# Advanced OpenMP

http://tinyurl.com/cq-adv-openmp-20160428

By: Bart Oldeman, Calcul Québec – McGill HPC

Bart.Oldeman@calculquebec.ca, Bart.Oldeman@mcgill.ca

## Partners and Sponsors

POLYTECHNIQUE MONTRÉAL

UQTR Savoir. Surprendre.

INNOVATION.CA CANADA FOUNDATION FOR INNOVATION | FONDATION CANADIENNE POUR L'INNOVATION

Université de Montréal

McGill

Québec

UNIVERSITÉ Concordia UNIVERSITY

UNIVERSITÉ LAVAL

risq LE RÉSEAU AU SERVICE DU SAVOIR

ÉTS

UNIVERSITÉ DE SHERBROOKE

canarie

UQÀM

compute canada | calcul canada

## Outline of the workshop

### Theoretical / practical introduction

- Parallelizing your serial code
- Revision of Introduction to OpenMP
- How do we run OpenMP codes (on the Guillimin cluster)?
- Advanced OpenMP topics:
  - Nested parallelism
  - OpenMP tasks
  - the OpenMP memory model
  - synchronization
  - performance tuning
  - tips, tricks, and pitfalls
  - some new features in OpenMP 4.0 and 4.5

## Outline of the workshop

Calcul Québec

Practical exercises on Guillimin

- Login, setup environment, launch OpenMP code
- Analyzing and running examples
- Modifying and tuning OpenMP codes

References, partly based on

- `http://tinyurl.com/OpenMP-Tutorial`
- `http://www.archer.ac.uk/training/course-material/2014/05/AdvancedOpenMP_Oxford/`
- `http://ircc.fiu.edu/sc13/AdvOpenMP_Slides.pdf`

## Parallelizing your serial code

Models for parallel computing

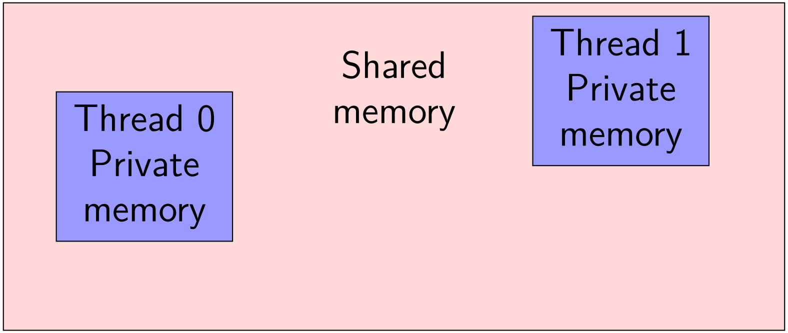
(as an ordinary user sees it ...)

- Implicit Parallelization — minimum work for you
  - Threaded libraries (MKL, ACML, GOTO, etc ....)
  - Compiler directives (OpenMP)
  - Good for desktops and shared memory machines
- Explicit Parallelization — work is required !
  - You tell what should be done on what CPU
  - Low-level option for shared memory machines: POSIX Threads (pthreads)
  - Solution for distributed clusters (MPI: shared nothing!)

## OpenMP — Shared Memory API

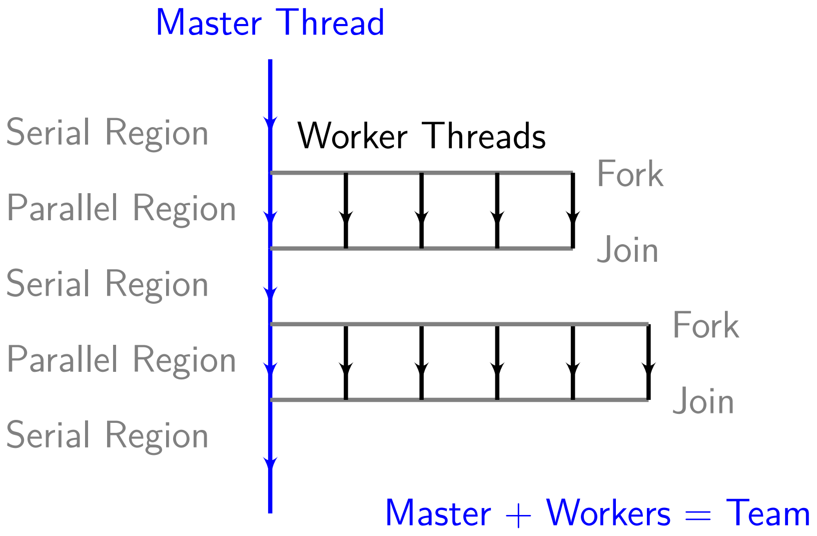
- Open Multi-Processing: An Application Program Interface for multi-threaded programs in a shared-memory environment.
- `http://www.openmp.org`
- Consists of
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Allows for relatively simple incremental parallelization.
- *Not* distributed, but can be combined with MPI (hybrid: see Advanced MPI workshop).

## Shared memory approach



- Most memory is shared by all threads.
- Each thread also has some private memory: variables explicitly declared private, local variables in functions and subroutines.

## OpenMP: fork/join model



Implementations use thread pools so worker threads sleep from join to fork.

## What is OpenMP for a user?

- OpenMP is NOT a language!
- OpenMP is NOT a compiler or specific product
- OpenMP is a de-facto industry standard, a specification for an Application Program Interface (API).
  - You use its directives, routines, and environment variables.
  - You compile and link your code with specific flags.
- History: version 1.0 (1997), 2.5 (2005), 3.0 (2008), 3.1 (2011), 4.0 (2013).
- Different implementations :
  - GCC (4.2+), Intel, PGI, Visual C++, Solaris Studio, CLang (3.7+), ...

## Basic features of OpenMP program

Calcul Québec

- Include basic definitions (`#include <omp.h>`, `INCLUDE 'omp_lib.h'`, or `USE omp_lib`).
- Parallel region declared by a directive of the form `#pragma omp parallel` (C) or `!$OMP PARALLEL` (Fortran), declaring which variables are private.
- Optional: code only compiled for OpenMP: use `_OPENMP` preprocessor symbol (C) or `!$` prefix (Fortran).

## Example: “Hello from N cores”

Calcul Québec

Fortran

```fortran
PROGRAM hello

!$ USE omp_lib

IMPLICIT NONE
INTEGER rank, size
rank = 0
size = 1

!$OMP PARALLEL PRIVATE(rank, size)

!$  size = omp_get_num_threads()
!$  rank = omp_get_thread_num()

WRITE(*,*) 'Hello from processor ',&
           rank, ' of ', size

!$OMP END PARALLEL

END PROGRAM hello
```

C

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main (int argc, char * argv[]) {
 int rank = 0, size = 1;
#ifdef _OPENMP
#pragma omp parallel private(rank, size)
#endif
 {
#ifdef _OPENMP
   rank = omp_get_thread_num();
   size = omp_get_num_threads();
#endif
   printf("Hello from processor %d"
      " of %d\n", rank, size );
 }
 return 0;
}
```

## Example: inner product via reduction

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 10000000
int main(void) {
    int *a, *b, ip, i; double t1, t2;
    a = malloc(N * sizeof(*a)); b = malloc(N * sizeof(*b));
    for (i = 0; i < N; i++) {
        a[i] = 2; b[i] = 3;
    }
    t1 = omp_get_wtime();
    ip = 0;
#pragma omp parallel for private(i) shared(a,b) reduction(+:ip)
    for (i = 0; i < N; i++) { ip += a[i] * b[i]; }
    t2 = omp_get_wtime();
    printf("Inner product = %d, time=%g\n", ip, t2-t1);
    return 0; }
```

## Compiling your OpenMP code

Calcul Québec

- NOT defined by the standard
- A special compilation flag must be used.
- On the Guillimin cluster:
  - `module load foss/2015b`
  - `gcc -fopenmp hello.c -o hello`
  - `gfortran -fopenmp hello.f90 -o hello`
  - `module load iomkl/2015b`
  - `icc -fopenmp hello.c -o hello`
  - `ifort -fopenmp hello.f90 -o hello`
  - `module load pomkl/2016.03`
  - `pgcc -mp hello.c -o hello`
  - `pgfortran -mp hello.f90 -o hello`

## Running your OpenMP code

- Important: environment variable `OMP_NUM_THREADS`.
  - `export OMP_NUM_THREADS=4`
  - `./hello`
    ```
    Hello from processor 2 of 4
    Hello from processor 0 of 4
    Hello from processor 3 of 4
    Hello from processor 1 of 4
    ```
  - `unset OMP_NUM_THREADS`
  - `pgcc -mp hello.c -o hello`
  - `./hello`
    ```
    Hello from processor 0 of 1
    ```
  - `gcc -fopenmp hello.c -o hello`
  - `./hello`
    ```
    Hello from processor 3 of 8 (...)
    ```

## OpenMP directives

Format: `sentinel directive [clause,]` where `sentinel` is `#pragma omp` or `!$OMP`. Examples:

- `#pragma omp parallel` (C), `!$OMP PARALLEL`, `!$OMP END PARALLEL` (Fortran): Parallel region construct.
- `#pragma omp for`: A *workshare* construct that makes a loop parallel (`!$OMP DO` in Fortran).
- `#pragma omp parallel for`: A combined construct: defines a parallel region that only contains the loop.
- `#pragma omp barrier`: A synchronization directive: all threads wait for each other here.

## Running your OpenMP code

- On your laptop or desktop, just compile and run your code as above.
- On Guillimin cluster, use batch system to submit non-trivial OpenMP jobs! Example: `hello.pbs`:

```
#!/bin/bash
#PBS -l nodes=1:ppn=6
#PBS -l walltime=00:05:00
#PBS -N hello
cd $PBS_O_WORKDIR
module load iomkl/2015b
export OMP_NUM_THREADS=6
./hello > hello.out
```

Submit your job:

```
$ qsub hello.pbs
```

## Exercise 1:

Calcul Québec

### Log in to Guillimin, setting up the environment

1) Log in to Guillimin:

```
ssh class##@guillimin.hpc.mcgill.ca
```

2) Check for loaded software modules:

```
$ module list
```

3) See all available modules:

```
$ module av
```

4) Load toolchain module (Intel+OpenMPI+MKL):

```
$ module load iomkl/2015b
```

5) Check loaded modules again

## Exercise 2: “Hello” program, compilation

1) Copy all files to your home directory:

```
$ cp -a /software/workshop/advomp/* ./
```

2) Compile your code:

```
$ ifort -fopenmp hello.f90 -o hello
$ icc -fopenmp hello.c -o hello
```

## Exercise 2: “Hello”, job submission

Calcul Québec

3) View the file “hello.pbs”:

```
#!/bin/bash
#PBS -l nodes=1:ppn=6
#PBS -l walltime=00:05:00
#PBS -N hello
cd $PBS_O_WORKDIR
module load iomkl/2015b
export OMP_NUM_THREADS=6
./hello > hello.out
```

## Exercise 2: “Hello”, submitting your job

4) Submit your job:

```
$ qsub hello.pbs
```

5) Check the job status:

```
$ qstat -u $USER
```

```
$ showq -u $USER
```

6) Check the output (`hello.out`)

## Exercise 2: “Hello”, compile and run

Calcul Québec

Alternatively, using interactive qsub, or on your own Mac/Linux/Cygwin/MSYS computer:

1) Interactive login:

```
$ qsub -I -l nodes=1:ppn=6,walltime=7:00:00
```

or create and then copy all files to a directory:

```
yourlaptop> git clone -b mcgill \
https://github.com/calculquebec/cq-formation-advanced-openmp.git
cd cq-formation-advanced-openmp
```

2) Compile your code:

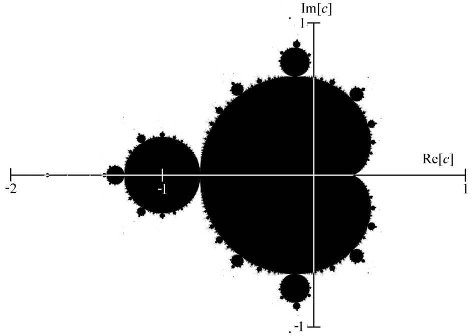
```
> gfortran -fopenmp hello.f90 -o hello
> gcc -fopenmp hello.c -o hello
```

3) Run your code:

```
> # can use any value here; default: number of cores
> export OMP_NUM_THREADS=6
> ./hello
```

## Exercise 3: "Mandelbrot"

Please see the files `area.f90`, `area.c`, and `area.pbs`. These compute the area of the Mandelbrot set.



Parallelize this program using `omp parallel do` or `omp parallel for` and measure the speedup.

## More OpenMP directives

- `#pragma omp atomic`
  Protects updates to shared variables.
- `#pragma omp critical`
  Locked section, threads can only enter sequentially.
- `#pragma omp single`
  Only one thread executes this section.
- `#pragma omp master`
  Only the master thread executes this section (NO implicit barrier!).

## OpenMP clauses

- Data scope: `private` and `shared`.
  `!$omp parallel private(i) shared(x)` The variable `i` is private to the thread but the variable `x` is shared with all other threads.
  Default: all variables shared except loop variables (C: outer, Fortran:all), and variables declared inside block.
- `!$omp parallel default(shared) private(i)`
  All variables are shared except `i`.
- `!$omp parallel default(none) private(i)`
  No default (*recommended!*), `i` is private.

## More OpenMP clauses

- `!$omp parallel for firstprivate(y) lastprivate(z)`
  The variable `y` is private, initialized from the corresponding variable before the parallel region. The variable `z` is private; the value from the last iteration is copied to the corresponding variable after the parallel region.
- `copyprivate` in `omp single copyprivate(x)`.
  Variable `x` is copied to the corresponding variable in all other threads after the `single` region.
- `nowait` in `#pragma omp for nowait`.
  A loop where threads do not wait for each other upon completion.

## scheduling clauses

Calcul Québec

- `schedule(static, 10000)` allocates chunks of 10000 loop iterations to every thread:

```
void addvectors(const int *a, const int *b, int *c, int n) {
  int i;
#pragma omp for schedule(static, 10000)
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

- Use `dynamic` instead of static to dynamically assign threads, if one finishes it is assigned the next chunk. Useful for unequal work within iterations.
- `guided` instead of dynamic: chunk sizes decrease as less work is left to do.
- `runtime`: use OMP_SCHEDULE environment variable.

## workshare (Fortran)

- Example:

```
integer a(10000), b(10000), c(10000), d(10000)

!$OMP PARALLEL
!$OMP WORKSHARE
  c(:) = a(:) + b(:)
!$OMP END WORKSHARE NOWAIT

!$OMP WORKSHARE
  d(:) = a(:)
!$OMP END WORKSHARE NOWAIT
!$OMP END PARALLEL
```

- Array assignments in Fortran are distributed among threads like loops.
- Note that `nowait` in Fortran comes at the end.

## threadprivate

- `!$omp parallel threadprivate(x)` Here the variable `x` must be a *global* or persistent variable, e.g. C: `static`, Fortran: `SAVE`, `COMMON`.
  The variable is then private to each thread and keeps its value between parallel regions.
- Example:

```
static int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
  counter++;
}
```

Each thread has its own global `counter` variable.

## OpenMP important library routines

```
int omp_get_max_threads(void);
```

Get maximum number of threads used here.

```
void omp_set_num_threads(int);
```

Set number of threads for next parallel region.

```
int omp_get_thread_num(void);
```

Get current thread number in parallel region.

```
int omp_get_num_threads(void);
```

Get number of threads in parallel region.

```
double omp_get_wtime(void);
```

Portable wall clock timing routine.

More exist, for example for locks and nested regions.

## OpenMP main environment variables

`OMP_NUM_THREADS`
: Sets the maximum number of threads used.

`OMP_SCHEDULE`
: Used for run-time scheduling.

`OMP_STACKSIZE`
: Sets stack size for private variables (for instance, `4M`).

`OMP_NESTED`
: Enables nested parallelism (for instance, `TRUE`).

More exist, for example to control nested parallelism.

## parallel: manual scheduling (SPMD)

Calcul Québec

- SPMD=Single Program Multiple Data, like in MPI.

```
void addvectors(const int *a, const int *b, int *c, int n) {
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
} ....
  int tid, nthreads, low, high;
#pragma omp parallel default(none) private(tid, nthreads,\
      low, high) shared(a, b, c, n)
  {
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    low = (n * tid) / nthreads;
    high = (n * (tid + 1)) / nthreads;
    addvectors(&a[low], &b[low], &c[low], high-low);
  }
```

- Calculate which thread does which loop iterations. Note: no barrier.

## SPMD vs. worksharing

- Worksharing (`omp for`/`omp do`) is easiest to implement.
- SPMD (do work based on thread ID) may give better performance but is harder to implement.
- SPMD like in MPI:
    - Instead of using large shared arrays, use smaller arrays private to threads: mark all (non-read-only) global and persistent (`static`/`SAVE`) variables `threadprivate`, and communicate using buffers and barriers.
    - Fewer cache misses using more private data may give better performance.

## sections (SPMD construct)

- Example:

```
#pragma omp parallel sections
  {
#pragma omp section
  addvectors(a, b, c, n);
#pragma omp section
  printf("hello world!\n");
#pragma omp section
  printf("I may or may not be the third thread\n");
  }
```

- The sections are individual code blocks that are distributed over the threads.
- More flexible alternative (OpenMP 3.0): `omp task`, useful when traversing dynamic data structures (lists, trees, etc.).

## Nested parallelism

- Nested parallelism is supported in OpenMP.
- If a `PARALLEL` directive is encountered within another `PARALLEL` directive, a new team of threads will be created.
- This is enabled with the `OMP_NESTED` environment variable or the `omp_set_nested` routine.
- If nested parallelism is disabled, the code will still executed, but the inner teams will contain only one thread.

## Nested parallelism (cont)

- Example:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
!$OMP PARALLEL DO
   do i = 1,n
      x(i) = 1.0
   end do
!$OMP SECTION
!$OMP PARALLEL DO
   do j = 1,n
      y(j) = 2.0
   end do
!$OMP END SECTIONS
!$OMP END PARALLEL
```

## Nested loops

For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the `collapse` clause:

- Argument is number of loops to collapse.
- Will form a single loop of length NxM and then parallelize and schedule that.
- Useful if N is close to the number of threads so parallelizing the outer loop may not have good load balance
- More efficient than using nested teams
- 
```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
    .....
  }
}
```

## Exercise 4: Mandelbrot scheduling/collapse

Experiment with scheduling and `collapse` in the Mandelbrot example and see if you can get a better speedup.

## OpenMP tasks

- Run independent tasks in parallel, example for linked list:

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p = listhead ;
    while (p) {
      #pragma omp task firstprivate(p)
      {
        process (p);
      }
      p=next (p) ;
    }
  }
}
```

## OpenMP tasks

- Or a binary tree...

```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
    { postorder(p->left); }
  if (p->right)
    #pragma omp task
    { postorder(p->right); }
  #pragma omp taskwait
  process(p->data);
}
```

- Without tasks would have needed to put the `process` arguments into an array, and use `omp for/do` on that.

## Memory model

OpenMP supports a relaxed-consistency shared memory model.

- Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
- These temporary views are made consistent only at certain points in the program.
- The operation which enforces consistency is called the `flush` operation

## Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
- All previous read/writes by this thread have completed and are visible to other threads
- No subsequent read/writes by this thread have occurred
- A `flush` operation is analogous to a fence in other shared memory APIs

## Flush operation

A `flush` operation is implied by OpenMP synchronizations, e.g.

- at entry/exit of parallel regions
- at implicit and explicit barriers
- at entry/exit of critical regions
- whenever a lock is set or unset ...

(but not at entry to worksharing regions or entry/exit of master regions)

## Flush operation

In order for a write of a variable on one thread to be guaranteed visible and valid on a second thread, the following operations must occur in the following order:

1. Thread A writes the variable
2. Thread A executes a flush operation
3. Thread B executes a flush operation
4. Thread B reads the variable

## Flush operation

Using flush correctly is difficult and prone to subtle bugs

- extremely hard to test whether code is correct
- may execute correctly on one platform/compiler but not on another
- bugs can be triggered by changing the optimization level on the compiler
- Don't use it unless you are 100% confident you know what you are doing!
- and even then ......

## Example: producer-consumer pattern

Calcul Québec

This will most likely lock thread 1:

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

```

while (!flag);
```

Fix using `flush` operations:

Thread 0

```
a = foo();
// ensure flag is written
// after a:
#pragma omp flush
flag = 1;

// ensure flag is written
// to memory:
#pragma omp flush
```

Thread 1

```
do {
  // ensure flag is read
  // from memory:
  #pragma omp flush
} while (!flag);
// ensure correct ordering
// of flushes
#pragma omp flush
b = a;
```

## Example: producer-consumer pattern

To be 100% correct need to use `atomic` as well, which implies a `flush` on the relevant variable.

Thread 0

```
a = foo();


#pragma omp flush
#pragma omp atomic write
flag = 1;
```

Thread 1

```
do {
  #pragma omp atomic read
  myflag = flag;
} while (!myflag);
#pragma omp flush
b = a;
```

Note 1: OpenMP 4.0: `atomic read seq_cst` and `atomic write seq_cst` eliminate the need for `flush`.

Note 2: Could also use `omp barrier` or OpenMP 4.0 task dependencies.

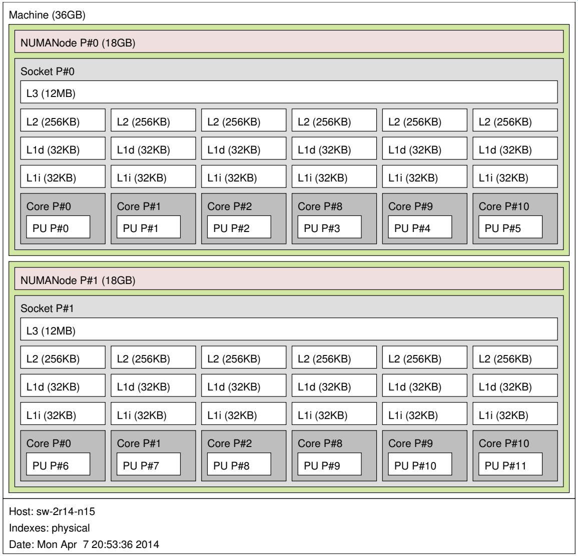
## Performance

Causes for poor performance in shared memory parallel programs:

1. Sequential code: see Amdahl's Law,
2. Communication, data affinity: same thread accesses same data, preferable large contiguous chunks,
3. Load imbalance (see Mandelbrot example, experiment with scheduling).
4. Synchronization, for instance `barrier` overhead.
5. Hardware resource contention, e.g. memory bandwidth.
6. Compiler (non-)optimization: parallel code inhibits some optimizations. Can often be avoided by making more data private.

## ccNUMA

Output from lstopo
`http://www.open-mpi.org/projects/hwloc/`



```
Host: sw-2r14-n15
Indexes: physical
Date: Mon Apr  7 20:53:36 2014
```

## Data locality - Memory access

Calcul Québec

- ccNUMA: cache coherent non-uniform memory access.
- “First touch”: memory is used closest to the core that first wrote to it: try to initialize variables/ array sections using the same thread that uses them later.
- Memory hierarchy:
    - Registers (few bytes, ultimate speed)
    - Caches L1, L2, L3 (2*32kB/core, 256kB/core, 12-20MB)
    - RAM (3 ch. 1333 Mhz, 4 ch. 1600 Mhz)
    - SWAP space
- Taking advantage of caches:
    - Avoid cache misses (between L1-L1, L1-L2, L2-L2, L2-L3, L3-RAM)
        - Contiguous access or reuse memory pages (4kB)
        - Avoid false sharing

## Data locality - Memory access (2)

- Contiguous access or reuse memory pages (4kB)
  - In a 2D or 3D array, depending on the convention of memory allocation, horizontal and vertical accesses have different average speed
  - Random accesses to the same pages in L1 cache
- Avoid false sharing
  - Memory lines (128 bytes) modified alternatively by many threads on different cores
  - Your structure instances must be aligned
  - Use local/private variables

## Data locality - Memory access (3)

Calcul Québec

- Intel environment variables:
  - `KMP_AFFINITY=scatter`: put threads far apart (may improve memory throughput).
  - `KMP_AFFINITY=compact`: put threads close together (improves synchronization overhead, data sharing).
  - `KMP_AFFINITY=<core_list>`: list of explicit cores to put threads on.
- GNU
  - `GOMP_AFFINITY=<core_list>`: list of explicit cores to put threads on.
- OpenMP 4.0
  - `OMP_PROC_BIND=spread` or `close`: like scatter/compact.
  - `OMP_PLACES=threads`, `cores`, or `sockets`: or a list like `{0,1},{6,7}`: restrict threads to hyperthreads, cores, sockets, or list (thread 0 on cores 0 and 1, 1 on 6 and 7).

## Tips and tricks, common errors

Calcul Québec

- The overhead of executing a parallel region is typically in the 1-5 microseconds range: depends on compiler, hardware, no. of threads. `omp barrier` overhead is around 0.4 $\mu$s.
- Use EPCC OpenMP microbenchmarks to do detailed measurements of overheads on your system: `www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking`.
- The sequential execution time has to be several times this to make it worthwhile parallelizing.
- If a section only sometimes takes long enough, use the `if` clause to decide parallelization at runtime.

## Tips and tricks, common errors

Calcul Québec

- Use `nowait` when you can but be careful!
- Mistyping the sentinel (e.g. `!OMP` or `#pragma opm`) typically raises no error message.
- Always, always use `default(none)`. Everybody suffers from “variable blindness”. Spot the bug!

```
#pragma omp parallel for shared (a,b,c,d,N,M)\
private(temp)
for(i=0;i<N;i++){
  for (j=0;j<M;j++){
    temp = b[i]*c[j];
    a[i][j] = temp * temp + d[i];
  }
}
```

## Tips and tricks, common errors

- Example:

```
do i=1,n
  ..... several pages of code referencing 100+
  variables
end do
```

- Determining the correct scope (private/shared/reduction) for all those variables is tedious, error prone and difficult to test adequately.
- Refactor sequential code to

```
do i=1,n
  call loopbody(......)
end do
```

## Tips and tricks, common errors

- Need to use `SAVE` or `static` correctly, but these variables are then shared by default:
  may need to make them `threadprivate`.
- If you have large private data structures, it is possible to run out of stack space: the size of thread stack apart from the master thread can be controlled by the `OMP_STACKSIZE` environment variable.

## Exercise 5: Inner product initialization

Calcul Québec

Consider the inner product example in `innerprod.c` and `innerprod.f90`. Does parallelizing the initialization make the main loop scale better, in particular with `KMP_AFFINITY=scatter`/`OMP_PLACES=sockets`?

## Exercise 6: Block matrix update

See blockmatrix.c and blockmatrix.f90:

```
   do k = 2, n
      do j = 2, n
!$omp parallel do default(shared) private(i)
         do i = 1, m
            x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
         end do
      end do
   end do
```

This program does not scale well. Can you fix it?

## Exercise 7: Segmentation fault

The program `omp_bug4.f` or `omp_bug4.c` causes a segmentation fault. Can you fix it?

## Exercise 8: EPCC syncbench benchmark

Experiment with `syncbench` and different `OMP_PLACES` settings.

## New in OpenMP 4.0/4.5

Calcul Québec

- Support for accelerators (GPUs, Intel Xeon Phi)
- Thread affinity support
- SIMD support for vectorization
- Thread cancellation
- Fortran 2003 support
- Tasks: groups, dependencies, abort
- User defined reductions
- Atomics: sequential consistency

Supported in new compilers (Intel 14.0+, GCC 4.9.1+). **OpenMP 4.5**: C/C++ array reductions, tasks, offload, etc. (GCC 6.1 only): `http://openmp.org/wp/2015/11/openmp-45-specs-released`

## Loop vectorization

- Compilers are now able to identify loops doing independent and identical operations:
  - No dependency between iterations (indices $i$ and $i - 1$, for example)
  - The execution path must be the same: be careful with `if`, `switch`, `break`, `while` and `for` statements
  - Function calls are allowed if they follow the above rules
  - It works very well with vectors or arrays
  - But sometimes the programmer needs to tell the compiler that iterations are independent.
- Example:

```
#pragma omp for simd
   for (i = 0; i < N; i++) {
      c[i] = a[i] * b[i];
   }
```

## Exercise 9: Intel vect./omp simd

Compile `filter.c` with the `-qopt-report` option:

```
$ icc -fopenmp -qopt-report=3
-qopt-report-phase=vec -o filter filter.c
```

Then, make the main loop vectorizable using `omp for simd`

## Further information:

- The standard itself, news, development, tutorials: http://www.openmp.org
- Intel tutorial on YouTube (from Tim Mattson): http://tinyurl.com/OpenMP-Tutorial
- More extensive Advanced OpenMP tutorials: http://www.archer.ac.uk/training/course-material/2015/07/advopenmp_manch https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/2014_sc_openmp/SC14_-_Advanced_OpenMP_Tutorial.pdf http://openmp.org/sc15
- Questions? Write the guillimin support team at guillimin@calculquebec.ca