Optimization on one core
OpenMP, MPI and hybrid programming
An introduction to the de-facto industrial standards

Vincent Keller
Nicolas Richart

July 10, 2019

Table of Contents

Lecture based on specifications ver 3.1

Releases history, present and future

- October 1997: Fortran version 1.0
- Late 1998: C/C++ version 1.0
- June 2000: Fortran version 2.0
- April 2002: C/C++ version 2.0
- June 2005: Combined C/C++ and Fortran version 2.5
- May 2008: Combined C/C++ and Fortran version 3.0
- **July 2011: Combined C/C++ and Fortran version 3.1**
- July 2013: Combined C/C++ and Fortran version 4.0
- November 2015: Combined C/C++ and Fortran version 4.5

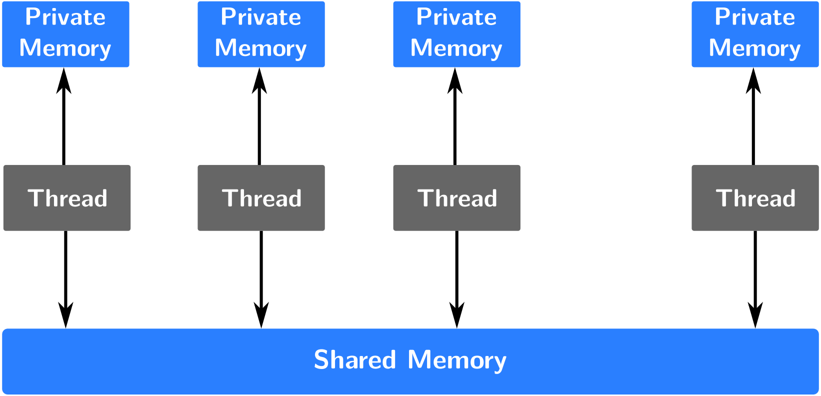
Terminology

- **thread :** an execution entity with a stack and a static memory (*threadprivate memory*)
- **OpenMP thread :** a *thread* managed by the OpenMP runtime
- **thread-safe routine :** a routine that can be executed concurrently
- **processor :** an HW unit on which one or more *OpenMP thread* can execute

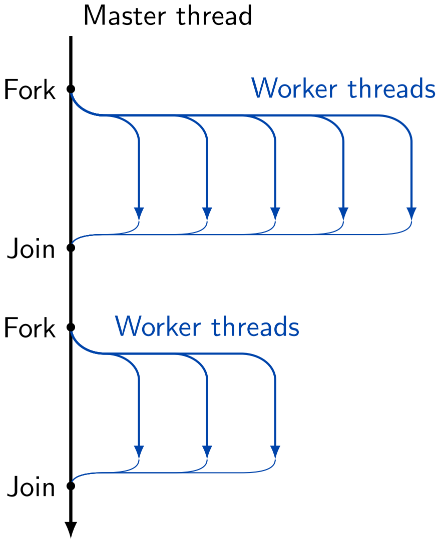
Execution and memory models

- Execution model : fork-join
- One heavy thread (process) per program (initial thread)
- leightweigt threads for parallel regions. threads are assigned to cores by the OS
- No implicit synchronization (except at the beginning and at the end of a parallel region)
- Shared Memory with shared variables
- Private Memory per thread with threadprivate variables

Memory model (simplified)



Execution model (simplified)



OpenMP and MPI/pthreads

- **OpenMP** $\neq$ OpenMPI
- All what you can do with OpenMP can be done with MPI and/or pthreads
- easier **BUT** data coherence/consistency

Syntax in C

OpenMP directives are written as pragmas: `#pragma omp`

Use the conditional compilation flag `#if defined _OPENMP` for the preprocessor

Compilation using the GNU gcc or Intel compiler:

```
gcc -fopenmp ex1.c -o ex1
```

Hello World in C

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) {
   int myrank=0;
   int mysize=1;
#if defined (_OPENMP)
#pragma omp parallel default(shared) private(myrank,
    mysize)
{
   mysize = omp_get_num_threads();
   myrank = omp_get_thread_num();
#endif
   printf("Hello from thread %d out of %d\n", myrank,
       mysize);
#if defined (_OPENMP)
}
#endif
   return 0;
```

Syntax in Fortran 90

OpenMP directives are written as comments: `!$omp omp`

Sentinels `!$` are authorized for conditional compilation (preprocessor)

Compilation using the GNU gfortran or Intel ifort compiler:

```
gfortran -fopenmp ex1.f90 -o ex1
```

Number of concurrent threads

The number of threads is specified in a hardcoded way (*omp_set_num_threads*()) or via an environment variable.

BASH-like shells :

```
export OMP_NUM_THREADS=4
```

CSH-like shells :

```
setenv OMP_NUM_THREADS 4
```

Components of OpenMP

- Compiler directives (written as comments) that allow work sharing, synchronization and data scoping
- A runtime library (libomp.so) that contains informal, data access and synchronization directives
- Environment variables

The `parallel` construct

Syntax

This is the mother of all constructs in OpenMP. It starts a parallel execution.

```
#pragma omp parallel [clause[[,] clause]...]
{
  structured-block
}
```

where *clause* is one of the following:

- `if` or `num_threads` : conditional clause
- `default(private | firstprivate | shared | none)` : default data scoping
- `private(`*list*`)`, `firstprivate(`*list*`)`, `shared(`*list*`)` or `copyin(`*list*`)` : data scoping
- `reduction({` *operator* | *intrinsic_procedure_name* `}` *: list*`)`

Data scoping

What is data scoping ?

- most common source of errors
- determine which variables are **private** to a thread, which are **shared** among all the threads
- In case of a private variable, what is its value when entering the parallel region **firstprivate**, what is its value when leaving the parallel region **lastprivate**
- The default scope (if none are specified) is **shared**
- most difficult part of OpenMP

The data sharing-attributes `shared` and `private`

Syntax

These attributes determines the scope (visibility) of a single or list of variables

```
shared(list1) private(list2)
```

- The `private` attribute : the data is private to each thread and non-initiatilized. Each thread has its own copy. Example : `#pragma omp parallel private(i)`
- The `shared` attribute : the data is shared among all the threads. It is accessible (and non-protected) by all the threads simultaneously. Example : `#pragma omp parallel shared(array)`

The data sharing-attributes `firstprivate` and `lastprivate`

Syntax

These clauses determines the attributes of the variables within a parallel region:

```
firstprivate(list1) lastprivate(list2)
```

- The `firstprivate` like `private` but initialized to the value before the parallel region
- The `lastprivate` like `private` but the value is updated after the parallel region

Worksharing constructs

Worksharing constructs are possible in three “flavours” :

- `sections` construct
- `single` construct
- `workshare` construct (only in Fortran)

The `single` construct

Syntax

```
#pragma omp single [clause[[,] clause] ...]
{
   structured-block
}
```

where *clause* is one of the following:

- `private(`*list*`)`, `firstprivate(`*list*`)`

Only one thread (usualy the first entering thread) executes the `single` region. The others wait for completion, except if the `nowait` clause has been activated

The `for` directive

Parallelization of the following loop

Syntax

```
#pragma omp for [clause[[,] clause] ... ]
{
   for-loop
}
```

where *clause* is one of the following:

- `schedule(`*kind[, chunk_size]*`)`
- `collapse(`*n*`)`
- `ordered`
- `private(`*list*`)`, `firstprivate(`*list*`)`, `lastprivate(`*list*`)`,`reduction()`

The `reduction(...)` clause (Exercise)

How to deal with

```
vec = (int*) malloc (size_vec*sizeof(int));
global_sum = 0;
for (i=0;i<size_vec;i++){
   global_sum += vec[i];
}
```

A solution with the `reduction(...)` clause

```
vec = (int*) malloc (size_vec*sizeof(int));
global_sum = 0;
#pragma omp parallel for reduction(+:global_sum)
   for (i=0;i<size_vec;i++){
      global_sum += vec[i];
   }
```

But other solutions exist !

The `schedule` clause

Load-balancing

clause	**behavior**
schedule(static [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads in a round-robin fashion. If *chunk_size* not specified system decides.
schedule(dynamic [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads when they request them until no chunk remains to be distributed. If *chunk_size* not specified default is 1.

The `schedule` clause

clause	behavior
schedule(guided [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads when they request them. Size of chunks is proportional to the remaining unassigned chunks. By default the chunk size is approx loop_count/number_of_threads.
schedule(auto)	The decisions is delegated to the compiler and/or the runtime system
schedule(runtime)	The decisions is delegated to the runtime system

A parallel `for` example

How to...

... parallelize the dense matrix multiplication $C = AB$ (triple for loop $C_{ij} = C_{ij} + A_{ik}B_{kj}$). What happens using different `schedule` clauses ?)

A parallel `for` example

```c
#pragma omp parallel shared(A,B,C) private(i,j,k,
    myrank)
{
   myrank=omp_get_thread_num();
   mysize=omp_get_num_threads();
   chunk=(N/mysize);
   #pragma omp for schedule(static, chunk)
   for (i=0;i<N;i++){
      for (j=0;j<N;j++){
         for (k=0;k<N;k++){
            C[i][j]=C[i][j] + A[i][k]*B[k][j];
         }
      }
   }
}
```

A parallel `for` example

```
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=1
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 0.33388209342956
 [DGEMM] Performance  [GF/s]: 0.59901385529736
 [DGEMM] Verification        : 2000000000.00000
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=2
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 0.18277192115783
 [DGEMM] Performance  [GF/s]: 1.09425998661625
 [DGEMM] Verification        : 2000000000.00000
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=4
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 9.17780399322509E-002
 [DGEMM] Performance  [GF/s]: 2.17917053085506
 [DGEMM] Verification        : 2000000000.00000
```

Synchronization

Synchronization constructs

Those directives are sometimes mandatory:

- `master` : region is executed by the master thread only
- `critical` : region is executed by only one thread at a time
- `barrier` : all threads must reach this directive to continue
- `taskwait` : all tasks and childs must reach this directive to continue
- `atomic (read | write | update | capture)` : the associated storage location is accessed by only one thread/task at a time
- `flush` : this operation makes the thread's temporary view of memory consistent with the shared memory
- `ordered` : a structured block is executed in the order of the loop iterations

The `master` construct

- Only the master thread execute the section. It can be used in any OpenMP construct

```c
#pragma omp parallel default(shared)
{
...
   #pragma omp master
   {
      printf("I am the master\n");
   }
...
}
```

Nesting regions

Nesting

It is possible to include parallel regions in a parallel region (i.e. nesting) under restrictions (cf. sec. 2.10, p.111, *OpenMP: Specifications ver. 3.1*)

Runtime Library routines

Usage

- The functions/subroutines are defined in the lib `libomp.so` / `libgomp.so`. Don't forget to include `#include <omp.h>`
- These functions can be called anywhere in your programs

Runtime Library routines

Timing routines

routine	behavior
omp_get_wtime	returns elapsed wall clock time in seconds.
omp_get_wtick	returns the precision of the timer used by omp_get_wtime

Environment variables

Usage

- Environment variables are used to set the ICVs variables
- under `csh` : `setenv OMP_VARIABLE "its-value"`
- under `bash` : `export OMP_VARIABLE="its-value"`

Environment variables

variable	what for ?
OMP_SCHEDULE	sets the run-sched-var ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
OMP_NUM_THREADS	sets the nthreads-var ICV that specifies the number of threads to use in parallel regions

The apparent "easiness" of OpenMP

"Compared to MPI, OpenMP is much easier"

In the reality

- Parallelization of a non-appropriate algorithm
- Parallelization of an unoptimized code
- Race conditions in shared memory environment
- Memory coherence
- Compiler implementation of the OpenMP API
- (Much) more threads/tasks than your machine can support

OpenMP Thread affinity

Affinity = on which core does my thread run ?

Show and set affinity with Intel executable

By setting the `export KMP_AFFINITY=verbose,SCHEDULING` you are able to see where the OS pin each thread

Show and set affinity with GNU executable

By setting the `export GOMP_CPU_AFFINITY=verbose,SCHEDULING` you are able to see where the OS pin each thread

OpenMP Thread affinity with compact

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY=verbose,compact
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,9}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {1,9}
 [DGEMM]   Compute time [s]    :   0.344645023345947
 [DGEMM]   Performance  [GF/s]:   0.580307233391397
 [DGEMM]   Verification        :    2000000000.00000
```

OpenMP Thread affinity with scatter

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY=verbose,scatter
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {4,12}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,9}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {5,13}
 [DGEMM]   Compute time [s]   :   0.204235076904297
 [DGEMM]   Performance  [GF/s]:   0.979263714301724
 [DGEMM]   Verification       :    2000000000.00000
```

OpenMP Thread affinity with explicit (a kind of pining)

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY='proclist=[0,2,4,6],explicit',verbose
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {6,14}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {2,10}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {4,12}
 [DGEMM]   Compute time [s]   :   0.248908042907715
 [DGEMM]   Performance  [GF/s]:   0.803509591990774
 [DGEMM]   Verification       :    2000000000.00000
```

"OpenMP-ization" strategy

- **STEP 1** : Optimize the sequential version:
  - Choose the best algorithm
  - "Help the (right) compiler"
  - Use the existing optimized scientific libraries
- **STEP 2** : Parallelize it:
  - Identify the bottlenecks (heavy loops)
  - "auto-parallelization" is rarely the best !

Goal

Debugging - Profiling - Optimization cycle. Then parallelization !

Tricks and tips

- **Algorithm**: choose the "best" one
- **cc-NUMA**: no (real) support from OpenMP side (but OS). A multi-CPU machine is not a real shared memory architecture
- **False-sharing**: multiple threads write in the same cache line
- **Avoid barrier**. This is trivial. Bus sometimes you can't
- **Small number of tasks**. Try to reduce the number of forked tasks
- **Asymetrical problem**. OpenMP is well suited for symetrical problems, even if tasks can help
- **Tune the schedule**: types, chunks...
- **Performance expectations**: a theoretical analysis using the simple Amdahl's law can help
- **Parallelization level**: coarse (SPMD) or fine (loop) grain ?

What's new with OpenMP 4.0 ?

- Support for new devices (`Intel Phi`, `GPU`,...) with `omp target`. Offloading on those devices.
- Hardware agnostic
- League of threads with `omp teams` and distribute a loop over the team with `omp distribute`
- SIMD support for vectorization `omp simd`
- Task management enhancements (cancelation of a task, groups of tasks, task-to-task synchro)
- Set thread affinity with a more standard way than `KMP_AFFINITY` with the concepts of `places` (a thread, a core, a socket), `policies` (spread, close, master) and `control settings` the new clause `proc_bind`