Programming in parallel

A (quick) overview

Pierre Beaujean (Belgian NCC for HPC)

v0.1a, November 2021

About me

https://pierrebeaujean.net/

- 2006: Ubuntu 6.06 LTS, from a friend
- 2007: Started learning programming from the *Site du Zéro* (now deceased!).
- 2009: chose to study chemistry instead, but keeps Linux
- 2015: Switching to *Zeste de Savoir* (I'm still there!)
- January 2015: Started and fulfilled a master's thesis in Quantum Chemistry
- September 2015: started (and recently fulfilled!) a PhD in Quantum Chemistry, as Teaching Assistant
- September 2017: started (and fulfilled) a bachelor in computer science (evening classes)
- September 2021: started to work for the Belgian NCC for HPC.

Preliminary notes

- Jokes and opinions are my own, not the one of the NCC Belgium ;)
- First of all, "Premature optimization is the root of all evil" (Sir Tony Hoare)
- Code optimization is an art, and the result depends on the target architecture (cache size, instruction set, connectivity, etc. See `lscpu`).
- Here, we will not look for the best optimized code (i.e., using manual optimization techniques such as manual loop unrolling or writing asm). We will just look at the effect of some modifications (and sometimes, the assembler).
- Codes are in C (but could be in Fortran), compiled with `gcc` (Intel compiler may provide better results on Intel architectures).
- Two simple examples from linear algebra (LAPACK): `*dot` and `*axpy`. If you ever need that, use libraries instead!

The examples for today

- *(s/d)dot*: dot product of two vectors: $r = \vec{x} \cdot \vec{y}$.

```
float sdot(int n, float* x, float* y) {
  float sum = .0f;
  if (n > 0) {
    for(int i=0; i < n; i++)
      sum += y[i] * x[i];
  }
  return accum;
}
```

Note: actually use a **Kahan sum**!

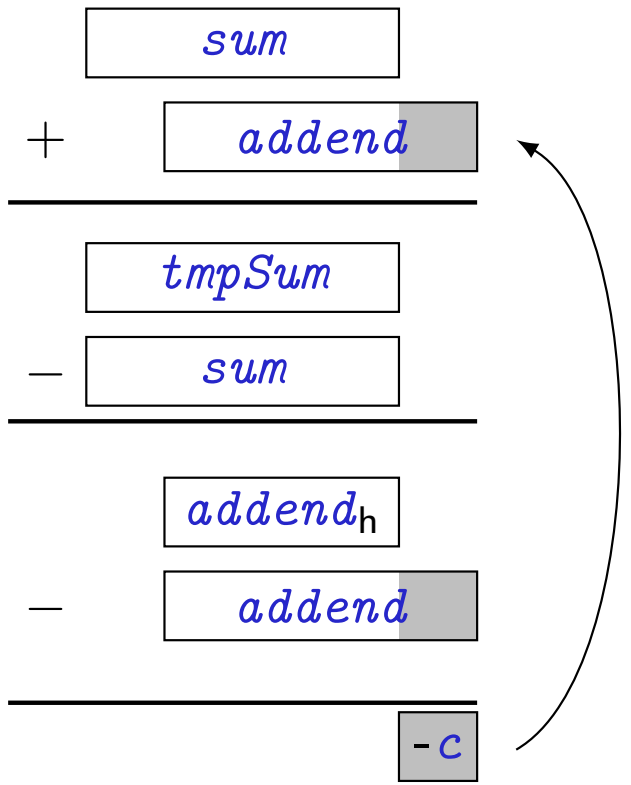
- *(s/d)axpy*: generalized vector addition: $\vec{y} = \vec{y} + a\,\vec{x}$.

```
void saxpy(int n, float alpha, float* x, float* y) {
  if (n > 0 && alpha != 0.f) {
    for(int i=0; i < n; i++)
      y[i] += alpha * x[i];
  }
}
```

Preamble: Kahan sum

https://en.wikipedia.org/wiki/Kahan_summation_algorithm

```
float sdot(int n, float* x, float* y) {
  float sum = .0f, c = .0f;
  float addend, tmpSum;
  if (n > 0) {
    for(int i=0; i < n; i++) {
      addend = y[i] * x[i] - c;
      tmpSum = sum + addend;
      c = (tmpSum - sum) - addend;
      sum = tmpSum;
    }
  }
  return sum;
}
```



Preamble: don't forget 'bout optimization

	*dot		*axpy	
	s	d	s	d
-O0	426.5	420.4	137.3	136.4
-O1	141.1	140.1	30.7	31.7

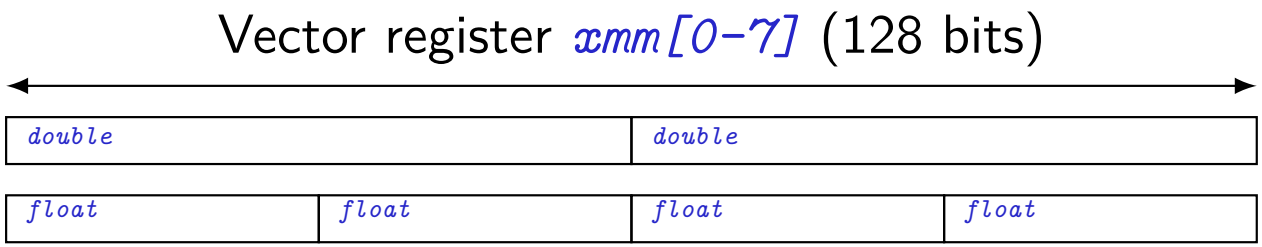
Table: Average running times (in ms) with `-N 1000`, on 2^{24} numbers.

- Ran on AMD EPYC 7542, and compiled with GCC 10.2
- Compilers does a decent job at optimizing (see `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` for the details)
- `-O2` and `-O3` include stuff we will see later on (but you **should** use `-O3`, or even `-Ofast`).
- If you are brave enough `objdump -dS -M intel <program, compiled with -g>`. Here, gcc chooses to use `muls(s|d)` (from the old SSE extension).

"SIMD for nothin', FLOPS for free" (old song)

SIMD

- SIMD = "simple instruction, multiple data" (in Flynn's taxonomy). Correspond to data level parallelism.
- Refers to (at least) two mechanisms: pipelining[1] and **packed instruction** / vector processing.
- Response to early graphic cards: MMX (64 bits registers, for integers) / 3DNow! ($+$`float`) $\rightarrow$ SSE (128 bit registers, $+$`double`) $\rightarrow$ AVX (256 and 512 bit registers)
- Example: SSE



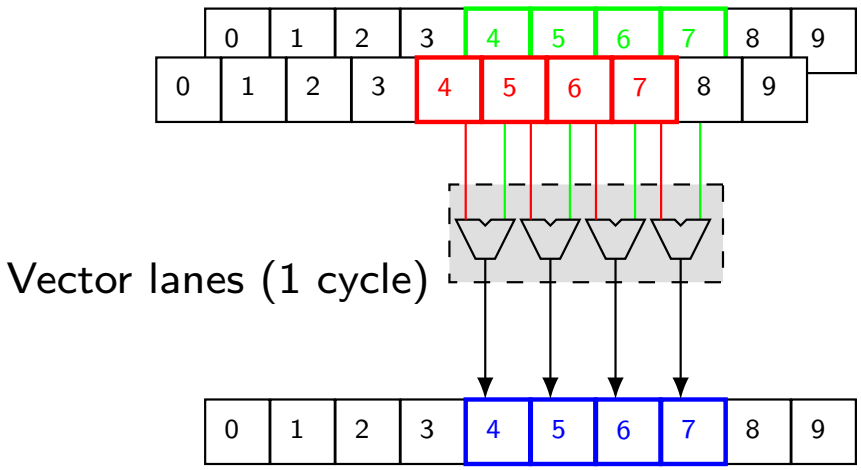
[1] See "Fonctions et concepts des ordinateurs" (INFOB2126/IHDCB142)

Vector processing

Examples with SSE (as modified by AVX):

```
float x[N], y[N], z[N];
for(i=0; i < N; i++)
  z[i] = x[i] + y[i];
```

The last line is translated into `vaddps xmm1,xmm2,xmm3` (among other things).



The program runtime for this part is (theoretically!) divided by the number of vector lanes.

Howto!

- Write assembler (**don't!**)
- Use compiler intrinsic (if you **really** have to):

```
#include <xmmintrin.h>
__m128 r1 = _mm_load_ps(&x[4*i]);
__m128 r2 = _mm_load_ps(&y[4*i]);
__m128 r0 = _mm_add_ps(r1, r2);
_mm_store_ps(&z[4*i],r0);
```

- Use SIMD classes (in C++)
- Add *#pragma omp simd* above the loop (other exists, but it depends on the compiler) and compile with *-fopenmp-simd* (gcc).
- Let the compiler do its job ... And maybe check its result. With gcc, *-ftree-vectorize* (or *-O2*) and *-fopt-info-loop-(optimized|missed)*. The compiler may refuse to optimize a loop, though.

Aliasing?

- If one compiles the serial code of the *axpy* example, the output is something like

```
1_serial.c:11:9: optimized: loop vectorized using 16 byte
vectors
1_serial.c:11:9: optimized:  loop versioned for
vectorization because of possible aliasing
(...)
```

  So the loop (here the one of *saxpy*) get vectorized (with 16-byte vector, so a 128 bit register), but also versioned.

- Here, the compiler is not able to determine whether x and y point to the same memory address, so it keeps both versions (and will choose at runtime).
- Use *restrict* to help:

```
void saxpy(int n, float alpha,
  float* restrict x, float* restrict y)
```

Results

	Extension	*saxpy*	*daxpy*
-O1	SSE	30.7	31.7
-O1 -ftree-vectorize	SSE + l.u.	16.0	27.3
-O1 -ftree-vectorize -march=native -mtune=native	AVX2 + l.u.	14.5	25.8
-O1 -lm -ftree-vectorize -mavx2	SSE + AVX2 + l.u.	16.2	27.3

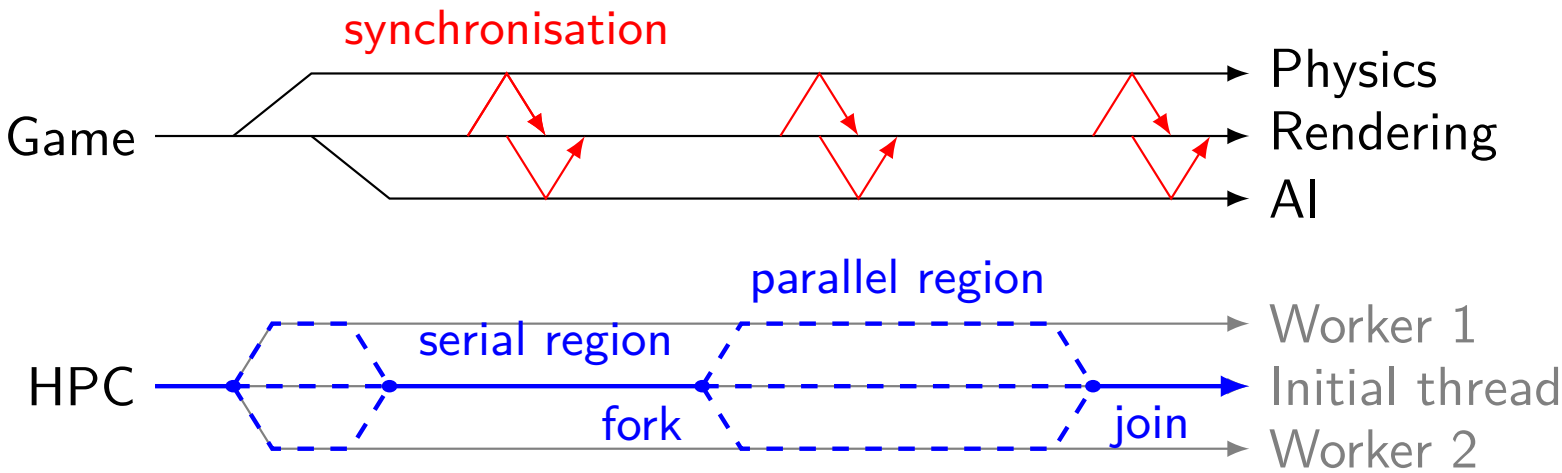
Table: Average running times (in ms) with *-N 1000*, on 2^{24} numbers.

- Use *-march=native -mtune=native* for best performances.
- *dot* does not vectorize ... **Why?**

Threading made easy: OpenMP

Threads

- Reminder: a thread is an independent sequence of instructions that can be managed by a scheduler. Multiple threads can exist for one process, with shared memory. May induce **race conditions** (e.g., data race).
- Using dedicated libraries (e.g., pthreads) may be cumbersome, especially since, in scientific computing, the paradigm is usually different.



- OpenMP is built on the fork/join paradigm, and generally does not use synchronization.

OpenMP

- Main paradigm in scientific computing for single-node applications!
- To use OpenMP, add directives (*#pragma omp*) and rely on the compiler for the underlying details. One of the advantages is that it allows an **incremental** adoption.
- But, of course, there are still two problems that the compiler cannot solve on its own:
  1. data dependencies:

```
for(i=2;i<N;i++) // Vietnam flashback, anyone?
  a[i] = a[i-1] + a[i-2];
```

  2. data race (more on that later):

```
int x = 0;
for(i=0;i<N;i++)
  x = x + a[i];
```

Hello, world!

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

int main() {
  #pragma omp parallel
  { // fork!
    printf("I'm thread %d of %d\n",
      omp_get_thread_num(), omp_get_num_threads());
  } // implicit join
}
```

Compile and execute:

```
$ gcc -o hello hello.c -fopenmp
$ OMP_NUM_THREADS=4 ./hello
I'm thread 0 of 4
I'm thread 2 of 4
I'm thread 3 of 4
I'm thread 1 of 4
```

Parallel $\neq$ Worksharing

- The content of the parallel region is executed by all threads:

```
#pragma omp parallel
{
  for(i=0;i<N;i++)
    a[i] = a[i]+ 1;
} // each thread execute the N iterations
```

- Explicit work sharing:

```
#pragma omp parallel
{
  int t = omp_get_thread_num();
  int n = omp_get_num_threads();
  int low = N * t / n, high = N * (t + 1) / n;
  for(int i = low; i < high; i++)
    a[i] = a[i] + 1;
} // each thread do its part
```

Parallel $\neq$ Worksharing

But there is an easier way:

- With a directive...

```
#pragma omp parallel
{
  #pragma omp for
  for(i=0;i<N;i++)
    a[i] = a[i]+ 1;
} // each thread do its part
```

- ... And directives can be combined:

```
#pragma omp parallel for
for(i=0;i<N;i++)
  a[i] = a[i]+ 1; // no curly braces required, but barrier
```

Note: of course, *#pragma omp for* must be in a parallel region (otherwise, it is only bound to the master thread). It may be useful for **orphaning**, though.

Say hello to data race!

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

int main() {
  int i, n; // ... why not?
  #pragma omp parallel
  {
    i = omp_get_thread_num(); n = omp_get_num_threads();
    printf("I'm thread %d of %d\n", i, n);
  }
}
```

Compile and execute gives (sometimes[2]):

```
$ OMP_NUM_THREADS=4 ./hello
I'm thread 2 of 4
I'm thread 2 of 4
I'm thread 3 of 4
I'm thread 2 of 4
```

[2] Data race is difficult to spot, since they do not happen every time ;)

First problem: communism. Let's privatize!

- Any variable declared outside a parallel region is shared by default.
- Either we declare them inside:

```
#pragma omp parallel
{
  int i = omp_get_thread_num(); // private
  int n = omp_get_num_threads(); // private
  printf("I'm thread %d of %d\n", i, n);
}
```

- Or we use the *private(list)* clause:

```
int i, n;
#pragma omp parallel private(i,n)
{
  i = omp_get_thread_num();
  n = omp_get_num_threads();
  printf("I'm thread %d of %d\n", i, n);
}
```

Privatization issue: initial value

```
int i = 1;
#pragma omp parallel private(i)
printf("Value is %d\n", i + 1);
```

gives

```
$ OMP_NUM_THREADS=1 ./tmp
Value is 1
```

Why? Because private variables are **not** initialized by default in the parallel region. You should use *firstprivate*, which initialize to its value in the main thread, for that:

```
i = 1;
#pragma omp parallel firstprivate(i)
printf("Value is %d\n", i + 1);
```

which gives

```
$ OMP_NUM_THREADS=1 ./tmp
Value is 2
```

Second problem: communism IS usefull!

```
int sum = 0;
#pragma omp parallel for
for (int i=0; i < N; i++)
  sum += a[i]; // numbers from 1 to 8
printf("Sum is %d\n", sum);
```

This gives (sometimes!):

```
$ OMP_NUM_THREADS=1 ./tmp
Sum is 36
$ OMP_NUM_THREADS=2 ./tmp
Sum is 26
```

The problem is, again, data race. But using *private(sum)* will not help, since we want the overall result.

The solution: reduction

The solution is the *reduction(op:list)* clause. *op* is a reduction operator, which can be either arithmetic (*+ - * max min*) or logical (*& && | ||*).

```
#pragma omp parallel for reduction(+:sum)
for (int i=0; i < N; i++)
  sum += a[i]; // will compute a private sum
// then, it will `op` (here, add) the results of all threads
```

Could have been written with a critical[3] or atomic region instead:

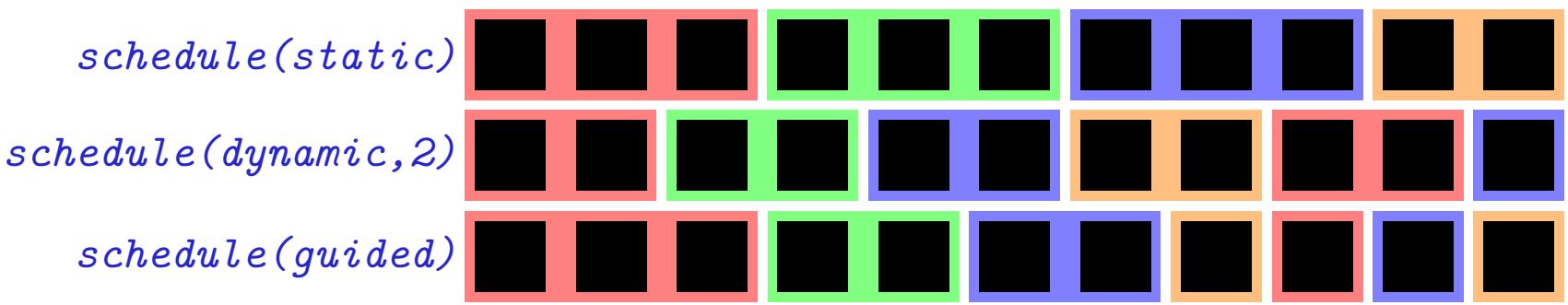
```
#pragma omp parallel for
for (int i=0; i < N; i++) {
  #pragma omp atomic
  sum += a[i];
}
```

... But you get an overhead due to synchronization!

[3] Equivalent to a mutex

Loop scheduling

- The *schedule(kind[,size])* clause specifies how iterations are divided into contiguous chunks (of a given *size*), and how these chunks are distributed to the threads. *kind* may be
  - *static* (default), iterations are divided into chunks, assigned to the threads. Each chunks contains the same number of iterations, except the last one.
  - *dynamic*, the chunks are requested then processed by the thread. Implies a little **overhead**.
  - *guided*, similar to *dynamic*, but the size of the chuncks depends on the number of remaining iteration.
- The two last are usefull when the runtime of an iteration is not constant in time: equal sharing of iteration $\neq$ equal workload.



Example: number of prime numbers

Computing $\pi(n)$, i.e., the number of primes less than or equals to n:

```
int n = 100000, not_prime = 2;
#pragma omp parallel for reduction(+:not_prime)
for(int i=2; i < n; i++) {
  for (int j=2; j < i; j++) { // large `i` requires more...
    if (i % j == 0) {
      not_prime++;
      break; // ... But iteration may be very short!
    }
  }
}
printf("Pi(%d) = %d\n", n, n - not_prime);
```

Here, scheduling is clearly interesting (but once again, the compiler cannot know that in advance)

Example: number of prime numbers

```
$ OMP_NUM_THREADS=4 ./pi
                          default      static     dynamic      guided
       n       pi(n)         time        time        time        time
-------------------------------------------------------------------
   65536        6542     0.38501     0.34878     0.21439     0.21869
  131072       12251     1.37137     1.46954     0.82317     0.81649
  262144       23000     4.95654     5.70368     3.97412     3.98117
  524288       43390    19.36867    19.65254    13.46914    12.13314
 1048576       82025    73.35597    76.84343    44.77434    44.33497
```

As expected, clear advantage of *dynamic* and *guided* over *static*.

Results (*axpy)

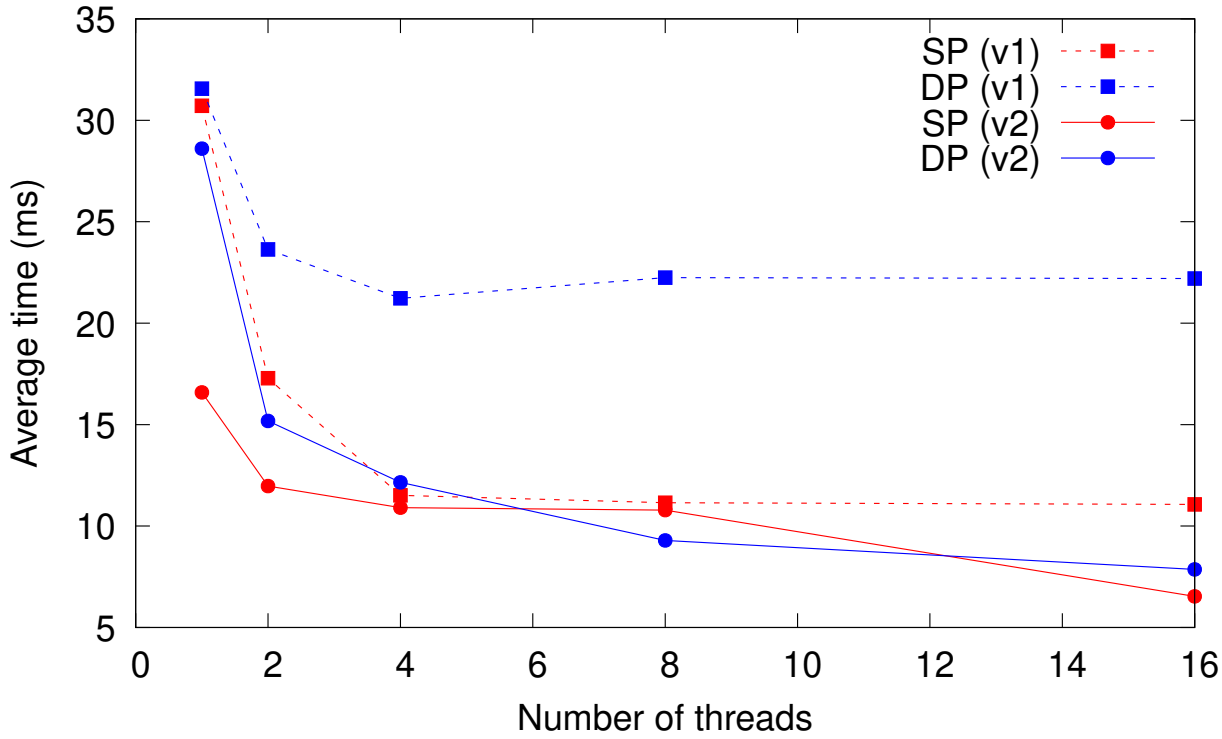


Figure: Results for *axpy* (2^{24} numbers) with *-N 1000* and different values of *OMP_NUM_THREADS*.

- SP faster than DP.
- Modest improvement for 8 and 16 threads (memory bandwidth).
- Second version adds SIMD and "first touch".

Results (*dot)

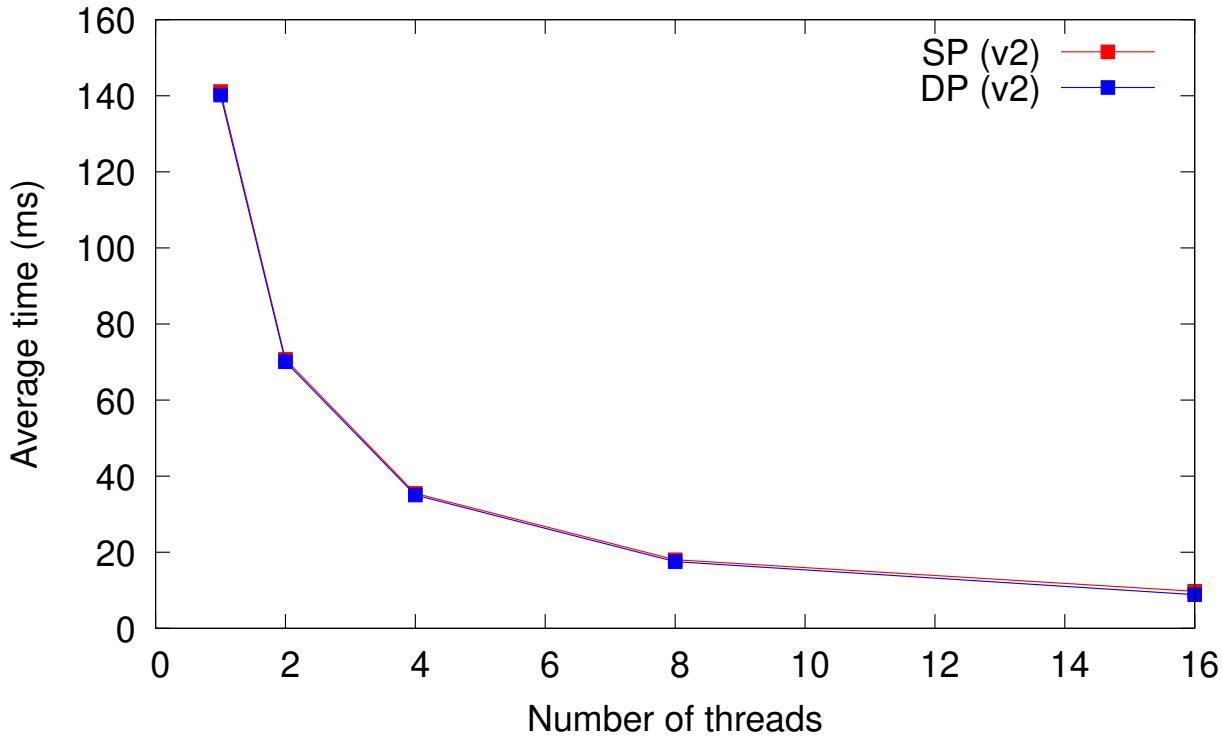


Figure: Results for *dot (2^{24} numbers) with -N 1000 and different values of OMP_NUM_THREADS.

- Not much difference between SP and DP.
- More work intensive loops $\rightarrow$ better speedup with a large number of threads.

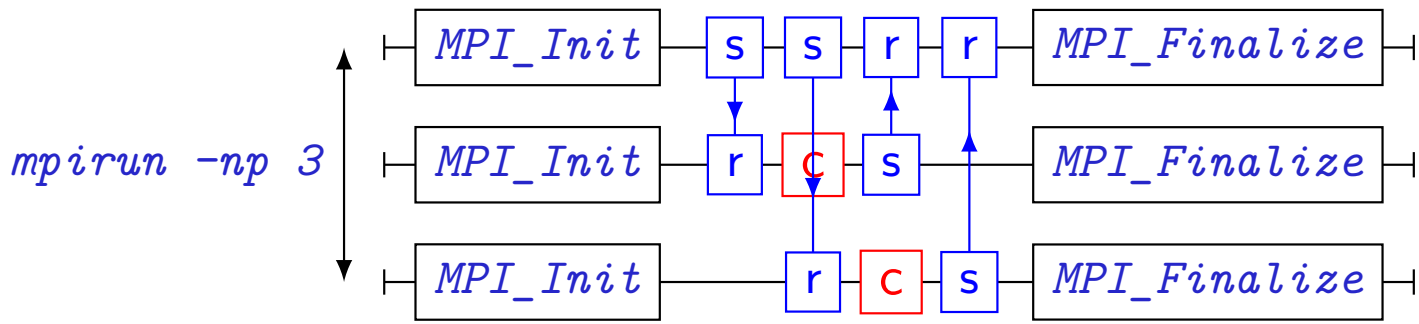
Going further

- Explicit synchronization (e.g., `critical`) ;
- Tasks (OpenMP 3.0) for better scheduling: allow parallelizing `while` loops and recursive call.
- Thread and memory management (NUMA, thread affinity, etc).

Harder, better, faster, stronger: enrolling other nodes with MPI

MPI?

- MPI = "message passing interface." It is a communication protocol for programming parallel computers. It defines a common and portable API.
- It provides different kinds of communication (point-to-point, collective, one-sided, ...), management, and parallel I/O.
- Consequence: sharing the workload is now **explicit** (and communication is a bottleneck)!
- The API is similar to the sockets (*i.e.*, *open* / *send* / *recv* / *close*), with some improvements:



- For maximal performances, MPI can be combined with OpenMP.

Hello, world!

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

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  int rank, size, len;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Get_processor_name(name, &len);
  printf("Rank=%d, size=%d (node=%s)\n", rank, size, name);
  MPI_Finalize();
  return 0;
}
```

Processes are grouped into **communicators**, *i.e.*, a group of processes that can communicate. In such group, each process is assigned a (unique) **rank**. Default is *MPI_COMM_WORLD* (all processes). The size of the communicator gives the number of processes in this communicator.

Compile and run

- MPI is generally not installed by default and comes in different flavors (implementations). You can install *openmpi* on your computer.
- There is a specific wrapper for the compilation *mpi[xx]* (which just add the relevant options), *e.g.*,

```
mpicc -o hello hello.c
```

- To use the application, you must use the launcher, *mpirun*, with the number of processes as argument:

```
mpirun -np 2 ./hello
```

- With the example given in previous slide, the output is

```
Rank=0, size=2 (node=raspi-cluster-master)
Rank=1, size=2 (node=raspi-cluster-master)
```

Sending

MPI passes data in the form of **messages**. They may be divided in two parts:

1. Message content: the data to be sent (MPI suppose it is array-like),
2. The "envelope": to which process the message is addressed.

$$\underbrace{\texttt{MPI\_Send(buff, count, datatype,}}_{\text{content}} \underbrace{\texttt{dest, tag, comm)}}_{\text{envelope}}$$

The message *tag* helps to discriminate between different messages addressed to the same process. It is user defined.

Sending

Sending data to a specific process may be done through one variant of

```
MPI_Send(
  void* buff,       // data
  int count,        // number of element to be sent
  MPI_Datatype dt, // type of an element
  int dest,         // recipient (rank)
  int tag,          // type of message
  MPI_Comm c,       // communicator
);
```

There is also *MPI_Isend*, which is **immediate** (non-blocking). *dt* may be

- *MPI_CHAR* / *MPI_UNSIGNED_CHAR* (or *MPI_BYTE*),
- *MPI_INT* / *MPI_UNSIGNED_INT*,
- *MPI_LONG* / *MPI_UNSIGNED_LONG*,
- *MPI_FLOAT* / *MPI_DOUBLE*,
- *MPI_PACKED* (*i.e.*, *struct*).

Receiving

Receiving data to a specific process may be done through one variant of

```
MPI_Recv(
  void* buff,       // data
  int count,        // number of element to be sent
  MPI_Datatype dt, // type of an element
  int source,       // sender (rank)
  int tag,          // type of message
  MPI_Comm c,       // communicator
  MPI_Status* s     // information on the message
);
```

Note that for a communication to succeed, the following should be met:

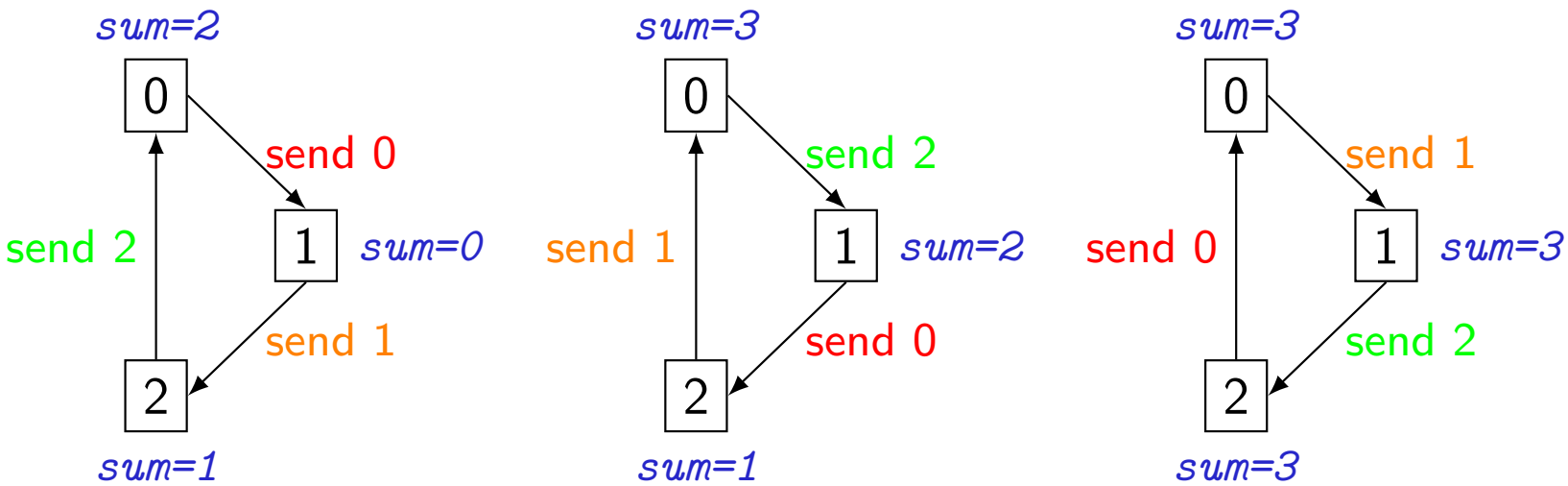
1. The same communicator must be used
2. Source and destination rank must match
3. The tag must match
4. The buffer should be large enough on the receive side.

Note that *count* is the **maximal** number of elements that can be received.

Example (ring communication)

Ring communication (sent to its next neighbor in the ring, receive from its previous). The goal is, *e.g.*, that every process should have the sum of the ranks at the end:

1. Set *sum = 0*, and *send_buffer = rank*,
2. Send the buffer to next neighbor,
3. Get from the previous in *receive_buffer*, adds it to *sum*,
4. *send_buffer = receive_buffer*,
5. Repeat from step 2 until all ranks have been propagated along the ring.



Example (synchronous)

```
send_buffer = rank;
for(int i=0; i < comm_size; i++) {
  if (rank % 2 == 0) { // WHY?
    MPI_Send(&send_buffer, 1, MPI_INT, next_rank, rank,
      MPI_COMM_WORLD);
    MPI_Recv(&receive_buffer, 1, MPI_INT, prev_rank, prev_rank,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(&receive_buffer, 1, MPI_INT, prev_rank, prev_rank,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&send_buffer, 1, MPI_INT, next_rank, rank,
      MPI_COMM_WORLD);
  }
  sum += receive_buffer;
  send_buffer = receive_buffer;
}
```

Example (asynchronous)

Both asynchronous call:

```
MPI_Request requests[2];
MPI_Isend(&send_buffer, 1, MPI_INT, next_rank, rank,
  MPI_COMM_WORLD, &requests[0]);
MPI_Irecv(&receive_buffer, 1, MPI_INT, prev_rank, prev_rank,
  MPI_COMM_WORLD, &requests[1]);
MPI_Waitall(2, requests, MPI_STATUS_IGNORE); // barrier
```

or asynchronous sent followed by synchronous receive:

```
MPI_Request request;
MPI_Isend(&send_buffer, 1, MPI_INT, next_rank, rank,
  MPI_COMM_WORLD, &request);
MPI_Recv(&receive_buffer, 1, MPI_INT, prev_rank, prev_rank,
  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Wait(&request, MPI_STATUS_IGNORE); // not required
```

Example: ping-pong

```
long N = 2 << i;
double* A = calloc(N, sizeof(double));
elapsed = MPI_Wtime();
for(int k=0; k < N_PING_PONG; k++) {
  if (rank == 0) {
    MPI_Send(A, N, MPI_DOUBLE, 1, 0,
      MPI_COMM_WORLD);
    MPI_Recv(A, N, MPI_DOUBLE, 1, 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(A, N, MPI_DOUBLE, 0, 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(A, N, MPI_DOUBLE, 0, 0,
      MPI_COMM_WORLD);
  }
}
elapsed = MPI_Wtime() - elapsed;
```

Elapsed time depends on the amount of data and asymptotically tend to the bandwith.

Example: ping-pong

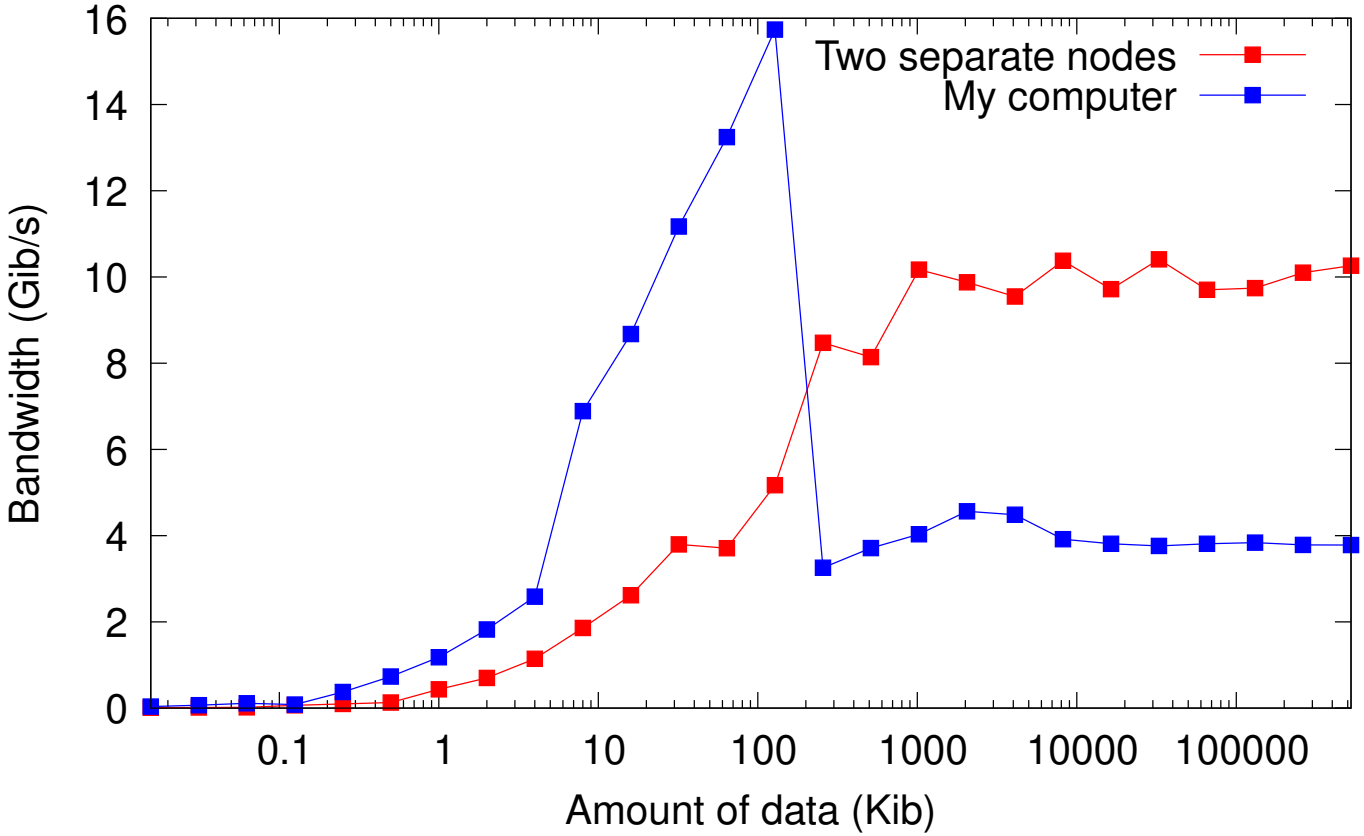
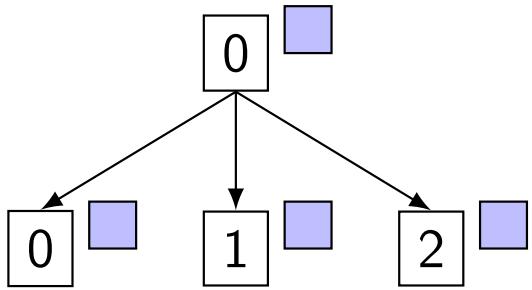


Figure: Bandwidth, as measured by the ping-pong example, in two different configurations

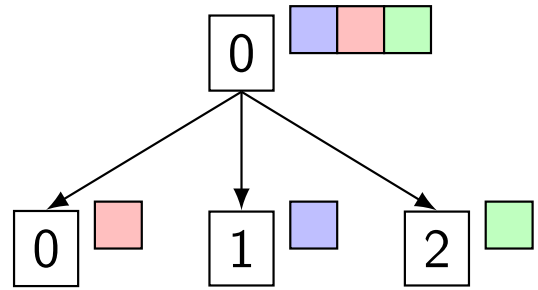
What (the hell!) did happen in both cases?

Other ways to communicate

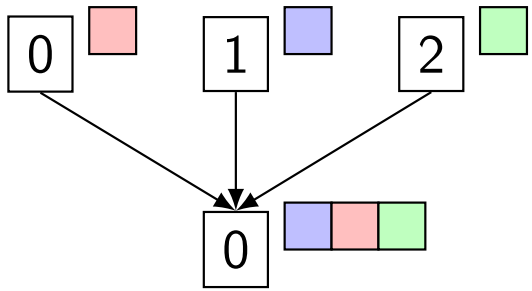
MPI_Bcast()



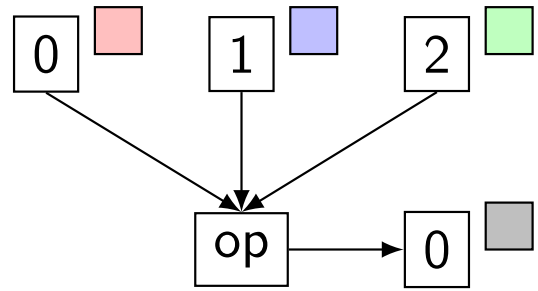
MPI_Scatter()



MPI_Gather()



MPI_Reduce()



... And others (asynchronous *I**, All-to-all *All**, vectors **v*, ...).

Results

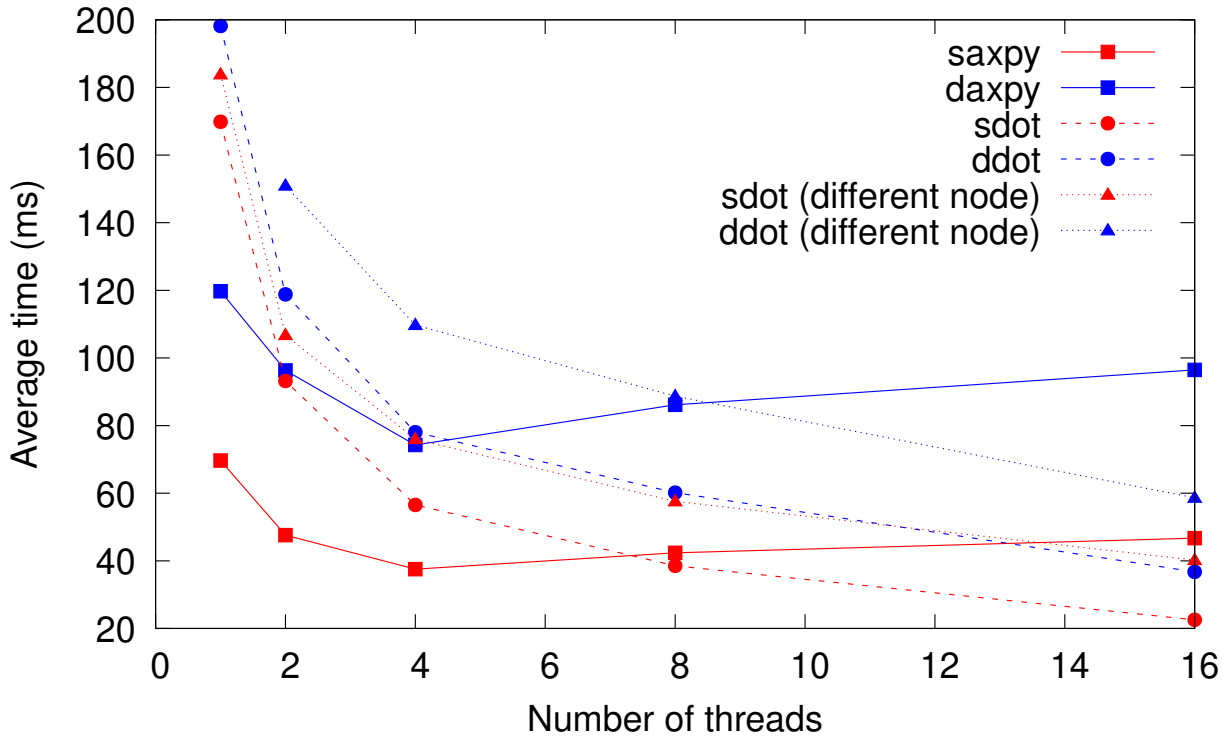


Figure: Results (2^{24} numbers) with `-N 1000` and different number of processes.

- Not so good in general.
- Again, more interesting for `*dot` (more work intensive) than for `*axpy` (bandwidth limited).
- Thanks to Infiniband, not so much impact when different nodes are used.

Next gen' stuffs: the GPUs

Why? How?

- CPU and GPU actually took different paths:
  - A CPU is very general purpose, with a complex set of operation (and prediction / speculative execution) and high clock rate.
  - A GPU is designed for *data stream*, by having a whole bunch of ALU, designed for bulk data handling. Thanks to Joule's law, the clock rate has to be smaller.
- There are different approaches to program with GPUs:
  1. Specific: NVIDIA CUDA, AMD HIP,
  2. Explicit cross-platform: OpenCL,
  3. Directive based: OpenACC, **OpenMP**.
- Though it is generally less difficult to manage (there is generally one device), communication is still the issue.
- Of course, I will use OpenMP (which actually uses CUDA/HIP as back-end). It works with `gcc`, but you will probably get better performances with `clang`.

A bit of (confusing) terminology

OpenMP	NVIDIA	AMD
SIMD Lane	Thread	Work item
Thread	Warp	Wavefront
Team	Thread block	Workgroup
League	Grid	??

- Each "worker" control SIMD lanes of size L_v, (each one of them being generally referred to as a *thread*, executed by a *core*)
- the N_w workers are grouped into a **team**,[4] (thus containing $L_v \times N_w$ cores). On CPU, there was only one team.
- There are up to N_t teams that work independently. This is where new instructions are needed.

[4] This is the OpenMP terminology

Offload and distribute

- To offload code to GPU, use *omp target*.
- To create a collection (league) of teams, use *omp teams*. As usual, parallel $\neq$ worksharing.
- To distribute iterations over the teams, use *omp distribute*.
- Combine that with the usual instruction to further distribute over threads.

```
#pragma omp target
{
  // distributed over all thread of all teams:
  #pragma omp teams distribute parallel for
  for(int i=0; i < N; i++)
    x[i] = 2.0 * x[i];
}
```

Actually, this code may not work...

Data management

OpenMP (in some implementation[5]) needs to know which data to move to and from the GPU. Use the *map* clause:

- *map(to: list)*: copy data to GPU,
- *map(from: list)*: copy data from GPU,
- *map(tofrom: list)*: copy data to and from the GPU,
- *map(alloc: list)*: directly allocate data on the GPU.

```
#pragma omp target map(tofrom: x[0:N])
{
  #pragma omp teams distribute parallel for
  for(int i=0; i < N; i++)
    x[i] = AX * x[i];
}
```

Notice that for an array, we have to give the number of data (since it may be a dynamic allocation). Use *target data* for permanent data management.

[5] At least GCC 10

Results

	*dot		*axpy	
	s	d	s	d
Serial (-O1)	141.1	140.1	30.7	31.7
OMP CPU (4)	35.5	35.0	10.9	12.1
OMP GPU	47.0	54.8	52.3	75.6

Table: Average running times (in ms) with `-N 1000`, on 2^{24} numbers.

- Compiled with `gcc -o xxx xxx.c -O1 -lm -fopenmp -foffload=-misa=sm_35`, with CUDA 11.1.
- Ran on NVIDIA RTX 2080 (should be `sm_75`!).
- And, well, that's awful... Again "thanks" to data transfer.

But wait, there is more! Introducing... Laplace.

- Laplace's equation: $\nabla^2 U = 0$. Used to model the diffusion phenomenons: U is unknown.
- A very common approximation is the iterative procedure based on the (Jacobi) 4-point stencil. One a grid with evenly spaced (of h) points, it reduces to

$$U^{k+1}(x, y) = \frac{U^k(x \pm h, y) + U^k(x, y \pm h)}{4}.$$

  where k is the iteration number.
- The values at the borders are fixed (Dirichlet)
- Excellent for GPU (or MPI): after upload, the data remains and are updated.

Figure: Visualization of U after 10 000 iterations on a 512x512 grid ($h = 1$).

Results

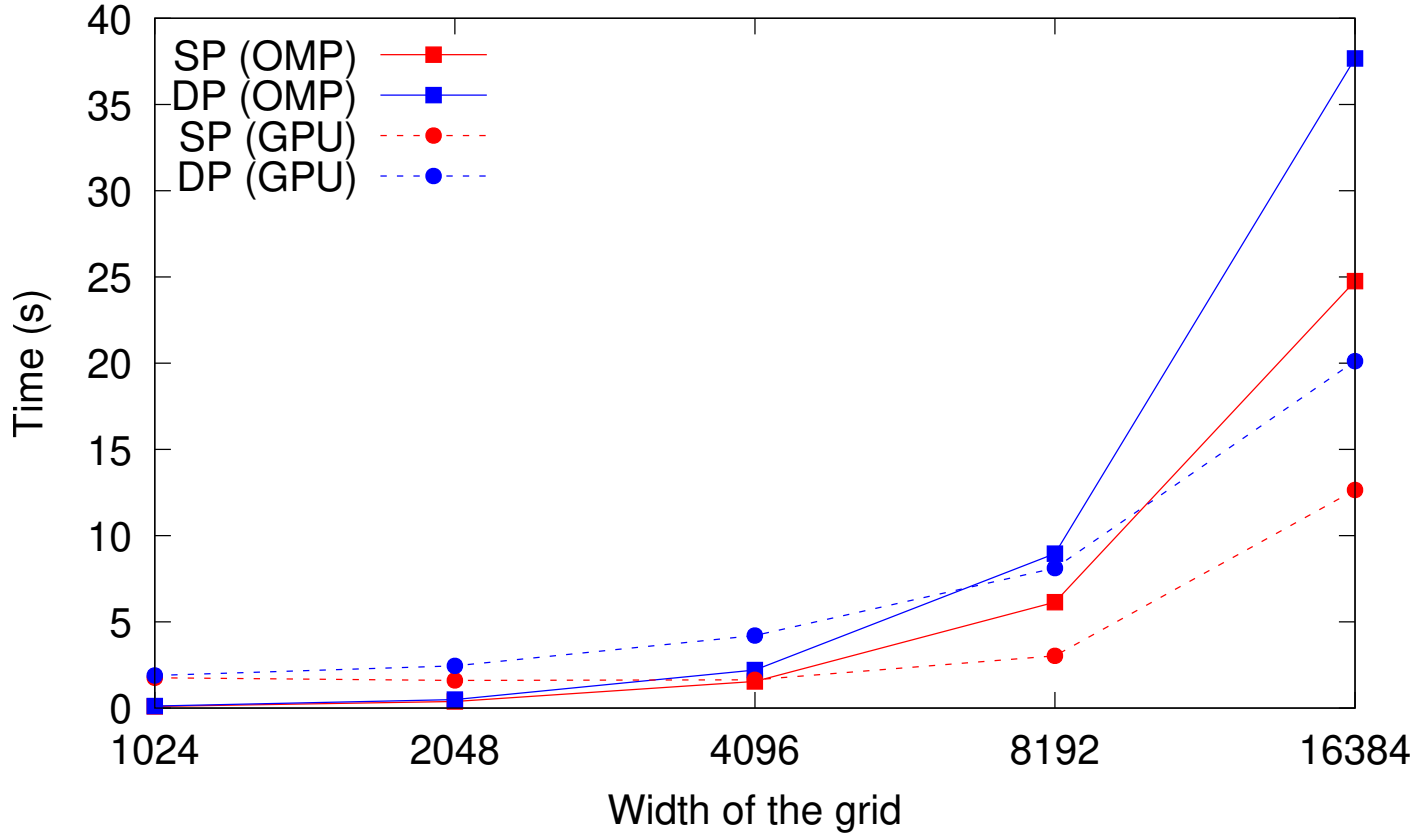


Figure: Results for the first 200 iteration with the size of the grid of the Laplace procedure on a CPU (with *OMP_NUM_THREADS=4*) and a GPU.

Now that gets interesting!

The end of the beginning

To conclude

- Again, optimization is an art! My goal was actually twofold:
  1. Convince you that compiler knows better than you, and
  2. Convince you that the directive approach allows you to gradually improve performance without (much) of rewrite.
- There is a clear trend towards GPUs, so the support will improve in the future (and a standard will probably emerge[6]). You just need to avoid data transfer as much as possible.
- This was just an introduction, so take a look around! The CÉCI is organizing training if you are interested to go further (`http://www.ceci-hpc.be/training.html`), and there is a student cluster at UNamur (called Hyades) if you want to play (in the frame of a lecture, though).
- If you really want to explore the topic, keep in mind that other languages also have concurrent tasks directly available as part of the language...

[6] It may already be there, with all libraries copying the CUDA style.