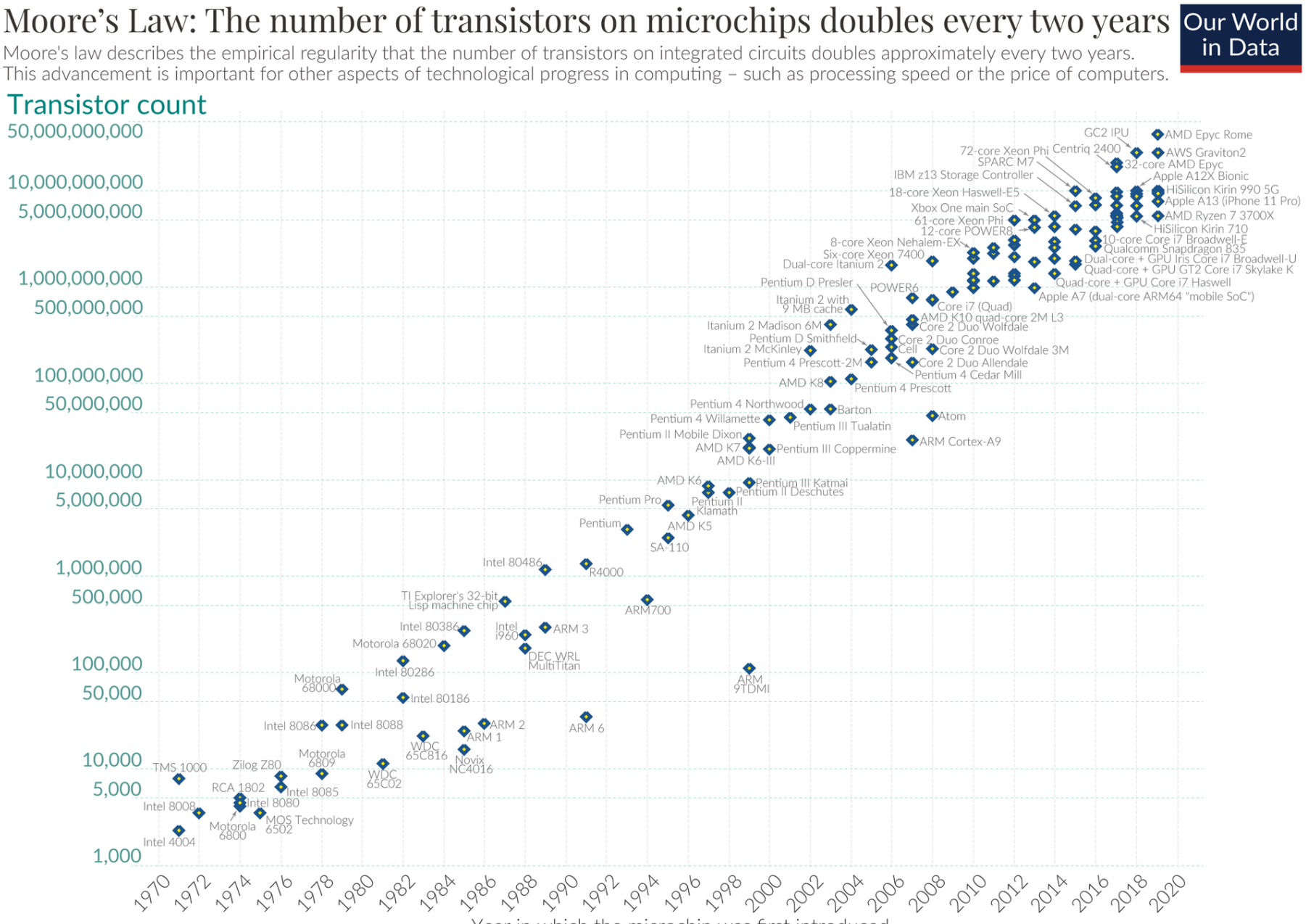
Calcul haute performance pour les systèmes embarqués (HPEC)
Lab 2 - Multicore & OpenMP

Bertrand LE GAL

IMS laboratory, CNRS UMR 5218
Digital Circuits and Systems team
Bordeaux-INP, University of Bordeaux
France

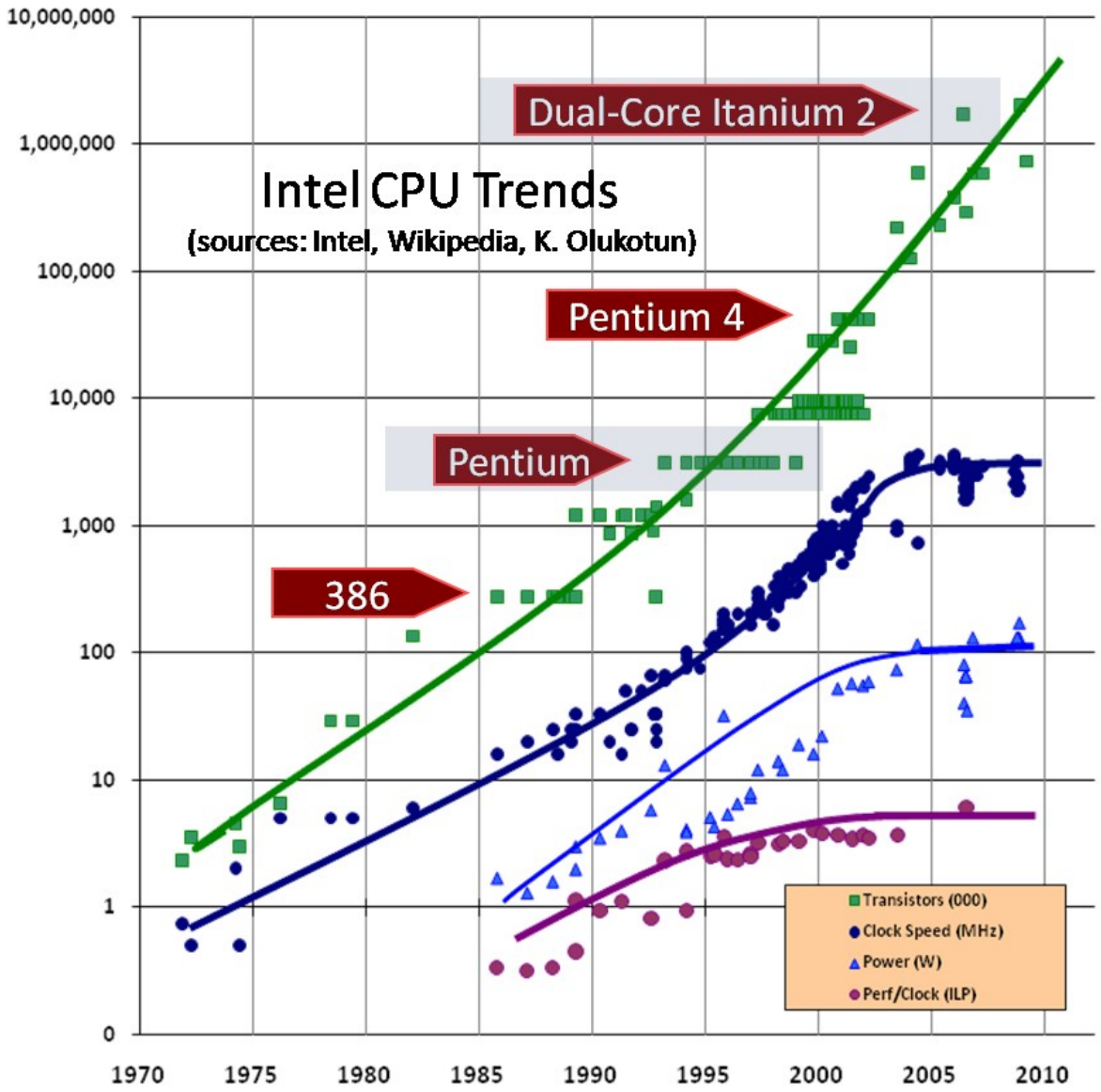
bertrand.legal@enseirb-matmeca.fr

Moore's law during the last decades



Moore's law and performance scaling

- Working frequencies
  - Up to 8 MHz en 1979
  - Up to 200 GHz in 1997,
  - Up to 4 GHz in 2006,
  - Up to 5 GHz in 2022,
  - Power issues (> 200W).
- Improvements
  - Lower working frequencies,
  - More computation cores,
  - Should improve global performances.
- Needs to upgrade applications.

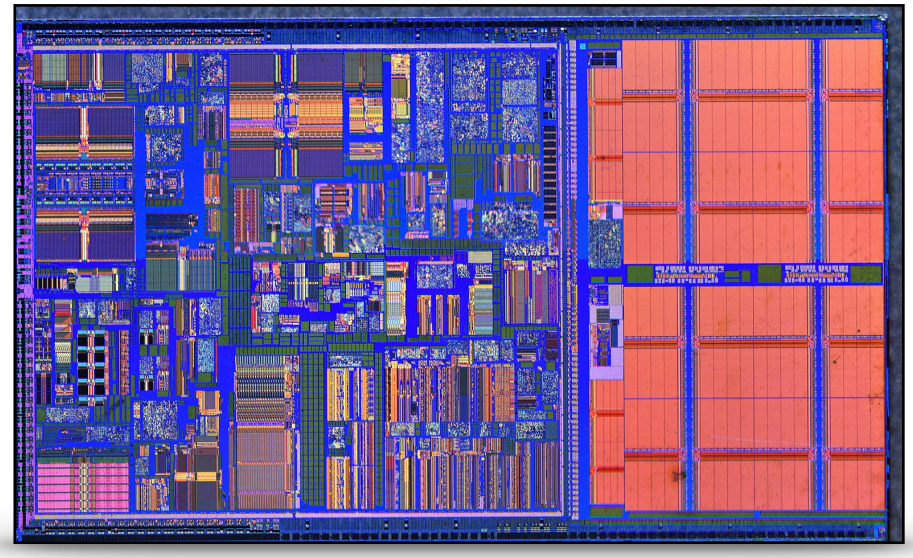


https://www.intel.com/pressroom/kits/quickreffam.htm

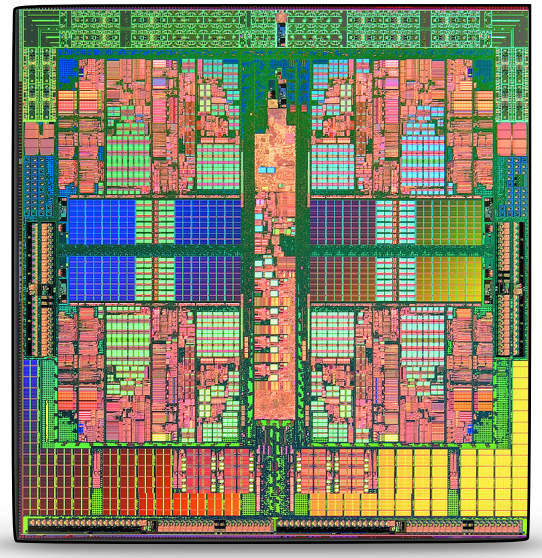
https://cs.stackexchange.com/questions/27875/moores-law-and-clock-speed

Introduction au pipeline

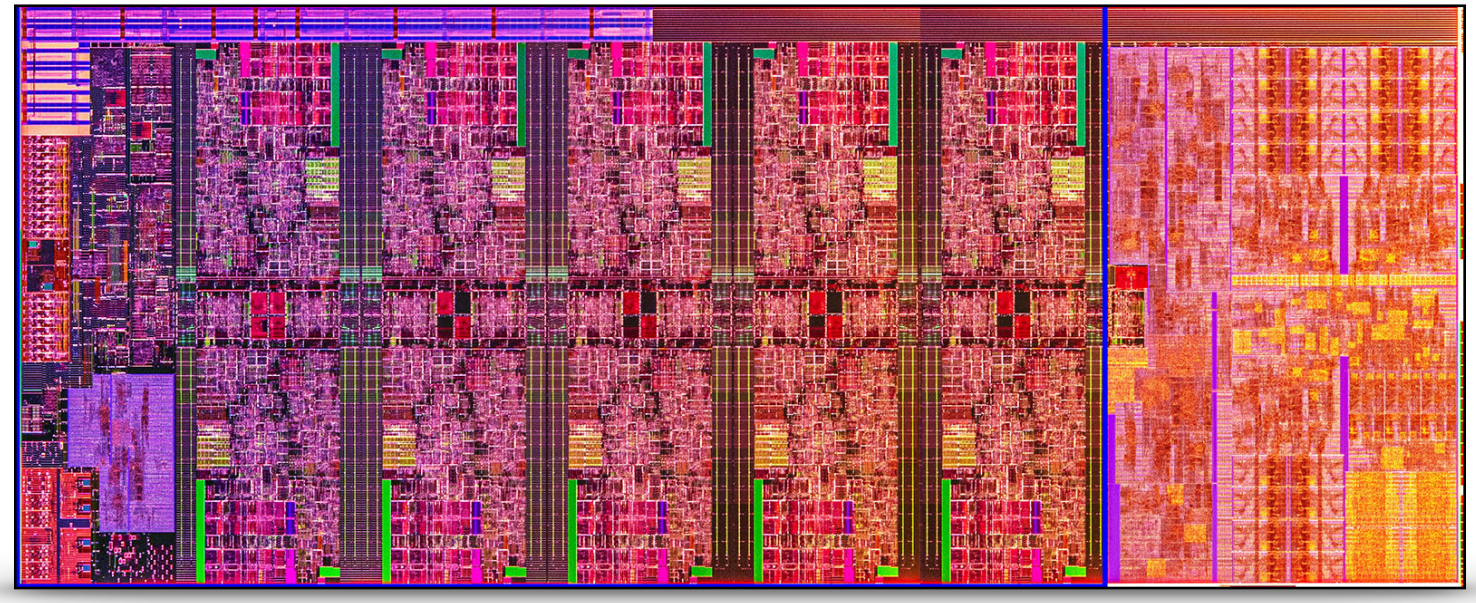
Pentium III core



Pentium core 2 duo



Comet Lake S-series CPUs (2022)



Introduction to OpenMP

- Parallel computing is (quite) old,
- OpenMP objective
  - Make parallel computing easy for software developers.
- A long way
  - OpenMP 1.0 for Fortran in 1997,
  - OpenMP 1.0 for C/C++ in 1998,
  - A lot of work,
  - OpenMP 5.2 in novembre 2021.
- Easy to use
  - Natively supported by recent C/C++ compilers and OS (except MacOs).

What is OpenMP ?

- OpenMP **is** a multi-vendor standard to perform shared-memory multithreading,
- OpenMP threads **share** a single executable, global memory, and heap (malloc, new),
- OpenMP **uses** the fork-join model,
- Using OpenMP **requires** no dramatic code changes,
- OpenMP **gives** you the biggest multithread benefit per amount of work you have to put in to using it !
- Add "pragmas" to your code and let the compiler work.

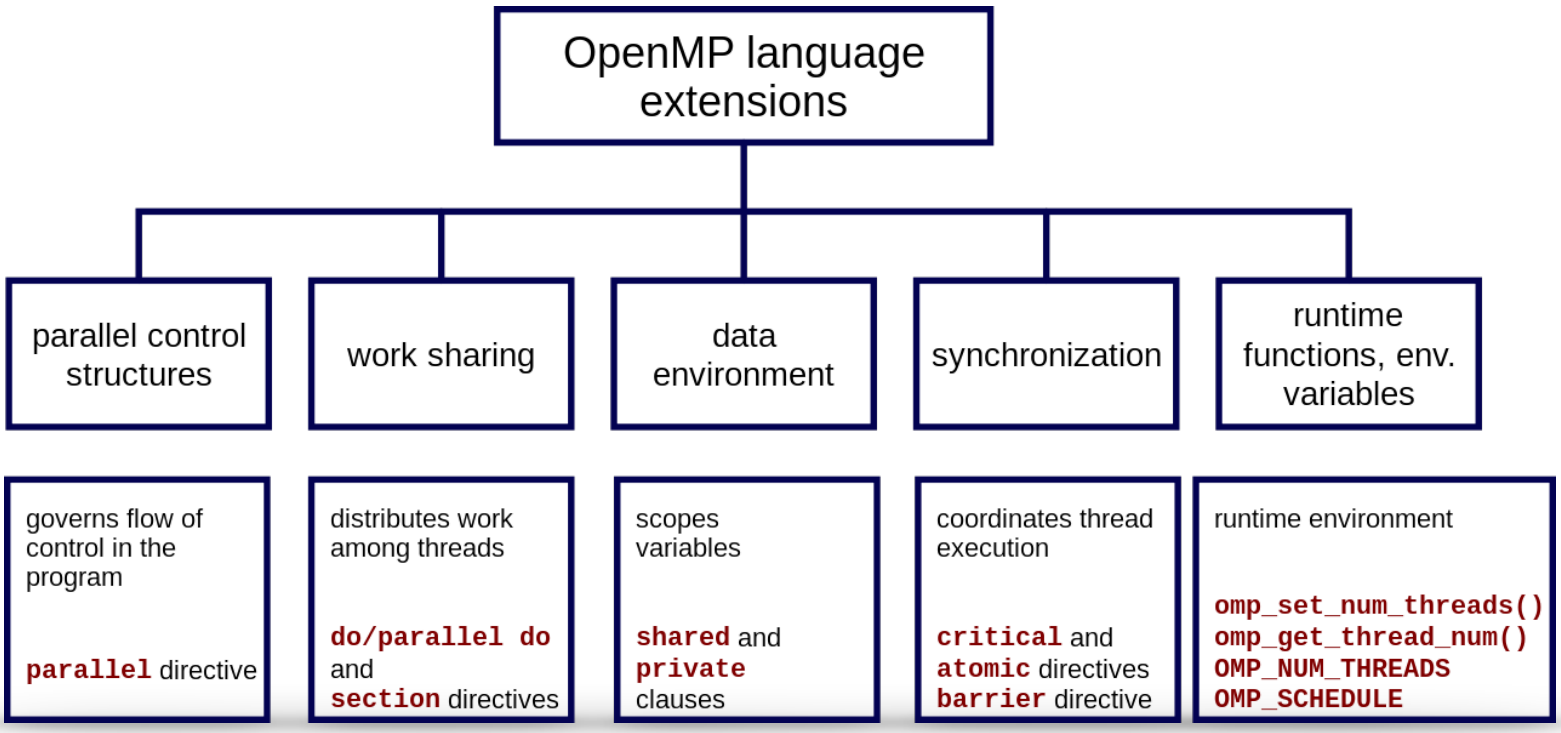
https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/openmp.1pp.pdf

What is not OpenMP ?

- OpenMP **doesn't** replace your brain !
- OpenMP **doesn't** check for data dependencies, data conflicts, deadlocks, or race conditions. You are responsible for avoiding those yourself
- OpenMP **doesn't** guarantee identical behavior across vendors or hardware, or even between multiple runs on the same vendor's hardware
- OpenMP **doesn't** guarantee the order in which threads execute, just that they do execute • OpenMP is not overhead-free
- OpenMP **doesn't** prevent you from writing code that triggers cache performance problems (such as in false-sharing), in fact, it makes it really easy.

The OpenMP pragmas

- Software designers have access to reserved keyword (« pragma ») to tell the compiler what they want,
- Pragmas can automatically discard/enable at runtime depending ion compiler options, improving the flexibility.



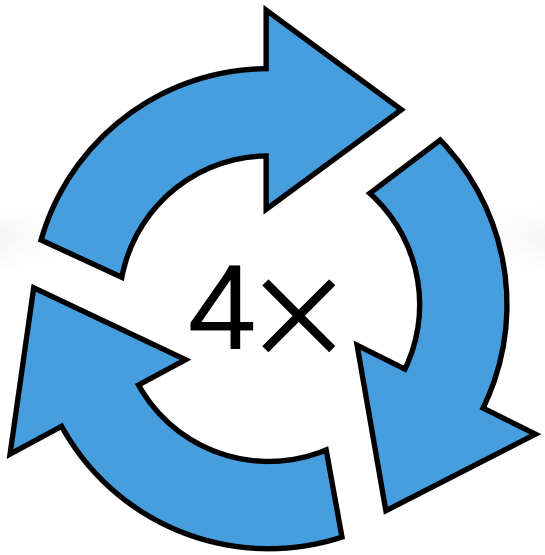
The (strange) OpenMP « hello world » version

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[] )
{
    omp_set_num_threads( 8 );

    #pragma omp parallel default(none)
    {
        printf("(thread #%d) Hello world !\n", omp_get_thread_num( ) );
    }

    return 0;
}
```

- g++ openmp_hello.cpp -o openmp_hello -O3 *-fopenmp*

The (strange) OpenMP « hello world » version

```
❯ ./openmp_hello
(thread #3) Hello world !
(thread #0) Hello world !
(thread #4) Hello world !
(thread #7) Hello world !
(thread #2) Hello world !
(thread #1) Hello world !
(thread #6) Hello world !
(thread #5) Hello world !
```




```
❯ ./openmp_hello
(thread #0) Hello world !
(thread #7) Hello world !
(thread #6) Hello world !
(thread #4) Hello world !
(thread #3) Hello world !
(thread #1) Hello world !
(thread #2) Hello world !
(thread #5) Hello world !
```

```
❯ ./openmp_hello
(thread #0) Hello world !
(thread #5) Hello world !
(thread #4) Hello world !
(thread #1) Hello world !
(thread #3) Hello world !
(thread #2) Hello world !
(thread #6) Hello world !
(thread #7) Hello world !
```

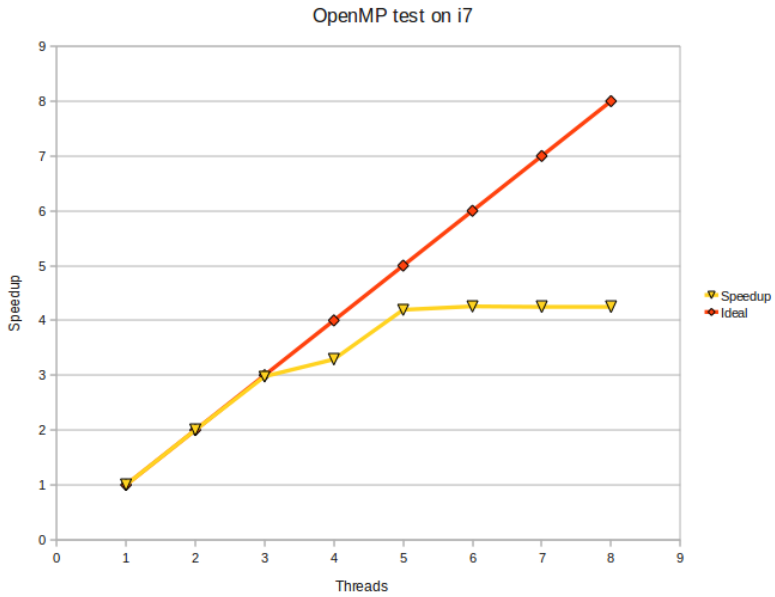
```
❯ ./openmp_hello
(thread #0) Hello world !
(thread #7) Hello world !
(thread #4) Hello world !
(thread #2) Hello world !
(thread #3) Hello world !
(thread #1) Hello world !
(thread #6) Hello world !
(thread #5) Hello world !
```

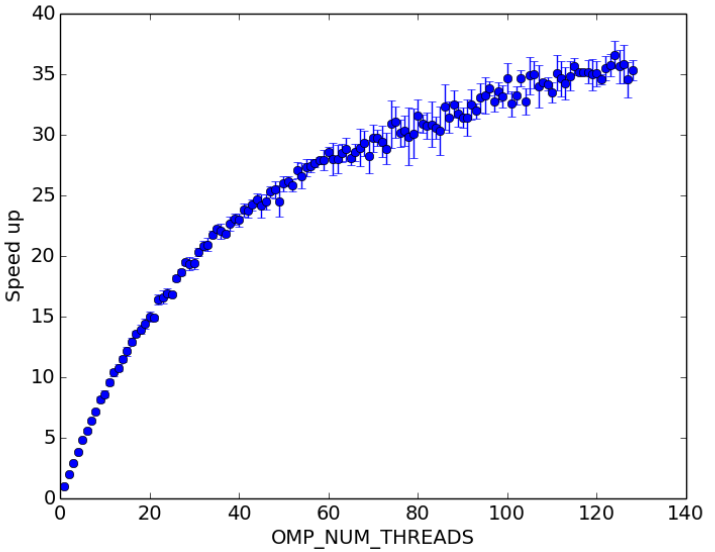
Configuring the execution platform

```
//
// Specify the amount of OpenMP threads we want to have available
//
void omp_set_num_threads(int num_threads);

//
// Asking how many cores the platform has
// (number of threads != number of physical processor cores)
//
int omp_get_num_procs(void);

//
// Asking how many OpenMP threads this program is using right now
//
int omp_get_num_threads( );


//
// Asking which thread number this one is:
//
int omp_get_thread_num( );
```

What piece of code could be parallelized ?

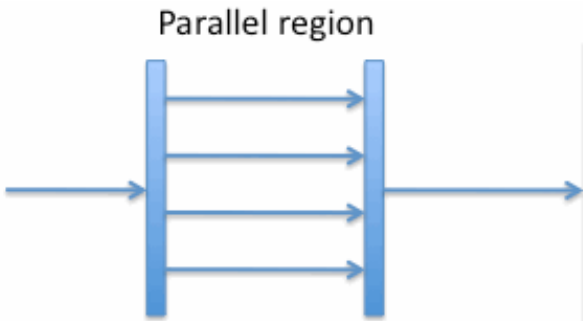
- Parallelization is useful for critical tasks that are independent,
  - Data independent,
  - Ctrl independent.
- For instance:
  - The for loops,
  - The independent function calls.
- The overall source codes could not benefits from parallelization,
  - Modifying algorithm could help in parallelization => more work.





Parallelization of **for** loops

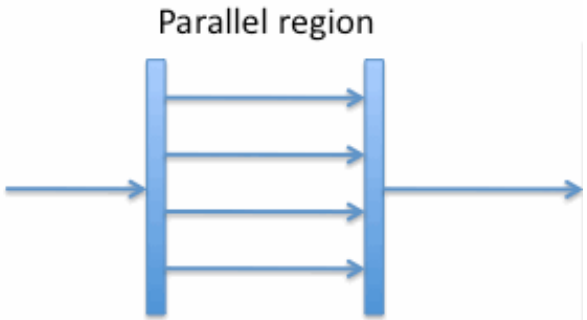
- A dedicated **pragma** is dedicated to for loop parallelization:
- In this example:
  - **y array** is shared across threads,
  - **x value** is private [0, 65536/T-1],
  - the **number of threads** is selected automatically at runtime.
- Thread synchronisation is done at the end of for loop.




```
#include <stdio.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[] )
{
    float y[65536];
    const float a = atof( argv[ 1 ] );
    const float b = atof( argv[ 2 ] );

    #pragma omp parallel for
    for (int x = 0; x < 65536; x += 1)
    {
        y[x] = a * ((float)x) + b;
    }

    //
    // Storing the [y] dataset in a file
    //

    return 0;
}
```

Parallelization of **tasks** (eg. function calls)

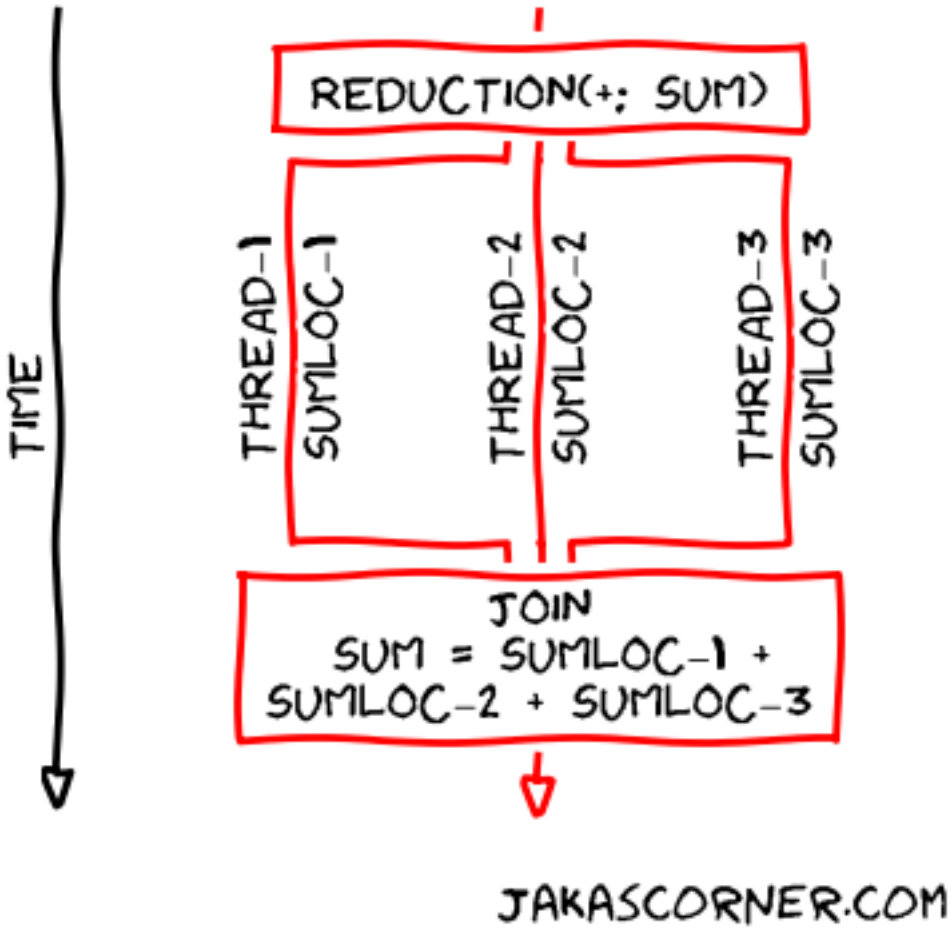
- A dedicated **pragma** is dedicated to for sub-task parallelization:
- In this example:
  - **parallel section** defines the parallel zone,
  - **section** describes the different sub-tasks,
  - the **number of threads** is selected automatically at runtime / specified.
- Synchronisation is done at the last brace of the **parallel section**.




```c
#include <stdio.h>
#include <omp.h>

void fA(){ printf("Task A\n"); }
void fB(){ printf("Task B\n"); }

int main()
{
    omp_set_num_threads( 2 );

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            fA();
        }
        #pragma omp section
        {
            fB();
        }
    }
    return 0;
}
```

What happen with real life applications ?

- In real life applications:
  - Processing and data are not always independent,
  - Algorithms transformation or source code adaptation are needed.
- Issues with shared result:
  - Many (write) actors at the same time,
- For example:
  - Data reduction (+, min, max, etc.)
  - Shared data are not enough (Write After Write issue).

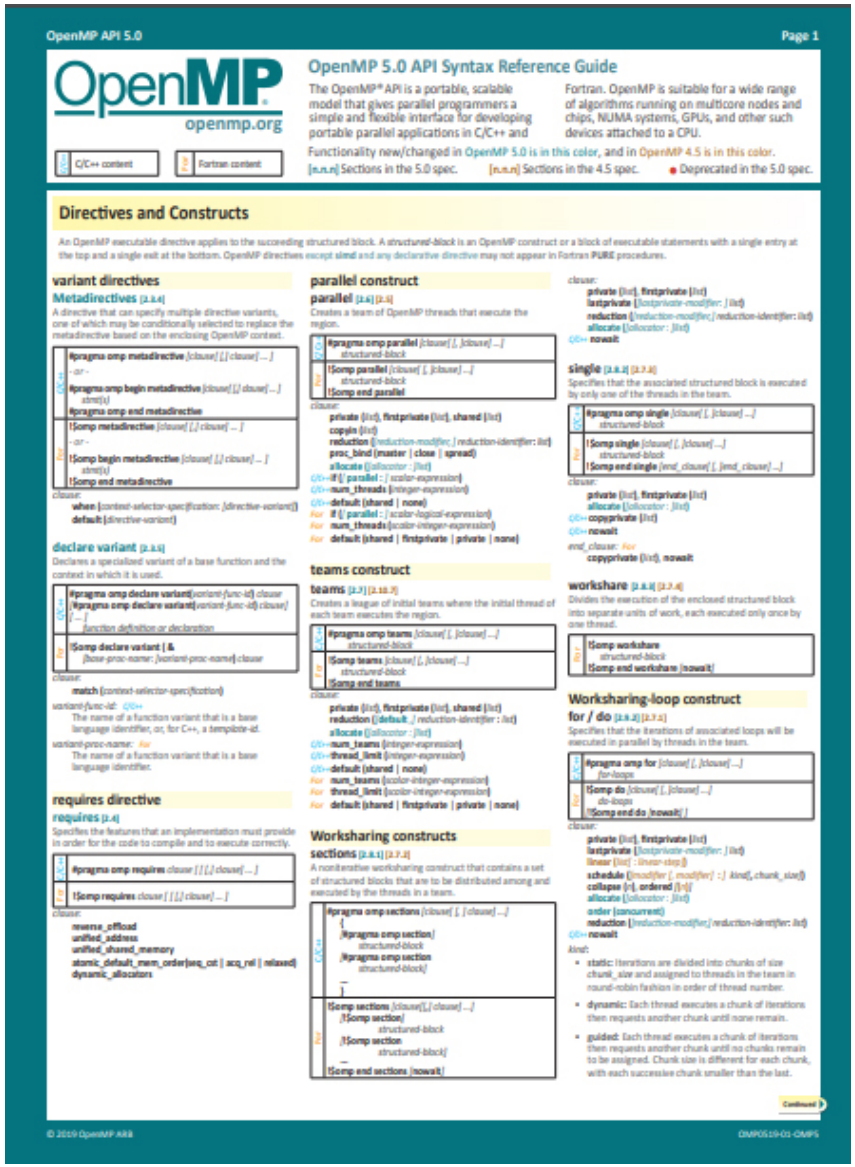


Sum reduction with OpenMP features

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[] )
{

   float* argv;
   float   sum = 0.f;

   #pragma omp parallel for shared(sum) reduction(+: sum)
   {
       for (auto i = 0; i < argc; i++)
       {
           sum += stof( argv[i] );
       }
   }

   return 0;
}
```

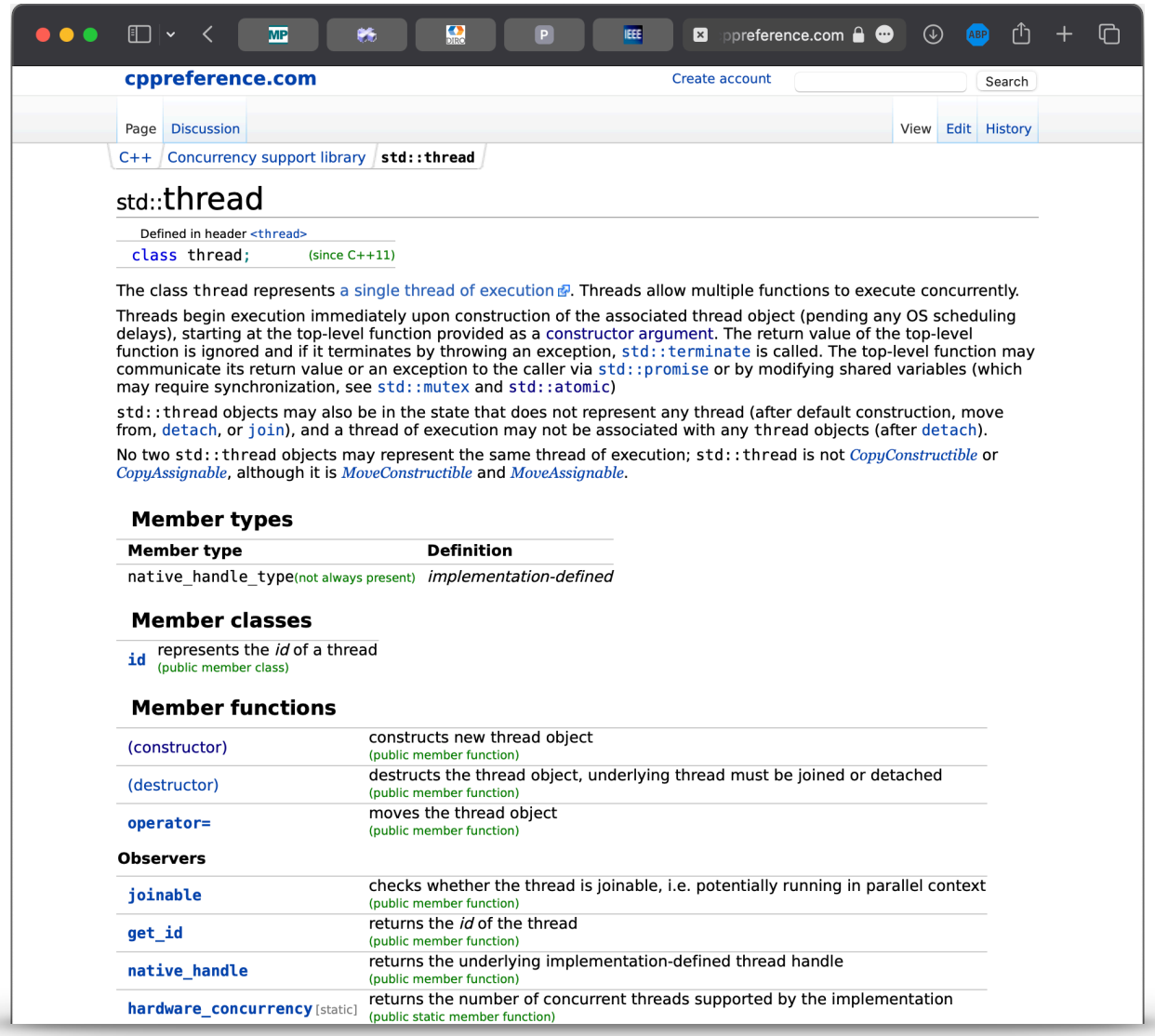
Conclusion concerning OpenMP library

- OpenMP is not a young framework…
  - A lot of tutorials, documents, source codes, … exist.
- OpenMP is not always interesting due to penalties
  - Large datasets,
  - High complexity kernels.
- The main difficulties comes from algorithm and your knowledge,
  - It needs transformation,
  - You need to understand it.



https://www.openmp.org/resources/refguides/

The std::thread parallelization approach



cppreference.com

Create account | Search

Page | Discussion | View | Edit | History

C++ / Concurrency support library / std::thread

std::thread

Defined in header <thread>

```
class thread;        (since C++11)
```

The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, std::terminate is called. The top-level function may communicate its return value or an exception to the caller via std::promise or by modifying shared variables (which may require synchronization, see std::mutex and std::atomic)

std::thread objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may not be associated with any thread objects (after detach).

No two std::thread objects may represent the same thread of execution; std::thread is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Member types

Member type	Definition
native_handle_type (not always present)	*implementation-defined*

Member classes

id	represents the *id* of a thread (public member class)

Member functions

(constructor)	constructs new thread object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
operator=	moves the thread object (public member function)
Observers	
joinable	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
get_id	returns the *id* of the thread (public member function)
native_handle	returns the underlying implementation-defined thread handle (public member function)
hardware_concurrency [static]	returns the number of concurrent threads supported by the implementation (public static member function)

Example of std::thread parallelization (1/2)

```
#include <iostream>
#include <thread>

void fA(){ printf("Task A\n"); }
void fB(){ printf("Task B\n"); }

int main()
{
   std::thread fst (fA);  // spawn new thread
   std::thread snd (fB);  // spawn new thread

   std::cout << "fA and fB now execute in //...\n";

   // synchronize threads:
   fst.join();  // pauses until first finishes
   snd.join();  // pauses until second finishes

   std::cout << "fA and fB completed.\n";

}
```

Example of std::thread parallelization (2/2)

```cpp
#include <iostream>
#include <thread>

void SUM(float* array, int start, int stop, float& val)
{
   for(int i = start, i < stop, i += 1)
      val += array[i];
}

int main()
{
   float array[1024]; // un tableau de data...
   float s1 = 0.f;
   float s2 = 0.f;
   std::thread fst (SUM, array,   0,  512, s1);
   std::thread snd (SUM, array, 512, 1024, s2);

   // synchronize threads:
   fst.join();  // pauses until first finishes
   snd.join();  // pauses until second finishes

   float sum = s1 + s2;
   // ... ... ...
}
```