We start @9:05

MPI and OpenMP in Scientific Software Development

Day 3

Maxim Masterov
SURF, Amsterdam, The Netherlands
27-29.05.2024

Contents

Overview

Day 1

- **@16:00 Dr. Matthias Möller (TU Delft)**
- MPI (recap)
- Connecting to Snellius supercomputer
- Jacobi solver
- Performance analysis & debugging tools
- Hands-on

Day 2

- **@12:00 Dr. Nicola Spallanzani (MaX CoE)**
- MPI communications
- Domain decomposition
- MPI topologies
- Hands-on

Day 3

- **@12:00 Dr. Remco Havenith (RUG)**
- MPI IO
- Advanced OpenMP
- Hybrid programming
- Hands-on

Contents

Overview

Day 1

- **@16:00 Dr. Matthias Möller (TU Delft)**
- MPI (recap)
- Connecting to Snellius supercomputer
- Jacobi solver
- Performance analysis & debugging tools
- Hands-on

Day 2

- **@12:00 Dr. Nicola Spallanzani (MaX CoE)**
- MPI communications
- Domain decomposition
- MPI topologies
- Hands-on

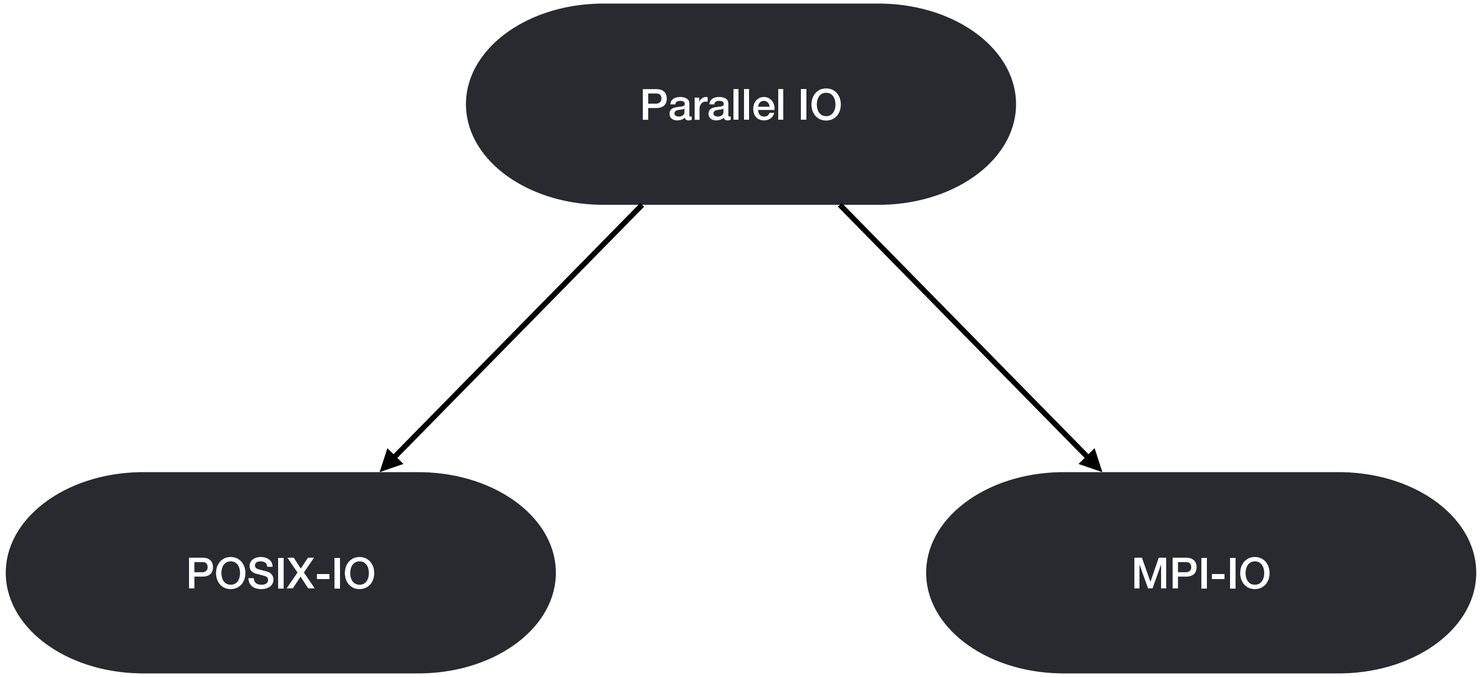
Day 3

- **@12:00 Dr. Remco Havenith (RUG)**
- MPI IO
- Advanced OpenMP
- Hybrid programming
- Hands-on

MPI IO

MPI IO

Parallel IO

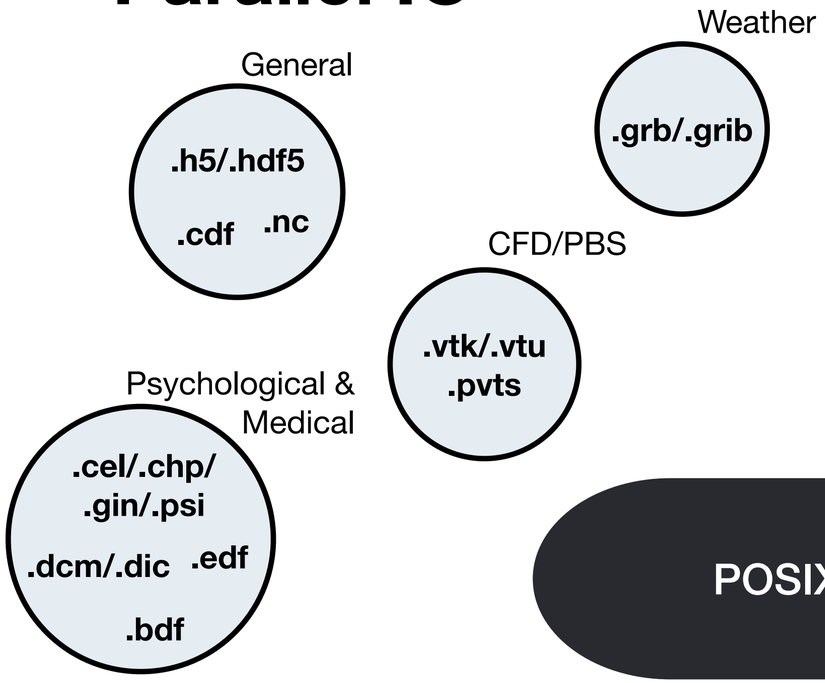


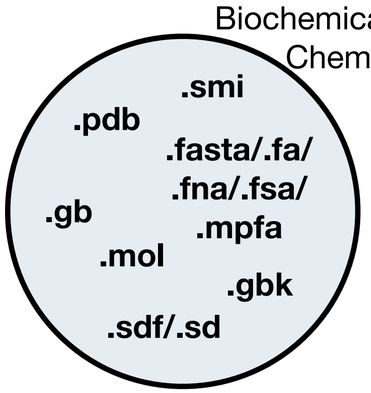
- The root process writes all data
- Each process writes a single independent file

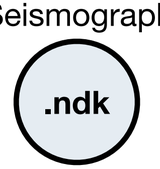
- Each process writes into the same single file

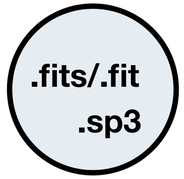
MPI IO

Parallel IO









- **Performance**
- **Scalability**
- **Robustness**

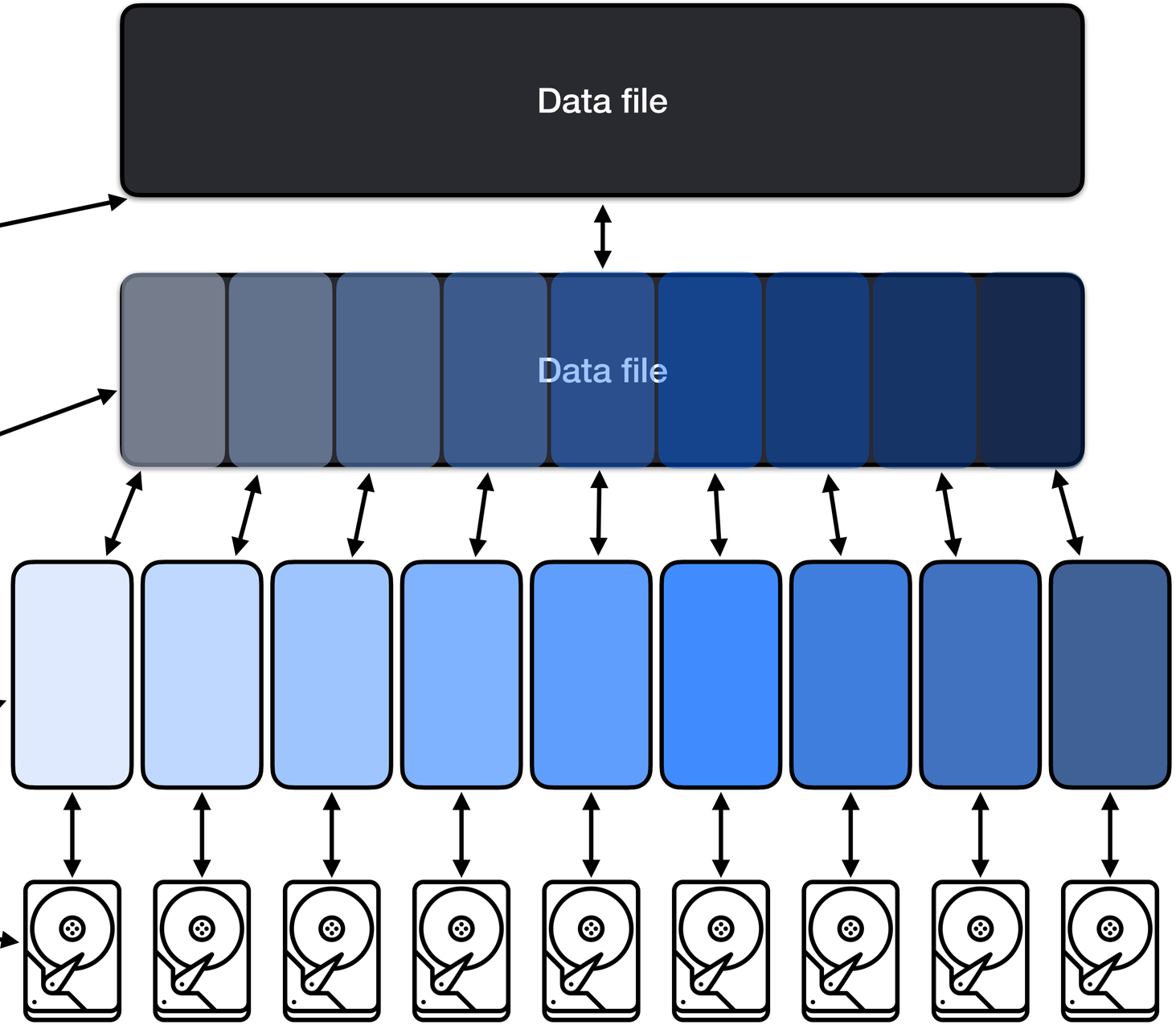
MPI IO

Parallel File Systems

One single file

FS automatically divides this file into stripes

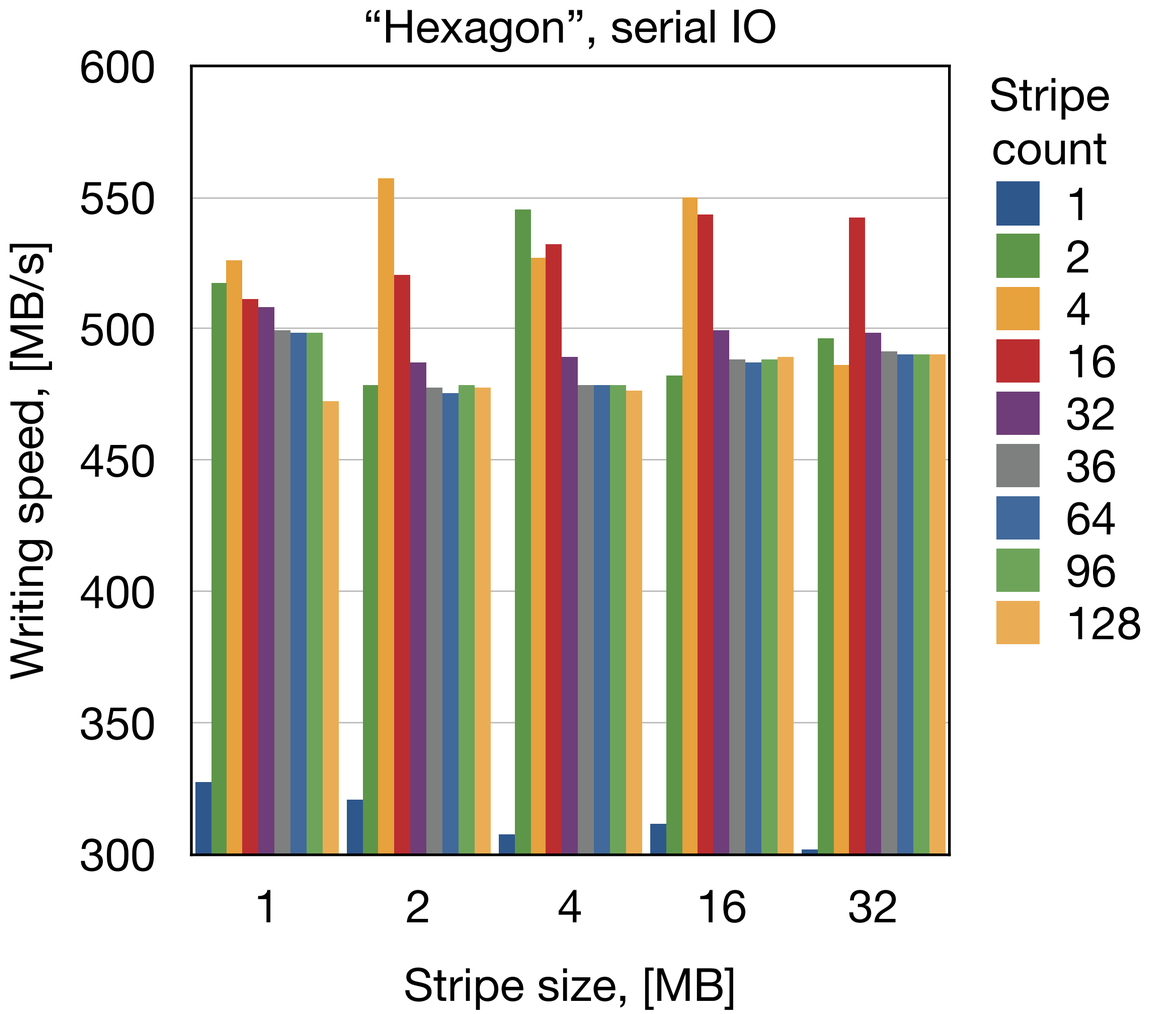
IO operation is split across multiple drives

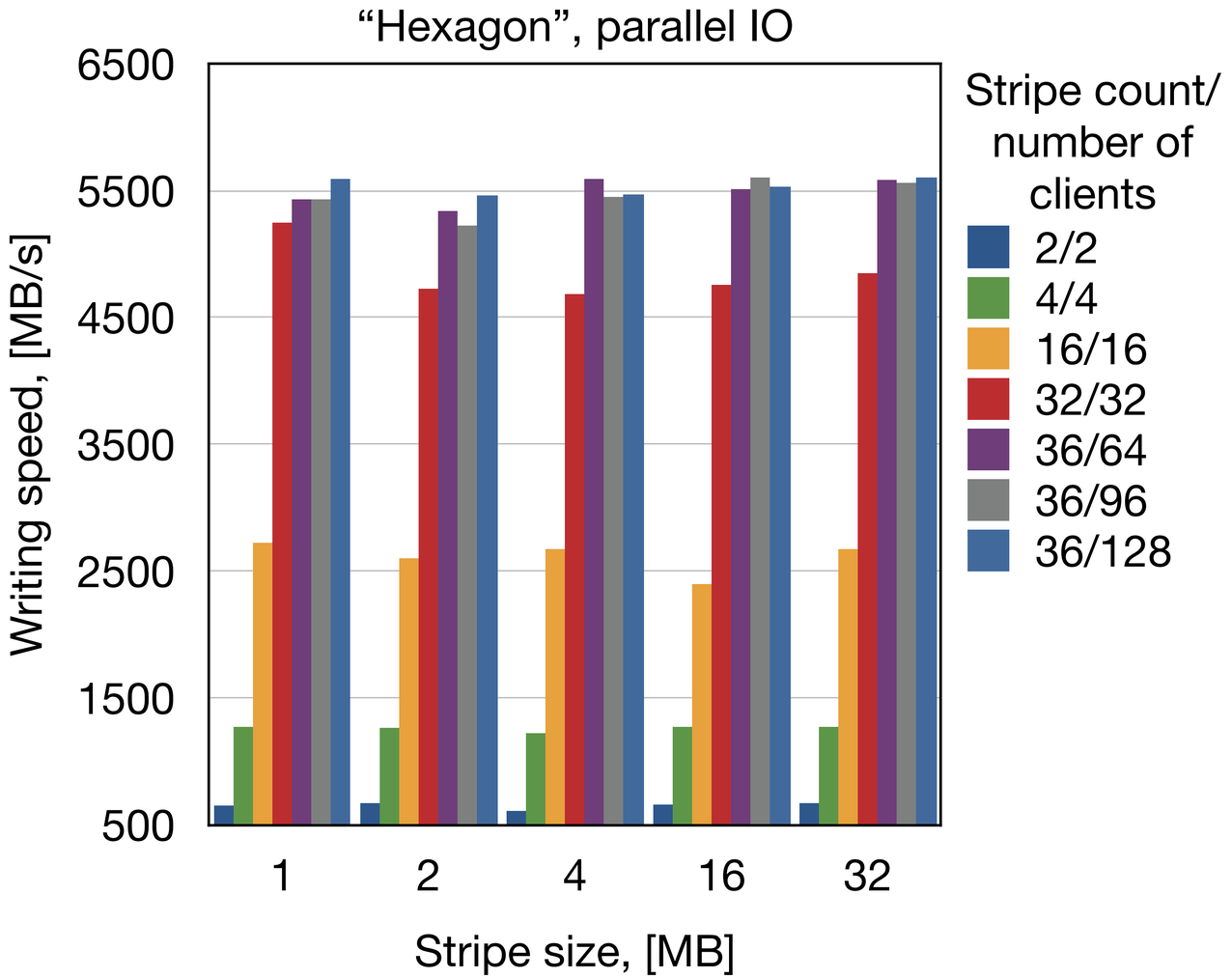


1. https://www.flaticon.com/free-icon/hard-disk-drive_2004609

MPI IO

File Systems Performance



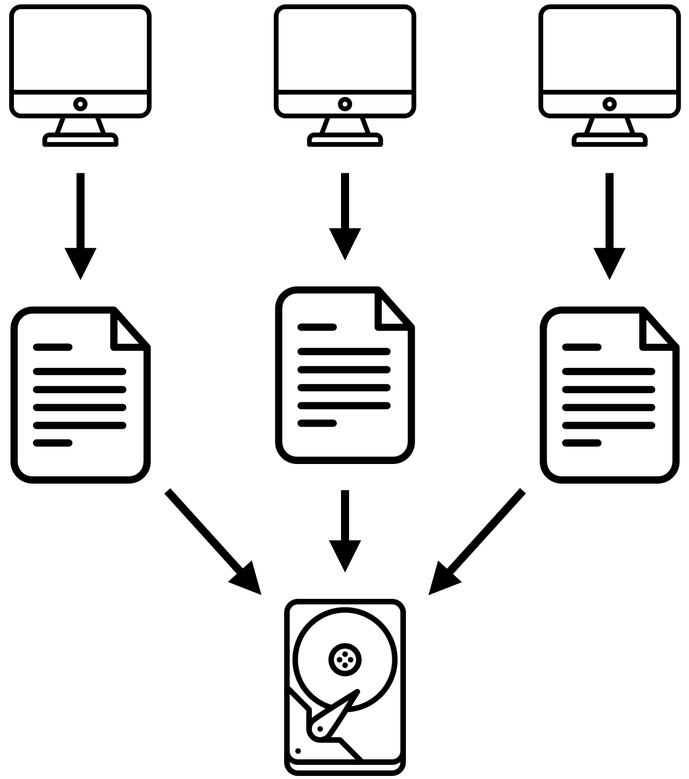


Data is taken from https://hpc-uit.readthedocs.io/en/latest/storage/lustre-performance.html

MPI IO

Independent files

- Each process performs IO operations on independent files concurrently
- Very simple implementation
- **Limited by the FS**
- **Does not scale (see previous point)**



C++

```cpp
std::ofstream out;
out.open(file_name);

for(...) { out << ... << “\n"; }

out.close();
```

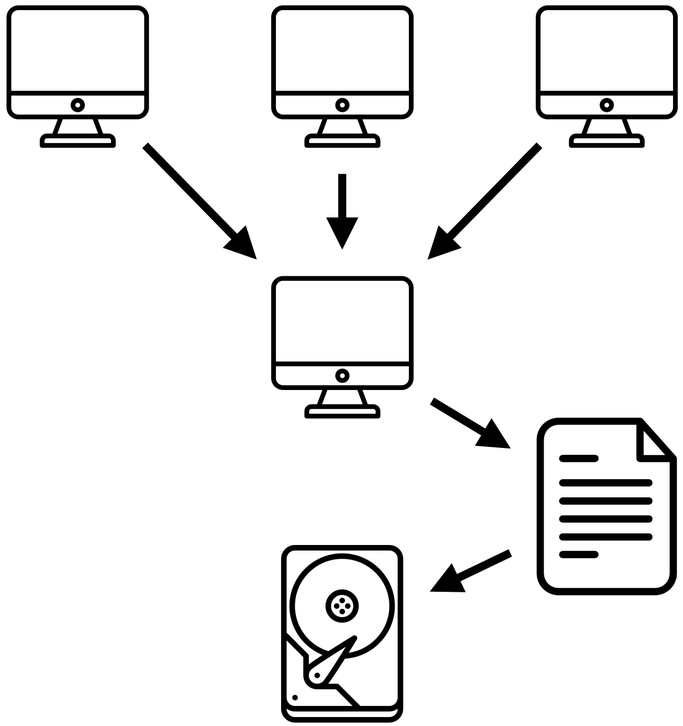
FORTRAN

```fortran
open(fh, file = file_name, status = 'new')

do i=1,100
    write(fh,*) …
end do

close(fh)
```

1. https://www.flaticon.com/free-icon/hard-disk-drive_2004609
2. https://www.flaticon.com/free-icon/computer_2344269
3. https://www.flaticon.com/free-icon/file_607674

MPI IO

Governing-process approach



- All IO operations are performed by the “root” process using POSIX or MPI library
- Data gathering is required (**MPI_Gather()**, **MPI_Gatherv()**)
- Relatively simple implementation
- **Does not scale (serial)**

```
int MPI_File_open(MPI_Comm comm,                // communicator
                  const char *filename,         // name of file to open
                  int amode,                    // file access mode
                  MPI_Info info,                // info object
                  MPI_File *fh);                // file handle

int MPI_File_close(MPI_File *fh);               // file handle
```

- MPI_MODE_RDONLY – read only
- MPI_MODE_RDWR – reading and writing
- MPI_MODE_WRONLY – write only
- MPI_MODE_CREATE – create the file if it does not exist
- MPI_MODE_EXCL – error if creating file that already exists
- MPI_MODE_DELETE_ON_CLOSE – delete file on close
- MPI_MODE_UNIQUE_OPEN – file will not be concurrently opened elsewhere
- MPI_MODE_SEQUENTIAL – file will only be accessed sequentially
- MPI_MODE_APPEND – set initial position of all file pointers to end of file

1. https://www.flaticon.com/free-icon/hard-disk-drive_2004609
2. https://www.flaticon.com/free-icon/computer_2344269
3. https://www.flaticon.com/free-icon/file_607674

MPI IO

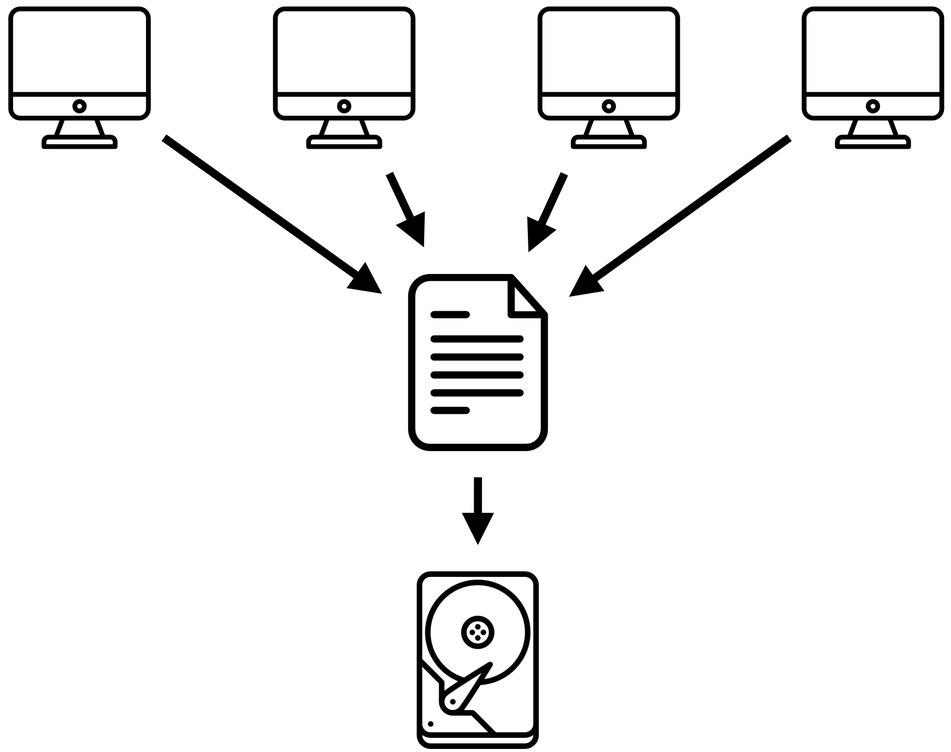
Shared file

```
int MPI_Gatherv(const void *sendbuf,          // starting address of send buffer
                int sendcount,                // number of elements in send buffer
                MPI_Datatype sendtype,        // data type of send buffer elements
                void *recvbuf,                // address of receive buffer
                const int recvcounts[],       // integer array (of length group size) containing the number of
                                              // elements that are received from each process (significant only
                                              // at root)
                const int displs[],           // integer array (of length group size). Entry i specifies the
                                              // displacement relative to recvbuf at which to place the incoming
                                              // data
                                              // from process i (significant only at root)
                MPI_Datatype recvtype,        // data type of recv buffer elements (significant only at root)
                int root,                     // rank of receiving process
                MPI_Comm comm);               // communicator

int MPI_Gather(const void *sendbuf,           // starting address of send buffer
               int sendcount,                 // number of elements in send buffer
               MPI_Datatype sendtype,         // data type of send buffer elements
               void *recvbuf,                 // number of elements for any single receive (integer, significant
                                              // only at root)
               int recvcount,                 // number of elements for any single receive (integer, significant only at root)
               MPI_Datatype recvtype,         // data type of recv buffer elements (significant only at root)
               int root,                      // rank of receiving process
               MPI_Comm comm);                // communicator
```

MPI IO

Shared file

- Each process performs IO operations with one **shared** file
- Implementation complexity depends on the data structure
- **Scalability depends on**
  - **implementation**
  - **file system**



1. https://www.flaticon.com/free-icon/hard-disk-drive_2004609
2. https://www.flaticon.com/free-icon/computer_2344269
3. https://www.flaticon.com/free-icon/file_607674

MPI IO

Definitions

file

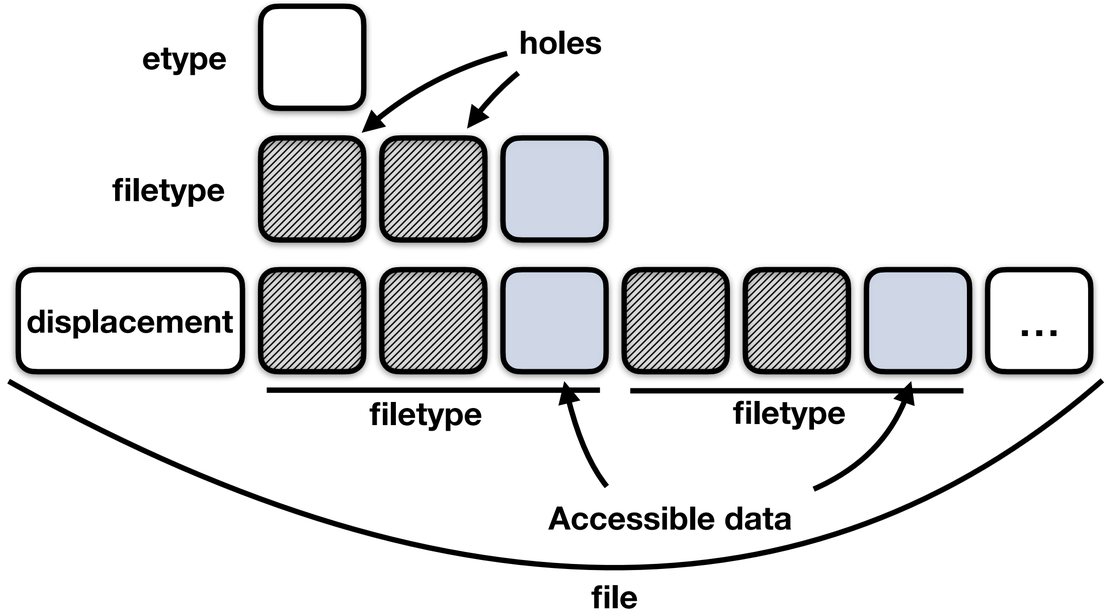
- (MPI file) is an ordered collection of typed data items

displacement

- An absolute byte position relative to the beginning of a file

etype (elementary datatype)

- is the unit of data access and positioning / offsets
- can be any MPI predefined or derived datatype
- typically the same at all processes



filetype

- the basis for partitioning a file among processes
- defines a template for accessing the file
- either a single **etype** or a derived MPI datatype constructed from multiple instances of the same **etype**
- different at each process

MPI IO

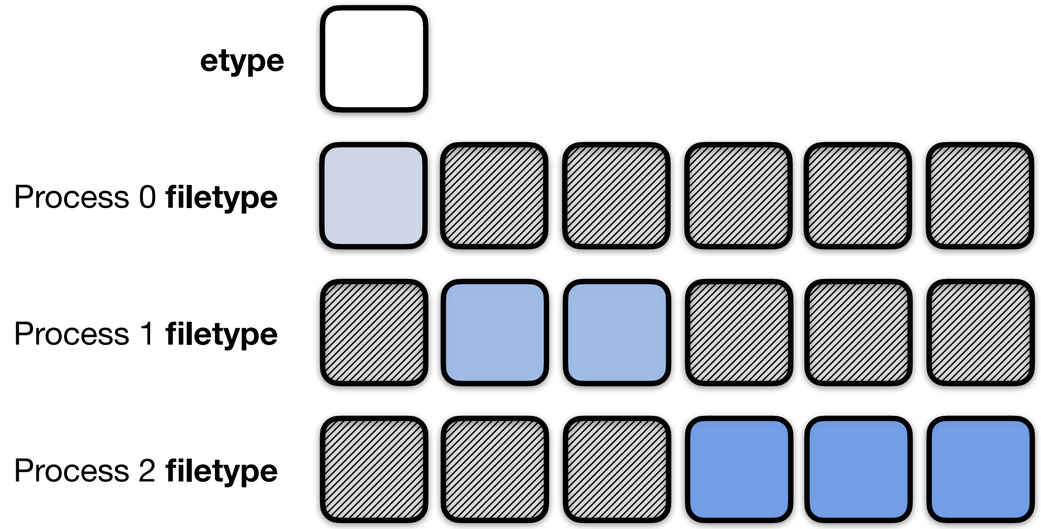
Definitions

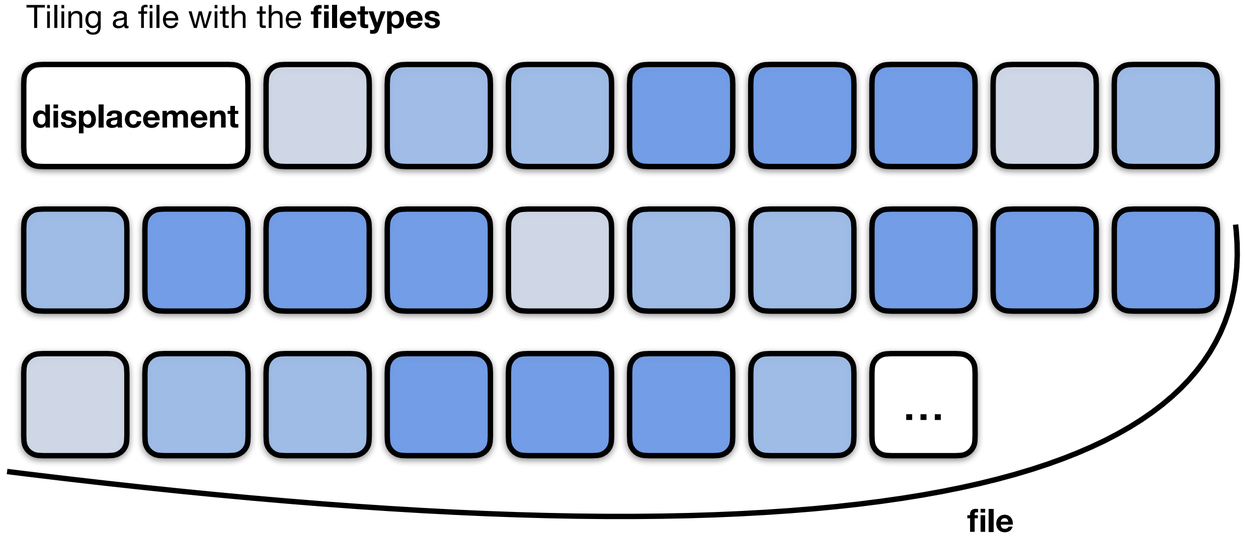
view

- defines the current set of data visible and accessible from an open file as an ordered set of **etypes**
- each process has its own view, defined by: a **displacement**, an **etype**, and a **filetype**
- the pattern described by a **filetype** is repeated, beginning at the **displacement**

offset

- a position in the file relative to the current view, expressed as a count of **etype**
- holes are skipped when calculating this position





MPI IO

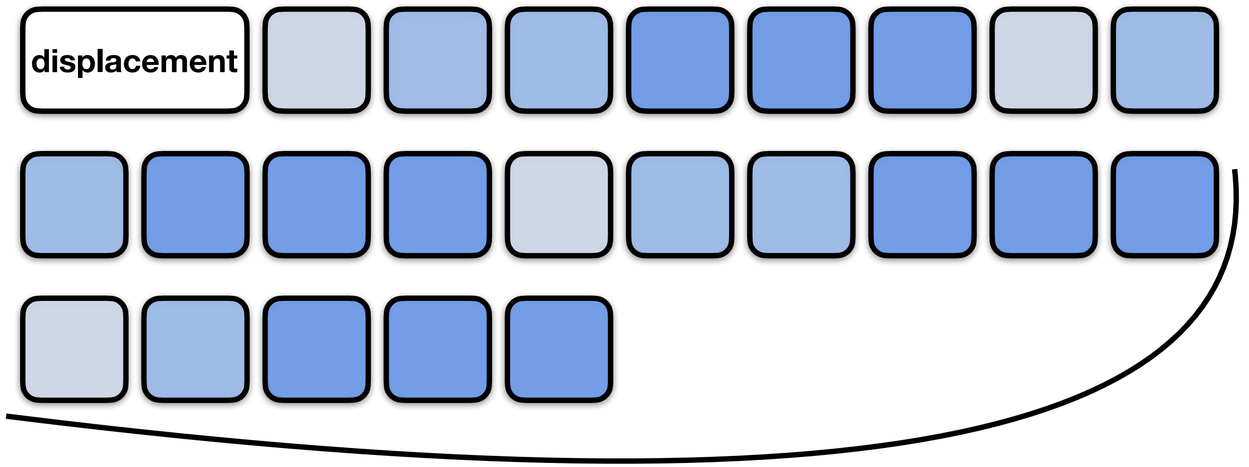
Definitions

file size and end of file

- the size of an MPI file, measured in bytes from the beginning of the file
- for any given **view**, the **end of file** is the offset of the first **etype** accessible in the current **view** starting after the last byte in the file

file pointer

- an implicit offset maintained by MPI
- individual **file pointers** are local to each process
- shared **file pointers** are shared by the group of processes



file size = sizeof(**displacement**) + sizeof(**view**), [B]

file handle

- an opaque object
- created by MPI_FILE_OPEN
- freed by MPI_FILE_CLOSE

MPI IO

Write to file

```c
/* Sets the file view */
int MPI_File_set_view(MPI_File fh,              // file handle
                      MPI_Offset disp,          // displacement
                      MPI_Datatype etype,       // elementary data type
                      MPI_Datatype filetype,    // file type
                      const char *datarep,      // data representation
                      MPI_Info info);           // info object

/* Write using individual file pointer */
int MPI_File_write(MPI_File fh,                 // file handle
                   const void *buf,             // initial address of buffer
                   int count,                   // number of elements in buffer
                   MPI_Datatype datatype,       // datatype of each buffer element
                   MPI_Status *status);         // status object

/* Collective write using individual file pointer */
int MPI_File_write_all(MPI_File fh,             // file handle
                       const void *buf,         // initial address of buffer
                       int count,               // number of elements in buffer
                       MPI_Datatype datatype,   // datatype of each buffer element
                       MPI_Status *status);     // status object
```

- Changes the process's view of the data in the file

- Will invoke multiple write/seeks
- May lock the file for other processes

- MPI library has a “global view” of the file
- Each process knows in advance where to write (collective IO)

MPI IO

Read from file

```c
/* Updates the individual file pointer */
int MPI_File_seek(MPI_File fh,                  // file handle
                  MPI_Offset offset,            // file offset
                  int whence);                  // update mode

/* Read using individual file pointer */
int MPI_File_read(MPI_File fh,                  // file handle
                  void *buf,                    // initial address of buffer
                  int count,                    // number of elements in buffer
                  MPI_Datatype datatype,        // datatype of each buffer element
                  MPI_Status *status);          // status object

/* Collective read using individual file pointer */
int MPI_File_read_all(MPI_File fh,              // file handle
                      void *buf,                // initial address of buffer
                      int count,                // number of elements in buffer
                      MPI_Datatype datatype,    // datatype of each buffer element
                      MPI_Status *status);      // status object
```

- MPI_SEEK_SET – the pointer is set to offset
- MPI_SEEK_CUR – the pointer is set to the current pointer position plus offset
- MPI_SEEK_END – the pointer is set to the end of the file plus offset

MPI IO

Complex data types

- One can create a **derived data types** and use collective operations in order **to simplify IO** operations.

  For an arbitrary 2D array:

```
int arr[nx_glob][ny_glob];                                      // declare a 2D array
// use the array…
int dims = 2;                                                   // dimensions of the array
int glob_sizes[dims] = {nx_glob, ny_glob};                      // global sizes of the array
int loc_sizes[dims] = {nx_loc, ny_loc};                         // local sizes of the array
int start_inds[dims] = {beg_glob_ind_x, beg_glob_ind_y};        // starting indices in the global view
MPI_Datatype subarray_type;                                     // the name of the new data type

MPI_Type_create_subarray(dims,              // number of dimensions
                         glob_sizes,        // number of elements of type oldtype in each dimension of the full array
                         loc_sizes,         // number of elements of type oldtype in each dimension of the subarray
                         start_inds,        // starting coordinates of the subarray in each dimension
                         MPI_ORDER_C,       // array storage order flag
                         MPI_INT,           // array element datatype
                         &subarray_type); // new datatype
MPI_Type_commit(&subarray_type);

MPI_Send(arr, 1, subarray_type, …);
```

MPI IO

Complex data types

```
/* Create a datatype for a subarray of a regular, multidimensional array */
int MPI_Type_create_subarray(int ndims,                  // number of array dimensions
                             const int size_array[],     // number of elements of type oldtype in each dimension
                                                         // of the full array
                             const int subsize_array[],  // number of elements of type oldtype in each dimension
                                                         // of the subarray
                             const int start_array[],    // starting coordinates of the subarray in each dimension
                             int order,                  // array storage order flag
                             MPI_Datatype oldtype,       // array element datatype
                             MPI_Datatype *newtype);     // new datatype

/* Commits the datatype */
int MPI_Type_commit(MPI_Datatype *type);                 // datatype
```

Hands-on #3.1

Hands-on

MPI-IO

- Implement in the Jacobi code:
  - MPI-write using all-to-one communication in ***IO/io.cpp:writeByRoot()***
    - Use **MPI_Gather()** and **MPI_Gatherv()** to assemble the temperature field and the grid on the root process
    - Use **MPI_File_write()** to write all the data into one file
    - Visualise the output using ***gnuplot:***

```
$ cp PostProcess/plot_mpi.plt .
$ gnuplot
gnuplot> load “plot_mpi.plt"
```

We continue @11:00

Hands-on

MPI-IO

- Implement in the Jacobi code:
  - MPI-write using a shared file and derived subarray datatype, see ***IO/io.cpp:writeByAll()***
    - Create a new data type only for the temperature field using **MPI_Type_create_subarray()**
    - Use **MPI_File_write_all()** and **MPI_File_write()** (compare the performance)
    - HINT: reduce the number of iterations (e.g. to 100), increase the grid size (e.g. to 100x100), use 8 processes
    - Visualise the output using ***gnuplot:***

```
gnuplot> set pm3d map
gnuplot> splot "output.dat" bin array=10x10 format='%lf' w image
```

Hands-on

MPI-IO

- Implement in the Jacobi code:
  - MPI-write using a shared file and derived structure and subarray datatypes, see ***IO/io.cpp:writeByAll()***
    - Create a new data type for the structure of coordinates and temperature field using **MPI_Type_create_struct()**
    - Provide new data type as an argument to **MPI_Type_create_subarray()**
    - Set the file view using the created types of structure and subarray
    - Use **MPI_File_write_all()** and **MPI_File_write()** (compare the performance)
    - Visualise the output using ***gnuplot:***

```
$ cp PostProcess/plot_mpi.plt .
$ gnuplot
gnuplot> load “plot_mpi.plt"
```

@12:00
Dr. Remco Havenith, RUG
“Accelerated Chemistry”

We continue @14:00

OpenMP-1

OpenMP
Basics

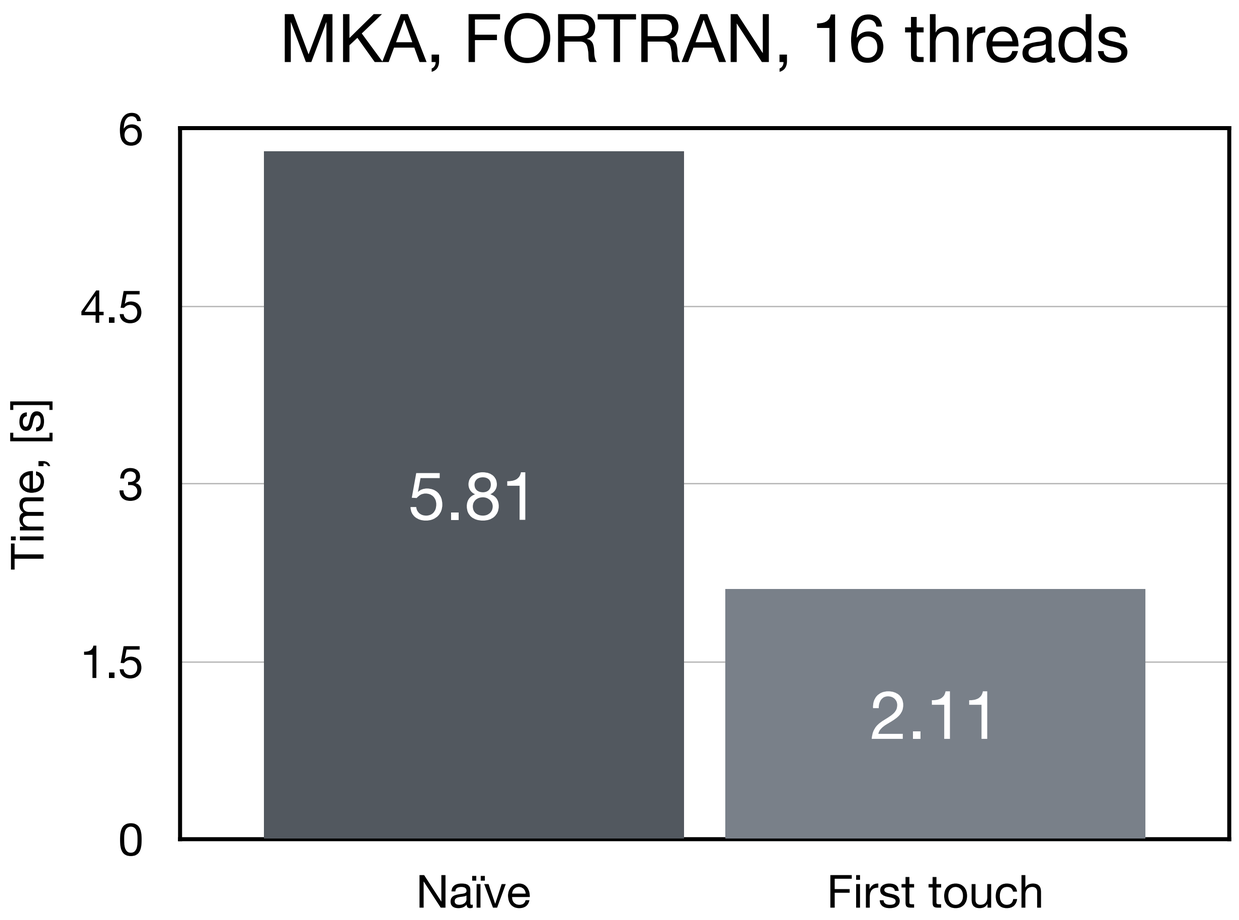
- Different implementations/vendors provide different level of support
- Officially, 21 compilers implement the OpenMP API[1]
- The most popular: **GNU**, **LLVM**, **Intel**

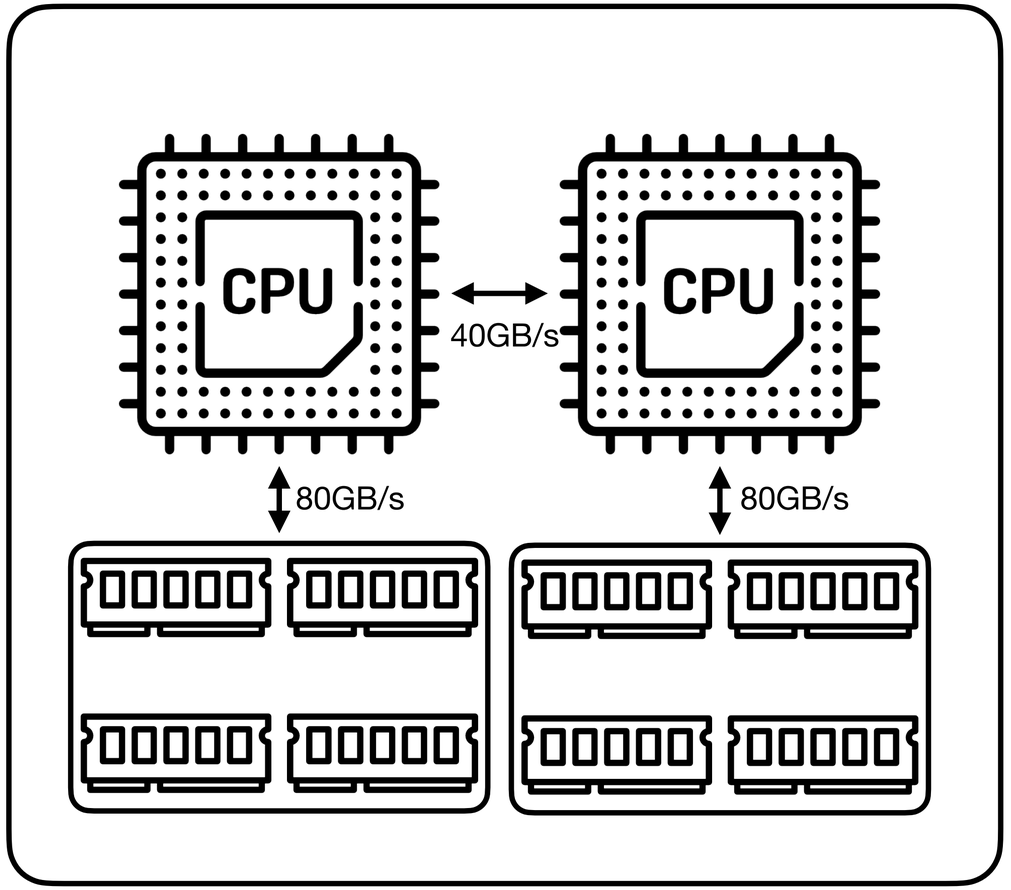
```
#pragma omp parallel for
for (int n = 0; n < N; ++n) {
    compute(…);
}
```

1. https://www.openmp.org/resources/openmp-compilers-tools/

OpenMP

Memory placement

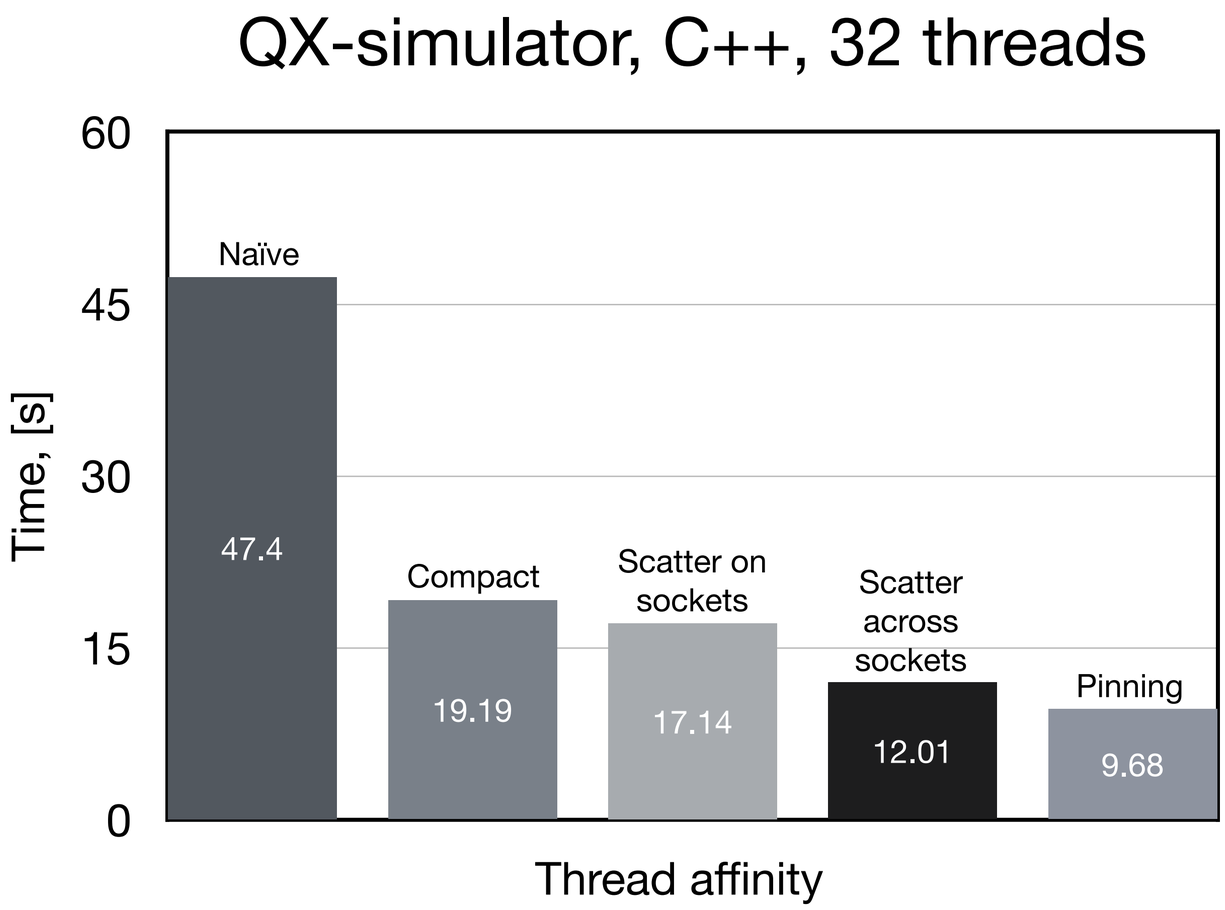


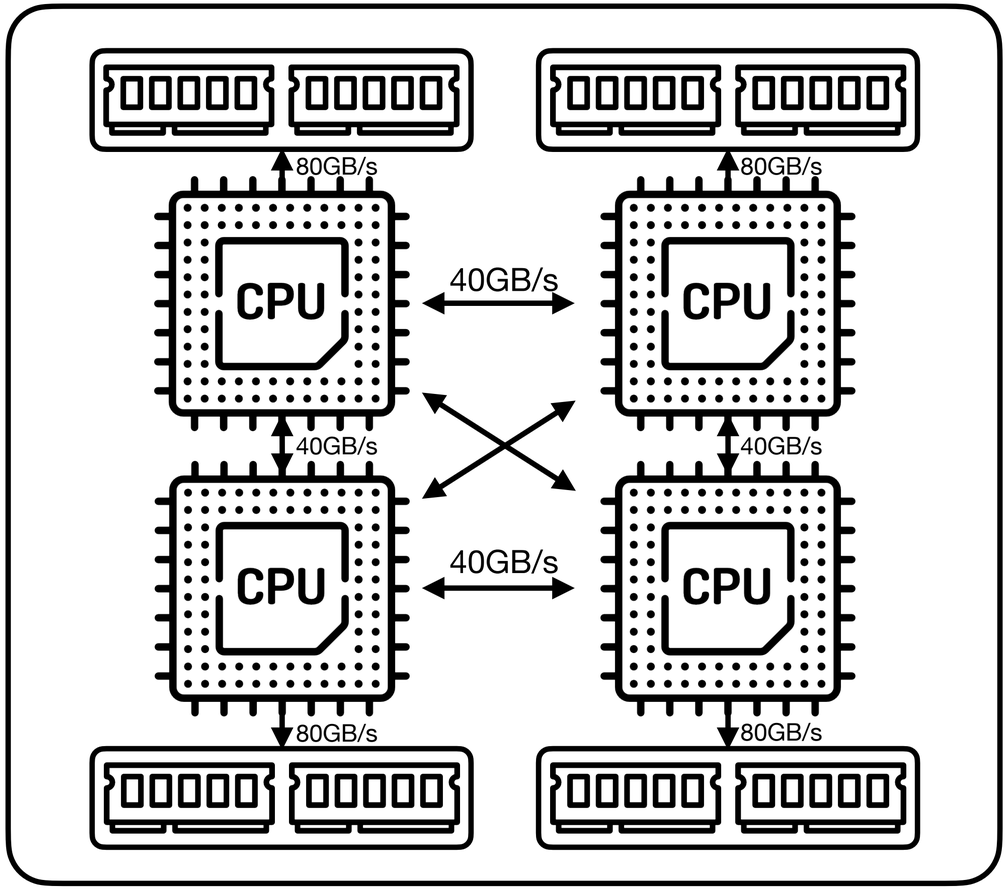


1. https://www.flaticon.com/free-icon/ram_900330
2. https://www.flaticon.com/free-icon/cpu_689379

OpenMP

Thread affinity



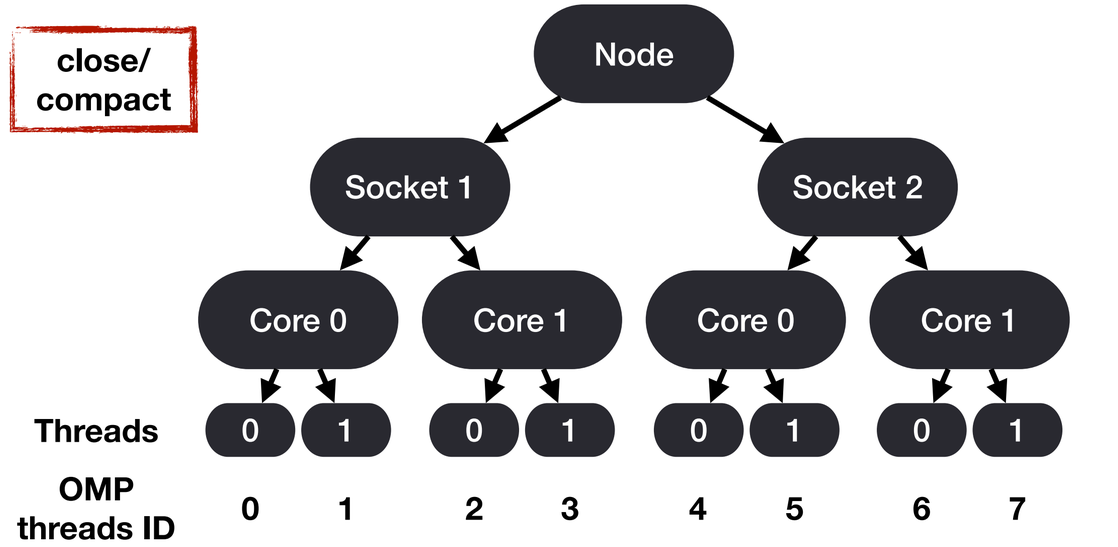


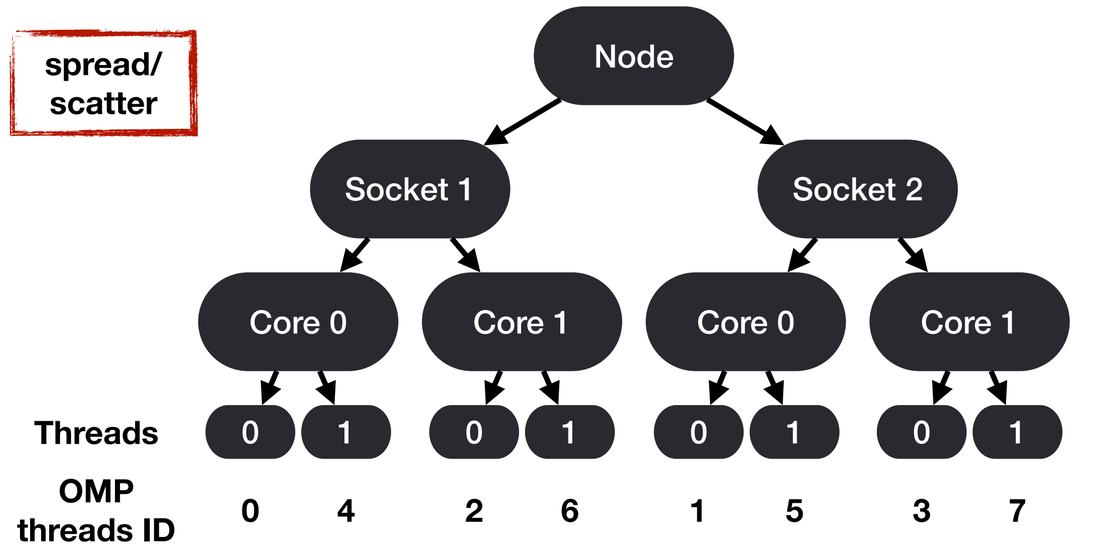
1. https://www.flaticon.com/free-icon/ram_900330
2. https://www.flaticon.com/free-icon/cpu_689379

OpenMP

Thread affinity

- **Common:**
  - **OMP_PROC_BIND**=true | false | close | spread
    - Specifies whether threads may be moved between CPUs
  - **OMP_PLACES**=threads | cores | sockets
    - Specifies on which CPUs the threads should be placed
- **GNU:**
  - **GOMP_CPU_AFFINITY**="0 3 1-2 4-15:2"
    - Bind threads to specific CPUs
- **Intel:**
  - **KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]**
    - modifier
      - *granularity = fine | thread | core | tile*
      - *type = balanced, compact, disabled, explicit, none, scatter*





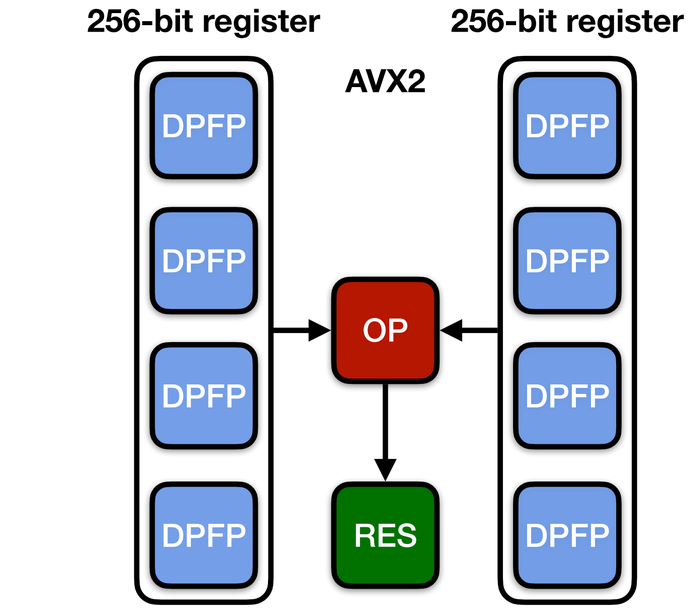
Read more: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html

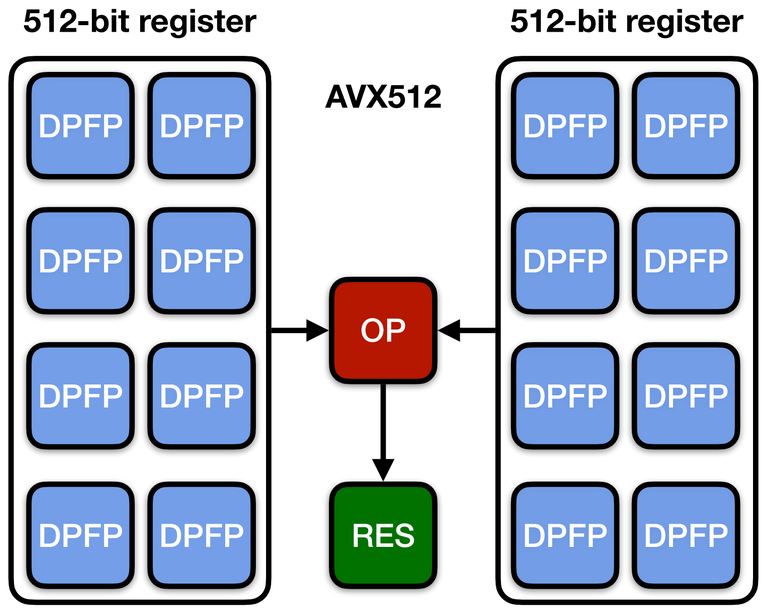
OpenMP

Latest

- **SIMD directive**: multiple iterations of the loop can be executed concurrently using SIMD instructions
- For successful vectorisation, the **data should be aligned**
- Some useful options to check the vectorisation
  - **GNU:** *-fopt-info-vec-optimized*
  - **Intel:** *-qopt-report=1 -qopt-report-phase=vec*

```
#pragma omp simd [clause[ [,] clause] ... ] new-line
for (int n = 0; n < N; ++n) {
    compute(…);
}
```





Hands-on #3.2

Hands-on

Filling the empty space

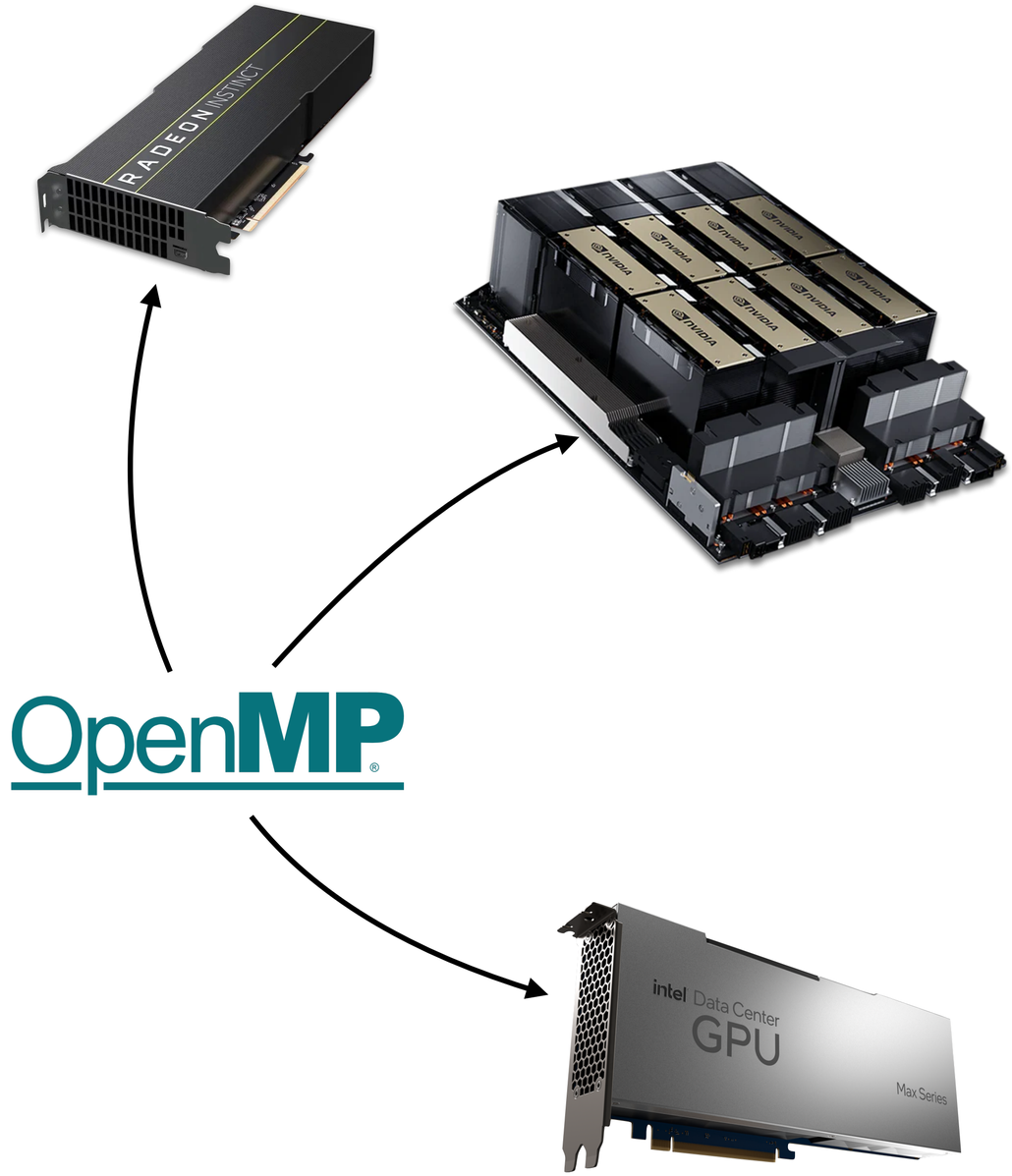
- Parallelise the main math operations in the Jacobi code (***Solver/solver.cpp***) with OpenMP
- Compile the code using the ***make_all.sh omp*** command
- Vectorise as many operations as possible and measure the elapsed time for **32x32 grid** (1 thread)
- Compare performance of the GNU and Intel compilers (modify ***make_all.sh*** accordingly)
  - For the GNU compiler (*module load 2022 foss/2022a*):
    - Set the ***extra_flags*** variable in ***omp*** clause to ***(-fopenmp -fopenmp-simd -fopt-info-vec-optimized)***
  - For the Intel compiler (*module load 2022 intel/2022a*):
    - *Set the* ***extra_flags*** variable in ***omp*** clause to ***(-qopenmp -qopenmp-simd -qopt-report-phase=vectorize)***

We continue @15:10

OpenMP-2

OpenMP
Offloading

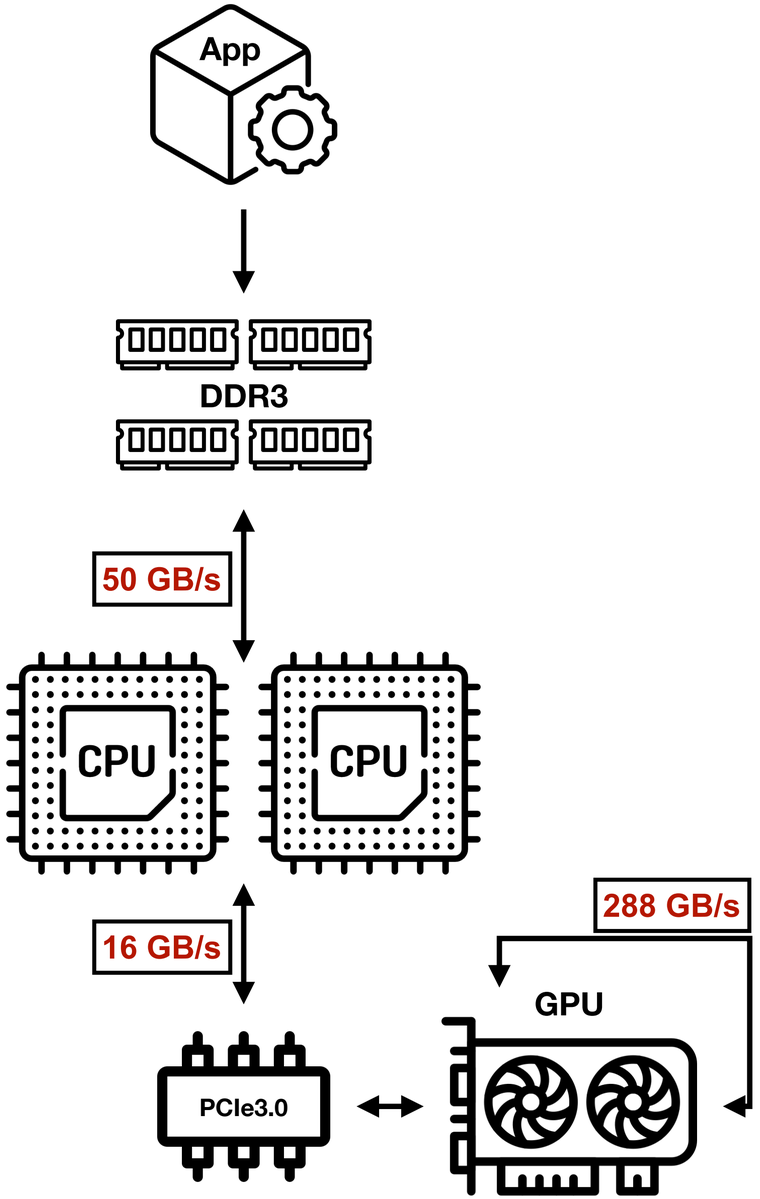
- The OpenMP-4.0/4.5 standard supports offloading to a device (GPU or MIC)
- **Pros:**
  - OpenMP native (no need for a 3rd party library)
  - Requires relatively simple modification of already parallelised code (if the code base is simple)
- **Cons:**
  - Not all compilers provide decent support
  - Not all GPUs are well supported
  - Not as efficient as low-level implementations (e.g. with CUDA or ROCm)



OpenMP

Offloading

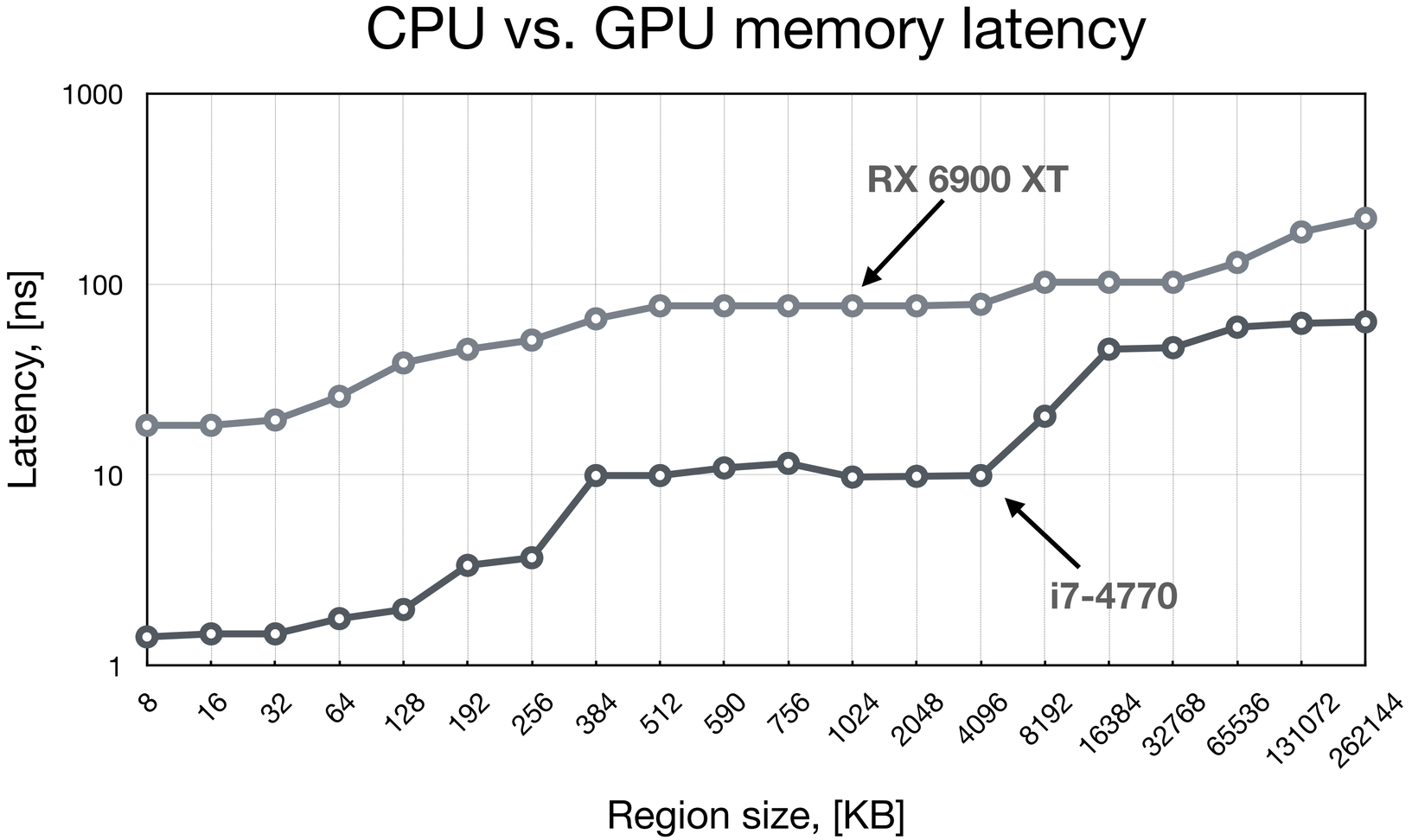
CPU	GPU
Task parallelism	Data parallelism
A few heavyweight cores	Many lightweight cores
High memory size	High memory throughput
Many diverse instruction sets	A few highly optimized instruction sets
Explicit thread management	Threads are managed by hardware

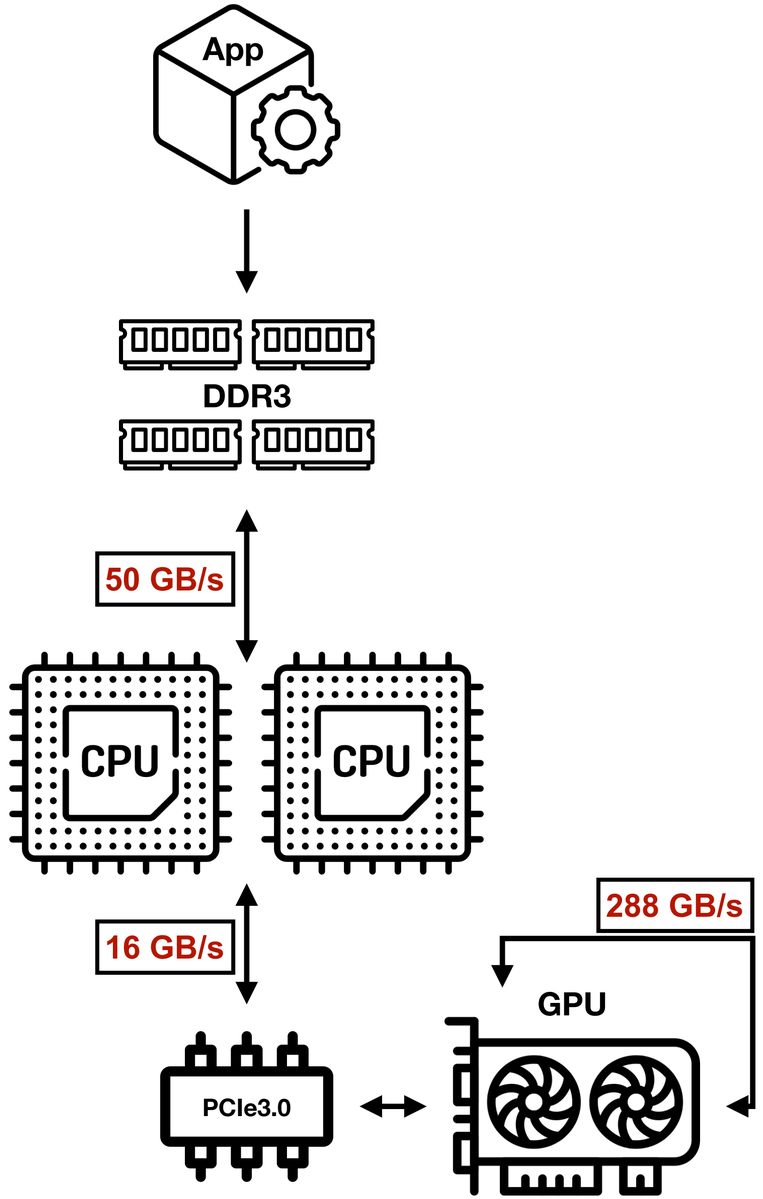


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/graphics-card_2000522
3. https://www.flaticon.com/free-icon/ram_900330
4. https://www.flaticon.com/free-icon/3d-modeling_4229105

OpenMP

Offloading



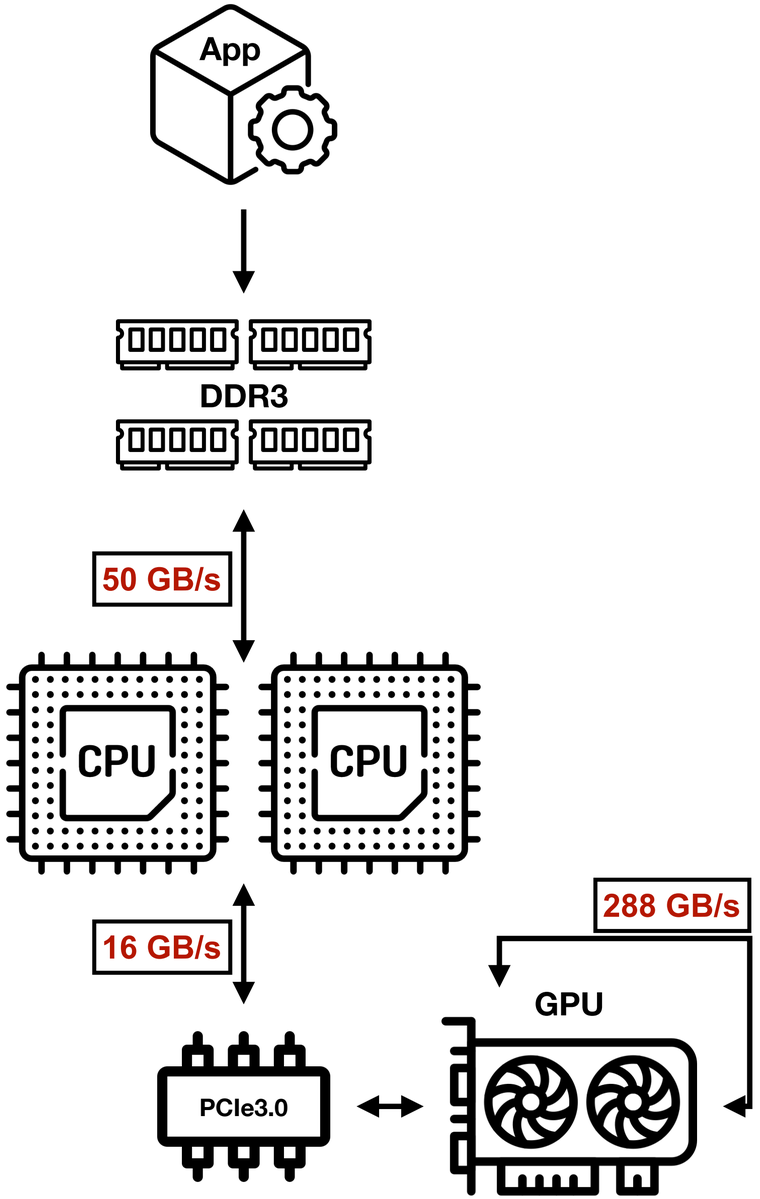


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/graphics-card_2000522
3. https://www.flaticon.com/free-icon/ram_900330
4. https://www.flaticon.com/free-icon/3d-modeling_4229105
5. Data for the plot: https://chipsandcheese.com/2021/04/16/measuring-gpu-memory-latency

OpenMP
Offloading

Rules of thumb:

- Memory transfer is expensive, avoid it
- Offload as much of the work as possible
- Avoid branches
- Preallocate and reuse the memory on a device

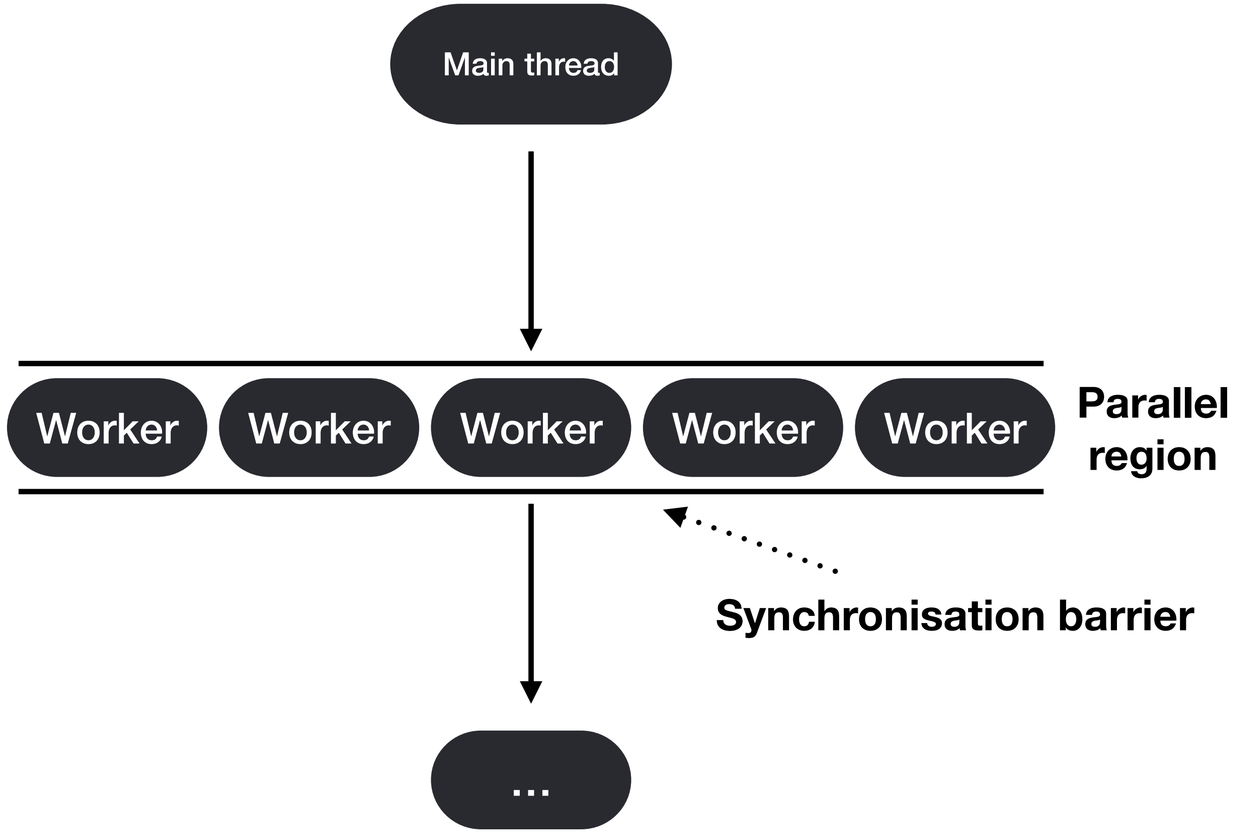


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/graphics-card_2000522
3. https://www.flaticon.com/free-icon/ram_900330
4. https://www.flaticon.com/free-icon/3d-modeling_4229105

OpenMP

Offloading

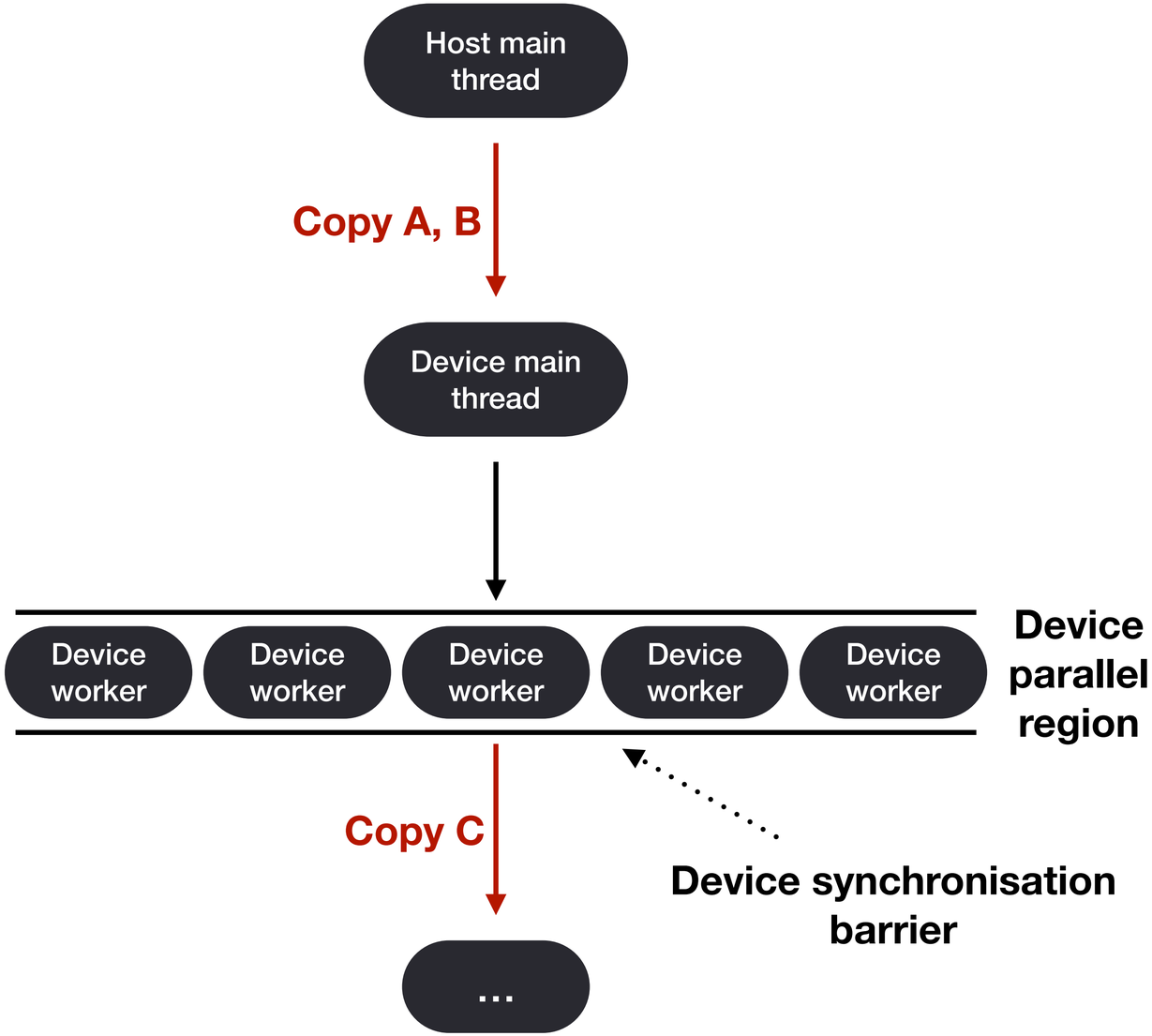
```
int N = 64;
double A[N][N], B[N][N], C[N][N];

…

#pragma omp parallel for
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            for(int k = 0; k < N; ++k) {
                C[i][j] = A[i][k] * B[k][j];
            }
        }
    }
```



OpenMP

Offloading

```
int N = 64;
double A[N][N], B[N][N], C[N][N];

…

#pragma omp target map(to: A, B) map(from: C)
{
 #pragma omp parallel for
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            for(int k = 0; k < N; ++k) {
                C[i][j] = A[i][k] * B[k][j];
            }
        }
    }
}
```



OpenMP

Offloading

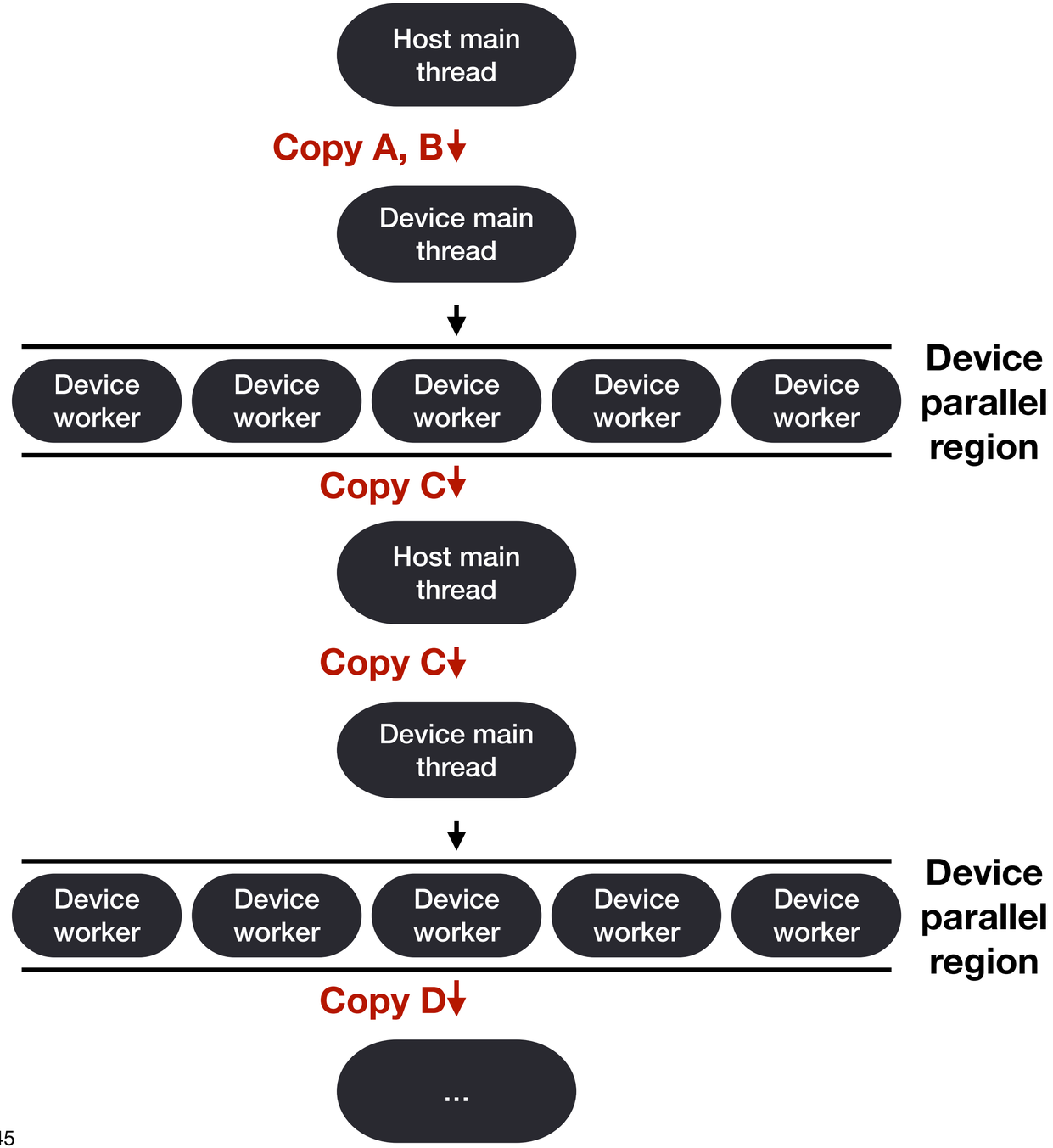
```
int N = 64;
double A[N][N], B[N][N], C[N][N];

…

#pragma omp target map(to: A, B) map(from: C)
{
 #pragma omp parallel for
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            for(int k = 0; k < N; ++k) {
                C[i][j] = A[i][k] * B[k][j];
            }
        }
    }
}
```

- **map(to:list)**: *read-only* data on the device. Variables in the list are initialised on the device using the original values from the host
- **map(from:list)**: *write-only* data on the device: initial value of the variable is not initialised. At the end of the target region, the values from variables in the list are copied into the original variables
- **map(tofrom:list)**: the effect of both a map-to and a map-from
- **map(alloc:list)**: data is allocated and uninitialised on the device
- **map(list)**: equivalent to **map(tofrom:list)**

OpenMP

Scope of data

```
int N = 64;
double A[N][N], B[N][N], C[N][N], D[N][N];

…

#pragma omp target map(to: A, B) map(from: C)
{
    // Calculate C = f(A, B)
}

#pragma omp target map(to: C) map(from: D)
{
    // Calculate D = f(C)
}
```

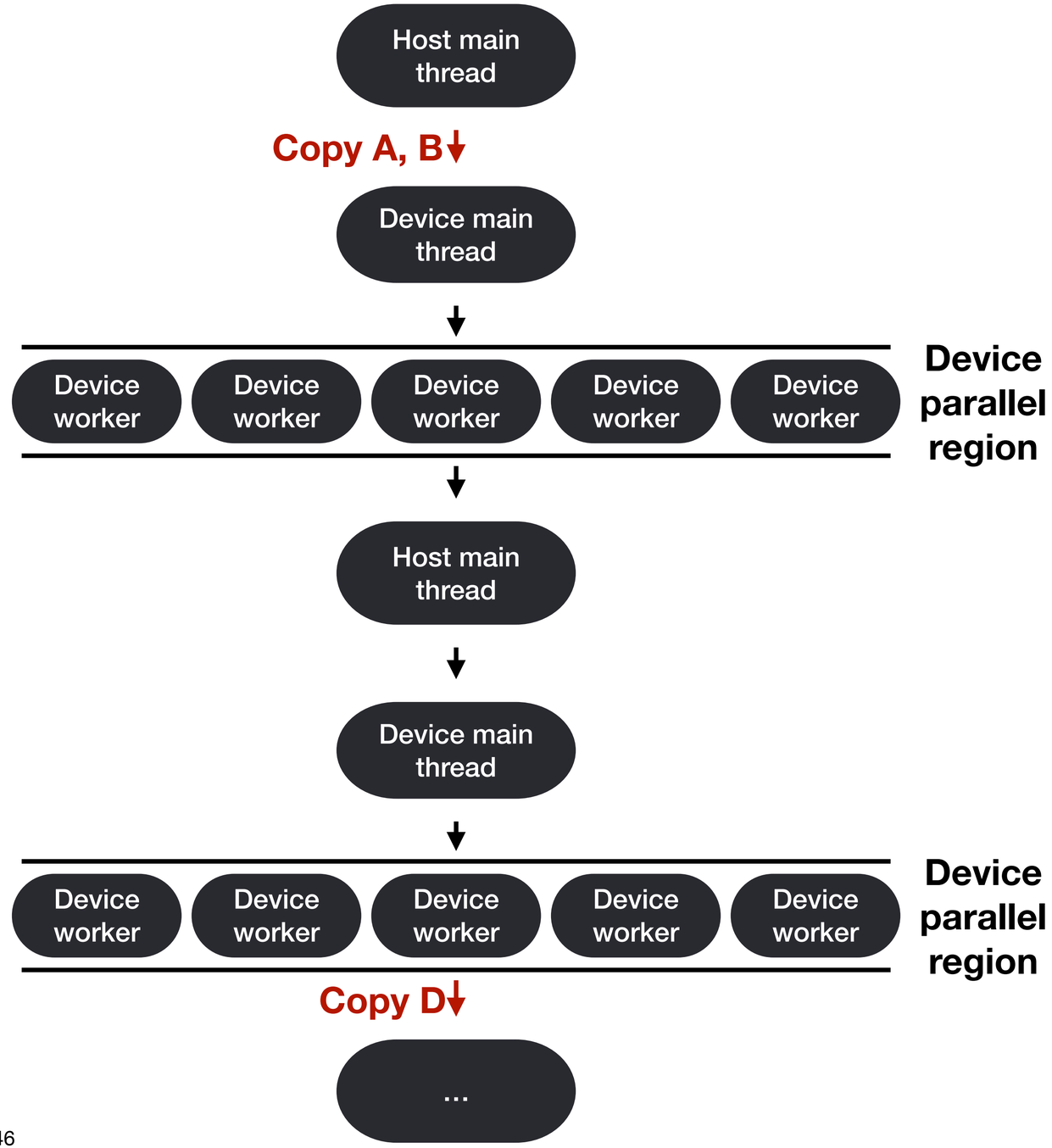


OpenMP

Scope of data

C is allocated and kept directly on the device

```
int N = 64;
double A[N][N], B[N][N], C[N][N], D[N][N];

…

#pragma omp target data map(alloc: C)
{
 #pragma omp target map(to: A, B)
 {
    // Calculate C = f(A, B)
 }

 #pragma omp target map(from: D)
 {
    // Calculate D = f(C)
 }
}
```



OpenMP

Offloading

- A device has at most one copy of each mapped variable
- The **map** clauses are ignored when data is already in the device scope
- The update of the data can be forced using the **update to/from** clause
- Data can be “pre-mapped” using **enter data** and **exit data** clauses

```
#pragma omp target map(tofrom: C)
{
    // Update C
}
#pragma omp target update from(C)
    // Use C on the host
```

```
#pragma omp target enter data map(alloc: C)
#pragma omp target map(to: C)
{
    // Use C
}
#pragma omp target exit data map(from: C)
```

OpenMP

Offloading

- Mapping rules:
  - for pointers **map(to:data[0:N])**
  - for static data **map(to:data)**
- If memory is allocated within a target region it cannot be brought to the host
- Use **target declare** to declare functions for offloading

```
// static array
double C[N];
…

#pragma omp target map(tofrom: C)
{
    // Update C
}
```

```
// dynamic array
double* C = (double*) malloc(N * sizeof(double));

…

#pragma omp target map(tofrom: C[0:N])
{
    // Update C
}
```

Allows to load a subset of an array!

```
#pragma omp declare target
void foo() {
    // Do some work
}
#pragma omp end declare target
```

OpenMP

Offloading

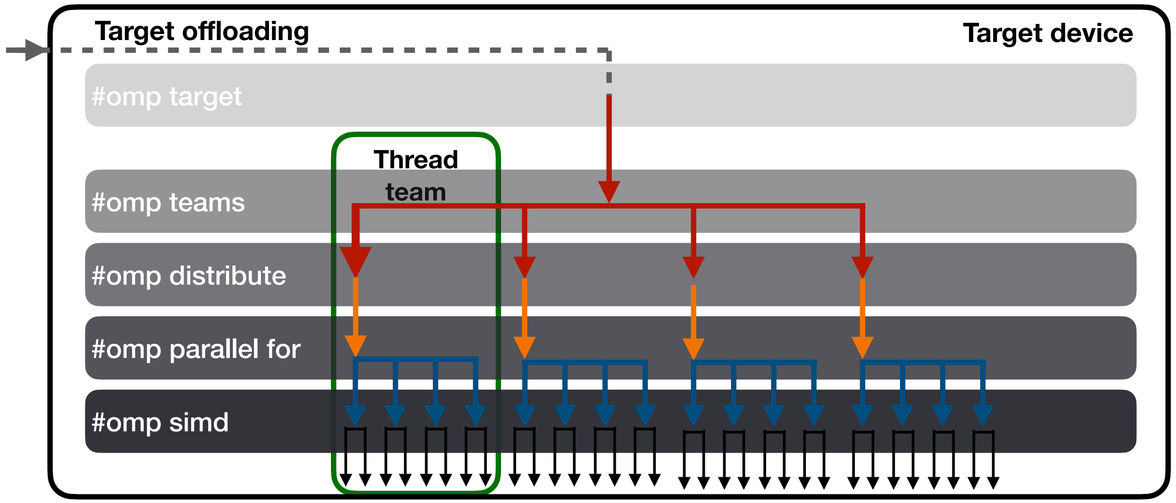
- In **C++** you can also **map an entire class** to the device
  - class can then be used on host or device
  - constructor/destructor used on host/device
- In **OpenMP5.0**
  - can also map individual methods
  - can map class static variables
  - can map class that have virtual functions, as long as they are not used on the device

```
#pragma omp declare target
class Foo {
private:
    int a, b;
public:
    Foo() : a(1), b(2) { }
    ~Foo() { }
    int sum() { return a + b; }
};
#pragma omp end declare target
```

```
class Foo {
private:
    int a, b;
public:
    Foo() : a(1), b(2) { }
    ~Foo() { }
#pragma omp declare target
    int sum() { return a + b; }
#pragma omp end declare target
};
```

OpenMP
Offloading

- Threads can be combined in **teams**
- Work can be **distributed** across **teams**




```
int N = 64;
double A[N][N], B[N][N], C[N][N];

…

#pragma omp target map(to: A, B) map(from: C)
 #pragma omp teams num_teams(4)
  #pragma omp distribute
   #pragma omp parallel for simd num_threads(3)
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            for(int k = 0; k < N; ++k) {
                C[i][j] = A[i][k] * B[k][j];
            }
        }
    }
```

OpenMP

Compiler flags

- Compiler flags

```
$ module use /sw/arch/Debian10/EB_testing/modulefiles/compiler
$ module load 2022
$ module load GCC/11.3.0
$ gcc -fopenmp -foffload=nvptx-none="-Ofast -misa=sm_35" main.cpp
```

```
-m32
-m64
-misa=ISA-string
-mmainkernel
-moptimize
-msoft-stack
-muniform-simt
-mgomp
```

- Profiling

```
$ #profile:
$ module load CUDAcore/11.8.0
$ nvprof ./a.out -s 32 32 -d 1 1
```

OpenMP

GPU offloading

- Example of intensive memory allocations (***nvprof***)

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.93%  2.95501s      5744  514.45us  483.43us  541.73us
_ZN6Solver13calculateNormER6Vector$_omp_fn$0
                   0.62%  18.646ms      5744  3.2460us  2.8160us  13.600us  [CUDA memcpy DtoH]
                   0.44%  13.271ms      5744  2.3100us  2.2400us  11.104us  [CUDA memcpy HtoD]
      API calls:   85.39%  31.2376s     11488  2.7191ms  8.9400us  7.1071ms  cuMemAlloc
                   8.22%  3.00754s      5744  523.60us  6.1710us  1.2464ms  cuCtxSynchronize
                   4.40%  1.60891s     11488  140.05us  3.6850us  1.1687ms  cuMemFree
                   0.51%  188.12ms         1  188.12ms  188.12ms  188.12ms  cuCtxCreate
                   0.47%  171.97ms      5744  29.938us  25.523us  144.83us  cuMemcpyDtoH
                   0.30%  110.65ms      5744  19.262us  13.155us  91.109us  cuMemcpyHtoD
                   0.30%  109.78ms      5744  19.111us  16.459us  826.53us  cuLaunchKernel
                   0.22%  79.001ms         1  79.001ms  79.001ms  79.001ms  cuCtxDestroy
                   0.10%  35.027ms        24  1.4595ms  1.2342ms  2.6770ms  cuLinkAddData
                   0.03%  11.231ms     22979     488ns     210ns  691.80us  cuCtxGetDevice
                   0.03%  10.235ms     11488     890ns     381ns  669.70us  cuMemGetAddressRange
```

Hands-on #3.3

Hands-on

OpenMP offloading

- Offload the main operations in the Jacobi solver
- Compile code with the *`gpu`* flag
- Measure the performance
- Use decent domain size, e.g. 100x100

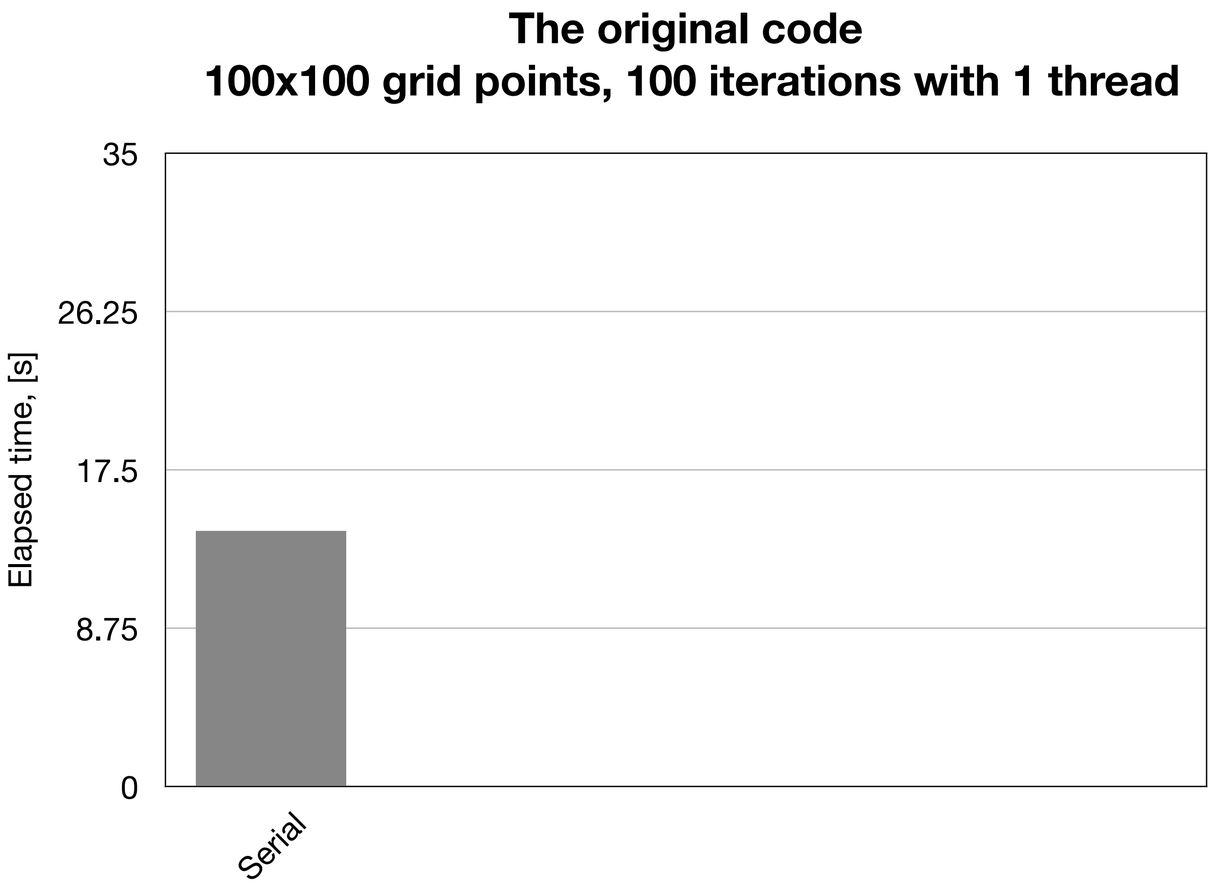
```
#!/bin/bash
#SBATCH -p gpu
#SBATCH –gpus-per-node=1
#SBATCH –job-name=gpu_offloading
#SBATCH –time=00:05:00

module load 2022 foss/2022a
module load CUDA/11.8.0

./make_all.sh gpu

./a.out -s 100 100 -d 1 1
```

```
$ module load 2022
$ module load GCC/11.3.0
$ gcc –fopenmp –foffload=nvptx-none="–Ofast –misa=sm_80” main.cpp
```

```
$ #profile:
$ module load CUDA/11.8.0
$ nvprof ./a.out –s 32 32 –d 1 1
```

OpenMP

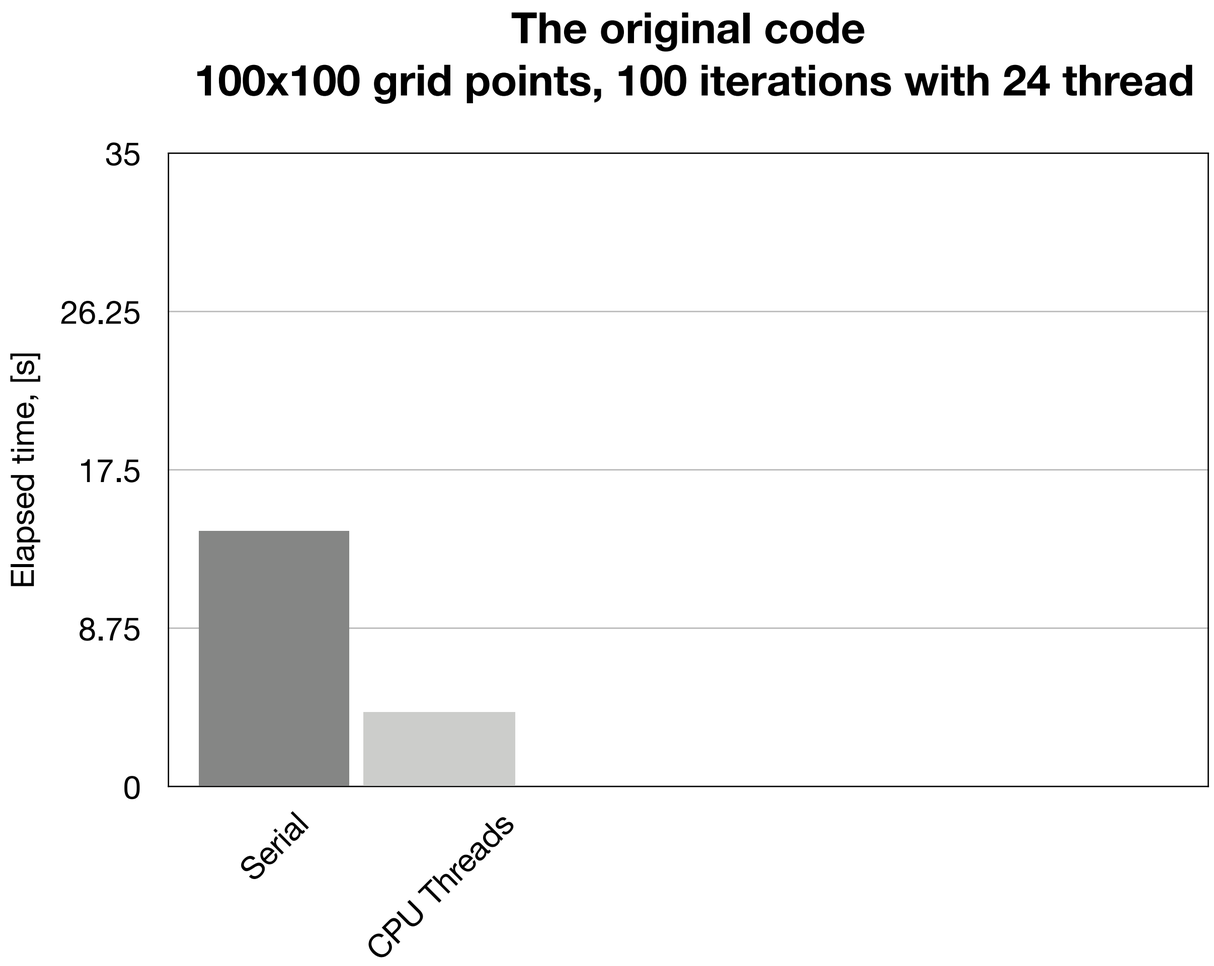
GPU offloading




```
    …
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {

#pragma omp parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

        calculateResidual(A, x, b, res, num_rows, num_cols);
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

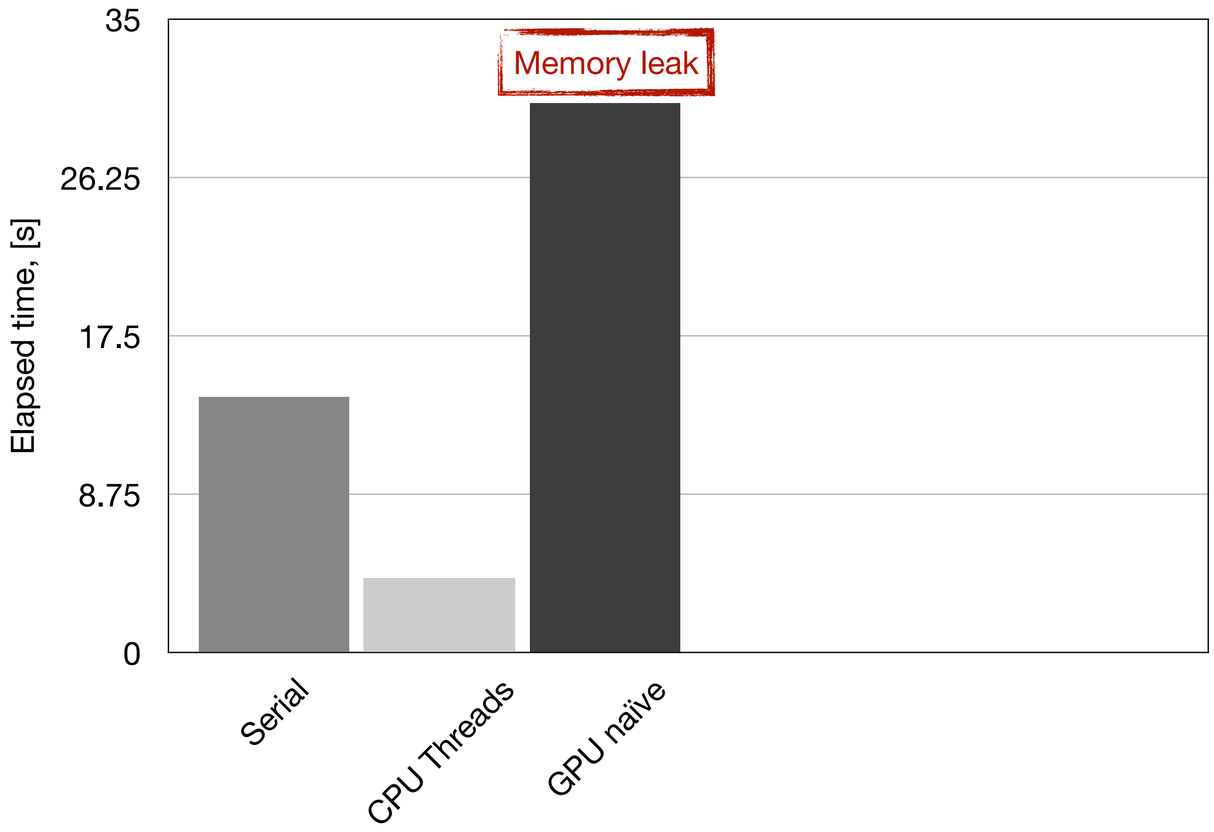
GPU offloading




```
    ...
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {

#pragma omp parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

        calculateResidual(A, x, b, res, num_rows, num_cols);
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

GPU offloading

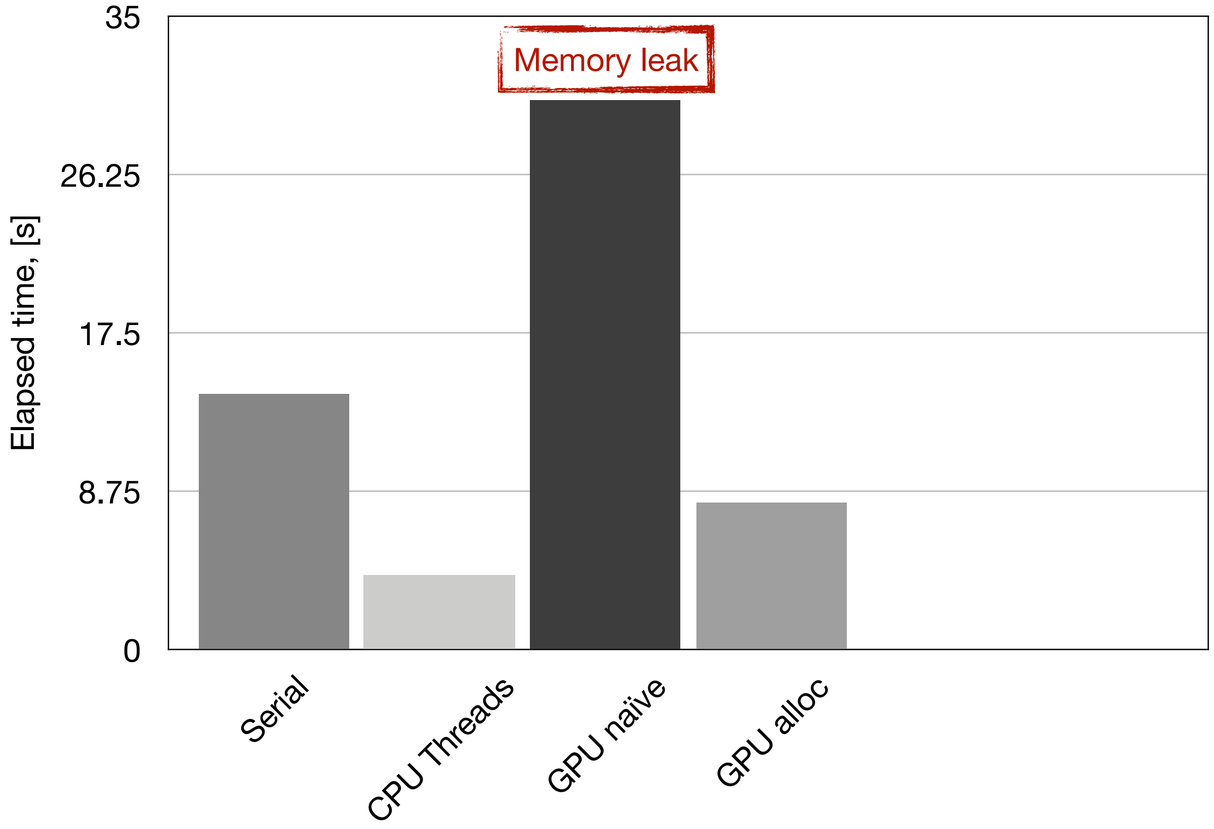



```
    …
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {

#pragma omp target map(to: A[0:num_elts_A], b[0:num_loc_elts_x],
                           x_old[0:num_loc_elts_x]) \
                   map(tofrom: x[0:num_loc_elts_x])
#pragma omp parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

        calculateResidual(A, x, b, res, num_rows, num_cols);
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

GPU offloading

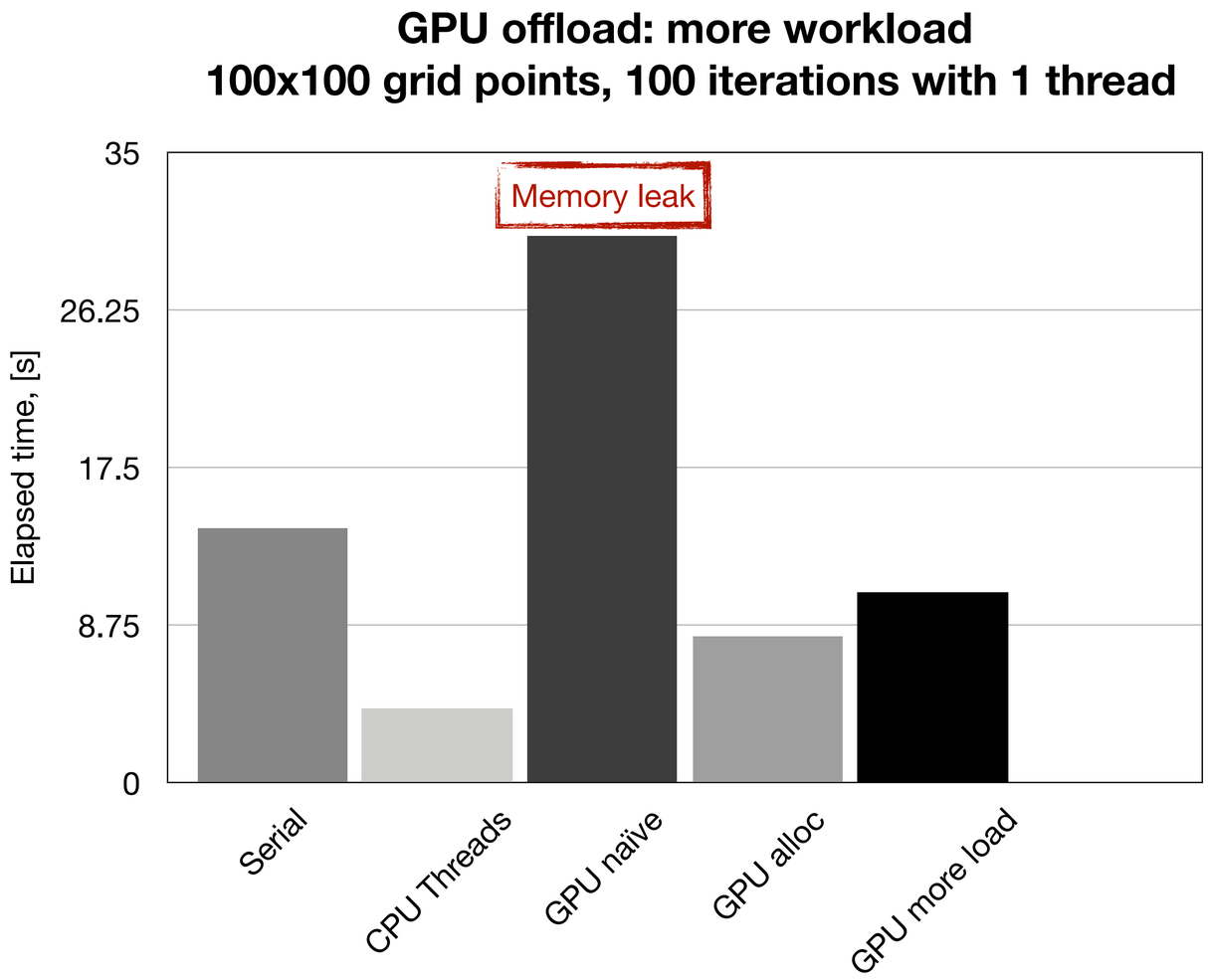
GPU offload: proper memory allocation
100x100 grid points, 100 iterations with 1 thread




```
    …
#pragma omp target data map(to: A[0:num_elts_A],
                                b[0:num_loc_elts_x],
                               x_old[0:num_loc_elts_x]) \
                        map(tofrom: x[0:num_loc_elts_x])
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {
#pragma omp target update to(x_old[0:num_loc_elts_x])
#pragma omp target
#pragma omp parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp target update from(x[0:num_loc_elts_x])

#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

        calculateResidual(A, x, b, res, num_rows, num_cols);

        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

GPU offloading



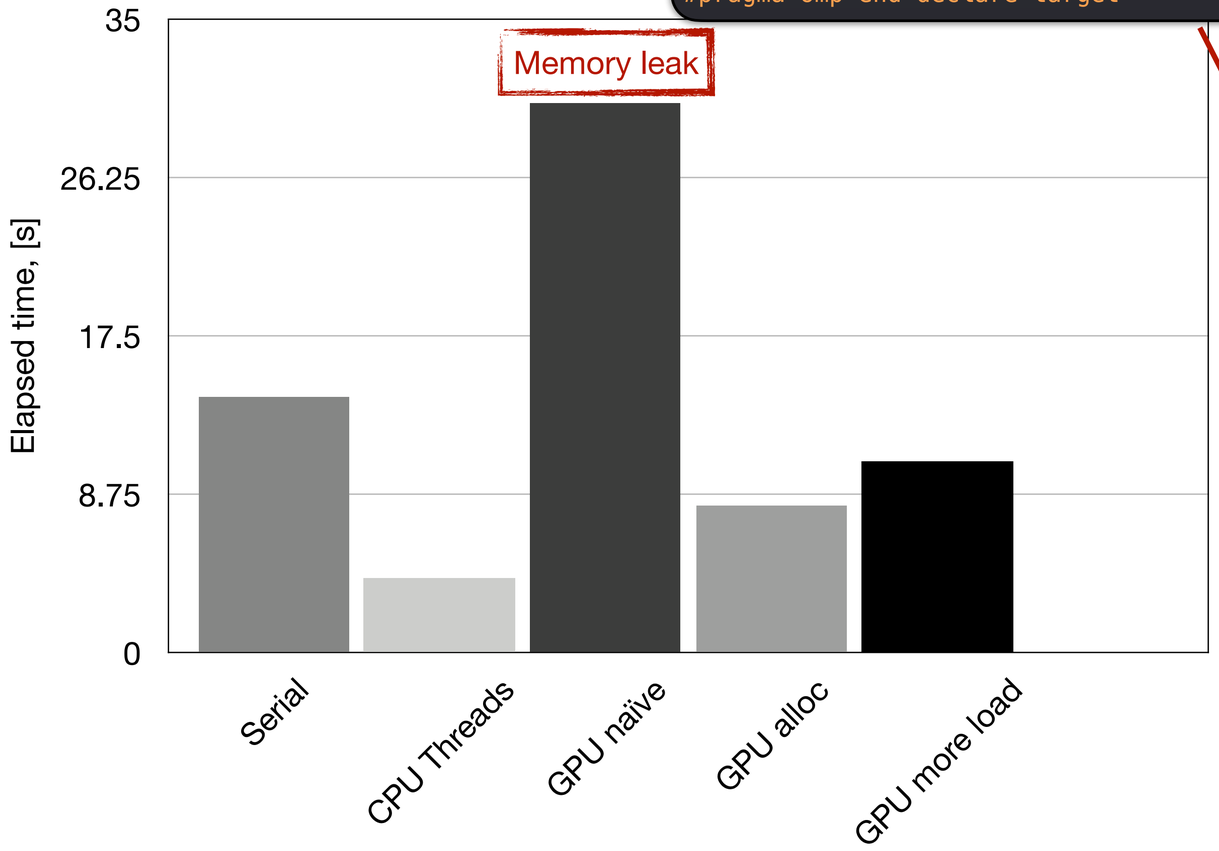

```
        …
#pragma omp target data map(to: A[0:num_elts_A],
                                b[0:num_loc_elts_x],
                                x_old[0:num_loc_elts_x],
                                res[0:num_loc_elts_x]) \
                        map(tofrom: x[0:num_loc_elts_x])
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {
#pragma omp target
#pragma omp parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp target
#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

#pragma omp target
        calculateResidual(A, x, b, res, num_rows, num_cols);

#pragma omp target map(from:residual_norm)
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

GPU offloading

GPU offload:
100x100 grid points, 10

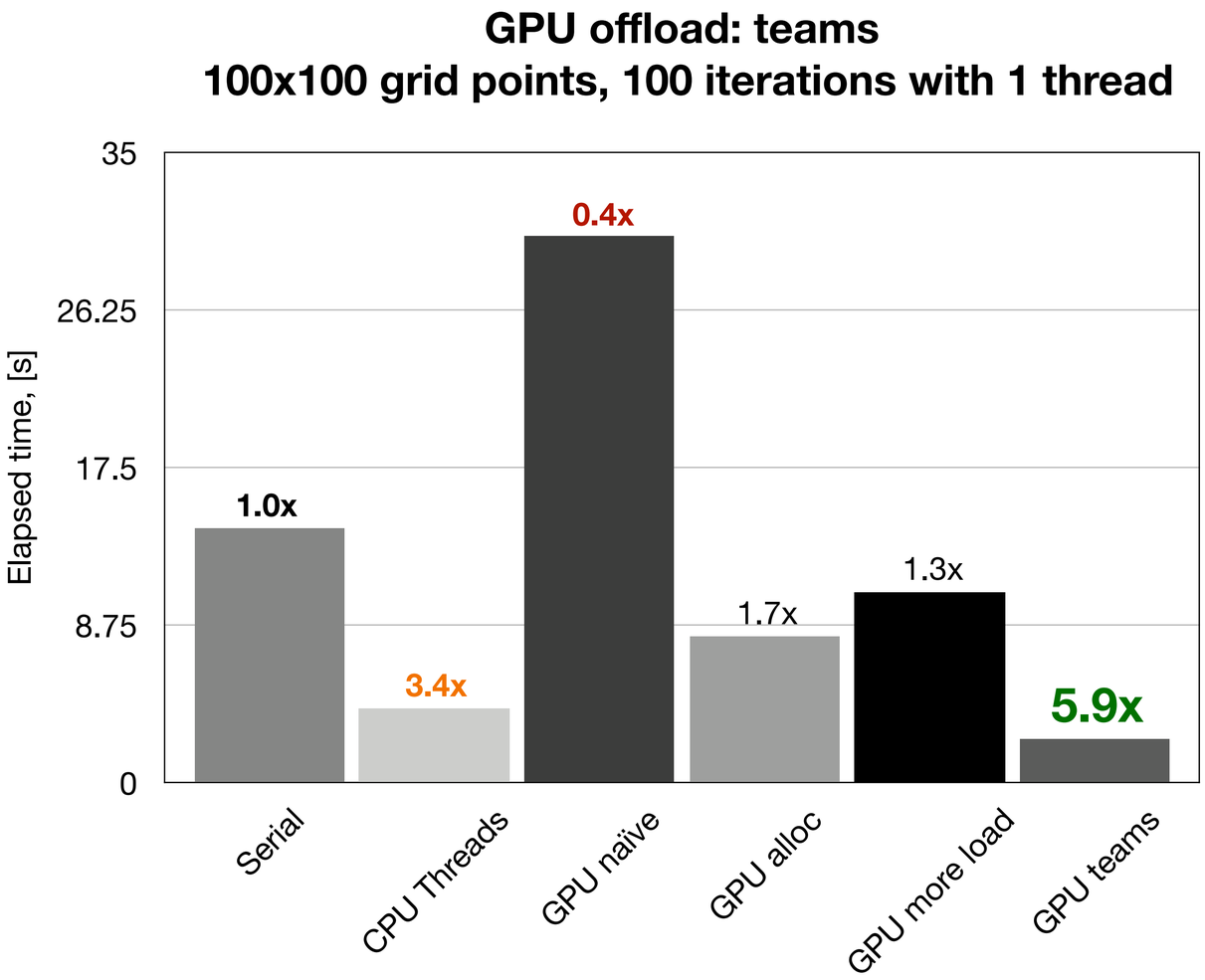
```
#pragma omp declare target
void Solver::calculateResidual(double *A, double *x, double *b,
                               double *res, int num_rows,
                               int num_cols) {

    copyVector(b, res, num_rows);

#pragma omp parallel for
    for(int i = 0; i < num_rows; ++i) {
        double sum = 0.0;
#pragma omp simd reduction(+:sum)
        for(int j = 0; j < num_cols; ++j) {
            sum += A[j + i * num_cols] * x[j];
        }
        res[i] -= sum;
    }
}
#pragma omp end declare target
```




```
data map(to: A[0:num_elts_A],
             b[0:num_loc_elts_x],
             x_old[0:num_loc_elts_x],
             res[0:num_loc_elts_x]) \
     map(tofrom: x[0:num_loc_elts_x])
 max_iter) && (residual_norm > tolerance) ) {

l for
 num_rows - 1; i >= 0; i--) {
diag = 1.;          // Diagonal element
sigma = 0.0;        // Just a temporary value

b[i];

duction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp target
#pragma omp parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

#pragma omp target
        calculateResidual(A, x, b, res, num_rows, num_cols);

#pragma omp target map(from:residual_norm)
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

OpenMP

GPU offloading




```
        …
#pragma omp target data map(to: A[0:num_elts_A],
                                b[0:num_loc_elts_x],
                                x_old[0:num_loc_elts_x],
                                res[0:num_loc_elts_x]) \
                        map(tofrom: x[0:num_loc_elts_x])
    while ( (iter < max_iter) && (residual_norm > tolerance) ) {
#pragma omp target teams
#pragma omp distribute parallel for
        for(int i = num_rows - 1; i >= 0; i--) {
            double diag = 1.;           // Diagonal element
            double sigma = 0.0;         // Just a temporary value

            x[i] = b[i];

#pragma omp simd reduction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp target teams
#pragma omp distribute parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

#pragma omp target teams
        calculateResidual(A, x, b, res, num_rows, num_cols);

#pragma omp target map(from:residual_norm)
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

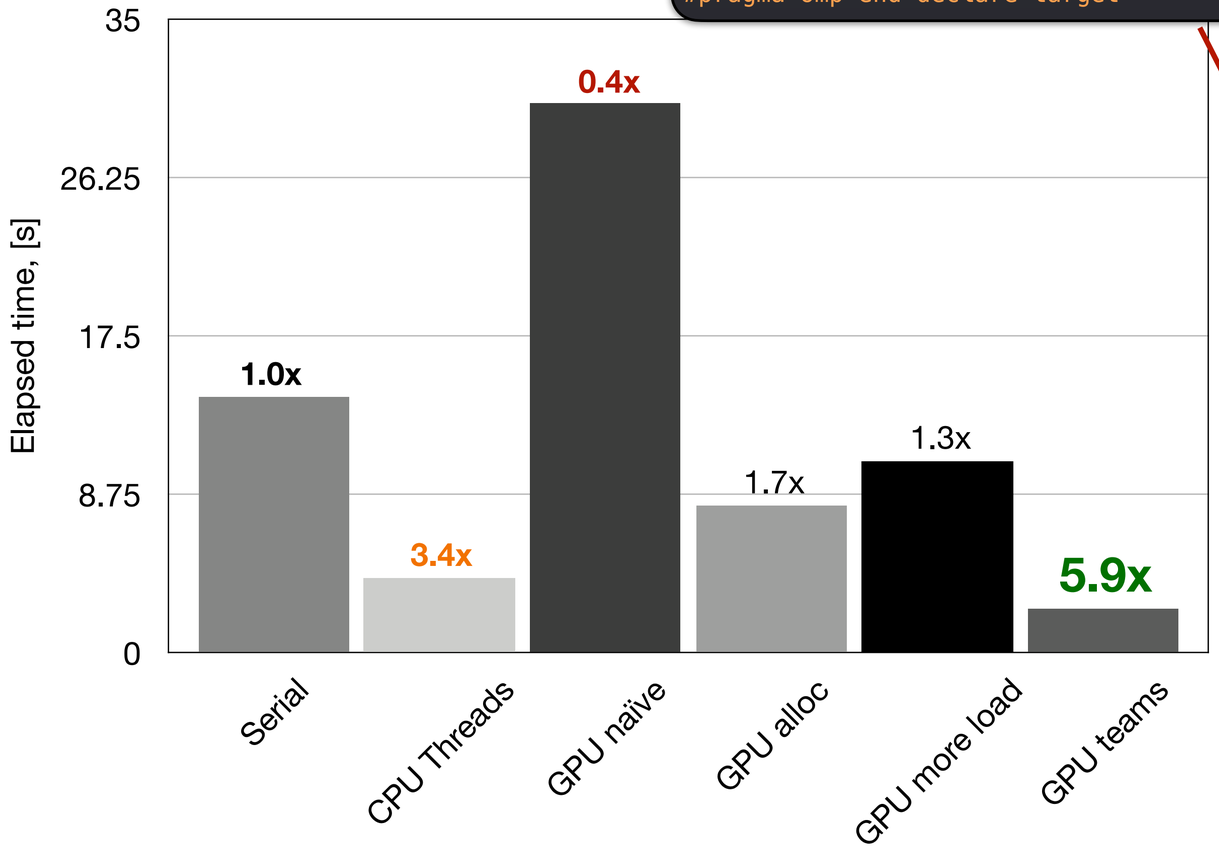
OpenMP

GPU offloading

```
#pragma omp declare target
void Solver::calculateResidual(double *A, double *x, double *b,
                               double *res, int num_rows,
                               int num_cols) {

    copyVector(b, res, num_rows);

#pragma omp distribute parallel for
    for(int i = 0; i < num_rows; ++i) {
        double sum = 0.0;
#pragma omp simd reduction(+:sum)
        for(int j = 0; j < num_cols; ++j) {
            sum += A[j + i * num_cols] * x[j];
        }
        res[i] -= sum;
    }
}
#pragma omp end declare target
```

```
data map(to: A[0:num_elts_A],
             b[0:num_loc_elts_x],
             x_old[0:num_loc_elts_x],
             res[0:num_loc_elts_x]) \
     map(tofrom: x[0:num_loc_elts_x])
 max_iter) && (residual_norm > tolerance) ) {
teams
ute parallel for
 num_rows - 1; i >= 0; i--) {
diag = 1.;          // Diagonal element
sigma = 0.0;        // Just a temporary value

b[i];

duction(+:sigma)
            for(int j = 0; j < num_cols; ++j) {
                sigma = sigma + A[j + i * num_cols] * x_old[j];
            }
            diag = A[i + i * num_cols];
            sigma -= diag * x_old[i];
            x[i] = (x[i] - sigma) * omega / diag;
        }

#pragma omp target teams
#pragma omp distribute parallel for simd
        for(int i = 0; i < num_loc_elts_x; ++i) {
            x[i] += (1 - omega) * x_old[i];
            x_old[i] = x[i];
        }

#pragma omp target teams
        calculateResidual(A, x, b, res, num_rows, num_cols);

#pragma omp target map(from:residual_norm)
        residual_norm = calculateNorm(res, num_rows) / b_norm;

        if (my_rank == 0)
            cout << iter << '\t' << residual_norm << endl;

        iter++;
    }
```

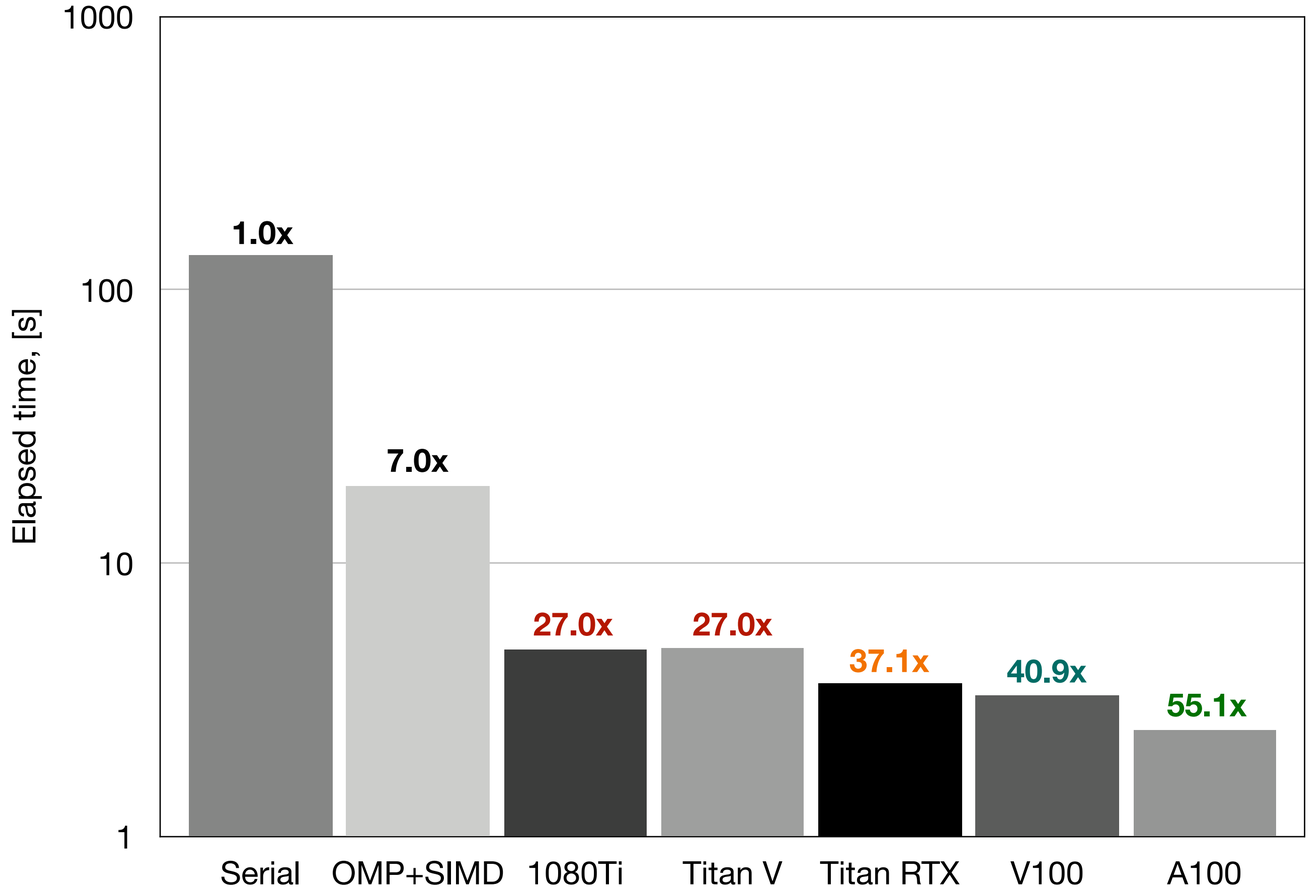
GPU offlo
100x100 grid points, 10



OpenMP

Offload

- **Larger case: 150x150 grid points**
- Performance of the Jacobi solver on host machine (CPU) and on the device (GPU)
- Evaluated iterations: 100
- CPU:
  - Intel Xeon Gold 5118, 2.30GHz, 2x12 cores
- NVIDIA GPUs:
  - GeForce 1080Ti, Titan V, Titan RTX, V100, A100

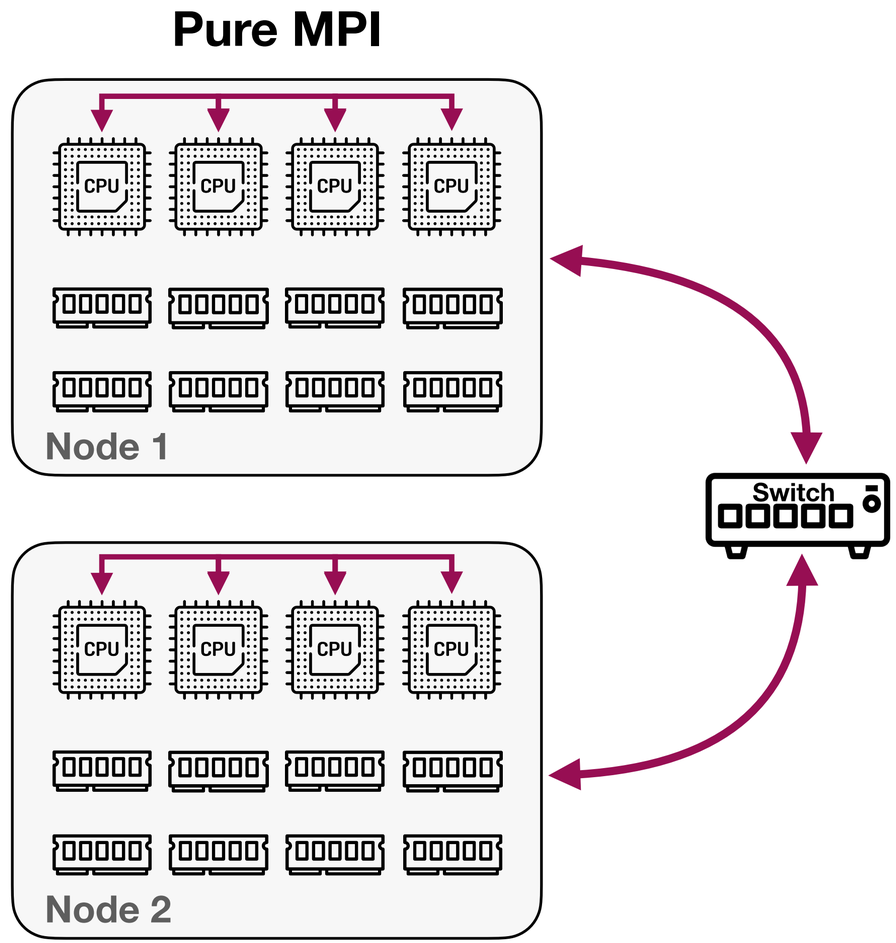


MPI+OpenMP

MPI+OpenMP

Hybrid

- Communications are expensive
- Intra-node communications can be reduced by using threads
- Often easy to implement with already MPI-parallelised code
- Often used in particle-based simulations

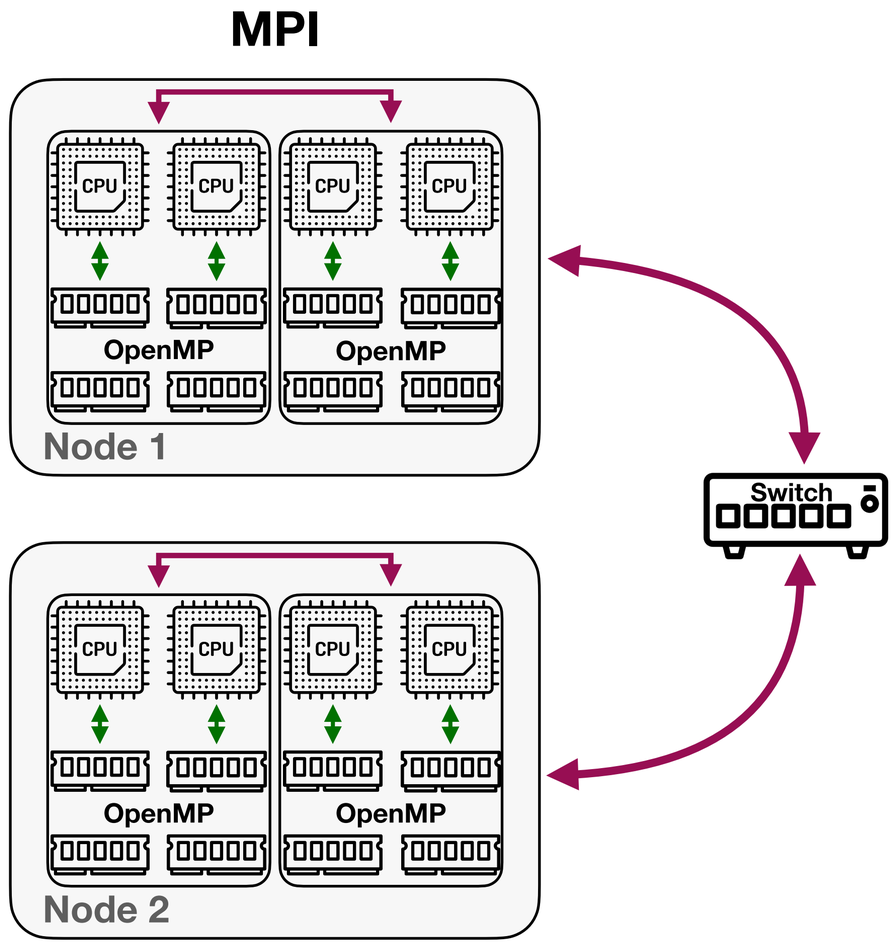


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/ram_900330
3. https://www.flaticon.com/free-icon/switch_1089006

MPI+OpenMP

Hybrid

- Idea - some (every) MPI ranks spawn a few threads
- Sometimes even using hyperthreading may pay off
- May help to reduce impact of the load imbalance
- MPI library may already have optimal implementation on a shared-memory node (hard to optimise)

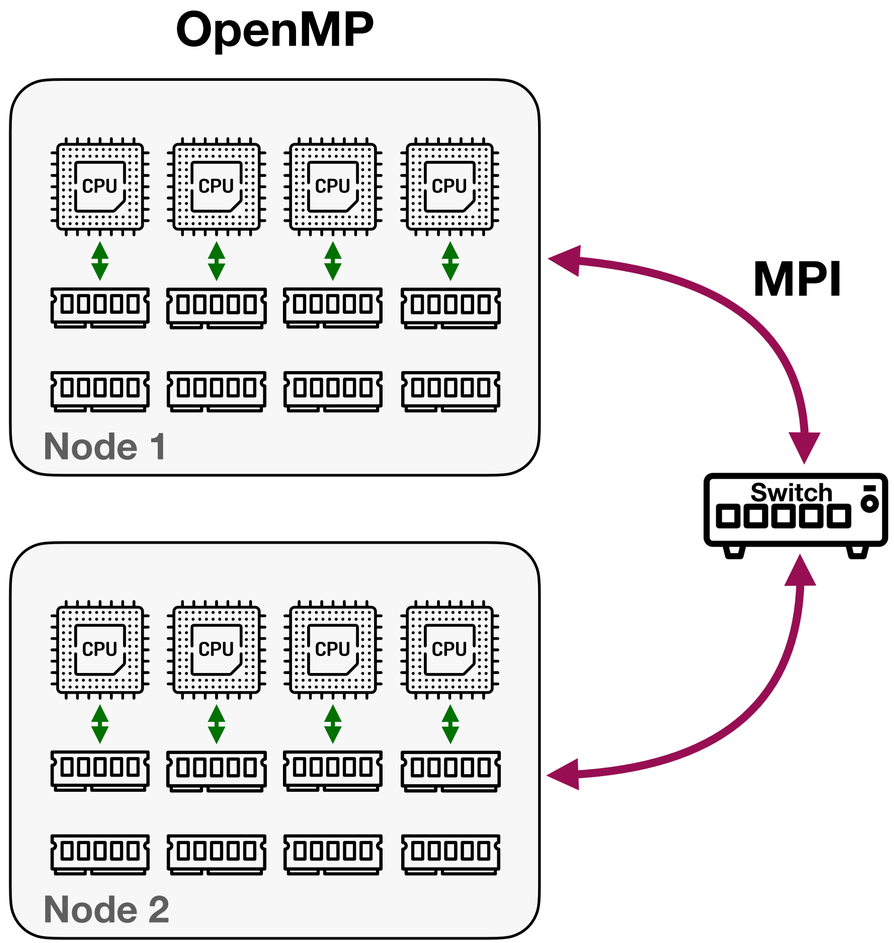


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/ram_900330
3. https://www.flaticon.com/free-icon/switch_1089006

MPI+OpenMP

Hybrid

- Idea - some (every) MPI ranks spawn a few threads
- Sometimes even using hyperthreading may pay off
- May help to reduce impact of the load imbalance
- MPI library may already have optimal implementation on a shared-memory node (hard to optimise)

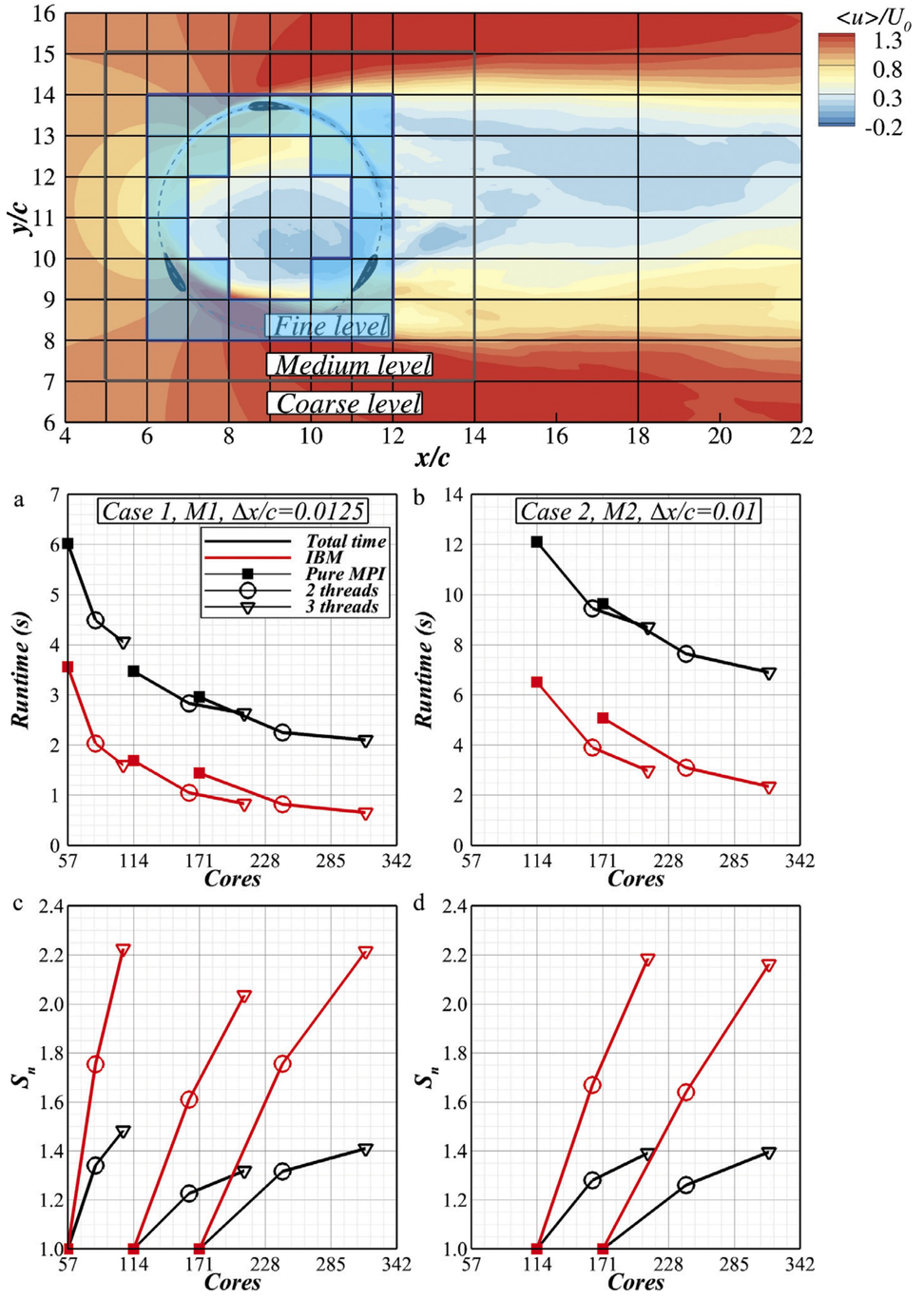


1. https://www.flaticon.com/free-icon/cpu_689379
2. https://www.flaticon.com/free-icon/ram_900330
3. https://www.flaticon.com/free-icon/switch_1089006

MPI+OpenMP

Hybrid

- Example: CFD + IBM[1]
- The whole domain is distributed across multiple processes
- However, only the blue region (Lagrangian grid) around the blades is evaluated with multiple threads
- This helped to improve the load imbalance

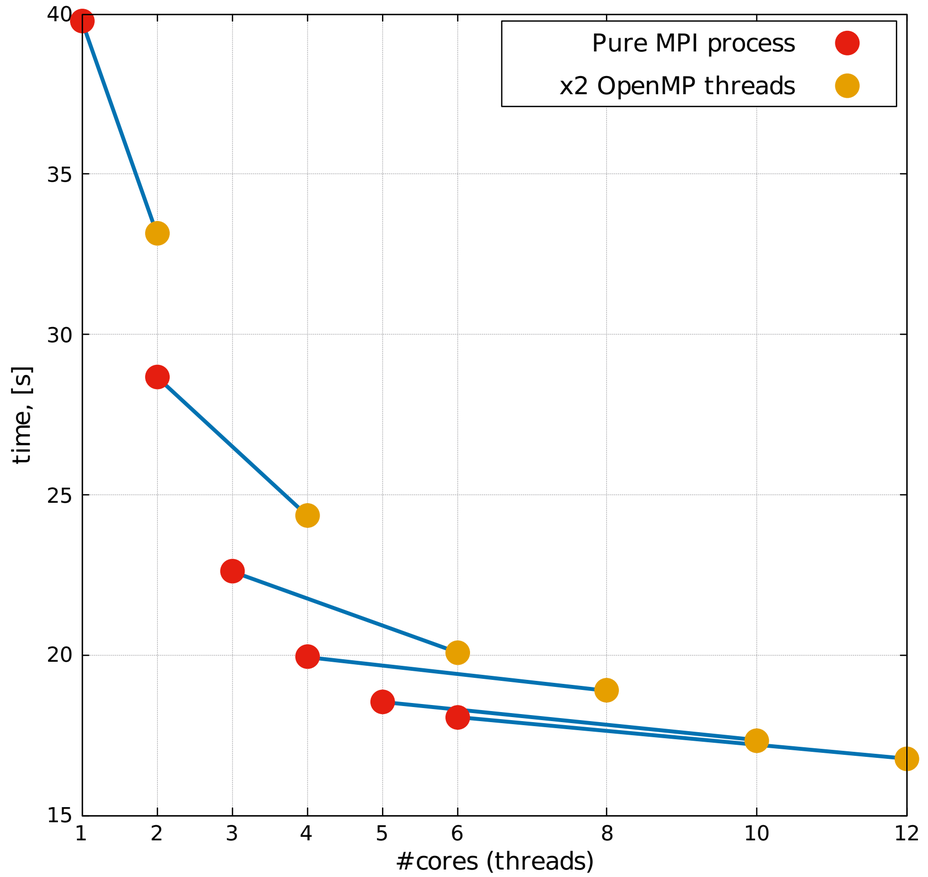


1. P. Ouro, B. Fraga, U. Lopez-Novoa, and T. Stoesser. Scalability of an Eulerian-Lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelisation. Computers & Fluids, 179:123 – 136, 2019

MPI+OpenMP

Hybrid

- Example: FoxBerry, multiphase CFD code
- Duct flow, 300 DoF, structured grid
- BiCGStab(2) + AMG for solving PPE
- MPI tasks correspond to physical cores
- OpenMP threads correspond to hyperthreads
- Performance with 6 MPI tasks and 2 hyperthreads is ~8% better, compared to pure MPI



Hands-on

Hands-on

MPI+OpenMP

```
$ cat hybrid.sh
#!/bin/bash
#SBATCH --job-name=hybrid
#SBATCH --output=hybrid.out
#SBATCH -N 2
#SBATCH --ntasks-per-node=64
#SBATCH --cpus-per-task=2
#SBATCH --partition=rome
#SBATCH --time 10

# Placement of threads as close as possible to each other
#SBATCH --distribution=block:block

module load 2022 foss/2022a

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

echo “Running $SLURM_NTASKS tasks with $SLURM_CPUS_PER_TASK threads”
# Note, we use 1D decomposition!
srun ./a.out -s 150 500 -d 1 $SLURM_NTASKS
$
$ ./make_all hybrid
$ sbatch hybrid.sh
$
```

- Compile the Jacobi code in a hybrid way
- Use this batch script on Snellius, or modify the script accordingly for Lisa
- Measure the performance with 2 and 4 threads per MPI process

Hands-on

MPI+OpenMP

Performance of the hybrid Jacobi solver (**200x500 DoF**), time represents execution of **only 100! iterations**. System: Snellius, AMD EPYC 7H12, 2x64 cores. Job always occupied a full node