Multicore and Multicore programming 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

Section 1

Introduction

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.

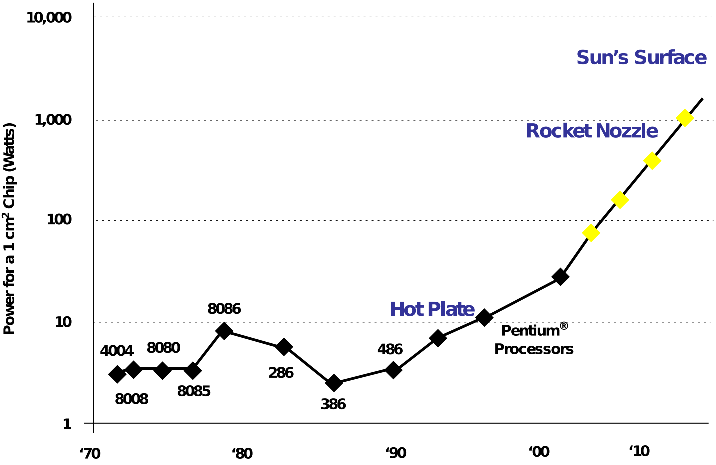
$$
\begin{aligned}
&P \simeq P_{dynamic} = CV^2 f \\
&P_{dynamic} = dynamic \quad power \\
&C = capacitance \\
&V = voltage \\
&f = frequency
\end{aligned}
$$

but

$$f_{max} \sim V$$

Power consumption and heat dissipation grow as f^3!

The Power wall



Source: Pat Gelsinger, Intel, ISSCC 2001

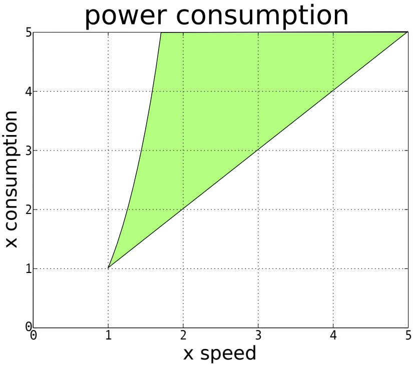
The Power wall

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.

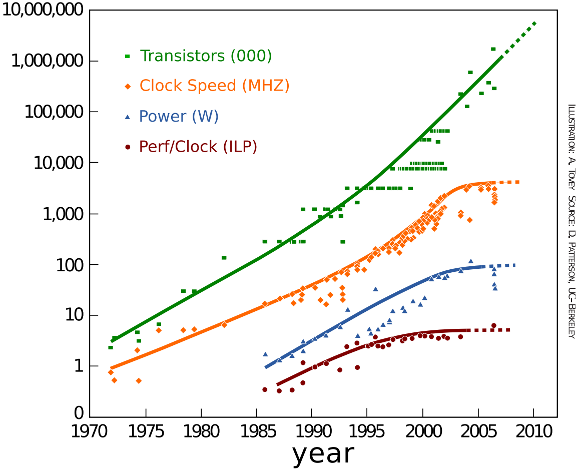
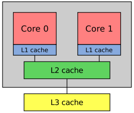


ILLUSTRATION: A. TOVEY SOURCE: D. PATTERSON, UC-BERKELEY

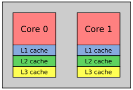
Examples of multicore architectures

Power 4 (2001)



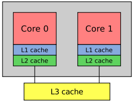
Core 0 Core 1
L1 cache L1 cache
L2 cache
L3 cache

Intel Itanium2 (2004)



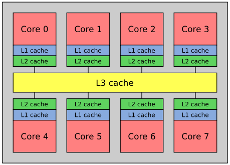
Core 0 Core 1
L1 cache L1 cache
L2 cache L2 cache
L3 cache L3 cache

AMD Opteron (2005)



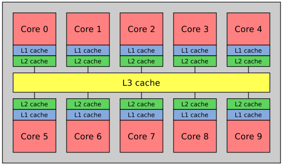
Core 0 Core 1
L1 cache L1 cache
L2 cache L2 cache
L3 cache

Power 7 (2010)



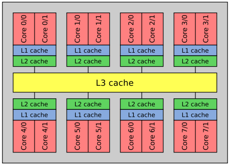
Core 0 Core 1 Core 2 Core 3
L1 cache L1 cache L1 cache L1 cache
L2 cache L2 cache L2 cache L2 cache
L3 cache
L2 cache L2 cache L2 cache L2 cache
L1 cache L1 cache L1 cache L1 cache
Core 4 Core 5 Core 6 Core 7

Intel Xeon-E7 (2011)



Core 0 Core 1 Core 2 Core 3 Core 4
L1 cache L1 cache L1 cache L1 cache L1 cache
L2 cache L2 cache L2 cache L2 cache L2 cache
L3 cache
L2 cache L2 cache L2 cache L2 cache L2 cache
L1 cache L1 cache L1 cache L1 cache L1 cache
Core 5 Core 6 Core 7 Core 8 Core 9

AMD Bulldozer (2011)



Core 0/0 Core 0/1 Core 1/0 Core 1/1 Core 2/0 Core 2/1 Core 3/0 Core 3/1
L1 cache L1 cache L1 cache L1 cache
L2 cache L2 cache L2 cache L2 cache
L3 cache
L2 cache L2 cache L2 cache L2 cache
L1 cache L1 cache L1 cache L1 cache
Core 4/0 Core 4/1 Core 5/0 Core 5/1 Core 6/0 Core 6/1 Core 7/0 Core 7/1

Conventional Multicores

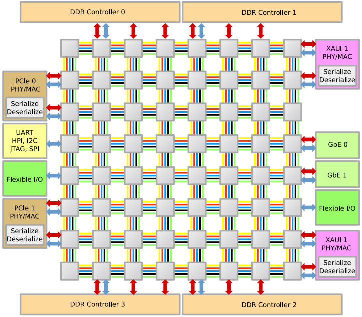
What are the problems with all these designs?

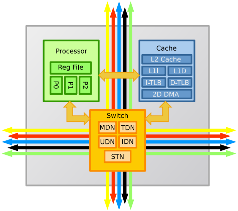
- Core-to-core communication. Although cores lie on the same piece of silicon, there is no direct communication channel between them. The only option is to communicate through main memory.
- Shared memory bus. On modern systems, processors are much faster than memory; example:
  Intel Woodcrest:
  - 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.



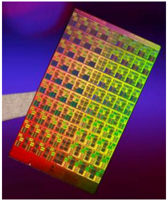


- 4.5 TB/s on-chip mesh interconnect
- 25 GB/s towards main memory
- no floating-point

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

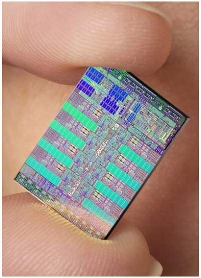


That makes $4(FLOPS) \times 80(tiles) \times 3.16GHz \simeq 1TFlop/s$. The first TFlop machine was the ASCII Red made up of 10000 Pentium Pro, taking 250 mq and 500 KW...

The IBM Cell

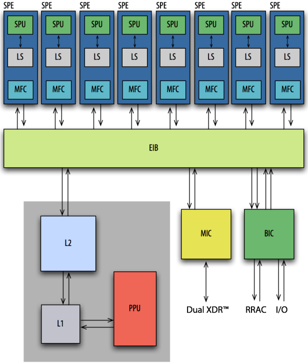
The *Cell Broadband Engine* was released in 2005 by the STI (Sony Toshiba IBM) consortium. It is an 9-way multicore processor.

- 1 control core + 8 working cores
- computational power is achieved through exploitation of two levels of parallelism:
  - vector units
  - multiple cores
- on-chip interconnect bus for core-to-core communications
- caches are replaced by explicitly managed local memories
- performance comes at a price: the Cell is very hard to program



The Cell: architecture

- one POWER Processing Element (PPE): this is almost like a PowerPC processor (it does not have some ILP features) and it is almost exclusively meant for control work.
- 8 Synergistic Processing Elements (SPEs) (only 6 in the PS3)
- one Element Interconnect Bus (EIB): on-chip ring bus connecting all the SPEs and the PPE
- one Memory Interface Controller (MIC) that connects the EIB to the main memory
- PPE and SPEs have different ISAs and thus we have to write different code and use different compilers



Hello world! example

The PPU and the SPUs have different ISAs (Instruction Set Architecture), therefore two different compilers must be used.

PPE code

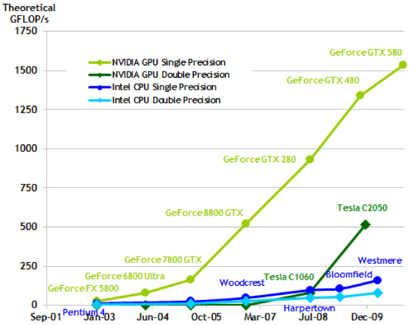
```c
#include <stdio.h>
#include <libspe.h>
#include <sys/wait.h>

extern spe_program_handle_t ;
extern hello_spu;

int main(void){
  speid_t speid[8];
  int status[8];
  int i;
  for (i=0;i<8;i++)
    speid[i] = spe_create_thread(0, &hello_spu,
                 NULL, NULL, -1, 0);
  for (i=0;i<8;i++){
      spe_wait(speid[i], &status[i], 0);

      printf("status = %d\n",
                    WEXITSTATUS(status[i]));
    }
  return 0;
}
```

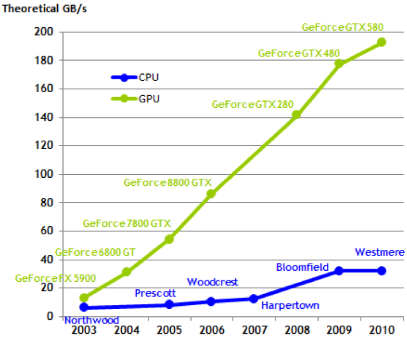
SPE code

```c
#include <stdio.h>

int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp){

  printf("Hello world (0x%llx)\n", speid);

  return 0;
}
```

Other computing devices: GPUs

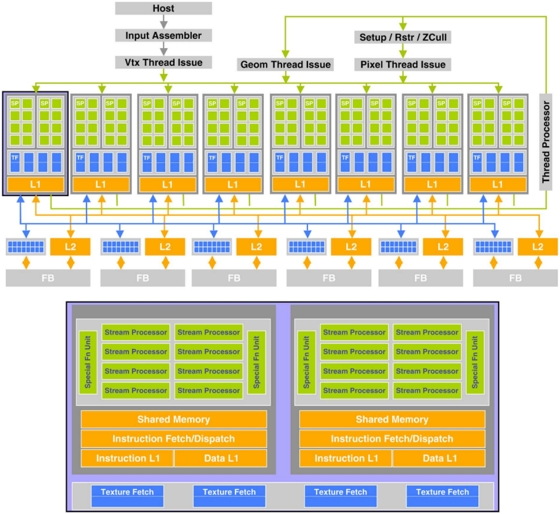
NVIDIA GPUs vs Intel processors: performance





Other computing devices: GPUs

NVIDIA GeForce 8800 GTX:



16 streaming multiprocessors of 8 thread processors each.

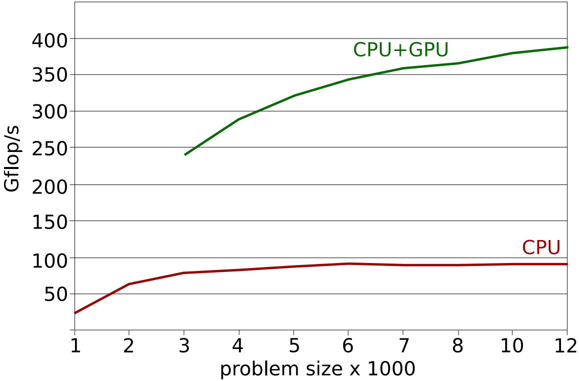
Other computing devices: GPUs

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:



Section 2

Single-core performance programming

Single-core programming

Simple matrix-vector multiply-add c=c+A*b in real, single precision with A of size 32

```
void matvec(float *A, float *c, float *b)
{
  int m, n;
  for (n = 0; n < 32; n++)
    for (m = 0; m < 32; m++)
      c[m] += A[n*32+m] * b[n];
}
```

- **good news**: standard C code will compile and run correctly on basically any computer
- **bad news**: no performance, i.e., 0.63 Gflop/s ($\sim$ 3% peak on a Core i7 @ 2.66 GHz)

Code available at:

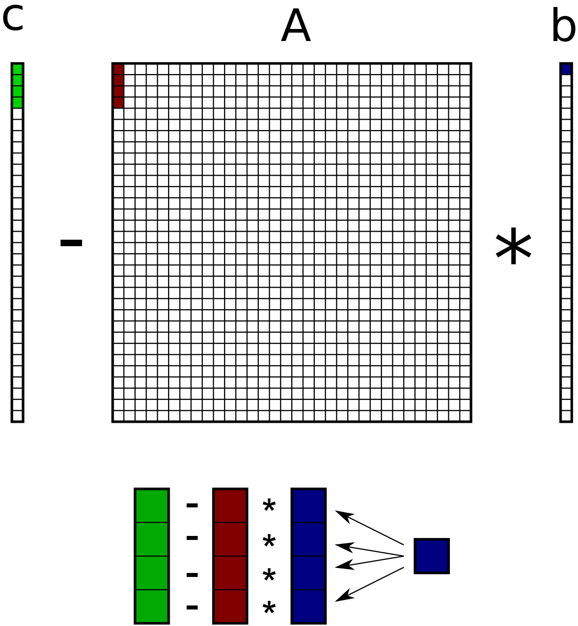
http://buttari.perso.enseeiht.fr/stuff/matvec.c

Vectorization

Most modern processors are equipped with vector units. They make it possible to perform the same operation on multiple data with a single instruction. For this reason they are also called SIMD (Single Instruction Multiple Data) units:

- x86 processors (Intel and AMD) have SSE (Streaming SIMD Extensions) have 128-bit vectors and can do either 4 single or 2 double precision operations
- Power processors (IBM) have AltiVec have 128-bit vectors and can do either 4 single or 2 double precision operations. AltiVecs moreover can also do fused multiply-add

Vectorization



Vectorization

The Intel compiler includes a library of intrinsics, i.e., instructions that are specific to the x86 architecture. They can be used to vectorize a code. GNU compilers have similar features as well as others (e.g., IBM XL)

```
#include "xmmintrin.h"

__m128 Av, cv, mul, bv;                   // declare vectors

for(j=0; j<32; j++){
  bv = _mm_load1_ps(&b[j]);               // splat one coefficient of b
  for(i=0; i<32; i+=4){
    Av  = _mm_loadu_ps(&A[j*N+i]);        // load 4 values in A(:,j)
    cv  = _mm_loadu_ps(&c[i]);            // load 4 coefficients of c
    mul = _mm_mul_ps(Av, bv);             // multiply
    cv  = _mm_add_ps(mul, cv);            // add
    _mm_storeu_ps(&c[i], cv);             // store the result in c
  }
}
```

Performance now is 1.26 Gflop/s, i.e., $\sim$ 6% of the peak

Unrolling

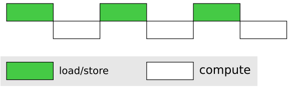
```
__m128 b;
__m128 Av0, mul0, c0; ...
__m128 Av7, mul7, c7;

c0 = _mm_loadu_ps(&c[0 ]); ...
c7 = _mm_loadu_ps(&c[28]);

for(j=0; j<B; j++){
  b   = _mm_load1_ps(&b[j]);

  Av0  = _mm_loadu_ps(&A[j*N + 0 ]); ...
  Av7  = _mm_loadu_ps(&A[j*N + 28]);

  mul0 = _mm_mul_ps(Av0, b); ...
  mul7 = _mm_mul_ps(Av7, b);

  c0  = _mm_add_ps(mul0, c0); ...
  c7  = _mm_add_ps(mul7, c7);
}

_mm_storeu_ps(&c[0 ], c0); ...
_mm_storeu_ps(&c[28], c7);
```

With a complete unrolling of the loop on rows, the c vector gets loaded only once into registers and then the result stored only once at the end.

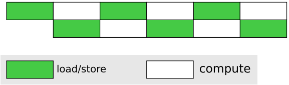
The performance is now 1.7 Gflop/s, i.e., $\sim$ 8% of the peak

Prefetching

```
__m128 b;
__m128 Av0, mul0, c0; ...
__m128 Av7, mul7, c7;
c0 = _mm_loadu_ps(&c[0 ]); ...
c7 = _mm_loadu_ps(&c[28]);

_mm_prefetch((void *)&A[0 ]); ...
_mm_prefetch((void *)&A[28]);

for(j=0; j<B; j++){
  _mm_prefetch((void *)&A[(j+1)*N + 0 ]); ...
  _mm_prefetch((void *)&A[(j+1)*N + 28]);

  b    = _mm_load1_ps(&b[j]);
  Av0  = _mm_loadu_ps(&A[j*N + 0 ]); ...
  Av7  = _mm_loadu_ps(&A[j*N + 28]);
  mul0 = _mm_mul_ps(Av0, b); ...
  mul7 = _mm_mul_ps(Av7, b);
  c0   = _mm_add_ps(mul0, c0); ...
  c7   = _mm_add_ps(mul7, c7);
}

_mm_storeu_ps(&c[0 ], c0); ...
_mm_storeu_ps(&c[28], c7);
```



When column j of A is being multiplied, column j+1 is pre-loaded into cache to reduce the latency of access to data

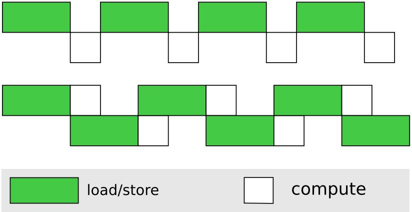


The performance is now 1.9 Gflop/s, i.e., $\sim$ 9% of the peak

Performance evaluation

An highly optimized implementation of the matrix-vector product (e.g., the SGEMV routine in MKL BLAS) achieves 3.6 Gflop/s but this is only 17% of the peak. Can we get any better performance?

The processor's SP peak is 21.28 Gflop/s. The memory bandwidth is 17 GB/s (i.e., 4.25 SP values per second). Thus one core is much faster than memory!



Prefetch can totally hide the memory transfers only if many more computations are done on each data in cache/registers. In this case it is possible to get very close to the processor's peak performance. This explains the difference between Level-1, Level-2 and Level-3 BLAS routines.

BLAS operations

- Level-1 BLAS: vector-vector operations like inner product or vector sum. $\mathcal{O}(n)$ operations are performed on $\mathcal{O}(n)$ data. Vectorizable but limited by bus speed
- Level-2 BLAS: matrix-vector operations like matrix-vector product. $\mathcal{O}(n^2)$ operations are performed on $\mathcal{O}(n^2)$ data. Vectorizable but limited by bus speed
- Level-3 BLAS: matrix-matrix operations like matrix-matrix product or rank-k update. $\mathcal{O}(n^3)$ operations are performed on $\mathcal{O}(n^2)$ data. Vectorizable and very efficient thanks to good exploitation of memory hierarchy

Section 3

OpenMP

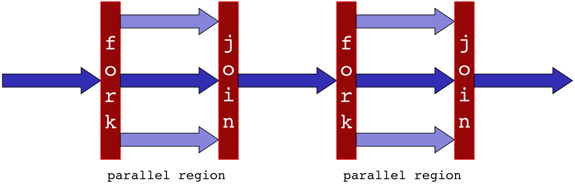
How to program multicores: OpenMP

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

How to program multicores: OpenMP

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

C/C++

```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

```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:

```fortran
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
  - 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

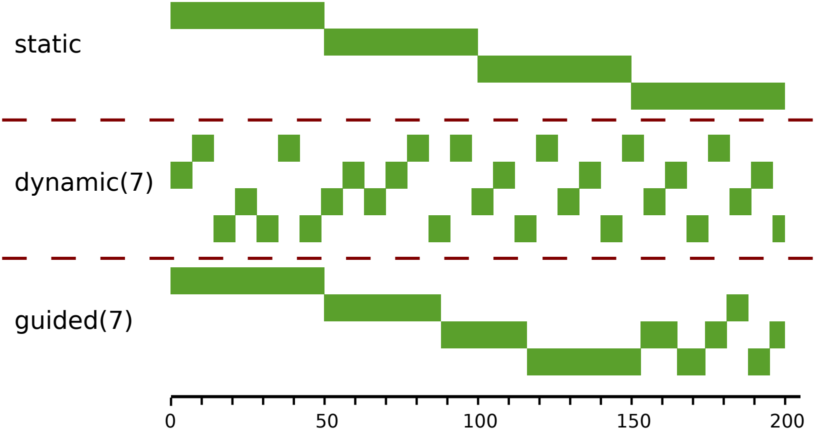
OpenMP: worksharing constructs

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

OpenMP: worksharing constructs

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)
    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
!$omp task
     call tsub(i)
!$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
```

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`: routine tests a lock, and sets it if it is available

OpenMP Locks

Examples:

omp_set_lock

```
!$OMP MASTER
! initialize the lock
call omp_init_lock(lock)
!$OMP END MASTER
...
! do work in parallel
...
call omp_set_(lock)
! exclusive access to data
...
call omp_unset_lock(lock)
...
! do more work in parallel
...
! destroy the lock
call omp_destroy_lock(lock)
```

omp_test_lock

```
!$OMP MASTER
! initialize the lock
call omp_init_lock(lock)
!$OMP END MASTER
...
! do work in parallel
...
if(omp_set_(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
   ...
end if
...
! do more work in parallel
...
! destroy the lock
call omp_destroy_lock(lock)
```

Section 4

OpenMP examples

Loop parallelism vs parallel region

Note that these two codes are essentially equivalent:

Loop parallelism

```
!$OMP PARALLEL DO
do i=1, n
   a(i) = b(i) + c(i)
end do
```

Parallel region

```
!$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
```

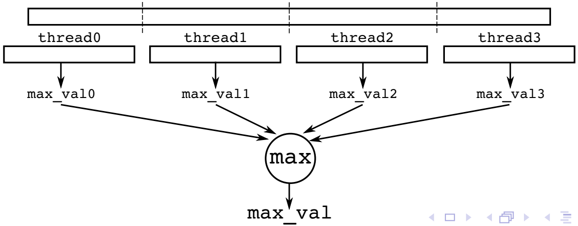
Loop parallelism is not always possible or may not be the best way of parallelizing a code.

Loop parallelism vs parallel region

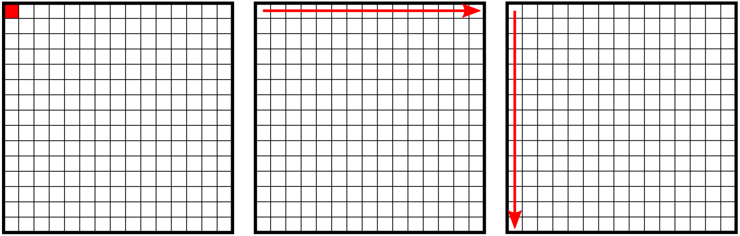
Another example: parallelize the `maxval(x)` routine which computes the maximum value of an array `x` of length `n`

Parallel region

```
!$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
```



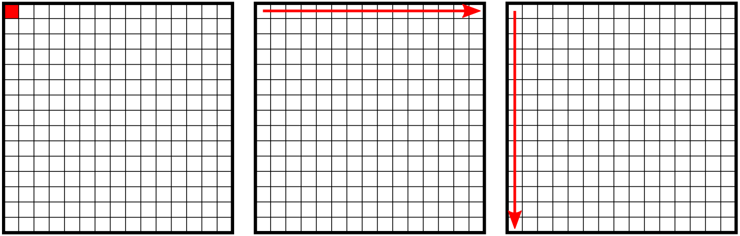
OpenMP MM product



```
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
```

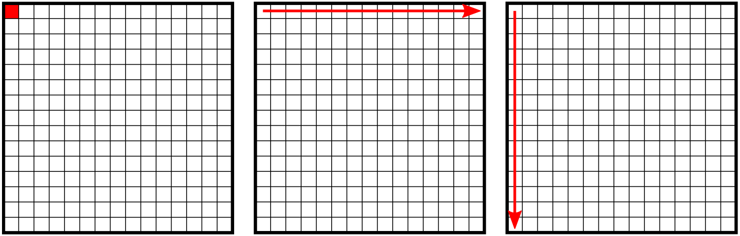
Sequential version

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 task
      end do
   end do
end do
end subroutine mmproduct
```

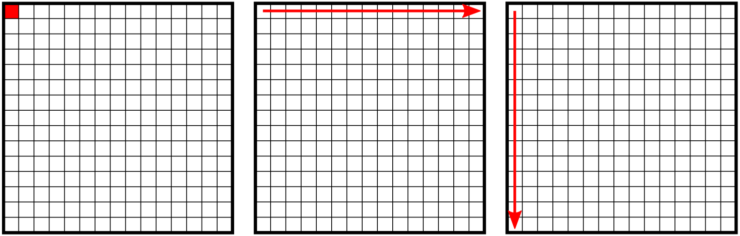
OpenMP MM product



```
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 task
      end do
   end do
end do
end subroutine mmproduct
```

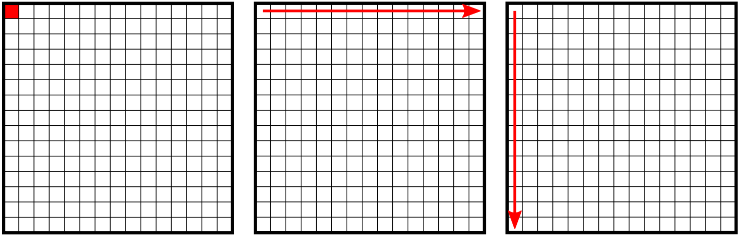
Incorrect parallel with **WAW**, **WAR** and **RAW** conflict on `c(i,j)`

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
end subroutine mmproduct
```

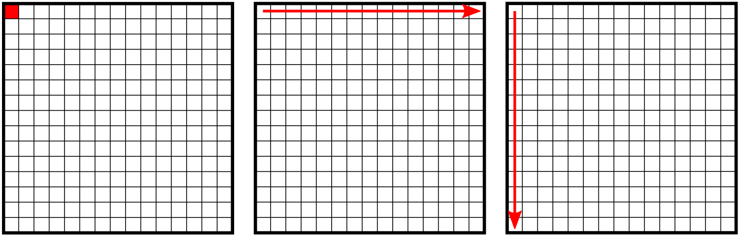
OpenMP MM product



```
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
end subroutine mmproduct
```

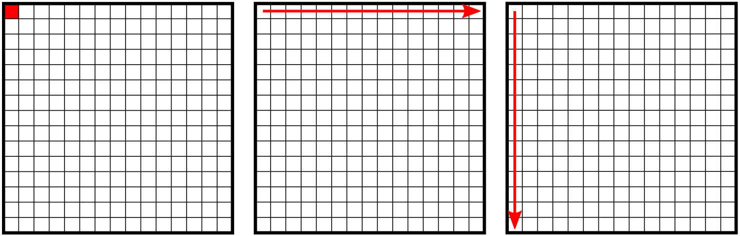
Incorrect parallel with **WAW**, **WAR** and **RAW** conflict on `c(i,j)`

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
end subroutine mmproduct
```

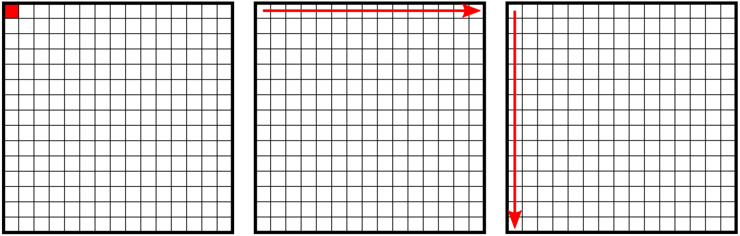
OpenMP MM product



```
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
end subroutine mmproduct
```

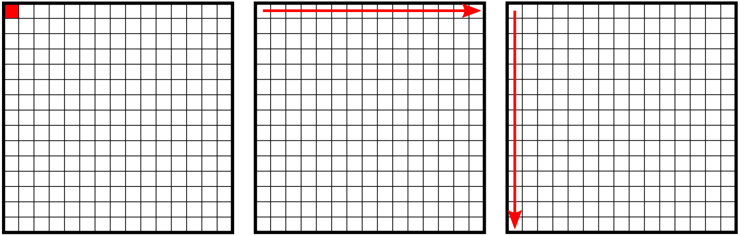
Correct parallel but enormous waste of memory (c is replicated)

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
```

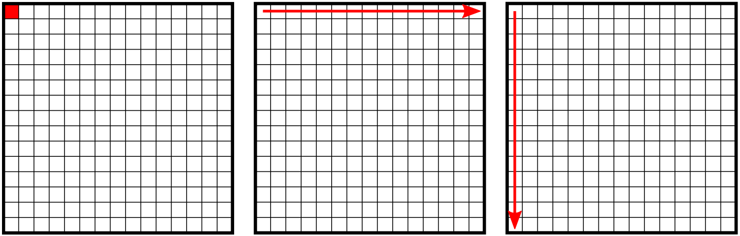
OpenMP MM product



```
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
```

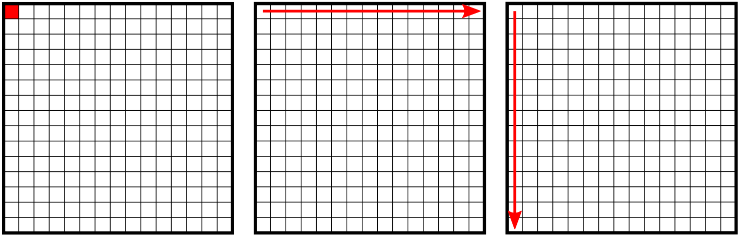
Correct parallel but low efficiency (many fork-join)

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
end subroutine mmproduct
```

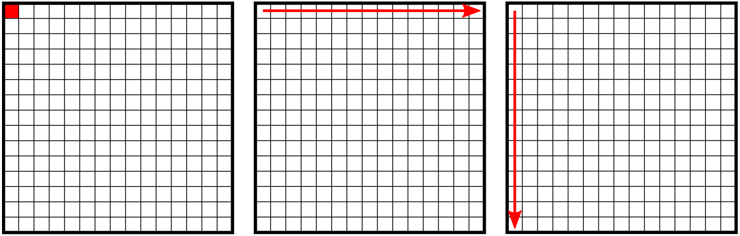
OpenMP MM product



```
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
end subroutine mmproduct
```

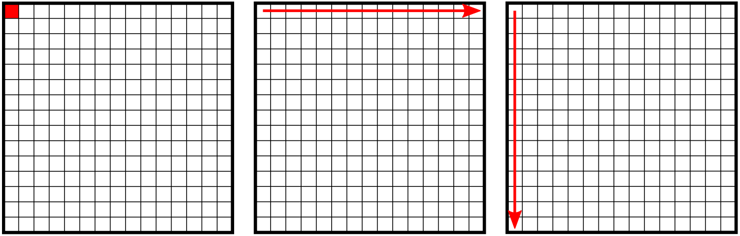
Correct parallel but still low efficiency

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
```

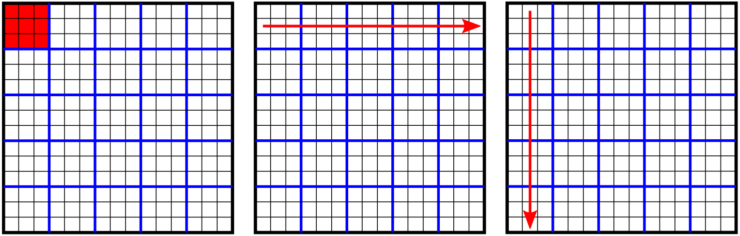
OpenMP MM product



```
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
```

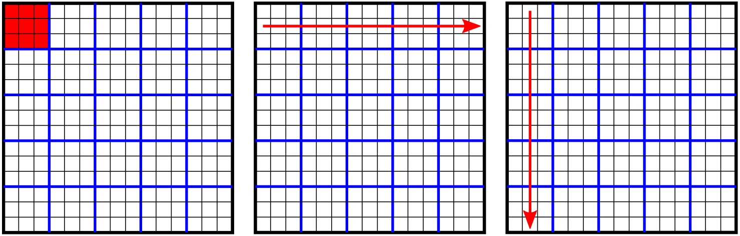
Correct parallel and good performance

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
```

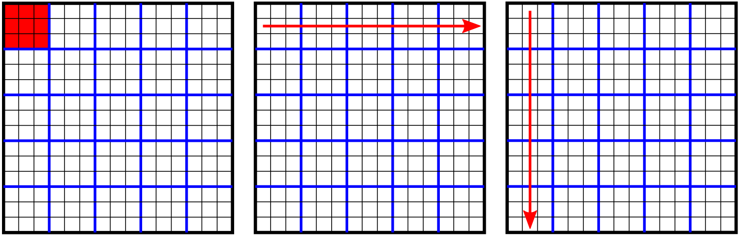
OpenMP MM product



```
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
```

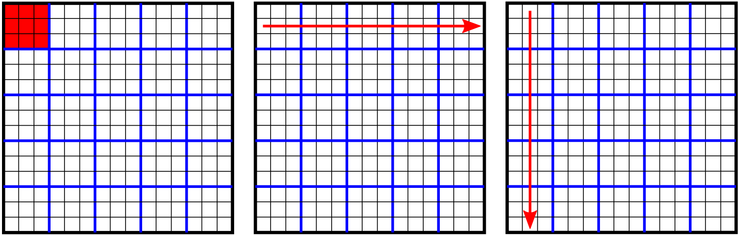
Optimized version by blocking

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
```

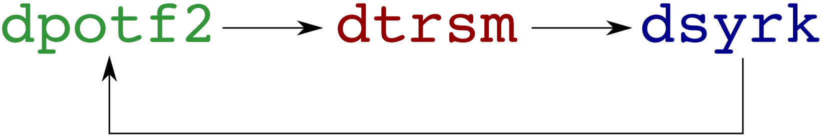
The unblocked Cholesky factorization is extremely inefficient due to a poor cache reuse. No level-3 BLAS operations possible.

The Cholesky factorization

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

```
do k=1, n, nb

   call dpotf2( a(k:k+nb-1,k:k+nb-1) )

   call dtrsm ( a(k+nb:n, k:k+nb-1), &
              & a(k:k+nb-1,k:k+nb-1) )

   call dsyrk ( a(k+nb:n,k+nb:n), &
              & a(k+nb:n, k:k+nb-1) )

end do
```

The blocked Cholesky factorization is highly efficient thanks to the usage of level-3 BLAS routines.


dpotf2
dtrsm
dsyrk

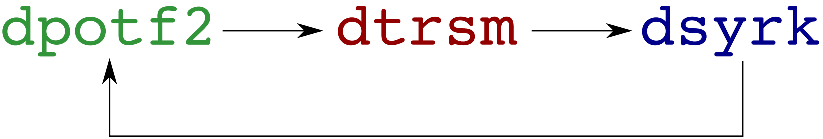
No potential for parallelism?

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, nb

   call dpotf2( a(k:k+nb-1,k:k+nb-1) )

   call dtrsm ( a(k+nb:n, k:k+nb-1), &
              & a(k:k+nb-1,k:k+nb-1) )

   call dsyrk ( a(k+nb:n,k+nb:n), &
              & a(k+nb:n, k:k+nb-1) )

end do
```

The blocked Cholesky factorization is highly efficient thanks to the usage of level-3 BLAS routines.



dpotf2 → dtrsm → dsyrk

No potential for parallelism? FALSE

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, nb

   call dpotf2( a(k:k+nb-1,k:k+nb-1) )

   do i=k+nb, n, nb
      call dtrsm ( a(i:i+nb-1, k:k+nb-1), &
                 & a(k:k+nb-1,k:k+nb-1) )
      do j=k+nb, i, nb
         call dpoup ( a(i:i+nb-1,j:j+nb-1), &
                    & a(i:i+nb-1, k:k+nb-1) &
                    & a(j:j+nb-1, k:k+nb-1) )
      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.

Blocked Cholesky: multithreading

First tentative:

```
!$omp parallel do
do k=1, n, nb
   call dpotf2( a(k:k+nb-1,k:k+nb-1) )

   do i=k+nb, n, nb
      call dtrsm ( a(i:i+nb-1, k:k+nb-1), a(k:k+nb-1,k:k+nb-1) )

      do j=k+nb, i, nb
         call dpoup ( a(i:i+nb-1,j:j+nb-1), a(i:i+nb-1, k:k+nb-1), a(j:j+nb-1, k:k+nb-1) )
      end do

   end do

end do
!$omp end parallel do
```

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( a(k:k+nb-1,k:k+nb-1) )
!$omp parallel do
   do i=k+nb, n, nb
      call dtrsm ( a(i:i+nb-1, k:k+nb-1), a(k:k+nb-1,k:k+nb-1) )

      do j=k+nb, i, nb
         call dpoup ( a(i:i+nb-1,j:j+nb-1), a(i:i+nb-1, k:k+nb-1), a(j:j+nb-1, k:k+nb-1) )
      end do

   end do
!$omp end parallel do
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( a(k:k+nb-1,k:k+nb-1) )

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

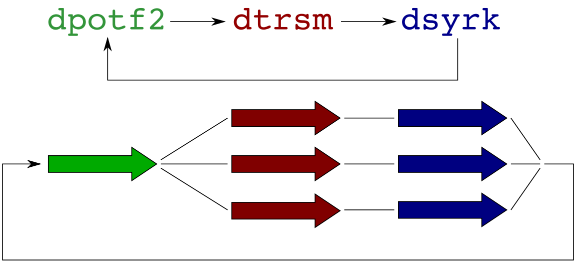
Fourth tentative:

```
do k=1, n, nb
   call dpotf2( a(k:k+nb-1,k:k+nb-1) )

!$omp parallel do
   do i=k+nb, n, nb
      call dtrsm ( a(i:i+nb-1, k:k+nb-1), a(k:k+nb-1,k:k+nb-1) )
   end do
!$omp end parallel do

!$omp parallel do
   do i=k+nb, n, nb
      do j=k+nb, i, nb
         call dpoup ( a(i:i+nb-1,j:j+nb-1), a(i:i+nb-1, k:k+nb-1), a(j:j+nb-1, k:k+nb-1) )
      end do
   end do
!$omp end parallel 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



dpotf2 → dtrsm → dsyrk

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

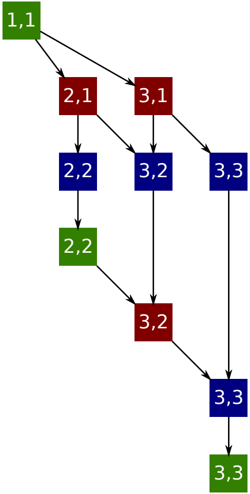
All the previous parallelization approaches are based on the assumption that step `step+1` can be started only when all the operations related to step `step` are completed. This constraint is too strict and can be partially relaxed.
Which conditions have to be necessarily respected?

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

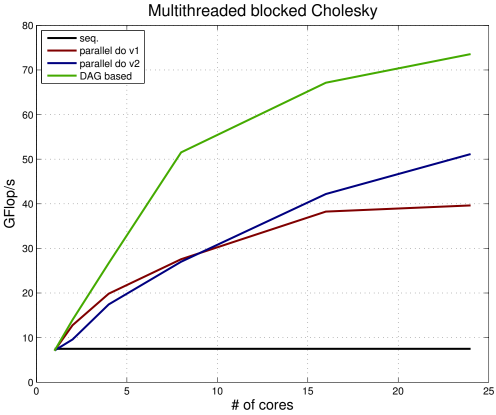
Blocked Cholesky: better multithreading

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: better multithreading



download the code at:

http://buttari.perso.enseeiht.fr/stuff/ompchol.F90

OpenMP: the task construct

```
recursive subroutine traverse ( p )
  type node
     type(node), pointer :: left, right
  end type node
  type(node) :: p
  if (associated(p%left)) then
!$omp task     ! p is firstprivate by default
     call traverse(p%left)
!$omp end task
  end if
  if (associated(p%right)) then
!$omp task      ! p is firstprivate by default
     call traverse(p%right)
!$omp end task
  end if
  call process ( p )
end subroutine traverse
```

Although the sequential code will traverse the tree in postorder, this is not true for the parallel execution since no synchronizations are performed.

OpenMP: the task construct

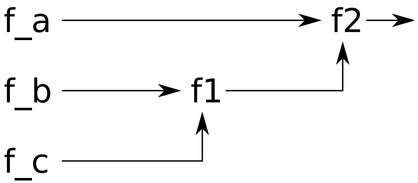
```fortran
recursive subroutine traverse ( p )
  type node
     type(node), pointer :: left, right
  end type node
  type(node) :: p
  if (associated(p%left)) then
!$omp task     ! p is firstprivate by default
     call traverse(p%left)
!$omp end task
  end if
  if (associated(p%right)) then
!$omp task      ! p is firstprivate by default
     call traverse(p%right)
!$omp end task
  end if
!$omp taskwait
  call process ( p )
end subroutine traverse
```

The TASKWAIT construct is a synchronization which forces a thread to wait until the execution of the children tasks is terminated.

OMP tasks: example

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

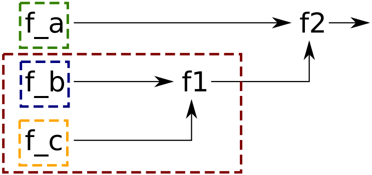
```
function  foo()
  integer  :: foo
  integer  :: a, b, c, x, y;

  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

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

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

!$omp task

!$omp task
  b = f_b()
!$omp end task

!$omp task
  c = f_c()
!$omp end task

!$omp taskwait

  x = f1(b, c)
!$omp end task

!$omp taskwait

  y = f2(a, x)

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

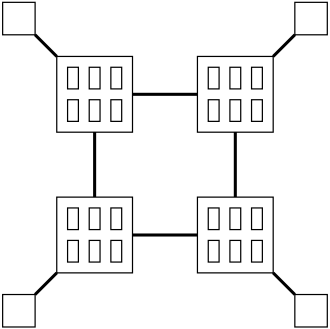


Section 5

OpenMP: odds & ends

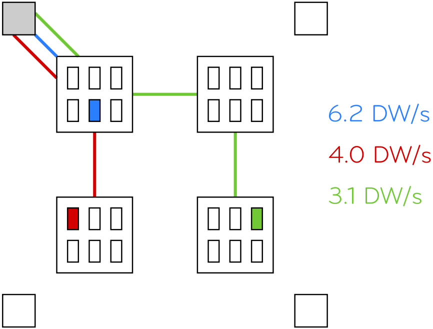
NUMA: Memory locality

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)



6.2 DW/s

4.0 DW/s

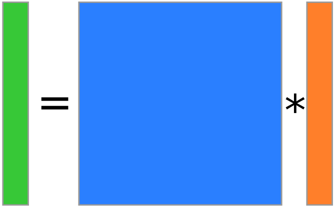
3.1 DW/s

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

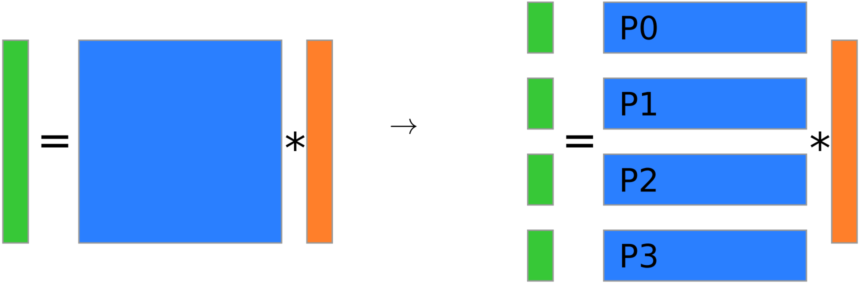
If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?



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

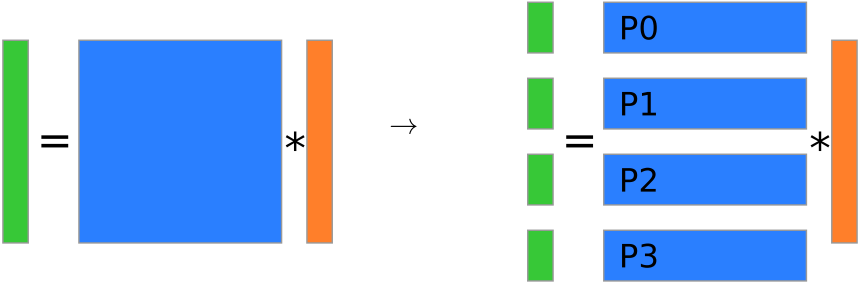
If an OpenMP parallel DGEMV (matrix operation) operation is not correctly coded on such an architecture, only a speedup of 1.5 can be achieved using all the 24 cores. Why?



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.

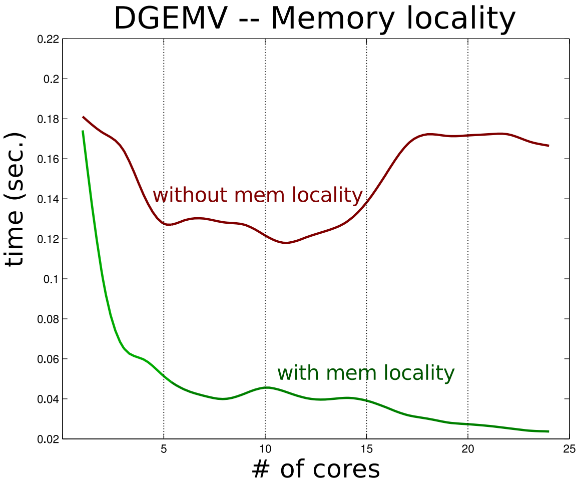
NUMA: memory locality

Implementing all this requires the ability to:

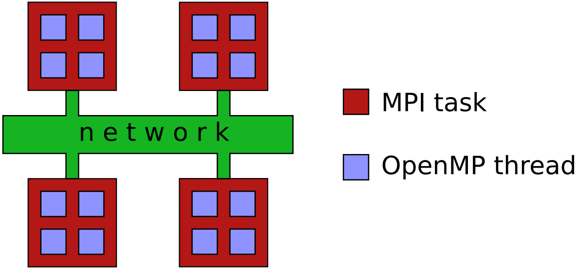
- control the placement of threads: we have to bind each thread to a single core and prevent threads migrations. This can be done in a number of ways, e.g. by means of tools such as `hwloc` which allows thread pinning
- control the placement of data: we have to make sure that one front physically resides 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)
```

Section 6

Mixed-precision Iterative refinement

Mixed-precision arithmetic

On modern systems, single-precision arithmetic has a clear performance advantage over double-precision arithmetic for the following reasons:

- **Vector instructions**: vector units can normally do twice as many SP operations as DP ones every clock cycle. For example SSE units can do either 4 SP or 2 DP.
- **Bus bandwidth**: because SP values are twice as small as DP ones, the memory transfer rate for SP values is twice as big as for DP
- **Data locality**: again, because SP data are twice as small DP, you can put twice as many SP values in the cache as DP

Mixed-precision arithmetic

Performance comparison between single and double precision arithmetic for matrix-matrix and matrix-vector product operations on square matrices.

	Size	SGEMM/	Size	SGEMV/
		DGEMM		DGEMV
AMD Opteron 246	3000	2.00	5000	1.70
Sun UltraSparc-IIe	3000	1.64	5000	1.66
Intel PIII Copp.	3000	2.03	5000	2.09
PowerPC 970	3000	2.04	5000	1.44
Intel Woodcrest	3000	1.81	5000	2.18
Intel XEON	3000	2.04	5000	1.82
Intel Centrino Duo	3000	2.71	5000	2.21

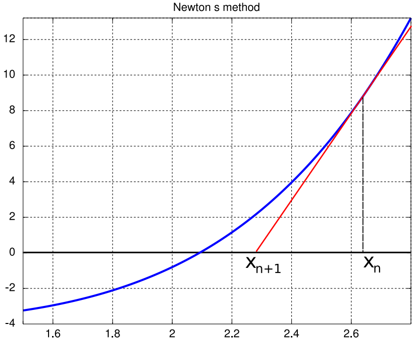
Mixed-precision arithmetic

Is there a way to do computations at the speed of single-precision while achieving the accuracy of double-precision? Sort of.
For some operations it is possible to do the bulk of computations in single-precision and then recover the accuracy to double-precision by means of an iterative method like the Newton's method.

Mixed-precision arithmetic

The Newton's method says that given an approximate root of the function $f(x)$, we can refine it through iterations of the type:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$



Mixed-precision arithmetic

The Newton's method can be applied, for example, to refine the root of the function $f(x) = b - Ax$ (equivalent to solving the linear system $Ax = b$).

$$x_{k+1} = x_k - A^{-1}r_k \quad where \quad r_k = b - Ax_k$$

This leads to the well known iterative refinement method:

```
$x_0 \leftarrow A^{-1}b$
repeat
    $r_k \leftarrow b - Ax_{k-1}$
    $z_k \leftarrow A^{-1}r_k$
    $x_k \leftarrow x_{x-1} - z_k$
until convergence
```

Mixed-precision arithmetic

The Newton's method can be applied, for example, to refine the root of the function $f(x) = b - Ax$ (equivalent to solving the linear system $Ax = b$).

$$x_{k+1} = x_k - A^{-1} r_k \quad where \quad r_k = b - Ax_k$$

This leads to the well known iterative refinement method:

```
$x_0 \leftarrow A^{-1} b$                 $O(n^3)$
repeat
    $r_k \leftarrow b - Ax_{k-1}$         $O(n^2)$
    $z_k \leftarrow A^{-1} r_k$           $O(n^2)$
    $x_k \leftarrow x_{x-1} - z_k$        $O(n)$
until convergence
```

Mixed-precision arithmetic

The Newton's method can be applied, for example, to refine the root of the function $f(x) = b - Ax$ (equivalent to solving the linear system $Ax = b$).

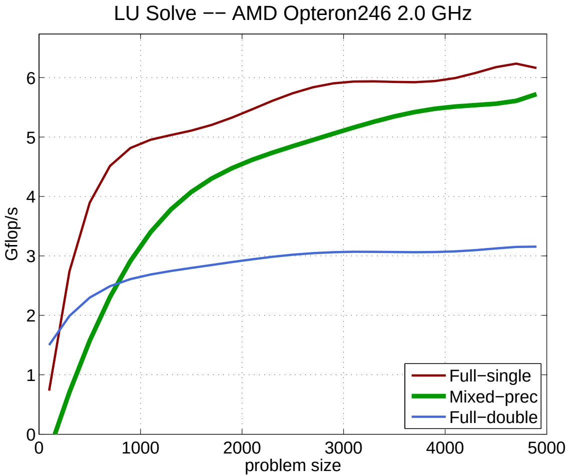
$$x_{k+1} = x_k - A^{-1} r_k \quad \textit{where} \quad r_k = b - Ax_k$$

This leads to the well known iterative refinement method:

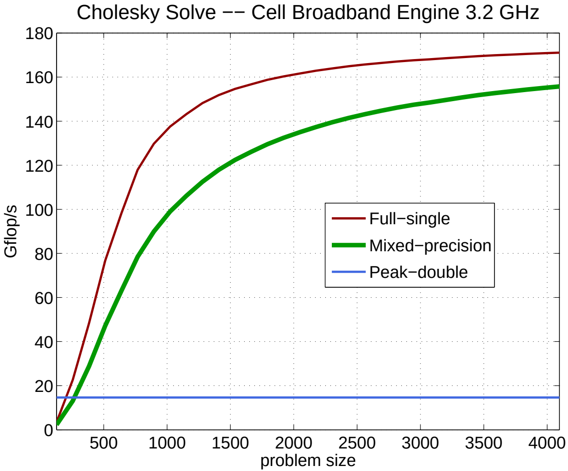
```
$x_0 \leftarrow A^{-1} b$                $\varepsilon_s$
repeat
    $r_k \leftarrow b - Ax_{k-1}$        $\varepsilon_d$
    $z_k \leftarrow A^{-1} r_k$          $\varepsilon_s$
    $x_k \leftarrow x_{x-1} - z_k$       $\varepsilon_d$
until convergence
```

We can perform the expensive factorization in single precision and then do the refinement in double.

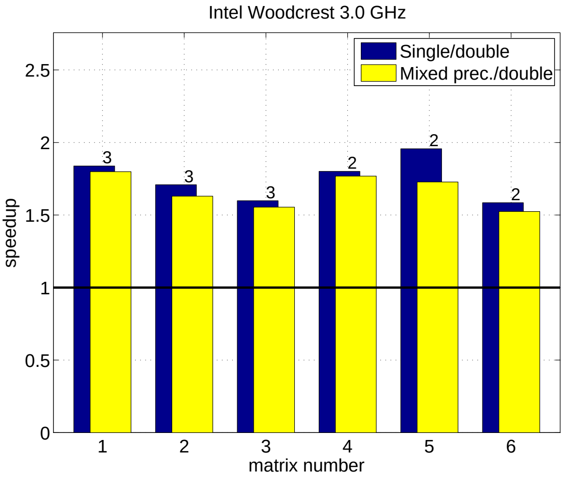
Mixed Precision Iterative Refinement: results



Mixed Precision Iterative Refinement: results



Mixed Precision Iterative Refinement: results



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

Reference and examples

The OpenMP reference document can be found at this address:
http://www.openmp.org/mp-documents/OpenMP3.1.pdf
The file contains detailed documentation about all the OpenMP directives (included those that were not discussed in the lectures) and many examples.
It is warmly recommended to study and analyze (at least) the following examples:

- A.1 parallel loop
- A.5 `parallel`
- A.12 `sections`
- A.14 `single`
- A.15.5 `task`
- A.15.6 `task`
- A.18 `master`
- A.19 `critical`
- A.21 `barrier`
- A.22.1 `atomic`
- A.22.2 `atomic`
- A.32.1 `private`
- A.32.2 `private`
- A.36 `reduction`