...]
               IF (scalar_logical_expression)
               PRIVATE (list)
               SHARED (list)
               DEFAULT (PRIVATE | SHARED | NONE)
               FIRSTPRIVATE (list)
               REDUCTION (operator: list)
               COPYIN (list)
               NUM_THREADS (scalar-integer-expression)

   block

!$OMP END PARALLEL
```

- The master is a member of the team and has thread number 0
- Starting from the beginning of the region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section.
- If any thread terminates within a parallel region, all threads in the team will terminate.

OpenMP: the PARALLEL construct

How many threads do we have? The number of threads depends on:

- Evaluation of the `IF` clause
- Setting of the `NUM_THREADS` clause
- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default - usually the number of CPUs on a node, though it could be dynamic

Hello world example:

```
program hello

  integer :: nthreads, tid, &
       & omp_get_num_threads, omp_get_thread_num

  ! Fork a team of threads giving them
  ! their own copies of variables

  !$omp parallel private(tid)
  ! Obtain and print thread id
  tid = omp_get_thread_num()
  write(*,'("Hello from thread ",i2)')tid

  ! Only master thread does this
  if (tid .eq. 0) then
     nthreads = omp_get_num_threads()
     write(*,'("# threads: ",i2)')nthreads
  end if

  ! All threads join master thread and disband
  !$omp end parallel

end program hello
```

- the `PRIVATE` clause says that each thread will have its own copy of the `tid` variable (more later)
- the `omp_get_num_threads` and `omp_get_thread_num` are runtime library routines

OpenMP: Data scoping

- Most variables are shared by default
- Global variables include:
  - Fortran: COMMON blocks, SAVE and MODULE variables
  - C: File scope variables, static
- Private variables include:
  - Loop index variables (in `!$OMP DO`) constructs
  - Stack variables in subroutines called from parallel regions
  - Fortran: Automatic variables within a statement block
- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
  - `PRIVATE`
  - `FIRSTPRIVATE`
  - `LASTPRIVATE`
  - `SHARED`
  - `DEFAULT`
  - `REDUCTION`
  - `COPYIN`

OpenMP: Data scoping

- `PRIVATE(list)`: a new object of the same type is created for each thread (uninitialized!)
- `FIRSTPRIVATE(list)`: Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.
- `LASTPRIVATE(list)`: The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
- `SHARED(list)`: only one object exists in memory and all the threads access it
- `DEFAULT(SHARED|PRIVATE|NONE)`: sets the default scoping
- `REDUCTION(operator:list)`: performs a reduction on the variables that appear in its list.

OpenMP: worksharing constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it
- Work-sharing constructs do not launch new threads

There are three main workshare constructs:

- `DO/for` construct: it is used to parallelize loops
- `SECTIONS`: used to identify portions of code that can be executed in parallel
- `SINGLE`: specifies that the enclosed code is to be executed by only one thread in the team.

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)


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


  !$omp end parallel

end program do_example
```

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: worksharing constructs

The DO/for directive:

```
!$OMP DO [clause ...]
         SCHEDULE (type [,chunk])
         ORDERED
         PRIVATE (list)
         FIRSTPRIVATE (list)
         LASTPRIVATE (list)
         SHARED (list)
         REDUCTION (operator | intrinsic : list)

   do_loop

!$OMP END DO  [ NOWAIT ]
```

This directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team

There is an implied barrier at the end of the construct

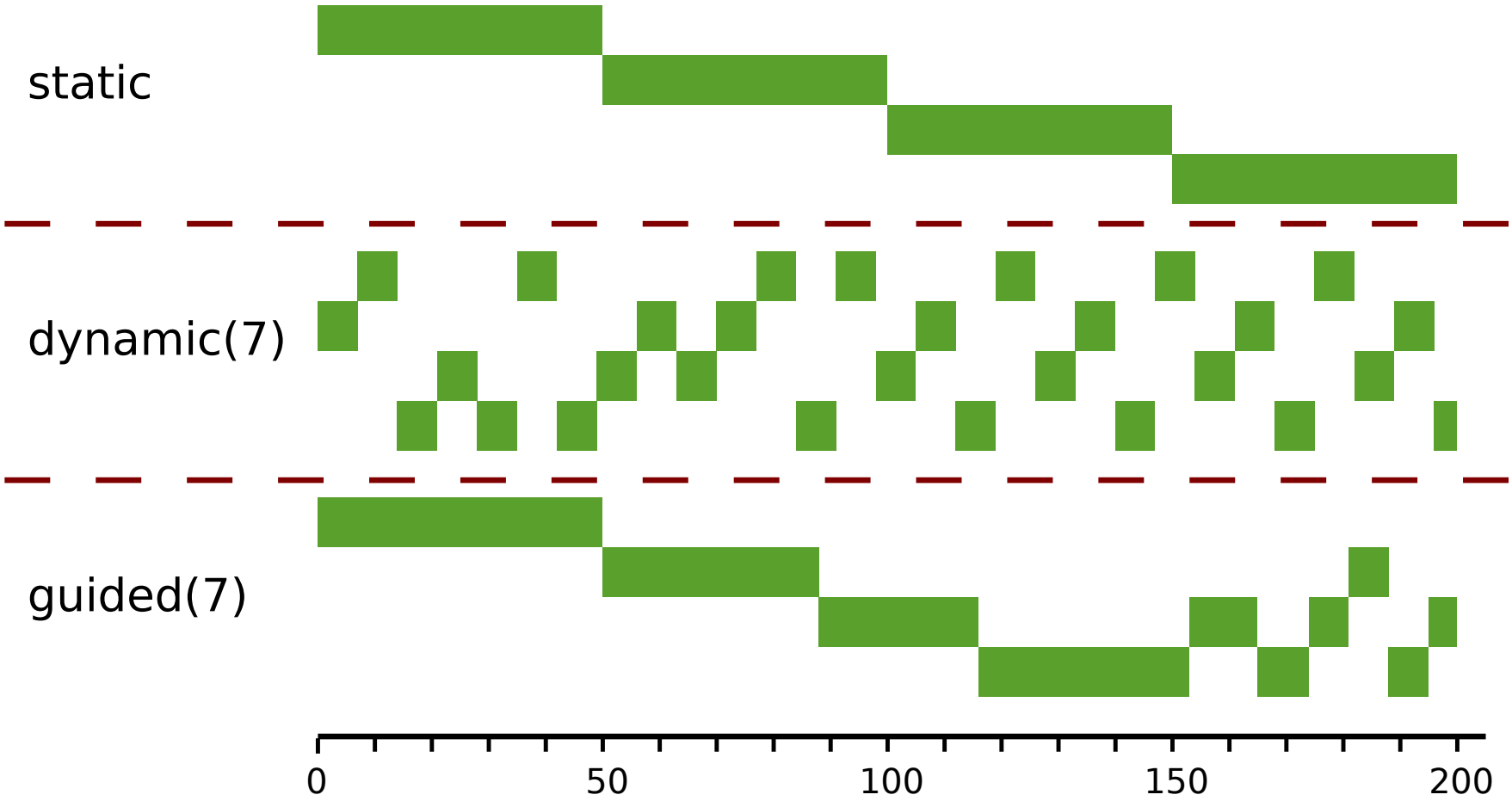
OpenMP: worksharing constructs

The SCHEDULE clause in the DO/for construct specifies how the cycles of the loop are assigned to threads:

- STATIC: loop iterations are divided into pieces of size *chunk* and then statically assigned to threads in a round-robin fashion
- DYNAMIC: loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another
- GUIDED: for a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations
- RUNTIME: The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE

OpenMP: worksharing constructs

Example showing scheduling policies for a loop of size 200



OpenMP: worksharing constructs

```
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,chunk) private(i)

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

  !$omp end parallel

end program do_example
```

OpenMP: worksharing constructs

The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

```
!$OMP SECTIONS [clause ...]
               PRIVATE (list)
               FIRSTPRIVATE (list)
               LASTPRIVATE (list)
               REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

    block

!$OMP  SECTION

     block

!$OMP END SECTIONS  [ NOWAIT ]
```

There is an implied barrier at the end of the construct

Example of the SECTIONS worksharing construct

```
program vec_add_sections

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

  ! some sequential code

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

  !$omp sections

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

  !$omp section
  do i = 1, n
     d(i) = a(i) * b(i)
  end do

  !$omp end sections
  !$omp end parallel

end program vec_add_sections
```

OpenMP: worksharing constructs

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.

```
!$OMP SINGLE [clause ...]
             PRIVATE (list)
             FIRSTPRIVATE (list)

    block

!$OMP END SINGLE [ NOWAIT ]
```

There is an implied barrier at the end of the construct

OpenMP: synchronization constructs

The CRITICAL construct enforces exclusive access with respect to all critical constructs with the same name in all threads

```
!$OMP CRITICAL [ name ]

    block

!$OMP END CRITICAL
```

The MASTER directive specifies a region that is to be executed only by the master thread of the team

```
!$OMP MASTER

    block

!$OMP END MASTER
```

The BARRIER directive synchronizes all threads in the team

```
!$OMP BARRIER
```

OpenMP: synchronization all-in-one example

```
!$omp parallel
! all the threads do some stuff in parallel
...

!$omp critical
! only one thread at a time will execute these instructions.
! Critical sections can be used to prevent simultaneous
! writes to some data
call one_thread_at_a_time()
!$omp end critical

...

!$omp master
! only the master thread will execute these instructions.
! Some parts can be inherently sequential or need not be
! executed by all the threads
call only_master()
!$omp end master

! each thread waits for all the others to reach this point
!$omp barrier
! After the barrier we are sure that every thread sees the
! results of the work done by other threads

...
! all the threads do more stuff in parallel

!$omp end parallel
```

OpenMP: synchronization constructs: ATOMIC

The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it.

```
!$OMP ATOMIC
    statement_expression
[!$OMP END ATOMIC]
```

What is the difference with CRITICAL?

```
!$omp atomic
x = some_function()
```

With ATOMIC the function `some_function` will be evaluated in parallel since only the update is atomical.

Another advantage:

```
!$omp critical
x[i] = v
!$omp end critical
```

```
!$omp atomic
x[i] = v
```

With `atomic` different coefficients of `x` will be updated in parallel

OpenMP: synchronization constructs: ATOMIC

With ATOMIC it is possible to specify the access mode to the data:

Read a variable atomically

```
!$omp atomic read
v = x
```

Write a variable atomically

```
!$omp atomic write
x = v
```

Update a variable atomically

```
!$omp atomic update
x = x+1
```

Capture a variable atomically

```
!$omp atomic capture
x = x+1
v = x
!$omp end atomic
```

atomic regions enforce exclusive access with respect to other atomic regions that access the same storage location x among all the threads in the program without regard to the teams to which the threads belong

OpenMP: reductions and conflicts

How to do reductions with OpenMP?

```
sum = 0
do i=1,n
   sum = sum+a(i)
end do
```

Here is a wrong way of doing it:

```
sum = 0
!$omp parallel do shared(sum)
do i=1,n
   sum = sum+a(i)
end do
```

What is wrong?

Concurrent access has to be synchronized otherwise we will end up in a WAW conflict!

Conflicts

- **Read-After-Write (RAW)**
  A data is read after an instruction that modifies it. It is also called true dependency

```
a = b+c
d = a+c
```

```
do i=2, n
   a(i) = a(i-1)*b(i)
end do
```

- **Write-After-Read (WAR)**
  A data is written after an instruction that reads it. It is also called anti-dependency

```
a = b+c
b = c*2
```

```
do i=1, n-1
   a(i) = a(i+1)*b(i)
end do
```

- **Write-After-Write (WAW)**
  A data is written after an instruction that modifies it. It is also called output dependency

```
c = a(i)*b(i)
c = 4
```

```
do i=1, n
   c = a(i)*b(i)
end do
```

OpenMP: reductions

We could use the CRITICAL construct:

```
sum = 0
!$omp parallel do shared(sum)
do i=1,n
!$omp critical
   sum = sum+a(i)
!$omp end critical
end do
```

but there's a more intelligent way

```
sum = 0
!$omp parallel do reduction(+:sum)
do i=1,n
   sum = sum+a(i)
end do
```

The reduction clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

OpenMP: the `task` construct

The TASK construct defines an explicit task

```
!$OMP TASK [clause ...]
             IF (scalar-logical-expression)
             UNTIED
             DEFAULT (PRIVATE | SHARED | NONE)
             PRIVATE (list)
             FIRSTPRIVATE (list)
             SHARED (list)
             DEPEND (dependence-type : list)
    block

!$OMP END TASK
```

When a thread encounters a TASK construct, a task is **generated** (not executed!!!) from the code for the associated structured block. The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.

OpenMP: the `task` construct

But, then, when are tasks executed? Execution of a task may be assigned to a thread whenever it reaches a task scheduling point:

- the point immediately following the generation of an explicit task
- after the last instruction of a task region
- in `taskwait` regions
- in implicit and explicit barrier regions

At a task scheduling point a thread can:

- begin execution of a tied or untied task
- resume a suspended task region that is tied to it
- resume execution of a suspended, untied task

OpenMP: the `task` construct

All the clauses in the `TASK` construct have the same meaning as for the other constructs except for:

- `IF`: when the `IF` clause expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed
- `UNTIED`: by default a task is tied. This means that, if the task is suspended, then its execution may only be resumed by the thread that started it. If, instead, the `UNTIED` clause is present, any thread can resume its execution

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.

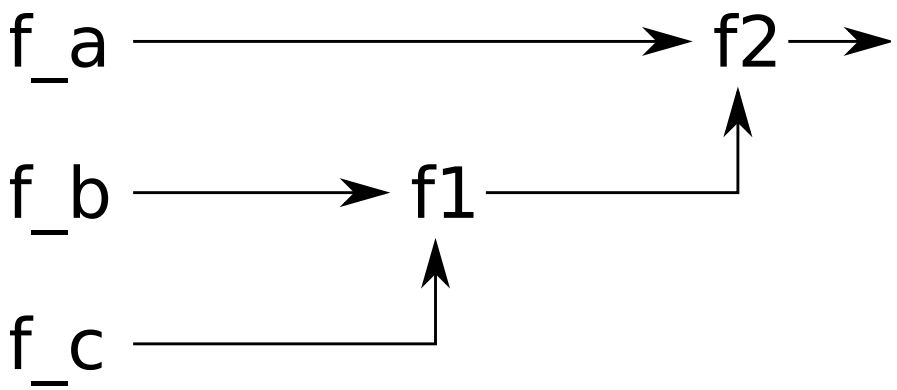
Task dependences are derived from the dependence-type of a depend clause and its list items, where dependence-type is one of the following:

- 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

Write a parallel version of the following subroutine using OpenMP tasks:

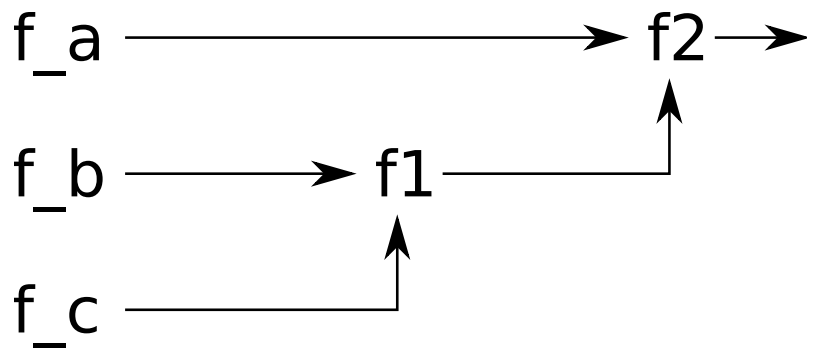
```
function  foo()
  integer  :: foo
  integer  :: a, b, c, x, y;

  a = f_a()
  b = f_b()
  c = f_c()
  x = f1(b, c)
  y = f2(a, x)

  return y;

end function foo
```



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

OpenMP Locks

Lock can be used to prevent simultaneous access to shared resources according to the schema

- acquire (or set or lock) the lock
- access data
- release (on unset or unlock) the lock

Acquisition of the lock is exclusive in the sense that only one threads can hold the lock at a given time. A lock can be in one of the following states:

- **uninitialized**: the lock is not active and cannot be acquired/released by any thread;
- **unlocked**: the lock has been initialized and can be acquired by any thread;
- **locked**: the lock has been acquired by one thread and cannot be acquired by any other thread until the owner releases it.

OpenMP Locks

Locks are used through the following routines:

- `omp_init_lock`: initializes a lock
- `omp_destroy_lock`: uninitializes a lock
- `omp_set_lock`: waits until a lock is available, and then sets it
- `omp_unset_lock`: unsets a lock
- `omp_test_lock`: tests a lock, and sets it if it is available

OpenMP Locks

Examples:

```
!$omp single
! initialize the lock
call omp_init_lock(lock)
!$omp end single
...
! do work in parallel
...
call omp_set_lock(lock)
! exclusive access to data
...
call omp_unset_lock(lock)
...
! do more work in parallel
...
!$omp barrier

! destroy the lock
!$omp single
call omp_destroy_lock(lock)
!$omp end single
```

```
!$omp single
! initialize the lock
call omp_init_lock(lock)
!$omp end single
...
! do work in parallel
...
10 continue
if(omp_test_lock(lock)) then
   ! the lock is available: acquire it
     and
   ! have exclusive access to data
   ...
   call omp_unset_lock(lock)
else
   ! do other stuff
   ! and check for availability later
   ...
   goto 10
end if
...
! do more work in parallel
...
!$omp barrier

! destroy the lock
!$omp single
call omp_destroy_lock(lock)
!$omp end single
```

Outline

Section 3

OpenMP examples

Loop parallelism vs parallel region

Note that these two codes are essentially equivalent:

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

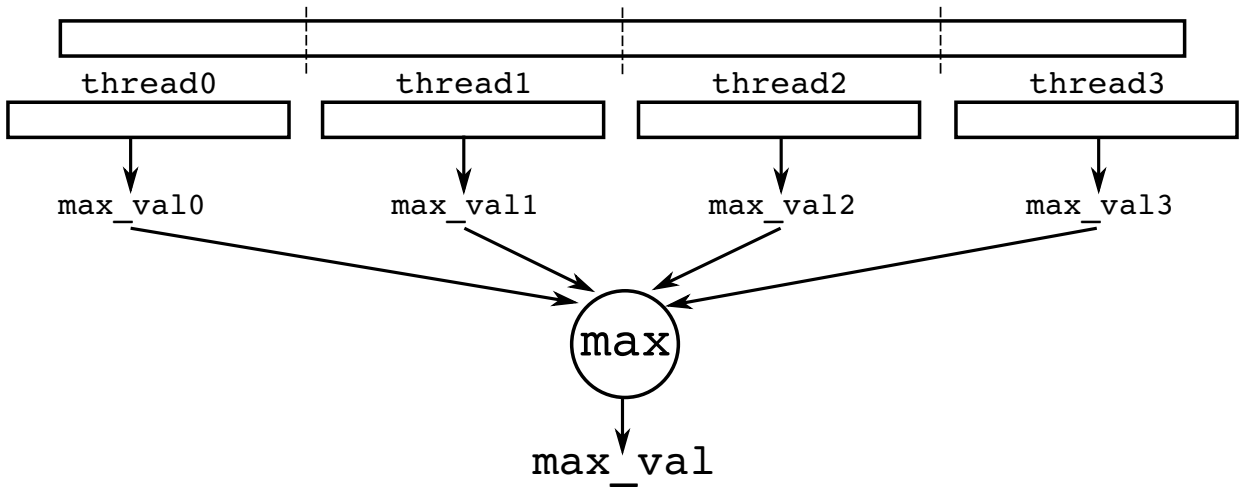
```
!$omp parallel private(iam, nth, b, nl, i)
iam = omp_get_thread_num()
nth = omp_get_num_threads()

! compute the number of loop iterations
! done by each thread
nl = (n-1)/nth+1

! compute the first iteration number
! for this thread
b  = iam*nl+1

do i=b, min(b+nl-1,n)
   a(i) = b(i) + c(i)
end do
!$omp end parallel
```

Loop parallelism is not always possible or may not be the best way of parallelizing a code.

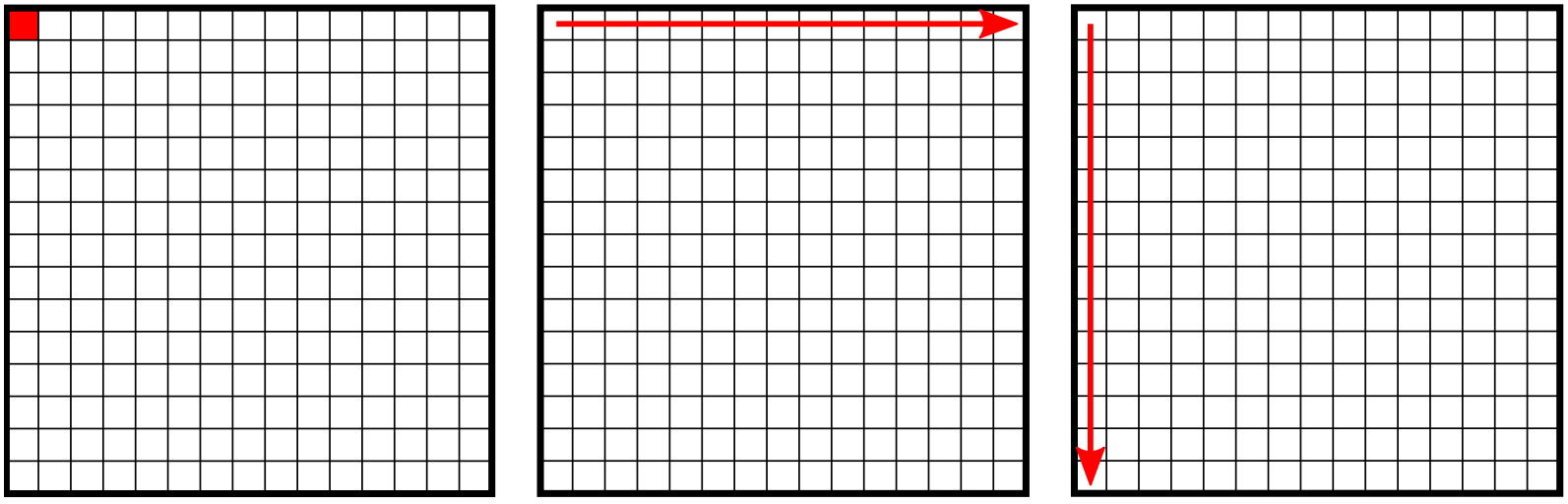
Loop parallelism vs parallel region

Another example: parallelize the `maxval(x)` routine which computes the maximum value of an array `x` of length `n`

```
!$omp parallel private(iam, nth, beg, loc_n, i) reduction(max:max_value)
iam = omp_get_thread_num()
nth = omp_get_num_threads()

! each thread computes the length of its local part of the array
loc_n = (n-1)/nth+1

! each thread computes the beginning of its local part of the array
beg = iam*loc_n+1

! for the last thread the local part may be smaller
if(iam == nth-1)
loc_n = n-beg;

max_value = maxval(x(beg:beg+loc_n-1))
!$omp end parallel
```



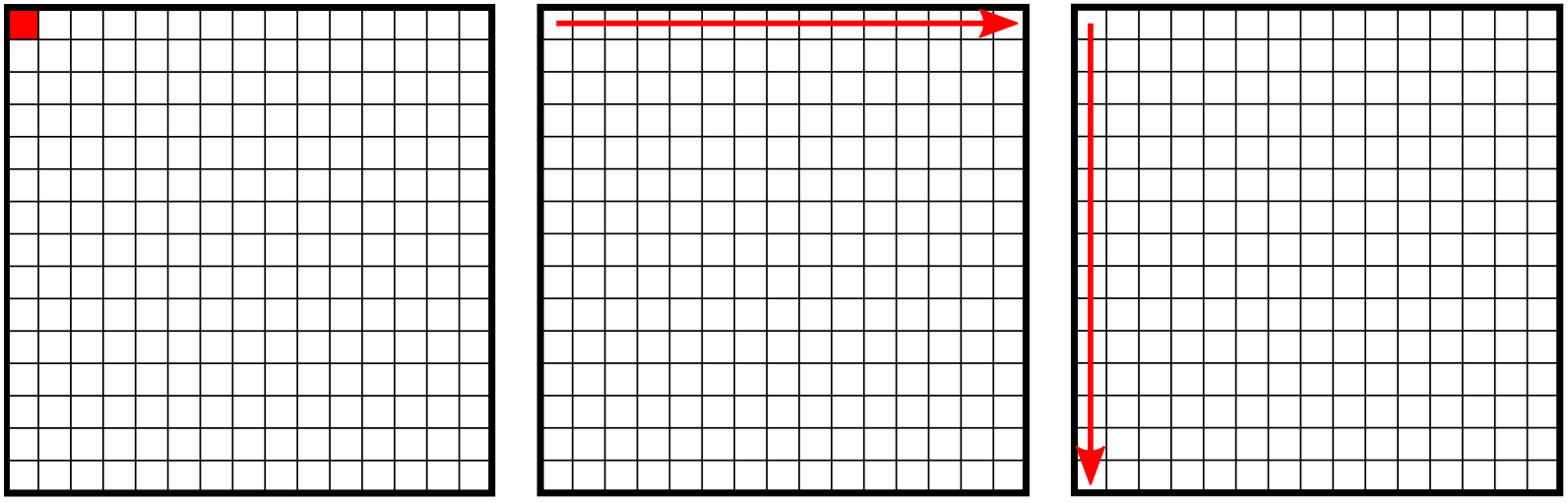
OpenMP MM product



```
subroutine mmproduct(a, b, c)
...

do i=1, n
   do j=1, n
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
   end do
end do

end subroutine mmproduct
```

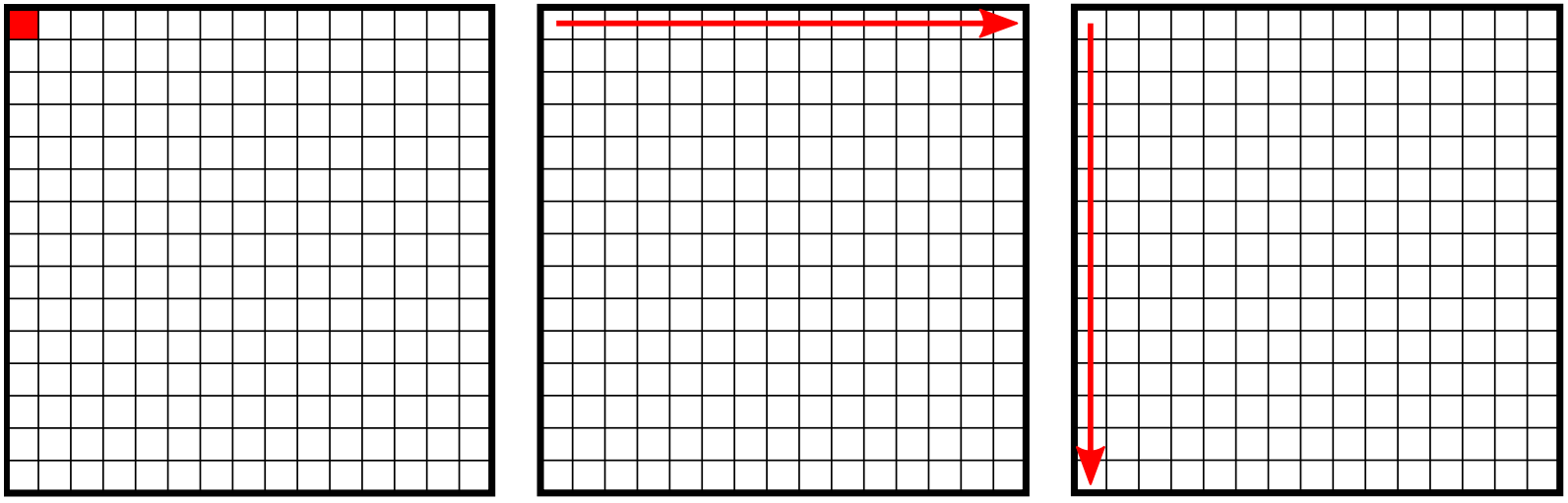
Sequential version

OpenMP MM product



```
subroutine mmproduct(a, b, c)
...
do i=1, n
   do j=1, n
      do k=1, n
         !$omp task
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
         !$omp end task
      end do
   end do
end do
end subroutine mmproduct
```

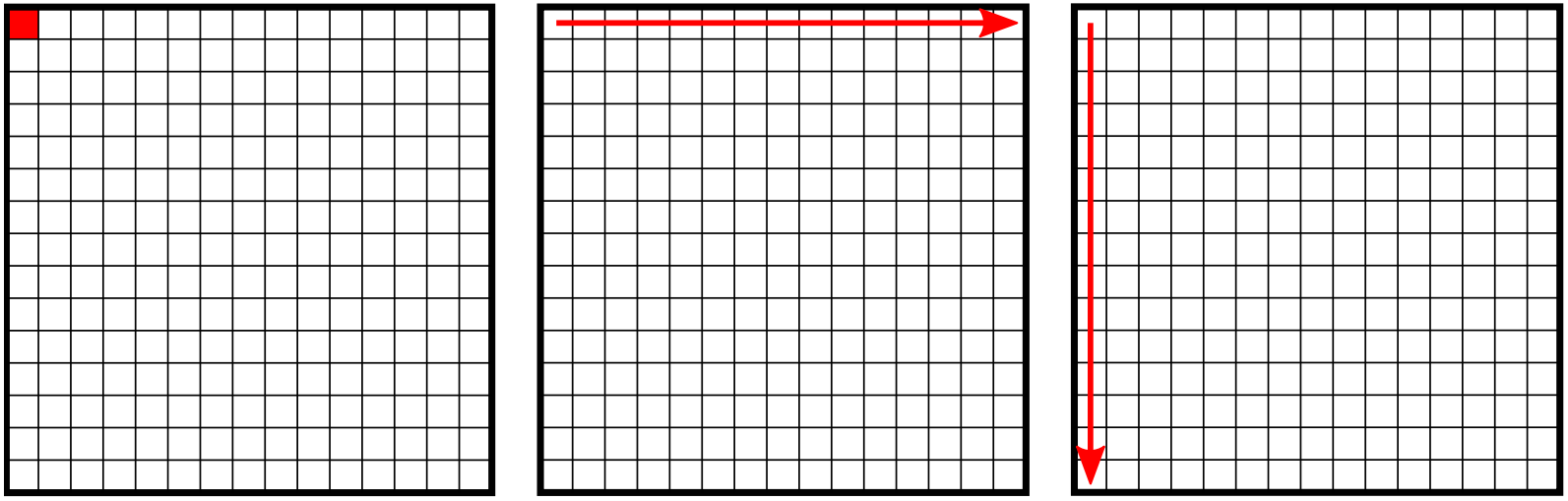
OpenMP MM product



```
subroutine mmproduct(a, b, c)
...
do i=1, n
   do j=1, n
      do k=1, n
         !$omp task
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
         !$omp end task
      end do
   end do
end do
end subroutine mmproduct
```

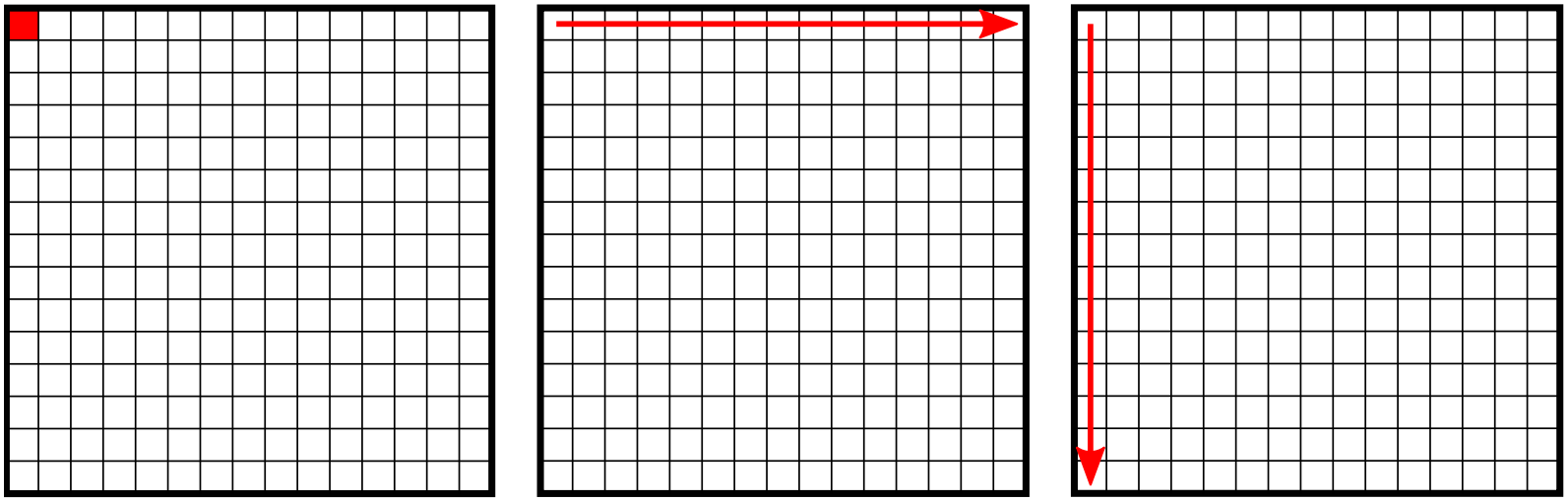
Incorrect parallel with **WAW**, **WAR** and **RAW** conflict on `c(i,j)`

OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j)
do i=1, n
   do j=1, n
      !$omp do
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
      !$omp end do
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

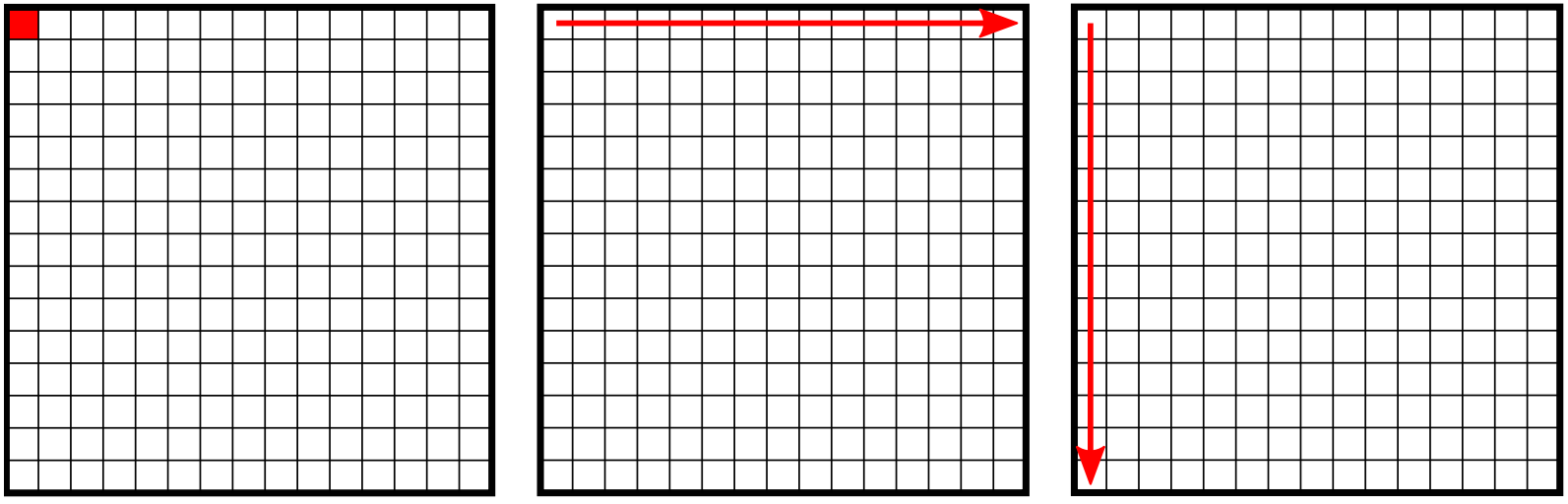
OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j)
do i=1, n
   do j=1, n
      !$omp do
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
      !$omp end do
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

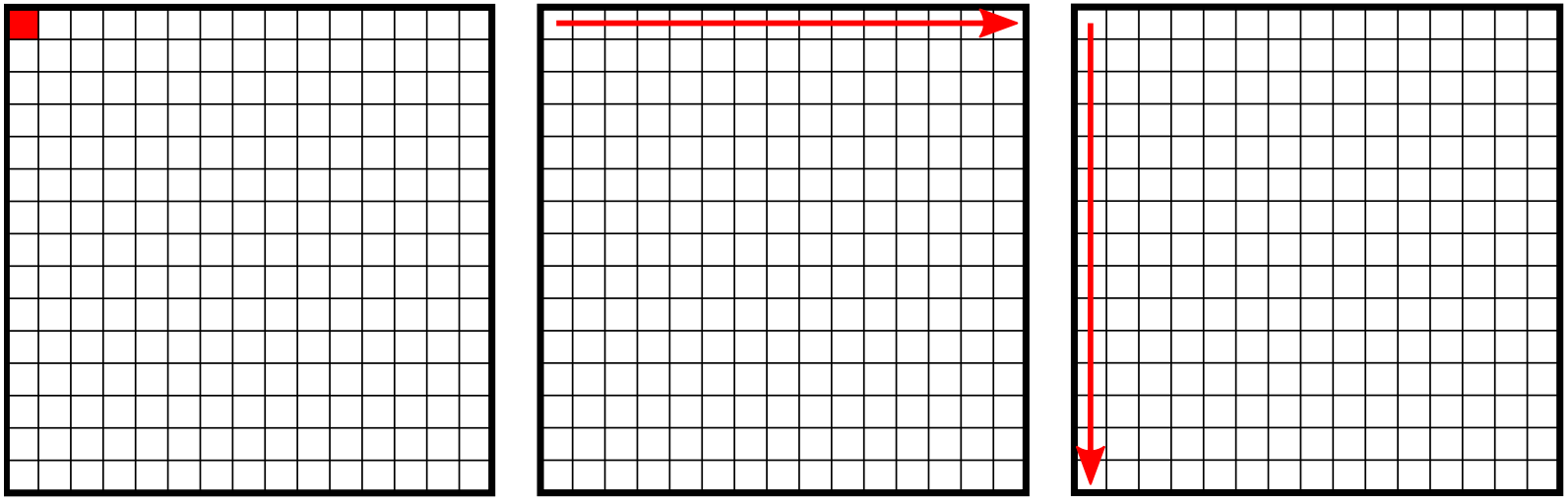
Incorrect parallel with **WAW**, **WAR** and **RAW** conflict on `c(i,j)`

OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel reduction(+,c) private(i,j)
do i=1, n
   do j=1, n
      !$omp do
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
      !$omp end do
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

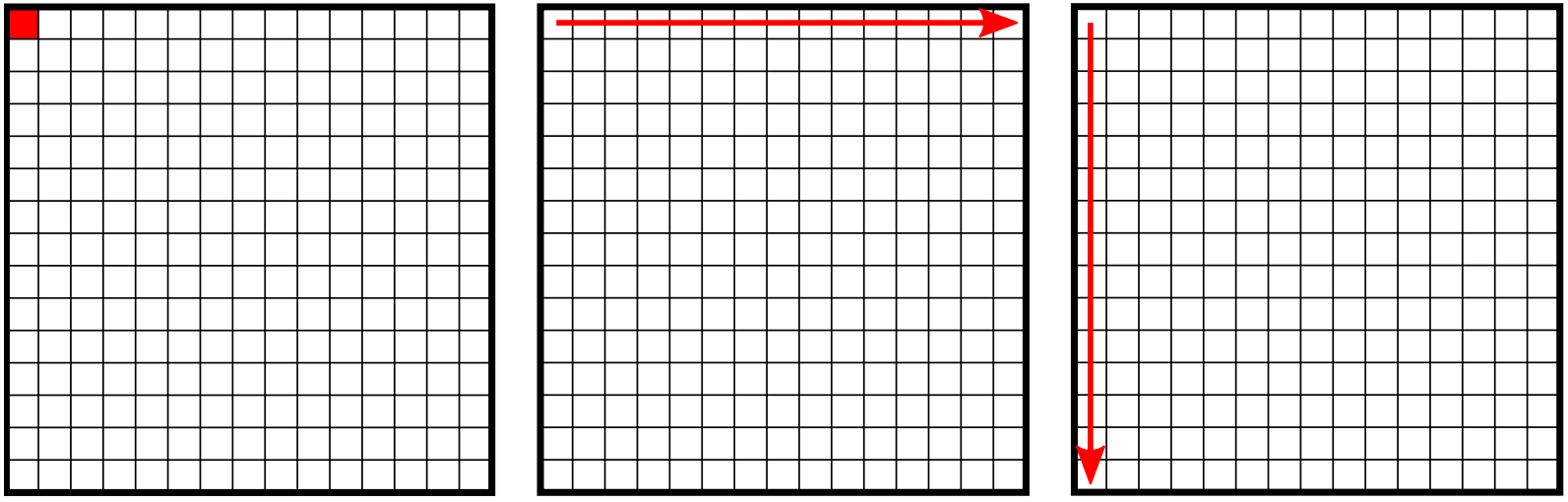
OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel reduction(+,c) private(i,j)
do i=1, n
   do j=1, n
      !$omp do
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
      !$omp end do
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

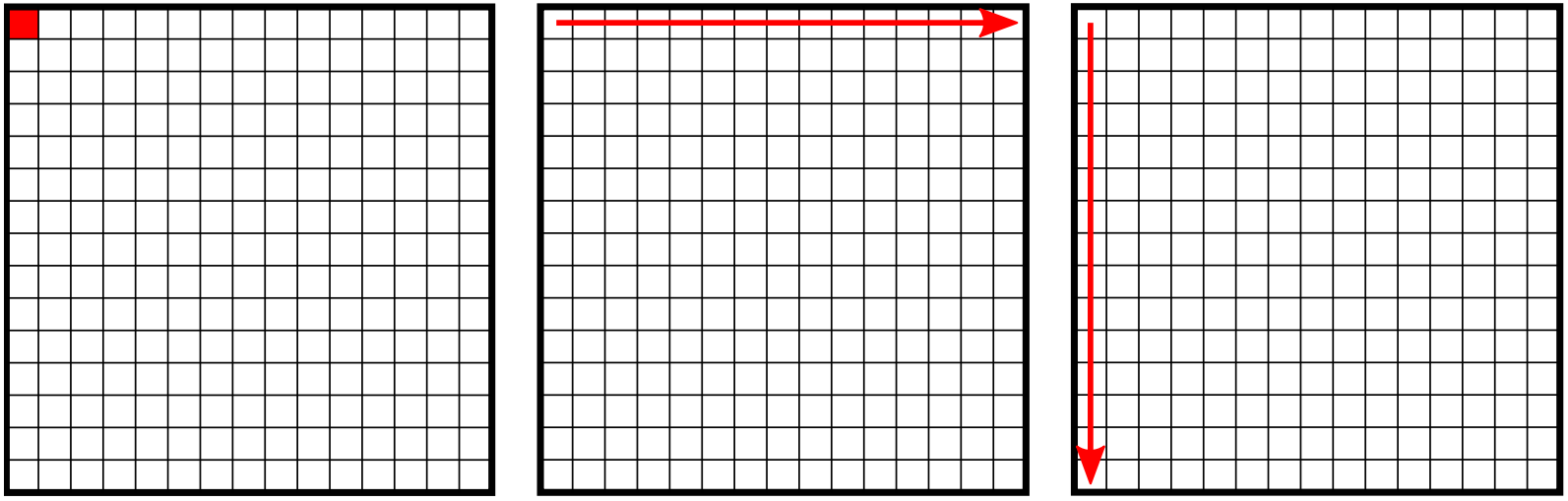
Correct parallel but enormous waste of memory (`c` is replicated)

OpenMP MM product



```
subroutine mmproduct(a, b, c)

do i=1, n
   do j=1, n
      acc = 0
      !$omp parallel do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      c(i,j) = c(i,j)+acc
   end do
end do
end subroutine mmproduct
```

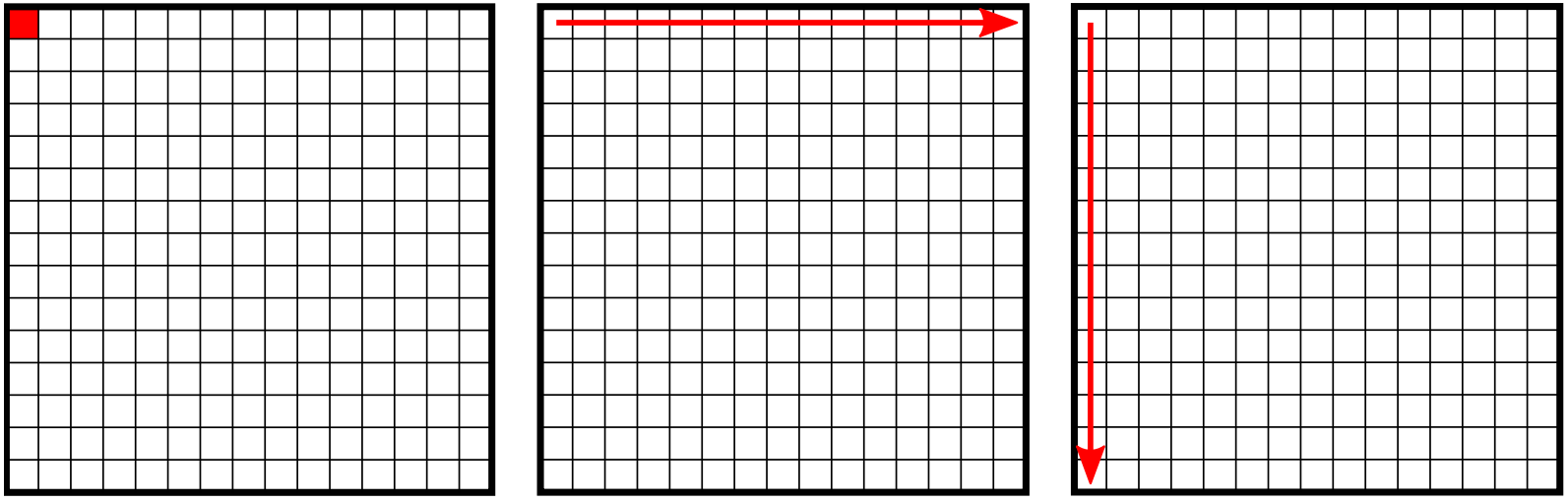
OpenMP MM product



```
subroutine mmproduct(a, b, c)

do i=1, n
   do j=1, n
      acc = 0
      !$omp parallel do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      c(i,j) = c(i,j)+acc
   end do
end do
end subroutine mmproduct
```

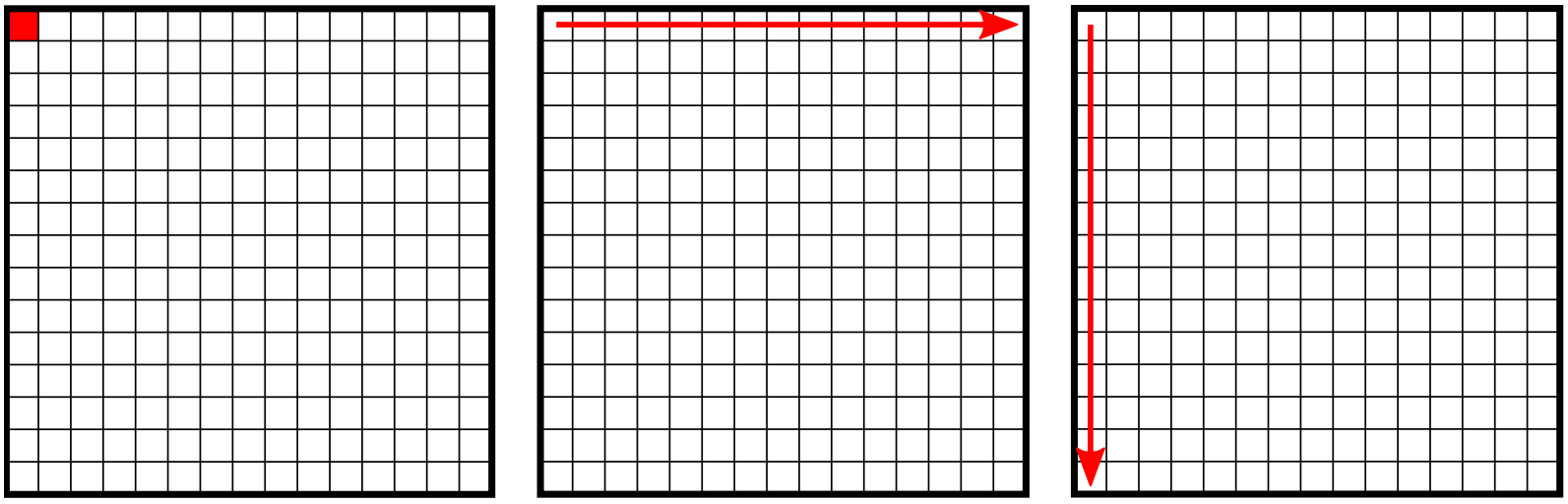
Correct parallel but low efficiency (many fork-join)

OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j,acc)
do i=1, n
   do j=1, n
      acc = 0
      !$omp do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      !$omp single
      c(i,j) = c(i,j)+acc
      !$omp end single
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

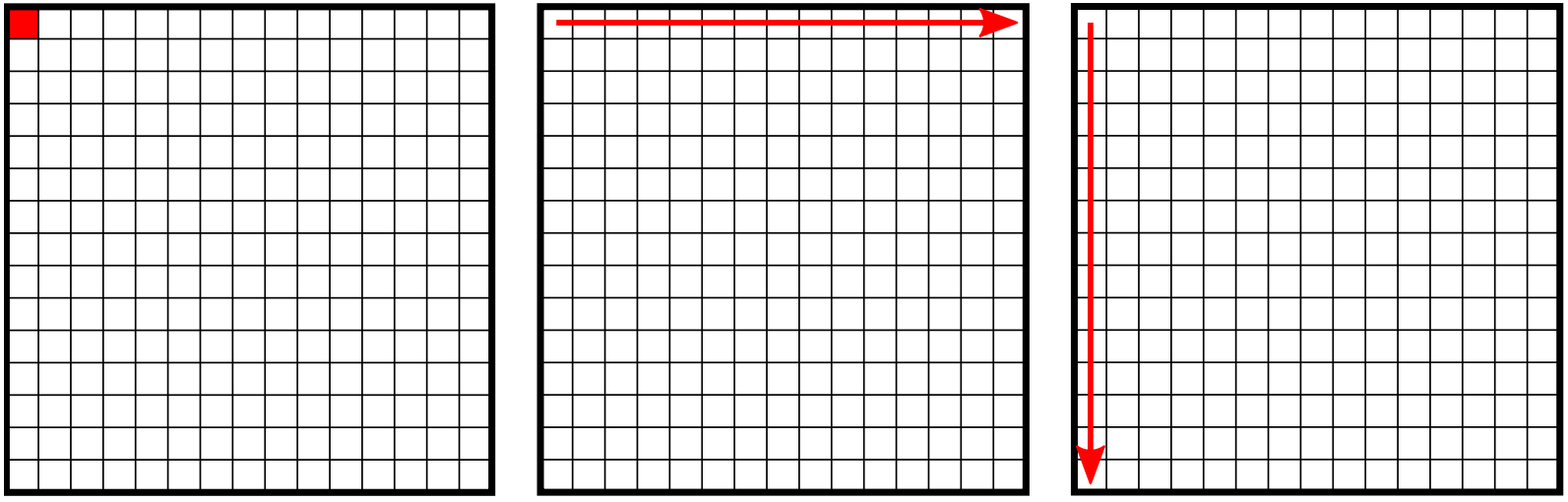
OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j,acc)
do i=1, n
   do j=1, n
      acc = 0
      !$omp do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      !$omp single
      c(i,j) = c(i,j)+acc
      !$omp end single
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

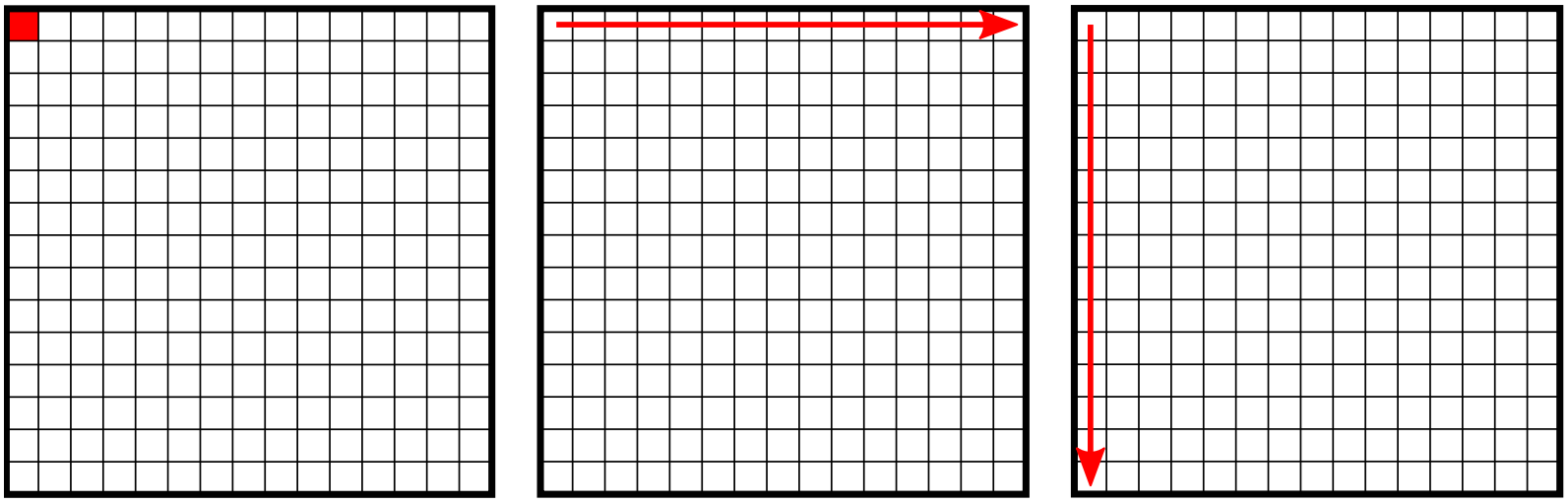
Correct parallel but still low efficiency

OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j,acc)
do i=1, n
   do j=1, n
      acc = 0
      !$omp do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      !$omp atomic update
      c(i,j) = c(i,j)+acc
      !$omp end atomic
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

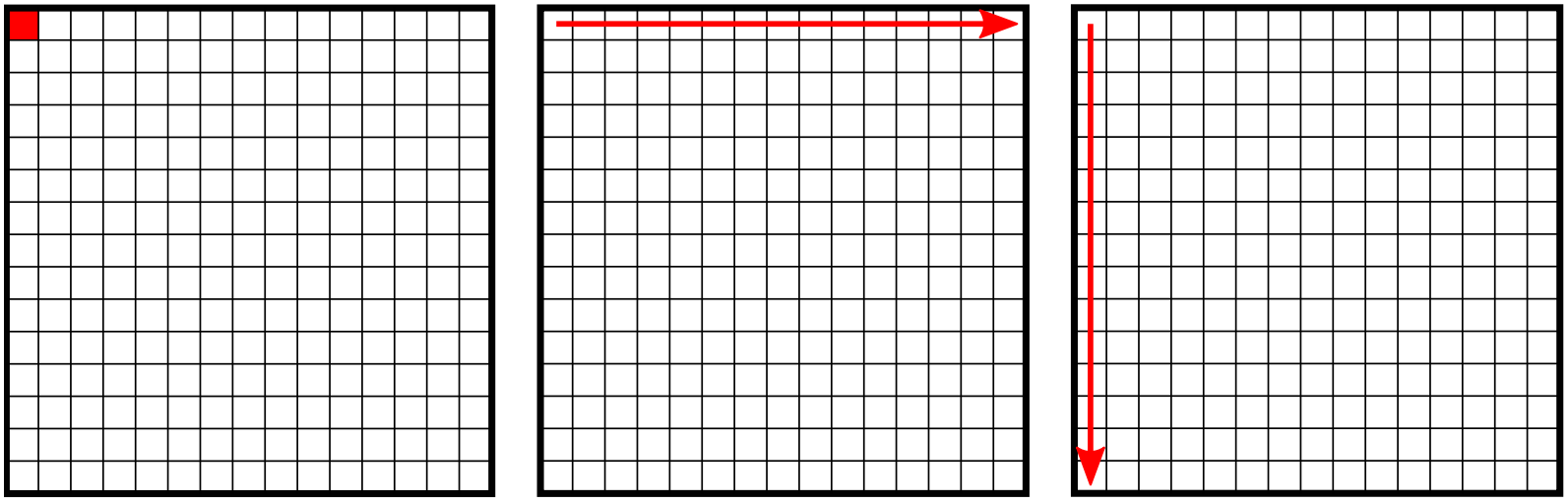
OpenMP MM product



```
subroutine mmproduct(a, b, c)
!$omp parallel private(i,j,acc)
do i=1, n
   do j=1, n
      acc = 0
      !$omp do reduction(+:acc)
      do k=1, n
         acc = acc+a(i,k)*b(k,j)
      end do
      !$omp end do
      !$omp atomic update
      c(i,j) = c(i,j)+acc
      !$omp end atomic
   end do
end do
!$omp end parallel
end subroutine mmproduct
```

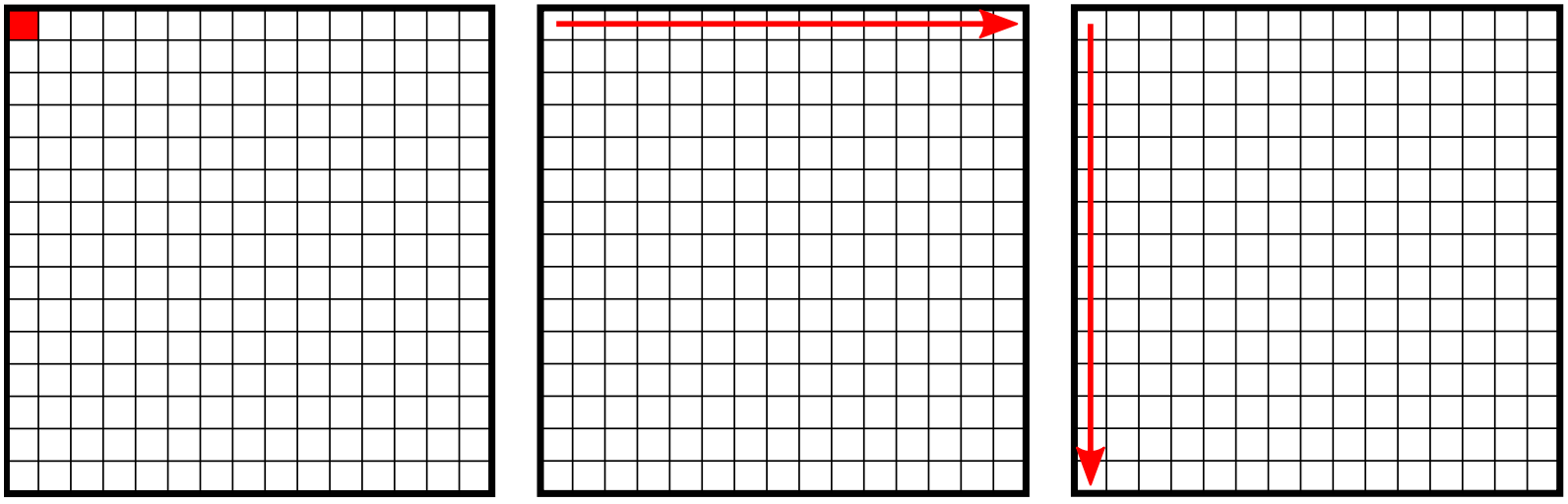
Slightly better but still not optimal

OpenMP MM product



```
subroutine mmproduct(a, b, c)

!$omp parallel do private(j,k)
do i=1, n
   do j=1, n
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
   end do
end do
!$omp end parallel do
end subroutine mmproduct
```

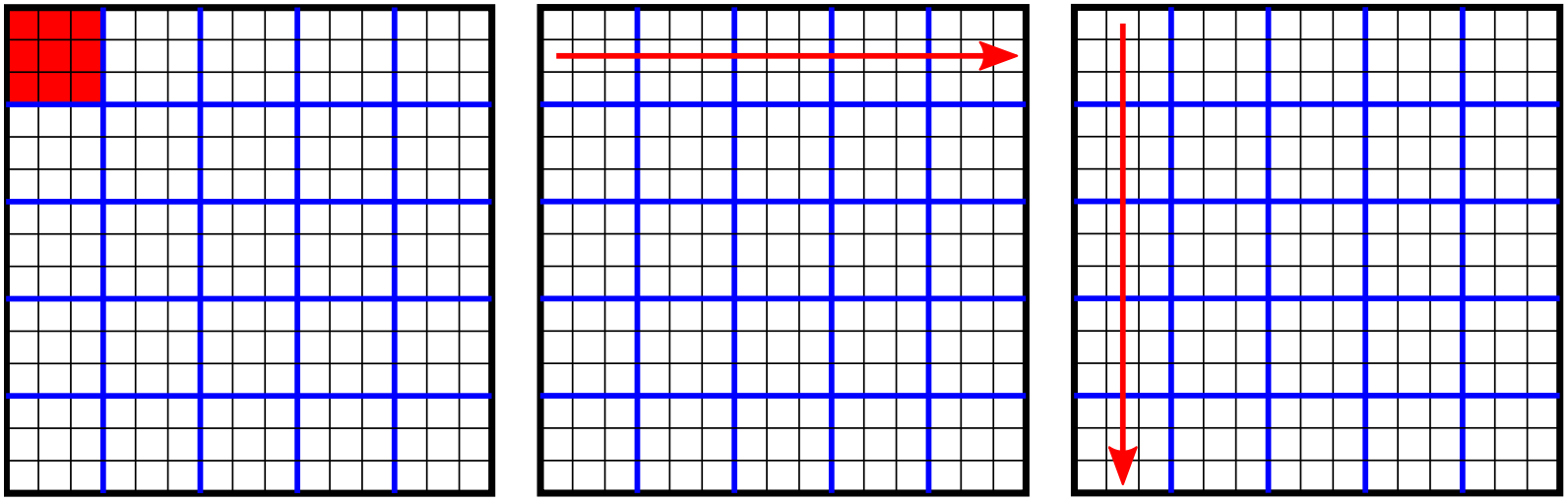
OpenMP MM product



```
subroutine mmproduct(a, b, c)

!$omp parallel do private(j,k)
do i=1, n
   do j=1, n
      do k=1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      end do
   end do
end do
!$omp end parallel do
end subroutine mmproduct
```

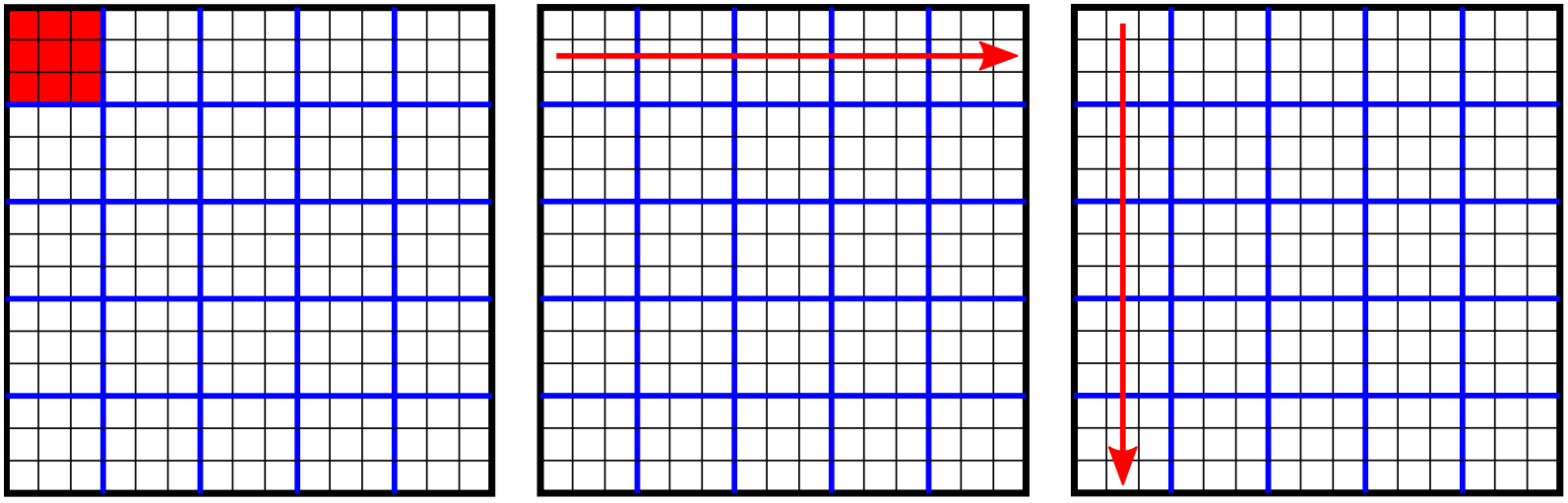
Correct parallel and good performance

OpenMP MM product



```
subroutine mmproduct(a, b, c)
...

do i=1, n, nb
   do j=1, n, nb
      do k=1, n, nb
         c(i:i+nb-1,j:j+nb-1) = c(i:i+nb-1,j:j+nb-1)+ &
            & matmul(a(i:i+nb-1,k:k+nb-1), b(k:k+nb-1,j:j+nb-1))
      end do
   end do
end do

end subroutine mmproduct
```

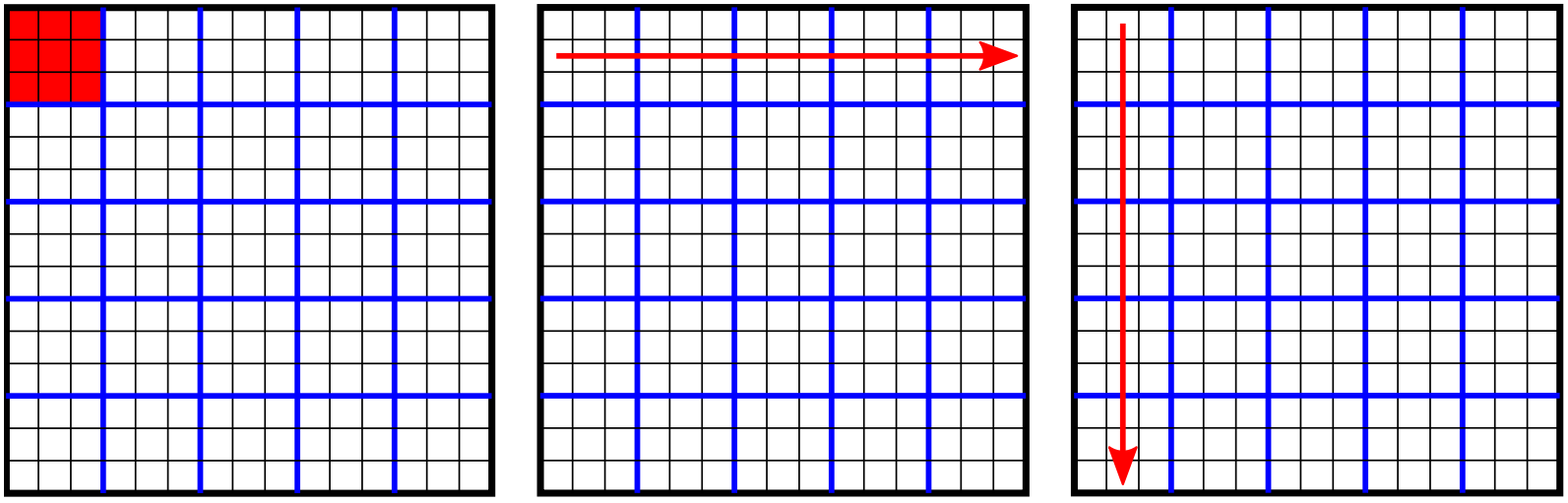
OpenMP MM product



```
subroutine mmproduct(a, b, c)
...

do i=1, n, nb
   do j=1, n, nb
      do k=1, n, nb
         c(i:i+nb-1,j:j+nb-1) = c(i:i+nb-1,j:j+nb-1)+ &
            & matmul(a(i:i+nb-1,k:k+nb-1), b(k:k+nb-1,j:j+nb-1))
      end do
   end do
end do

end subroutine mmproduct
```

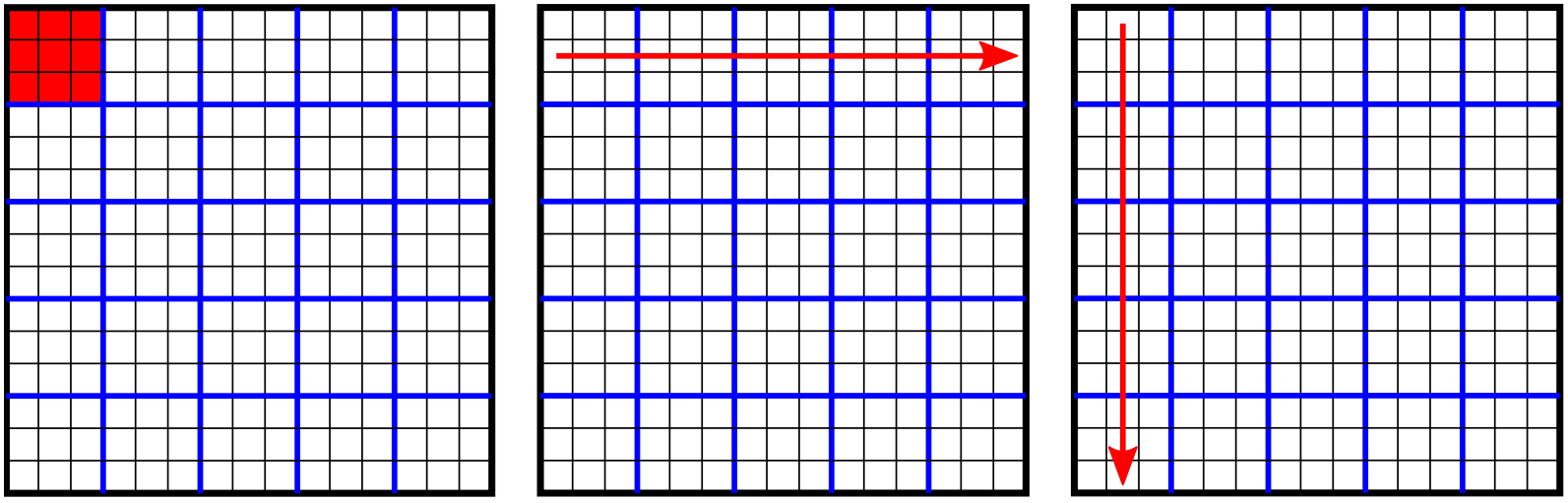
Optimized version by blocking

OpenMP MM product



```
subroutine mmproduct(a, b, c)
...
!$omp parallel do
do i=1, n, nb
   do j=1, n, nb
      do k=1, n, nb
         c(i:i+nb-1,j:j+nb-1) = c(i:i+nb-1,j:j+nb-1)+ &
            & matmul(a(i:i+nb-1,k:k+nb-1), b(k:k+nb-1,j:j+nb-1))
      end do
   end do
end do
!$omp parallel end do
end subroutine mmproduct
```

OpenMP MM product



```
subroutine mmproduct(a, b, c)
...
!$omp parallel do
do i=1, n, nb
   do j=1, n, nb
      do k=1, n, nb
         c(i:i+nb-1,j:j+nb-1) = c(i:i+nb-1,j:j+nb-1)+ &
            & matmul(a(i:i+nb-1,k:k+nb-1), b(k:k+nb-1,j:j+nb-1))
      end do
   end do
end do
!$omp parallel end do
end subroutine mmproduct
```

Optimized parallel version

OpenMP MM product

```
subroutine mmproduct(a, b, c)
...
!$omp parallel do
do i=1, n, nb
   do j=1, n, nb
      do k=1, n, nb
         c(i:i+nb-1,j:j+nb-1) = c(i:i+nb-1,j:j+nb-1)+a(i:i+nb-1,k:k+nb-1)*b(k:k+nb
     -1,j:j+nb-1)
      end do
   end do
end do
!$omp parallel end do
end subroutine mmproduct
```

```
    1  Threads --->   4.29  Gflop/s
    2  Threads --->   8.43  Gflop/s
    4  Threads --->  16.57  Gflop/s
    8  Threads --->  31.80  Gflop/s
   16  Threads --->  55.11  Gflop/s
```

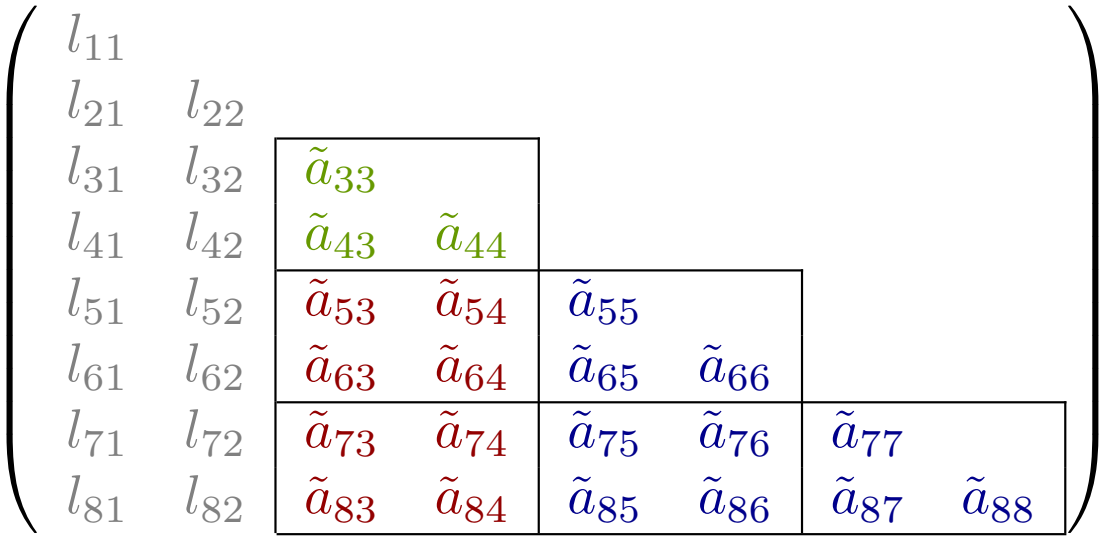
The Cholesky factorization

$$\begin{pmatrix}
l_{11} & & & & & & & \\
l_{21} & l_{22} & & & & & & \\
l_{31} & l_{32} & \tilde{a}_{33} & & & & & \\
l_{41} & l_{42} & \tilde{a}_{43} & \tilde{a}_{44} & & & & \\
l_{51} & l_{52} & \tilde{a}_{53} & \tilde{a}_{54} & \tilde{a}_{55} & & & \\
l_{61} & l_{62} & \tilde{a}_{63} & \tilde{a}_{64} & \tilde{a}_{65} & \tilde{a}_{66} & & \\
l_{71} & l_{72} & \tilde{a}_{73} & \tilde{a}_{74} & \tilde{a}_{75} & \tilde{a}_{76} & \tilde{a}_{77} & \\
l_{81} & l_{82} & \tilde{a}_{83} & \tilde{a}_{84} & \tilde{a}_{85} & \tilde{a}_{86} & \tilde{a}_{87} & \tilde{a}_{88}
\end{pmatrix}$$

```
do k=1, n

   a(k,k) = sqrt(a(k,k))

   do i=k+1, n
      a(i,k) = a(i,k)/a(k,k)

      do j=k+1, n
         a(i,j) = a(i,j) - a(i,k)*a(j,k)
      end do
   end do

end do
```

The unblocked Cholesky factorization is extremely inefficient due to a poor cache reuse. No level-3 BLAS operations possible.

The Cholesky factorization




```
do k=1, n/nb
   call dpotf2( Ab(k,k) )
   do i=k+1, n/nb
      call dtrsm ( Ab(i,k), Ab(k,k) )
      do j=k+1, i
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
      end do
   end do
end do
```

The matrix can be logically split into blocks of size $nb \times nb$ and the factorization written exactly as the non blocked where operations on single values are replaced by equivalent operations on blocks. `Ab` is the same matrix but with a block storage

Blocked Cholesky: multithreading

First tentative:

```
!$omp parallel do
do k=1, n/nb
   call dpotf2( Ab(k,k) )

   do i=k+1, n/nb
      call dtrsm ( Ab(i,k), Ab(k,k) )

      do j=k+1, i
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
      end do

   end do

end do
!$omp end parallel
```

WRONG!

This parallelization will lead to incorrect results. The steps of the blocked factorization have to be performed in the right order.

Blocked Cholesky: multithreading

Second tentative:

```
do k=1, n/nb
   call dpotf2( Ab(k,k) )
   !$omp parallel do
   do i=k+1, n/nb
      call dtrsm ( Ab(i,k), Ab(k,k) )

      do j=k+1, i
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
      end do

   end do
   !$omp end parallel
end do
```

WRONG!

This parallelization will lead to incorrect results. At step `step`, the `dpoup` operation on block `a(row,col)` depends on the result of the `dtrsm` operations on blocks `a(row,step)` and `a(col,step)`. This parallelization only respects the dependency on the first one.

Blocked Cholesky: multithreading

Third tentative:

```
do k=1, n/nb
   call dpotf2( Ab(k,k) )

   do i=k+1, n/nb
      call dtrsm ( Ab(i,k), Ab(k,k) )
      !$omp parallel do
      do j=k+1, i
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
      end do
      !$omp end parallel
   end do

end do
```

CORRECT!

This parallelization will lead to correct results. Because, at each step, the order of the `dtrsm` operations is respected, once the `dtrsm` operation on block `a(row,step)` is done, all the updates along row `row` can be done independently. Not really efficient.

Blocked Cholesky: multithreading

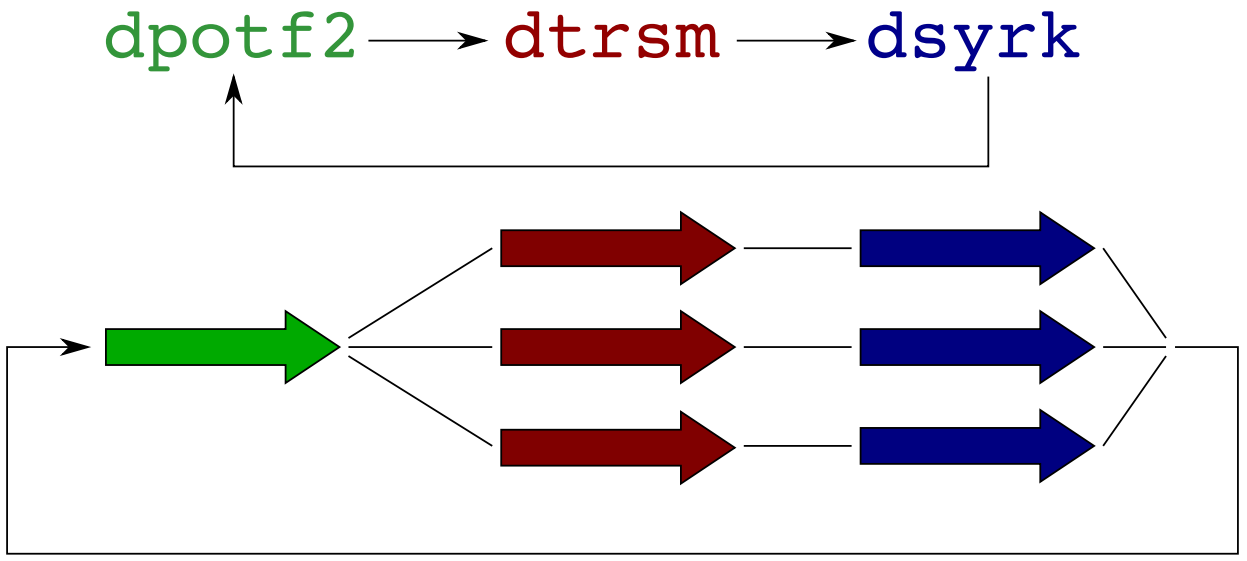
Fourth tentative:

```
do k=1, n/nb
   call dpotf2( Ab(k,k) )

   !$omp parallel do
   do i=k+1, n/nb
      call dtrsm ( Ab(i,k), Ab(k,k) )
   end do
   !$omp end parallel

   do i=k+1, n/nb
      !$omp parallel do
      do j=k+1, i
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
      end do
      !$omp end parallel
   end do

end do
```

CORRECT and more EFFICIENT!

All the `dtrsm` operations at step `step` are independent and can be done in parallel. Because all the `dtrsm` are done before the updates, these can be done in parallel too. But not optimal.

Blocked Cholesky: multithreading



Fork-join parallelism suffers from:

- poor parallelism: some operations are inherently sequential and pose many constraints to the parallelization of the whole code
- synchronizations: any fork or join point is a synchronization point. This makes the parallel flow of execution extremely constrained, increases the idle time, limits the scalability

Blocked Cholesky: better multithreading

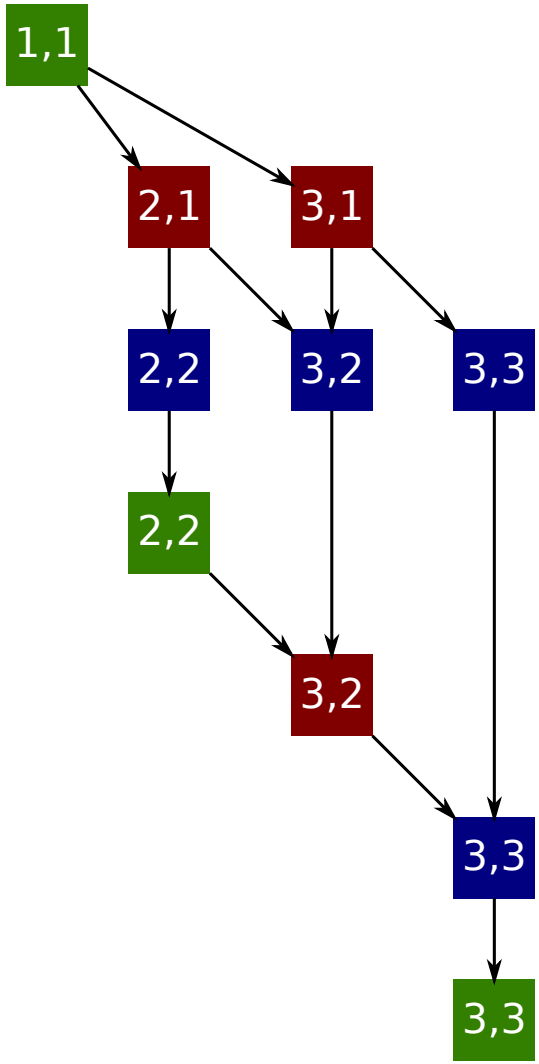
All the previous parallelization approaches are based on the assumption that step `step+1` can be started only when all the operations related to step `step` are completed. This constraint is too strict and can be partially relaxed.
Which conditions have to be necessarily respected?

1. the `dpotf2` operation on the diagonal block `a(step,step)` can be done only if the block is up to date with respect to step `step-1`
2. the `dtrsm` operation on block `a(row,step)` can be done only if the block is up to date with respect to step `step-1` and the `dpotf2` of block `a(step,step)` is completed
3. the `dpoup` of block `a(row,col)` at step `step` can be done only if the block is up to date with respect to step `step-1` and the `dtrsm` of blocks `a(row,step)` and `a(col,step)` at step `step` are completed

Blocked Cholesky: better multithreading

How is it possible to handle all this complexity? The order of the operations may be captured in a Directed Acyclic Graph where nodes define the computational tasks and edges the dependencies among them. Tasks in the DAG may be dynamically scheduled.

- fewer dependencies, i.e., fewer synchronizations and high flexibility for the scheduling of tasks
- no idle time
- adaptativity
- better scaling



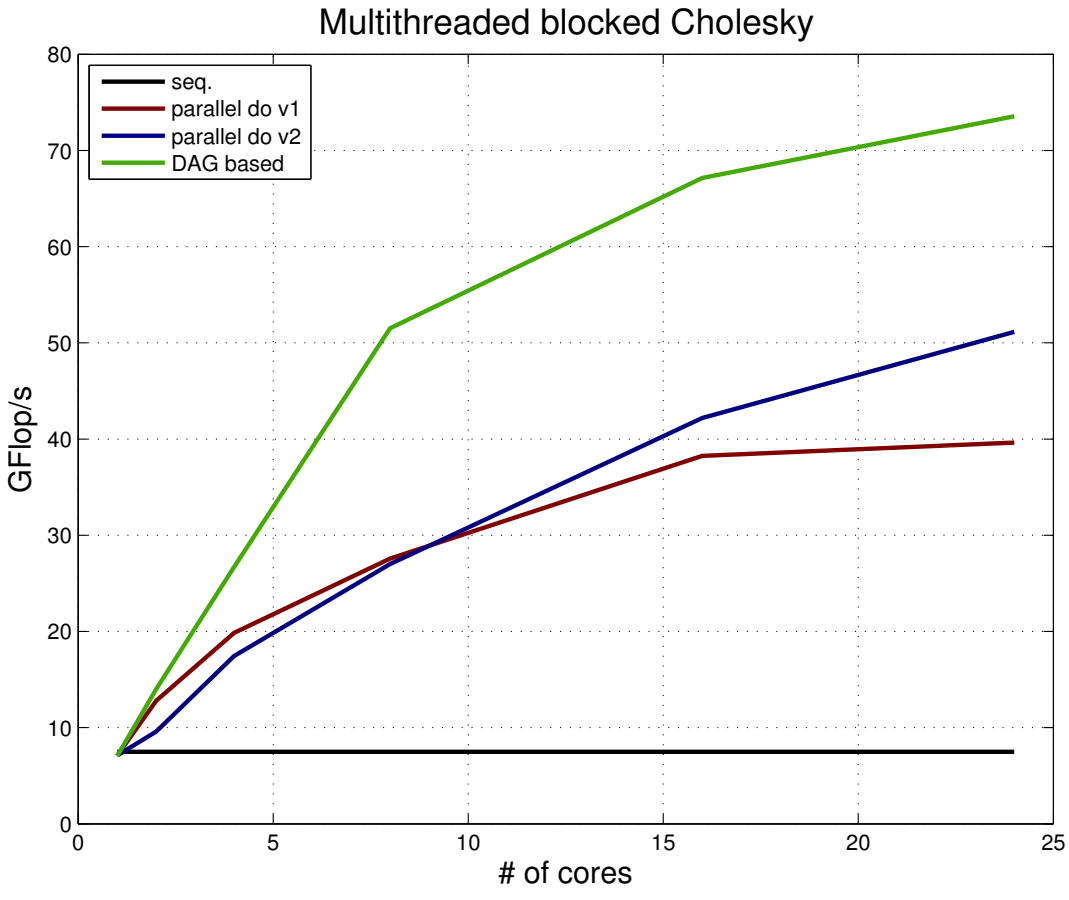
Blocked Cholesky: multithreading

DAG parallelism:

```
!$omp parallel
!$omp single nowait
do k=1, n/nb
   !$omp task depend(inout:Ab(k,k)) firstprivate(k)
   call dpotf2( Ab(k,k) )
   !$omp end task

   do i=k+1, n/nb
      !$omp task depend(in:Ab(k,k)) depend(inout:Ab(i,k)) firstprivate(i,k)
      call dtrsm ( Ab(i,k), Ab(k,k) )
      !$omp end task
      do j=k+1, i
         !$omp task depend(in:Ab(i,k),Ab(j,k)) depend(inout:Ab(i,j))
         !$omp& firstprivate(i,j,k)
         call dpoup ( Ab(i,j), Ab(i,k), Ab(j,k) )
         !$omp end task
      end do
   end do

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

OpenMP is capable to automatically build the DAG by looking at the specified dependencies and then schedule the tasks accordingly

Blocked Cholesky: better multithreading



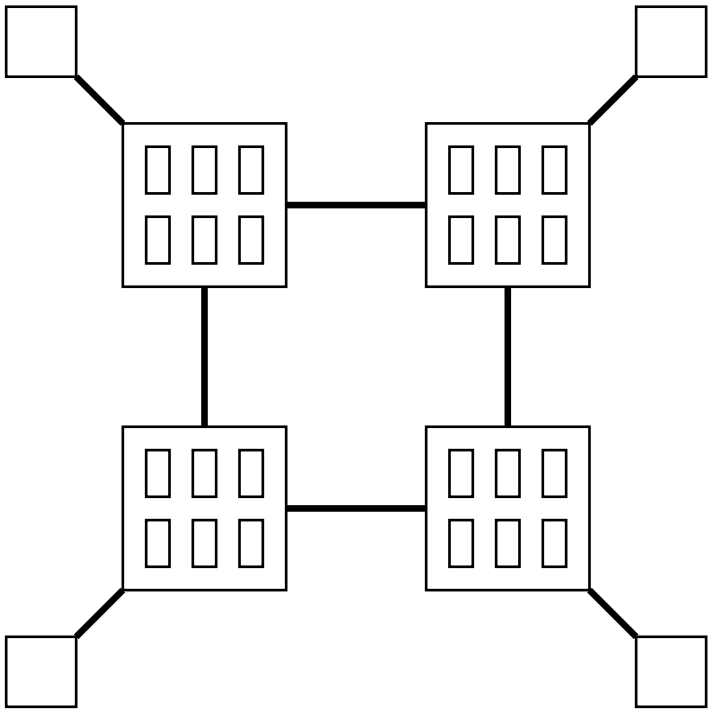
Outline

Section 4

OpenMP: odds & ends

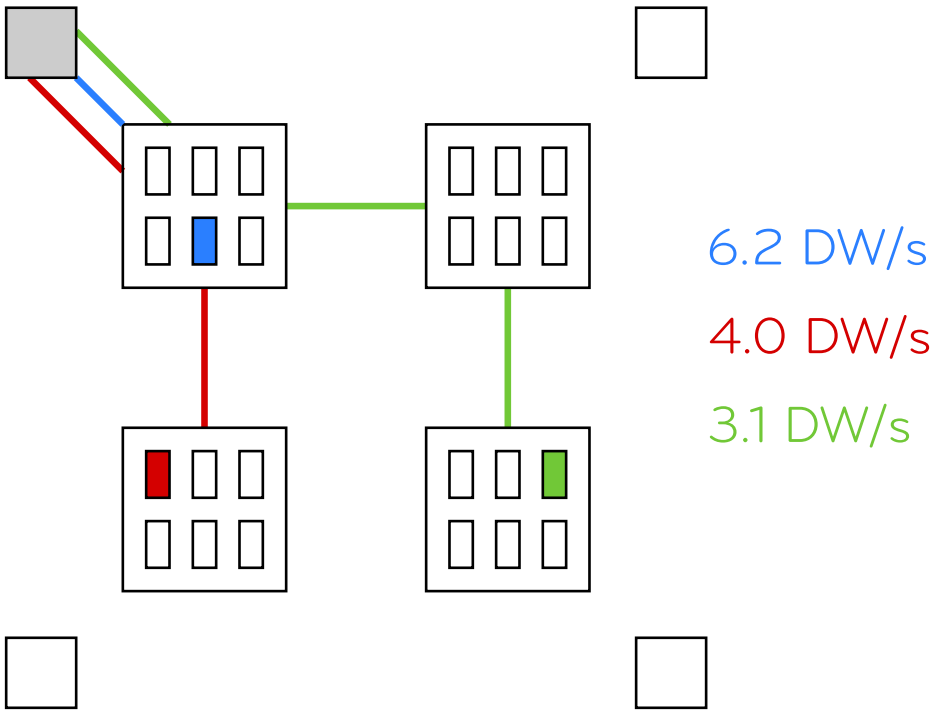
NUMA: Memory locality

Even if every core can access any memory module, data will be transferred at different speeds depending on the distance (number of hops)



NUMA: Memory locality

Even if every core can access any memory module, data will be transferred at different speeds depending on the distance (number of hops)



NUMA: memory locality

If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?

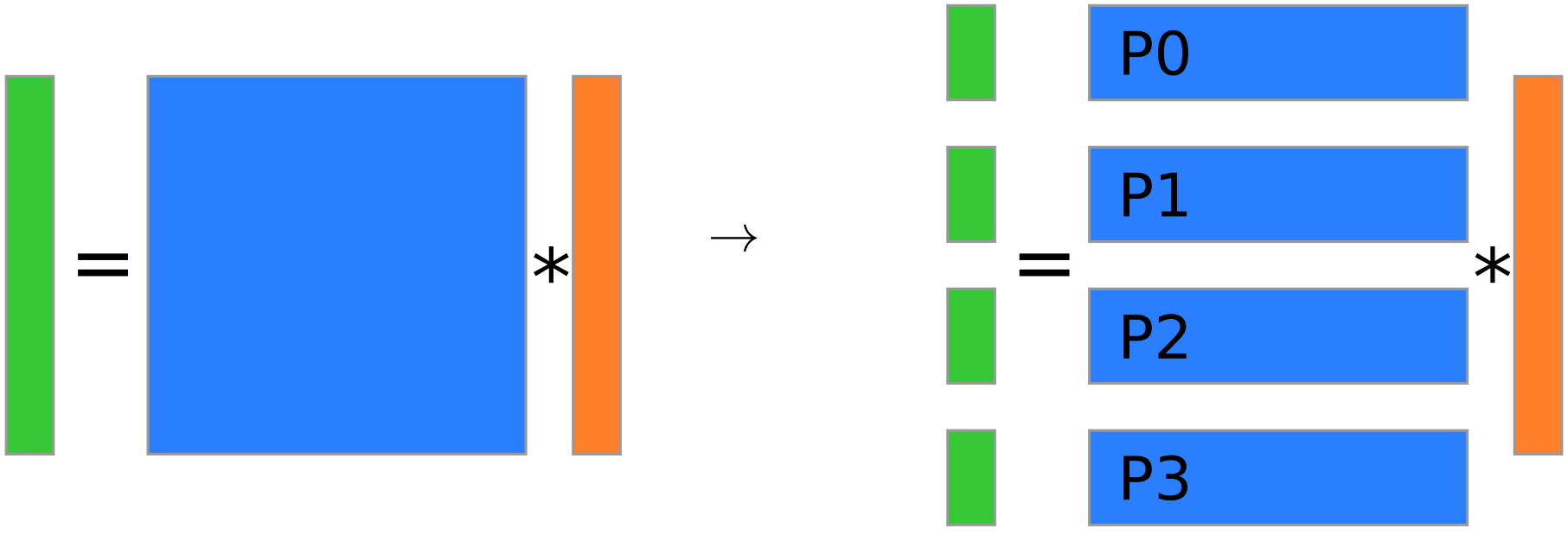
NUMA: memory locality

If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?

If all the data is stored on only one memory module, the memory bandwidth will be low and the conflicts/contentions will be high.

NUMA: memory locality

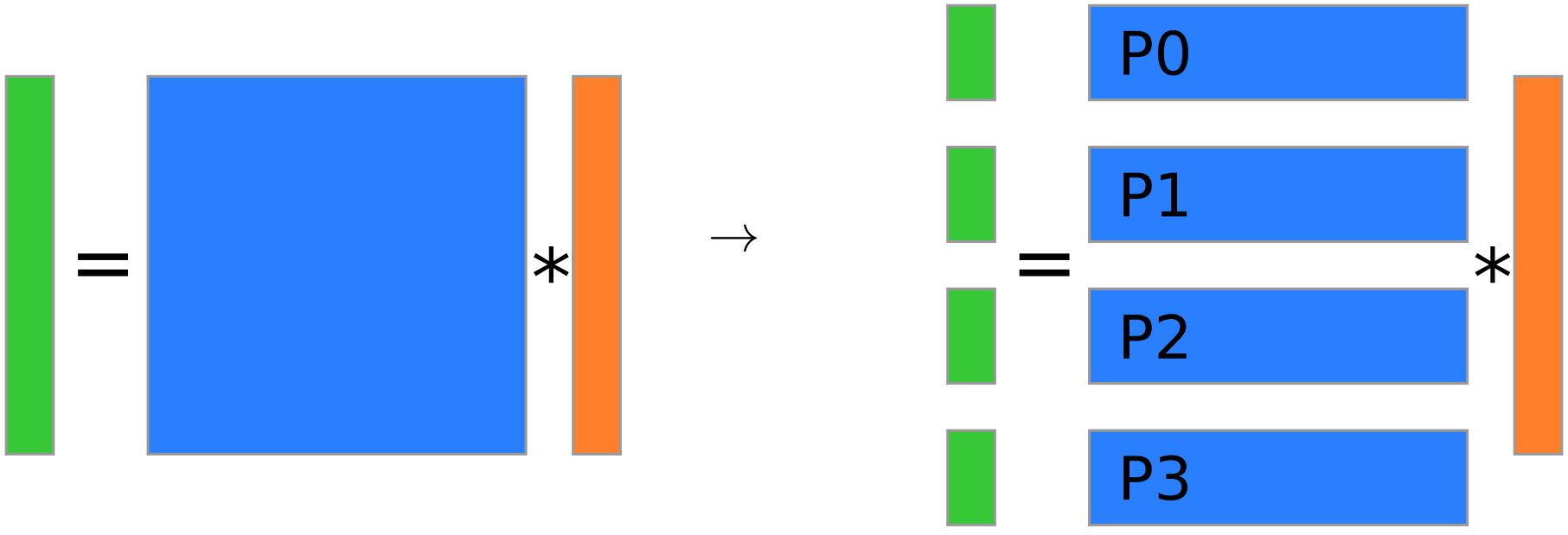
If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?



If all the data is stored on only one memory module, the memory bandwidth will be low and the conflicts/contentions will be high.

NUMA: memory locality

If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?



If all the data is stored on only one memory module, the memory bandwidth will be low and the conflicts/contentions will be high. When possible, it is good to partition the data, store partitions on different memory modules and force each core to access only local data.

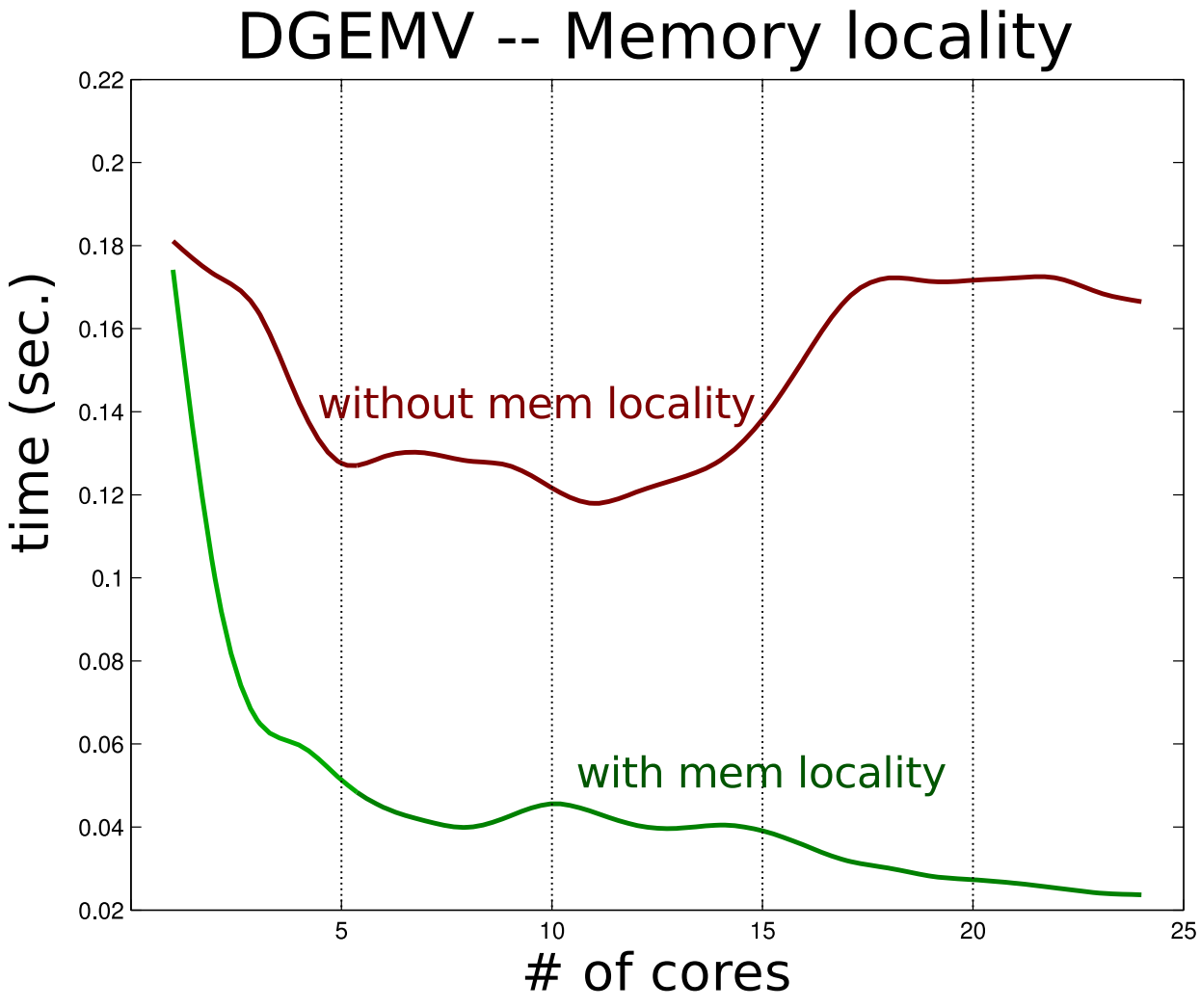
NUMA: memory locality

Implementing all this requires the ability to:

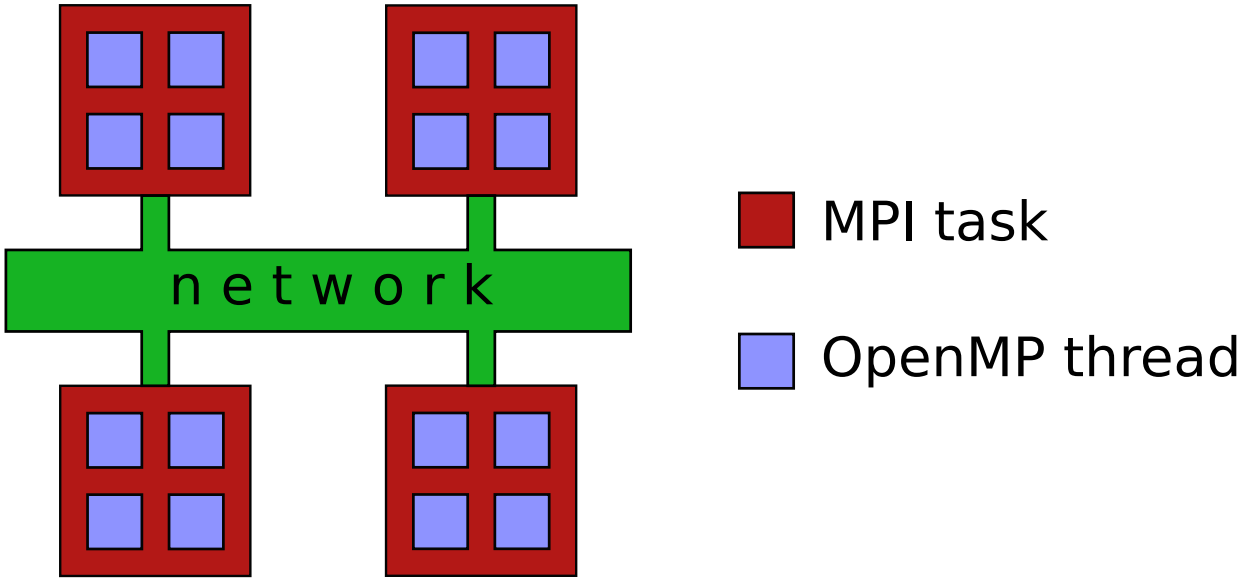
- control the placement of threads: we have to bind each thread to a single core and prevent threads migrations. This can be done in a number of ways, e.g. by means of tools such as `hwloc` which allows thread pinning
- control the placement of data: we have to make sure that one front physically resides on a specific NUMA module. This can be done with:
  - the first touch rule: the data is allocated close to the core that makes the first reference
  - `hwloc` or `numalib` which provide NUMA-aware allocators
- detect the architecture we have to figure out the memory/cores layout in order to guide the work stealing. This can be done with `hwloc`

NUMA: memory locality

When this optimization is applied much better performance and scalability is achieved:



Hybrid parallelism



How to exploit parallelism in a cluster of SMPs/Multicores? There are two options:

- Use MPI all over: MPI works on distributed memory systems as well as on shared memory
- Use an MPI/OpenMP hybrid approach: define one MPI task for each node and one OpenMP thread for each core in the node.

Hybrid parallelism

```
program hybrid

  use mpi

  integer  :: mpi_id, ierr, mpi_nt
  integer  :: omp_id, omp_nt, &
              & omp_get_num_threads, &
              & omp_get_thread_num

  call mpi_init(ierr)

  call mpi_comm_rank(mpi_comm_world, mpi_id, ierr)
  call mpi_comm_size(mpi_comm_world, mpi_nt, ierr)

!$omp parallel
  omp_id = omp_get_thread_num()
  omp_nt = omp_get_num_threads()

  write(*,'("Thread ",i1,"(",i1,") &
        & within MPI task ",i1,"(",i1,")")') &
        & omp_id,omp_nt,mpi_id,mpi_nt

!$omp end parallel

end program hybrid
```

result

```
Thread 0(2) within MPI task 0(2)
Thread 0(2) within MPI task 1(2)
Thread 1(2) within MPI task 1(2)
Thread 1(2) within MPI task 0(2)
```

Outline

Appendix: routines for blocked Cholesky

- `dpotf2`: this LAPACK routine does the unblocked Cholesky factorization of a symmetric positive definite matrix using only the lower or upper triangular part of the matrix
- `dtrsm`: this BLAS routine does the solution of the problem AX=B where A is a lower or upper triangular matrix and B is a matrix containing multiple right-hand-sides
- `dgemm`: this BLAS routine performs a product of the type C=alpha*A*B+beta*c where alpha and beta are scalars, A, B and C are dense matrices
- `dsyrk`: this BLAS routine performs a symmetric rank-k update of the type A=B*B'+alpha*A where alpha is a scalar, A is a symmetric matrix and B a rank-k matrix updating only the upper or lower triangular part of A
- `dpoup`: this routine (not in BLAS nor in LAPACK) calls the `dgemm` or the `dsyrk` routine to perform an update on an off-diagonal block or a diagonal block, respectively