Theory-Part-II.pdf

Bioinformatica

Computacion de alto rendimiento

2º Grado en Bioinformática

Escuela Superior de Comercio Internacional
Universidad Pompeu Fabra

NEW YORK BURGER
A fuego, but lento
Calle Pelayo, 5.
08001 Barcelona
¡Únete y recibe una bebida de regalo!
ONE WAY
Gana premios y descuentos únicos por unirte a nuestro programa One Way ;)

Introduction to OpenMP

A **CPU**, or Central Processing Unit, is the primary component of a computer system responsible for executing instructions and performing calculations. It is often referred to as the "brain" of the computer because it carries out the majority of the processing tasks required for the system to function.

The CPU interprets and executes instructions from the computer's memory, performing basic arithmetic, logical, control, and input/output (I/O) operations.
Modern CPUs consist of multiple processing cores, which allow for parallel execution of instructions and improved performance.

A **core** is an individual processing unit within a CPU that is capable of executing instructions independently. This allows parallelism, meaning that multiple tasks or threads can be executed concurrently.

When we run a program in a computer, that program is going to be a sequential program (sequence of machine instructions) and those instructions are going to be referred to as threads. So, a **thread** or task refers to a unit of execution within a program. It represents a sequence of instructions that can be scheduled and executed independently by a CPU core or a processing unit.

Each core will have its own memory, the Cache Memory L1, L2, L3 (L3 is shared between all cores). And the whole CPU will have a common Main Memory.
This is a shared memory system because the memory (L3 and main memory) is shared between all cores.

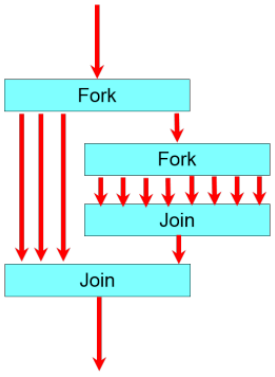
If I have a lot of money, I can use two different machines (MareNostrum) and connect them through the network. But this is a distributed memory system because we can't see the memory of the other machine, only mine.

When running the applications, you normally specify the number of threads you want to use, but in each case the argument is different:

- For example, -t 10 means that you want to use 10 threads in BWA
- In other programs you need to use -threads 10

OpenMP uses a fork-join model

- The master thread spawns a team of threads that joins at the end of the parallel region
- Threads in the same team can collaborate to do work



The idea here is that you have a sequential application (1 thread running) and at some point I am able to open several threads of execution and run the program in parallel.

All threads are going to do the same type of instructions with a different subset of data.

The ideal situation is to run one thread per core (having extra threads does not pay off sometimes). If I have 4 cores, dividing my application into 4 threads is enough in most of the cases. If I have more threads, they will have to wait until the cores are free (the threads are going to run 4 at a time).

There is hyperthreading technology, which means that a core has enough resources to hold 2 threads at a time. So, physically we have one core but I have extra stuff to run 2 threads simultaneously.

OpenMP defines a relaxed memory model

- Threads can see different values for the same variable
- Variables can be shared or private to each thread

We need to know 3 things:

- How do we open and close multiple threads
- Guarantee that the access of the data is safe. Multiple threads can not modify the data at the same time.
- Synchronize threads and memory and wait for termination of tasks.

Constructs or pragmas are going to be added into our programs.

- #pragma omp (then we add the construct that we want)

Creation of threads

We need to specify the parallel construct. We will open a parallel block that will be runned in parallel:

- #pragma omp parallel [clauses]

Main clauses are:

- Num_threads(integer)
- if (expression)
- shared (list-variables)
- private ()
- firstprivate ()
- reduction ()


McDonald's
“SOY CREW DE McDONALD’S,
Y POR SUPUESTO QUE
MI TRABAJO Y MI PASIÓN
SON COMPATIBLES”
¿TE VIENES?
My CREW
Mi trabajo. Mi pasión. Mi gente.

Computacion de alto rendimiento

Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

How do we specify the number of threads? There are 3 mechanisms:

- Use an environment variable: **OMP_NUM_THREADS** → It is used to specify the default number of threads before the program starts running. In the command line
  - *OMP_NUM_THREADS=4 ./file*

- We can use a function: **omp_set_num_threads** → Modify the number of threads during the execution. Inside the program
  - *omp_set_num_threads(4);*

- We can use the "**num_threads()**" pragma directive in each block. In each pragma
  - *#pragma omp parallel num_threads(N)*

Example 1.

```
#include <omp.h>

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

printf ("Starting...",\n);

/* Fork a team of threads  (assuming that OMP_NUM_THREADS
 variable is set to 4) */

#pragma omp parallel
 {
   printf("Hello World from thread = %d\n",
          omp_get_thread_num());

  }

printf ("Finishing...", \n);

}
```

How do I specify in Linux that the variable OMP_NUM_THREADS is equal to 4?

export OMP_NUM_THREADS=4

The output is going to be:

Starting … **# This is done sequentially (only one thread)**

Hello World from thread = 0 **# This is done in parallel (4 threads)**
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3

Finishing … **# This is done sequentially (only one thread)**

This is the order that we will see (opening and joining forks act as a synchronization point). But inside the parallel block, each thread will give the output as soon as they end (so they can be not organized).

If we want to give a concrete order, we need to specific clauses:

- #pragma omp ordered

If we are doing a loop, we use the ordered clause along with the ordered construct:

- # pragma omp for ordered
  for (int i = 0; i < 10; i++) {
  #pragma omp ordered {...

Example 2.

```
#include <omp.h>

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

printf ("Starting...",\n);

/* Fork a team of threads  (assuming that OMP_NUM_THREADS
 variable is set to 4) */

#pragma omp parallel
 {
 int i;
 for (i=0; i<2; i++)
   printf("Hello World from thread = %d iteration = %d \n",
          omp_get_thread_num(), i);

  }

printf ("Finishing...", \n);
}
```

Starting …
Hello World from thread 0 iteration 0
Hello World from thread 1 iteration 0
Hello World from thread 2 iteration 0
Hello World from thread 3 iteration 0
Hello World from thread 0 iteration 1
Hello World from thread 1 iteration 1
Hello World from thread 2 iteration 1
Hello World from thread 3 iteration 1
Finishing …

Important: We are going to see iteration 0 before iteration 1!

Example 3.

```
#include <omp.h>

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

 /* Fork a team of threads with each thread having a private
 tid variable */
 #pragma omp parallel
   {
   /* Obtain and print thread id */
   int tid;
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid);
   /* Only master thread does this */
   if (tid == 0)
     {
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
     }
   } /* All threads join master thread and terminate */
```

Each thread has a private tid variable because this variable is inside the parallel block.

Hello World from thread = 0
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
Number of threads = 4

Number of threads can not appear before “Hello World from thread = 0”.

Sequential Computation of pi

```
static long num_steps = 100000;
double step;
main() {
    int i;
    double x, sum = 0.0, pi = 0.0;

    step = 1.0/(double) num_steps;
    for (int i = 0; i < num_steps; ++i) {
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x*x);
    }
    pi = step *sum;
}
```

Parallel Computation of pi

```
static long num_steps = 100000;
double step;
main() {
    int i;
    double x, sum = 0.0, pi = 0.0;

    step = 1.0/(double) num_steps;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (int i = 0; i < num_steps; ++i) {
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x*x);
    }
    pi = step *sum;
}
```

X has to be a private variable.
Sum has to be global, since it holds the final result.

Data Sharing Attribute Clauses

Because OpenMP is based upon the shared memory programming model, most variables are shared by default.

- Global variables include:
    - File scope variables, static. If the variable is declared at the top of the program, it is going to be shared.

- Private variables include:
    - Loop index variables
    - Stack variables in subroutines called from parallel regions. If a variable is defined inside a parallel block, it is going to be private.

The global variables can be declared into private variables by adding the clause “private”. There are many other clauses:

- SHARED
- PRIVATE

- **FIRSTPRIVATE**: We will have a private variable but its initial value will be equal to the original value of the shared variable. It has the value that it had before entering the parallel block.
  If it is not defined, it will have an unknown value.

- **LASTPRIVATE**: The variable inside the construct combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable. So, all threads will have a private variable with the same initial value. But the value copied back into the original variable object is obtained from the last iteration or section of the enclosing construct.

- **REDUCTION**: Used to collapse the values of private variables into a global variable. We use “reduction(operator : list)”
    - Operators → +, -, *, min, max
    - The compiler creates a private copy of each variable in list that is properly initialized to the identity value

Synchronization

Some OpenMP synchronization mechanisms:

- **#pragma omp barrier** → Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
    - Some constructs have an implicit barrier at the end, for example the parallel construct.

```
#pragma omp parallel
{
      foo ()

#pragma omp barrier     % Forces all ocurrences to happen
      bar ()            % before all bar ocurrences

}    % Implicit barrier at the end of the parallel region
```

- **#pragma omp critical (name)** → Provides a region of mutual exclusion where only one thread can be working at any given time.

```
int x=1, y=0;
#pragma omp parallel num_threads (4)
{
#pragma omp critical (x)
      x++;                  % Different names: one thread
#pragma omp critical (y)    % can update x while another
      y++;                  % updates y
}
```

- **#pragma omp atomic** [update || read || write]→ Ensures that a specific storage location is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads
    - By default it is set to update

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i] *= 2;
}
```

Example 4.

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

int main(){
   int x=2;

   #pragma omp parallel num_threads(2) shared(x)
   {
   if (omp_get_thread_num() == 0) {
        x = 5;
   } else {
        printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   }
   #pragma omp barrier
   if (omp_get_thread_num() == 0) {
        printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   } else {
        printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   }
   }
}
```

1: Thread 1: x = 2 or 5 (normally it will be a 2)
2: Thread 0: x = 5
3: Thread 1: x = 5

There is only one box for x!

Useful Routines

- int omp get num threads()
- int omp get thread num()
- void omp set num threads()
- int omp get max threads()
- double omp get wtime()

Loop parallelism

#pragma omp for [clauses]
for (… ; … ; ...)

Where some possible clauses are:

- private
- firstprivate
- reduction
- schedule (type)
- nowait
- collapse (n)
- ordered (n)

The loop affected by the pragma is going to be automatically divided by the compiler.

- Loop iterations must be independent.
- The default data-sharing attribute is shared
- It can be merged with the parallel construct. This is used if we didn't open a parallel fork beforehand.
    - #pragma omp parallel for

Example 5.

```
int main ()  {
int i, id, MAX = 4;

  omp_set_num_threads (4) ;
  #pragma omp parallel for private (i, id)
  for (i = 0; i< MAX ; i++)
  {
    id = omp_get_thread_num();
    printf ("(%d) Hello iter. = %d \n", id, i);
  }
}
```

0 Hello iter = 0
1 Hello iter = 1 # The number of the thread can change, but the iterations are ordered.
2 Hello iter = 2
3 Hello iter = 3

The private(i) is not necessary, since it is private by default.

Example 6.

```
int main ()  {
int i, id, MAX = 4;
  omp_set_num_threads (4) ;
  #pragma omp parallel
  {
  #pragma omp for private (i, id)
    for (i = 0; i< MAX ; i++)
    {
      id = omp_get_thread_num();
      printf ("(%d) Hello iter. = %d \n", id, i);
    }
   printf ("End of first loop\n");
  #pragma omp for private (i, id)
    for (i = 0; i< 2*MAX ; i++)
    {
      id = omp_get_thread_num();
      printf ("(%d) Goodbye iter. = %d \n", id, i);
    }
  }
   printf ("End of second loop\n");
```

0 Hello iter = 0
1 Hello iter = 1 # The number of the thread can change, but the iterations are ordered.
2 Hello iter = 2
3 Hello iter = 3

End of first loop
End of first loop
End of first loop
End of first loop

0 Goodbye iter = 0
0 Goodbye iter = 1
1 Goodbye iter = 2
1 Goodbye iter = 3
2 Goodbye iter = 4
2 Goodbye iter = 5
3 Goodbye iter = 6
3 Goodbye iter = 7

End of second loop

Example 7.

```
int main ()  {
int i, id, MAX = 4;

  omp_set_num_threads (4) ;
  #pragma omp parallel private (i, id)
  {
    id = omp_get_thread_num();
    for (i = 0; i< MAX ; i++)
       printf ("(%d) Hello iter. = %d \n", id, i);
  }
  #pragma omp parallel for private (i, id)
  for (i = 0; i< MAX ; i++)
  {
    id = omp_get_thread_num();
    printf ("(%d) Goodbye iter. = %d \n", id, i);
  }
}
```

0 Hello iter = 0
0 Hello iter = 1
0 Hello iter = 2
0 Hello iter = 3
1 Hello iter = 0
1 Hello iter = 1
1 Hello iter = 2
1 Hello iter = 3
2 Hello iter = 0
2 Hello iter = 1
2 Hello iter = 2
2 Hello iter = 3
3 Hello iter = 0
3 Hello iter = 1
3 Hello iter = 2
3 Hello iter = 3
0 Goodbye iter = 0
1 Goodbye iter = 1
2 Goodbye iter = 2
3 Goodbye iter = 3

Example 8.

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

int main ()  {

  int tmp =10;

  #pragma omp parallel num_threads(3)
   #pragma omp for firstprivate (tmp) lastprivate (tmp)
       for (int j= 0; j<9; ++j)
            tmp = tmp+j;
   printf ("Final value of tmp =%d\n", tmp);
}
```

Schedules

We can distribute the iterations in different schedules:

- **Static, N:** The iteration space is broken in chunks of approximately size N/num_threads. We can also define the size N of the chunks.
    - Low overhead
    - Can have load imbalance problems

- **dynamic, N:** Threads dynamically grab chunks of N iterations until all iterations have been executed. So, we are not distributing all the iterations at the beginning, but only a size N. The other iterations will be distributed to threads that are available.
  By default, N = 1
    - High overhead
    - Solve imbalance problems

- **guided, N:** Variant of dynamic. The size of the chunks decreases as the threads grad iterations, but it is at least of size N. If no chunk is specified, N=1

Nowait

When we have a for loop, there is an implicit barrier at the end. So, threads will not move on until the loop has been finished unless we specify it using the clause Nowait.

This allows to overlap of the execution of non-dependent loops or tasks.

#pragma omp for nowait

Collapse

It is useful when we have nested loops, but the nest must traverse a rectangular iteration space.

- For example N iterations by M iterations. Instead of having 2 layers of iterations, we take the N·M iterations and distribute them.

If we do not collapse, we are going to distribute the outer loop. If we have 2 threads, one thread will do N/2 iterations of the outer loop and its corresponding iterations of the inner loop. The other thread will do the other N/2 iterations and its corresponding iterations of the inner loop.

Ordered

The order of the operations inside the loop are non-deterministic.

If we want to establish an order, we have 2 mechanisms:

- **Ordered**
- **Doacross**: We say that we need to do something before doing another thing, we create a dependance. We make sure that a previous iteration has been completed before doing another thing.

Single

Only one thread if the team executes the structured block
There is an implied barrier at the end.

#pragma omp single [clauses]

where clauses can be:

- private
- firstprivate
- nowait

Memory consistency

OpenMP lacks consistency memory model, which means that threads are modifying variables in the local memory, even threads that have a shared variable, they have a local copy of the variable. From time to time, this shared variable is going to be updated to the global memory.

At some particular point, the value of a thread affecting a global variable has not been made visible. This affects the order of operations and the behavior of the program.

If we want to make sure that the global copies of the variable are in the global memory, we need to use the flush construct, which will update the variables into the global space.

#pragma omp flush (list)

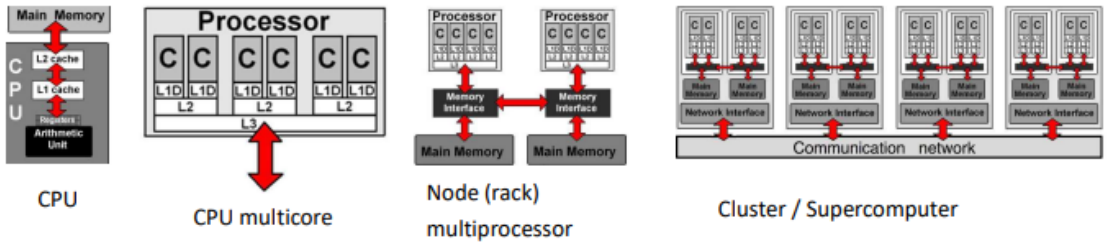
We are ensuring that we do not have a local copy of a shared variable that has not been updated in the global space.

SLURM

We have seen that a processor has several cores, I can run a thread in each core and we have several layers of cache memory.

I can put several processors on top of a board and we will have a server machine.

Any processor in this kind of machine can access any memory in the system, even though some bunch of memory is closer to one processor than another.



This is called NUMA Architecture. The time it takes to access one particular data depends on which memory band it is using.

If I have many racks, I can connect them together through the network and I will have a supercomputer.

If I want to increase the efficiency, I can add a GPU inside each one of these racks.

How do I use this supercomputer?

We do not have direct access to all these nodes of the system (it's not like working with a laptop).

We need to connect to an entry point machine, establishing an ssh connection (secure connection) to the machine. Once you have logged in, you can have access to the resources of the system.

We must take into consideration that there are hundreds of thousands of users.
To manage this, we add an extra layer on top of the OS that will manage many users that will try to run many applications on many systems. This is called **Batch Queue System**.

- The Batch Queue System that we will use is **SLURM**

SLURM performs 3 basic operations:

- Allocates machines or resources to a user
- Starts and controls the execution
- Once it finishes the execution, it manages a queue of pending jobs.

All the machines that we have are splitted into partitions. So, machines are classified in partitions, so that you submit a job to a particular partition (set of machines).

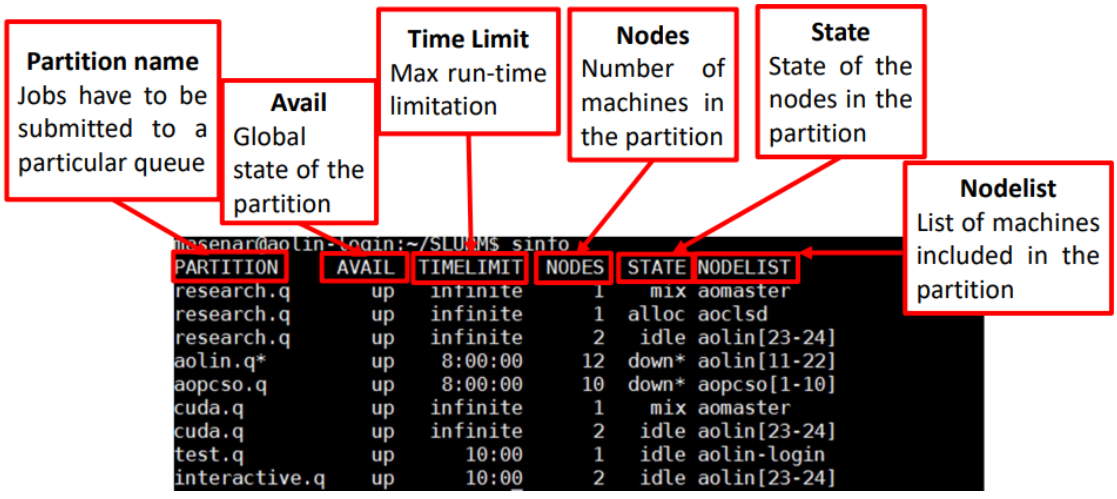
- Interactive partition
- Execution partition

User Commands:

- Information of all partitions on a cluster: sinfo
- Job submission: sbatch, srun
- Job deletion: scancel
- Job status: squeue –u
- Queue list: squeue
- Resource allocation: salloc
- Job control: scontrol
- User accounting: acctmgr

Job specification

- script directives: #SBATCH (SLURM)

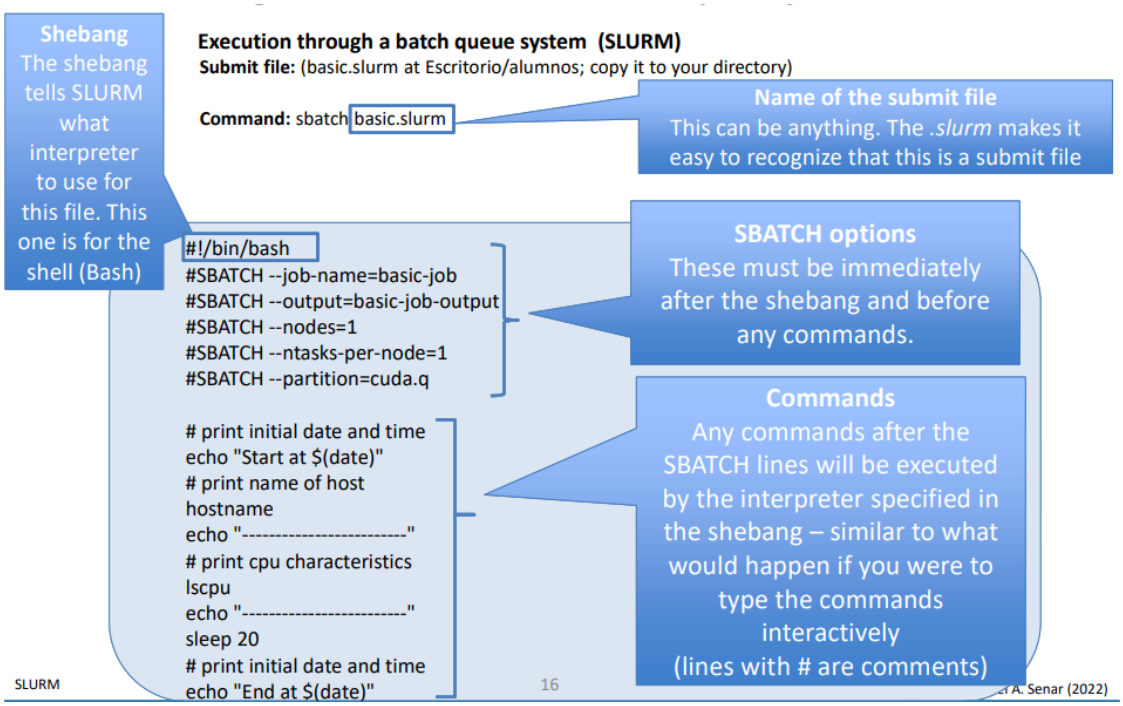


There is no interactivity between the program and the user. You just submit the job to the system and the system decides when the job will be runned.
Thus, if your program requires some input, you need to modify the slurm file so that all the input can be read at any time.

The output will be given in a file, not in the terminal.

What do we submit to SLURM? sbatch name.slurm

The slurm file is composed of a header and shell script operations



The 5 files after the Shebang are only understood by the SLURM.

In this case we are using one node (**#SBATCH –nodes = 1**) and within that node we have multiple CPUs and within these CPUs we have multiple cores.

How many cores do we need to run this file? Just one. For this reason **#SBATCH –ntasks-per-node = 1**.

So, we are specifying the number of threads. So, we are requesting one node but only one of its cores (element that only has one thread).

If we have an MPI application, we are going to use more than 1 node.
If we have an OpenMP application, we are going to use more than 1 task per node (multiple threads).

Command	What it does
--nodes	Number of nodes requested
--time	Maximum walltime for the job – in DD-HH:MM:SS format
--mem	Real memory (RAM) required per node – can use KB, MB and GB units – default is MB Request less memory than total available on the node
--ntasks	Number of tasks to run
--tasks-per-node	Number of tasks that will be launched per node
--cpus-per-task	Number of processors required by each task
--mem-per-cpu	Minimum of memory required per allocated CPU – default is 1GB
--output	Filename where STDOUT will be directed – default is slurm-<jobid>.out
--error	Filename where STDERR will be directed – default is slurm-<jobid>.out
--job-name	How the job will show up in the queue

Environmental variables

They can be used in the command section of a submit file (passed to scripts or programs via arguments). Cannot be used within an #SBATCH directive → Use Replacement Symbols instead

Environment Variable	Description
SLURM_JOB_ID	batch job id assigned by SLURM upon submission
SLURM_JOB_NAME	user-assigned job name
SLURM_NNODES	number of nodes
SLURM_NODELIST	list of nodes
SLURM_NTASKS	total number of tasks
SLURM_QUEUE	queue (partition)
SLURM_SUBMIT_DIR	directory of submission
SLURM_TASKS_PER_NODE	number of tasks per node

Imagine that we want to write the hostname into a file that has the same name as the job (basic-job). How can we do this?

hostname > $SLURM_JOB_ID_hostname.txt

Replacement Symbols

Symbol	Value
%A	Job array's master job allocation number
%a	Job array ID (index) number
%j	Job allocation number (job id)
%t	Task identifier (rank) relative to current job
%N	Short host name – (name of the first node in the job)
%u	User name
%x	Job name

A number can be placed between % and the following character to zero-pad the result

Example:

- #SBATCH –output = job%j.out → Create job777.out for job_id=777
- #SBATCH –output = job%9j.out → Create job00777.out for job_id=777

Array Job Submissions

We can use the same submission script to run multiple jobs that are similar. You have several jobs doing the same stuff and SLURM is going to create several instances of these jobs.

You need to specify: **#SBATCH –array=<array number>**

If we use #SBATCH –array=1, 5, 10 i will have 1 main job and 3 childs with name "common_ID.1, .5, .10"

These array jobs are useful when you need to perform the same operations in different files.

Example 9. What is this job doing?
In this case we have a single job with a for loop!

```
#!/bin/bash
#SBATCH --job-name=for-job
#SBATCH --partition=execution

# print initial date and time
echo "Start at $(date)"

for value in {0..4};
do
  echo $value >> for_$value.out
done

# sleep 10 seconds
sleep 10
# print final date and time
echo "End at $(date)“
```

It prints the Start Date

Creates 5 different files with the following information:

- for_0.out with value 0
- for_1.out with value 1
- for_2.out with value 2
- for_3.out with value 3
- for_4.out with value 4

Then it sleeps for 10 seconds and prints the Final Date.

Note that the general output of the job is going to be stored in: slurm-1777887.out

Since every time we submit a job and we do not specify the output file, 2 files are going to be created:

- slurm-job_ID.out → It will contain the Start Date and End Date
- slurm-job_ID.err

If we submit this job again, we will add in the same files the same information again (no overwrite). Since we are using >>, which means that we append at the end of the file.

Example 10. What is this job doing?
In this case we are submitting an array job. So, 5 instances of the same job are going to be executed. In other words, I will have in the system 5 independent jobs with the same ID with a personalized sub-ID (slurm-178727.0, slurm-178727.1… for example).

```
#!/bin/bash
#SBATCH --job-name=array-job
#SBATCH --partition=execution
#SBATCH --array=0-4

# print initial date and time
echo "Start at $(date)"

echo $SLURM_ARRAY_TASK_ID >>
array_$SLURM_ARRAY_TASK_ID.out

# sleep 10 seconds
sleep 10

# print final date and time
echo "End at $(date)“
```

Since they are independent jobs, they will run at the same time.

Prints the Start Date in each of the 5 slurm-ID_sub-ID.out.

Creates 5 different files with the following information:

- array_0.out with value 0
- array_1.out with value 1
- array_2.out with value 2
- array_3.out with value 3
- array_4.out with value 4

Then it sleeps for 10 seconds and prints the Final Date in each of the 5 slurm-ID_sub-ID.out

Job Dependencies

Allows you to queue multiple jobs that depend on the completion of one or more previous jobs. Maybe because the job before generates files that are going to be used by the next job.

When submitting the job, use the –d argument followed by specification of what jobs and when to execute.

So, the syntax is the following:
sbatch -d <when to execute> : <job_id>

Instead of -d, we can also say –dependency

Common cases:

- after → After the specified jobs have begun execution
- afterany → After the specified jobs have been completed
- afterok → After successful completion
- afternotok → After non-successful completion

If we want so specify multiple jobs, we use: sbatch -d <when to execute> : <job_id> : <job_id>

Job dependencies in a script

It is used when you want to execute multiple jobs that are dependent. You are not going to wait for each task to be completed… So you create a BASH file with the dependencies.

```
#!/bin/bash

# print initial date and time
echo "Start dependentjob at $(date)"
dia=$(date)
echo $dia

# first job - no dependencies
jid1=$(sbatch --partition=aolin.q job1.slurm | cut -f 4 -d' ')
echo $jid1

# multiple jobs can depend on a single job
jid2=$(sbatch  --partition=aolin.q --dependency=afterok:$jid1 job2.slurm | cut -f 4 -d' ')
jid3=$(sbatch  --partition=aolin.q --dependency=afterok:$jid1 job3.slurm | cut -f 4 -d' ')

# a single job can depend on multiple jobs
jid4=$(sbatch  --partition=aolin.q --dependency=afterany:$jid2:$jid3 job4.slurm | cut -f 4 -d' ')
jid5=$(sbatch --partition=aolin.q --dependency=afterany:$jid4 job5.slurm | cut -f 4 -d' ')

# show dependencies in squeue output:
squeue -u $USER -o "%.8A %.4C %.10m %.20E"

# print final date and time
echo "End dependent job at $(date)"
```

When submitting a slurm file (sbatch bla bla bla), we obtain the following output:
Submitted batch job <ID>

If we want to obtain the ID of this job, we can use: **cut -f 4 -d ' '**

We will not submit this script, since it is not a SLURM script. It's a BASH script and therefore we will run it!

srun

The srun command allows the immediate execution of a particular job. We have a set of nodes that can be used to run immediately a particular job.

If we use a sbatch submission, we are submitting the job to the system and the system decides when to run it (when the resources are available).

The normal scenario is that we have a particular partition with a small number of nodes available to run interactive jobs using the srun command.
Before using the srun command, we normally use the salloc command to allocate a particular machine.

Using srun to run several jobs

Srun can also be used to run several jobs

Let's take a look at this SLURM job:

```
#!/bin/bash
#SBATCH -N 2          (We ask for 2 nodes  :      --nodes)
#SBATCH -n 2          (We will run 2 tasks  :      --ntasks)
#SBATCH -c 1          (Each task uses one core : --cpus-per-task)
#SBATCH -p execution

test.cmd
```

test.cmd

```
#!/bin/bash
date
echo "I'm sleeping on..."
hostname
sleep 40
echo "Done sleeping at"
date
```

I am requesting 2 nodes, I will run 2 tasks and each task is going to run one task per cpu.

So, I want to run 2 instances of this program.

The output of this:
If we run this example, only 1 instance is going to run. The system will provide you 2 nodes but only one instance of the program is going to be executed.

If we want to use both machines, we need to use srun. So, inside the SLURM file, i need to specify → srun test.cmd

It will run as many times as we specify in the number of tasks.

Resource specification for parallel jobs

We have several options to specify how many machines I can claim, how many tasks I want to start…

--nodes=number (number of nodes to use; equiv. -N)
--ntasks=number (number of processes to start; equiv. -n)
--ntasks-per-node=number (number of tasks assigned to a node)
--cpus-per-task=number (number of resources (i.e. cores) used by each task; the number of resources (cores) assigned to the job will be the total_tasks number * cpus_per_task number; equiv. -c)
--ntasks-per-socket (number of tasks to run per CPU socket)
--gres=gpu:number (number of GPU assigned to a node)

With all these elements, you can submit large applications. Imagine that you have a MPI application where every process has already been parallelized using OpenMP (each process has multiple threads). So, I have 2 parallelization layers:

- MPI processes
- OpenMP threads within each process

If I have a machine with several nodes, with several CPUs in each node, with several cores in each CPU. The question is, if I want to run this MPI application on this system and I want to take advantage of the maximum parallelism available in my system, how should I submit the job?

Imagine that I have 4 MPI processes and 4 nodes with 2 CPUs in each node with 4 cores in each CPU. How would you submit a job?

Dependencies

There are different types of dependencies, but the only one that really matters is the loop carried dependency, which will prevent parallelization (the other ones can be handled).

A data dependence exists if the computation of one data point requires other data previously computed. If the required data are computed in another loop iteration, the dependence is called “loop carried”.

- Loop carried dependence are often a problem because they assume an execution order that does not exist in a parallel loop.

Example 1

In the first case, we can see that to compute X[i] we need the information from the previous iteration.

```
for (i=1; i<N; i++)
   X[i] = (X[i]+X[i-1])/2;
```

So, to compute element 3, we need to have previously computed element 2. So, there is a dependency. The element that causes the dependence needs to be written before and then read it (read after write dependence).

Example 2

Element Y is generating the dependence.

Because all the iterations after the iteration i==1 need to have variable Y initialized to 1.

```
for (i=1; i<N; i++){
   if (i==1) then
     y=i;
   X[i])=y;
}
```

If there is no order of execution, another iteration can be executed before and Y will have an unknown value (since it has not been initialized) and we will obtain a wrong result.

Anti dependence

This is a “backwards” data dependence in that one iteration requires data that would be modified “later” in the serial case, implying an order that is not there in the parallel case.

Here we have a write after read dependency and therefore I can not parallelize it immediately.

```
for (i=0; i<N-1; i++)
   X[i] = (Y[i]+X[i+1])/2;
```

This can be handled easily using a new array (Z) equal to the initial values of X[i] that will never be modified. Then:

X[i] = (Y[i] + Z[i+1]) / 2;

Output dependence

This is a data dependence that implies a serial loop order, usually by relying on a specific loop iteration being executed last, and using a variable from inside the loop outside of it.

Output dependencies occur when something computed at the last iteration of the loop is used after.

In this case, “a” is causing the dependency.

Thus, we can not parallelize directly.

This can be handled using the lastprivate(a) clause.

```
for (i=0; i<N; i++){
    a = (Y[i]+X[i])/2;
    Z[i] = a;
}
Result = sqrt (a+b);
```

Removing dependencies

Type	Removal Techniques
Loop Carried Flow Dependence	Reduction() clause (OpenMP) Loop skewing Induction variable elimination
Loop Carried Anti Dependence	Auxiliary array
Loop Carried Output Dependence	Lastprivate () clause (OpeMP)
Non Loop Carried	Not necessary

Flow dependence: reduction()

Flow dependence can be removed by reduction if the operation that causes it does not depend on order.

Each thread will have a private sum which will be reduced into a global sum variable!

```
for (i=0; i<N; i++)
   sum += X[i]/2;
```

```
#pragma omp for reduction (+:sum)
for (i=0; i<N; i++)
   sum += X[i]/2;
```

Flow dependence: loop skewing

If the computation of one array element in one iteration depends on an element of another array from a "previous" iteration, shifting computations to another iteration ("loop skewing") solves the problem.

```
for (i=1; i<N; i++){
   X[i]= (X[i]+ Y[i-1])/2;
   Y[i]= Y[i] + Z[i];
}
```

```
X[1]= (X[1]+Y[0])/2;
#pragma omp for
for (i=1; i<N; i++){
   Y[i]= Y[i] + Z[i];
   X[i+1]= (X[i+1]+ Y[i])/2;
}
Y[N-1]=Y[N-1]+Z[N-1];
```

Can't be parallelized because iteration i needs Y element from iteration i-1

Order reversed. Regrouping one line makes dependency non-loop carrying

Only in special cases

Flow dependence: elimination of induction variables

In some cases, variables that establish a data dependence can be eliminated by reference to the loop index.

```
factor = 1;
for (i=0; i<N; i++){
   X[i]= factor * Y[i];
   factor = factor / 2;
}
```

```
#pragma omp for
for (i=0; i<N; i++)
  X[i]= Y[i] * square (0.5, i-1)
factor = square (0.5, N-1);
```

Factor establishes an unnecessary dependence...

... and might as well be kicked out of the loop (using an appropriate function). If it is used later, we may compute it outside.

Only in special cases

Anti dependencies: auxiliary array

Anti dependencies can be resolved by copying the needed data into a new array that contains the needed elements as they were before the parallel loop was executed.

```
for (i=0; i<N-1; i++)
   X[i]= (Y[i] + X[i+1])/2;
```

```
copy (X, X_old);
#pragma omp for
for (i=0; i<N-1; i++)
  X[i]= (Y[i] + X_old[i+1])/2;
```

Since nothing has happened to X[i+1] when it is needed in serial...

...we can save the unaltered X in "auxiliary" X_old before the loop and eliminate the dependence.

Output dependencies: lastprivate()

Output dependencies occur when the value of a variable that is used inside and outside of the loop depends on a specific iteration being executed last. The lastprivate () clause takes care of this.

```
for (i=0; i<N-1; i++){
   a = (Y[i] + X[i])/i;
   Z[i] = a;
}
Result = sqrt (a+b);
```

```
#pragma omp for lastprivate (a)
for (i=0; i<N-1; i++){
   a = (Y[i] + X[i])/i;
   Z[i] = a;
}
Result = sqrt (a+b);
```

The implicit assumption is that iteration N is last to alter a ...

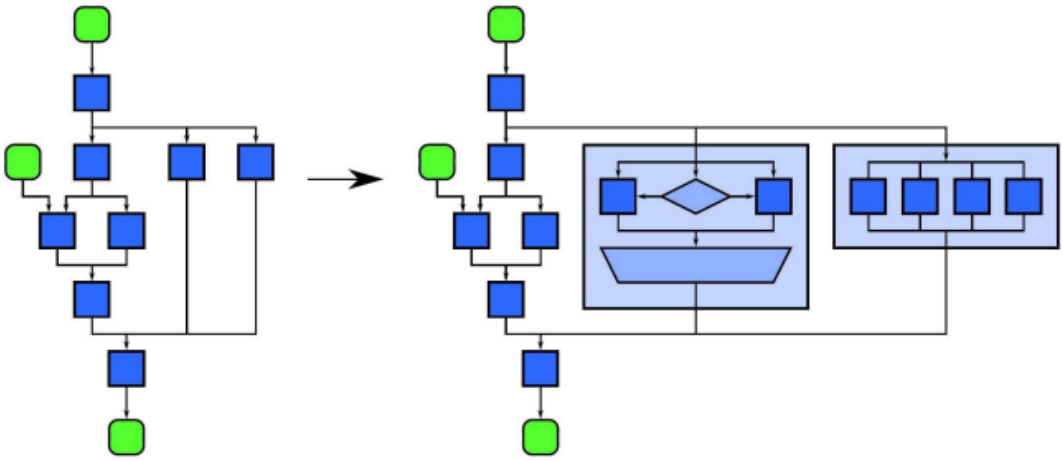
...which is exactly the effect of lastprivate(a)

Parallel patterns

1. Nesting patterns
2. Serial / Parallel Control Patterns
    a. Fork-join
    b. Map
    c. Stencil
    d. Reduction
    e. Scan
    f. Recurrence
3. Serial / Parallel Data Management Pattern
    a. Pack
    b. Pipeline
    c. Geometric decomposition
    d. Gather
    e. Scatter

Nesting patterns

I have a workflow and there are some dependencies. But maybe I can parallelize some parts of the workflow and obtain a nesting parallelism. Meaning that something can be done in parallel and the things that can be done in parallel can also be parallelized.



Parallel Control Patterns: Fork-join

Allows control flow to fork into multiple parallel flows, then rejoin later. So, I have 2 synchronization points.

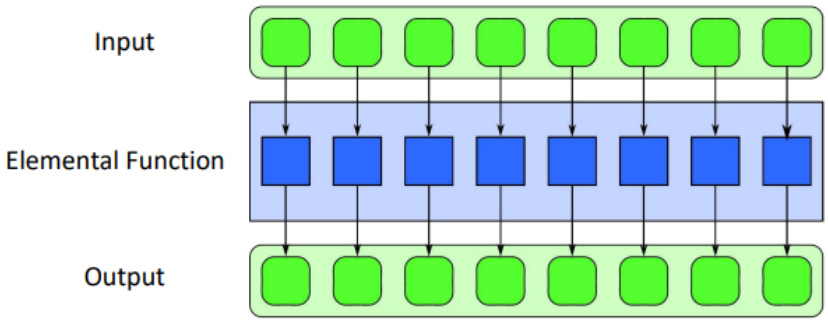
Note that a join is different than a barrier:

- Join → Only one thread continues
- Barrier → All threads continue

Parallel Control Patterns: Map

Performs a computation over every element of a collection that is independent.

Each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection.



Not Jacobi because it has neighborhood relationships. Only if we use different arrays we can use it.

One correct example is adding two arrays.

Parallel Control Patterns: Stencil

I want to compute an element and to do it, I depend on some of the neighbors. The number of neighbors can vary.

Boundary conditions must be handled carefully.

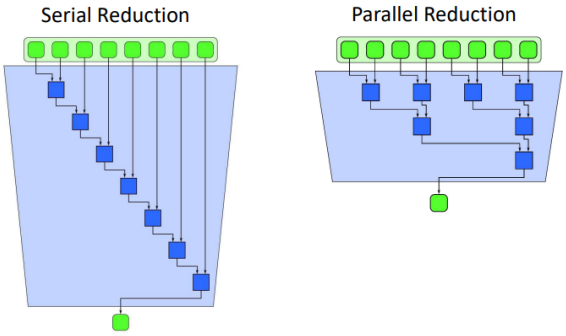
In this case, Jacobi fits this pattern.

Many physical problems can be approximated by this stencil approach:

- I have a material (steel, water…) and its elements can be splitted.
- I can know the evolution of this material. If we want to compute how the heat is transferred in a material, we can compute the heat at some point and compute the heat the next instant using the information from the neighbors.
- The temperature at each point depends on the temperature at the surrounding N elements.

Parallel Control Patterns: Reduction

Combines every element in a collection using an associative “combiner function”.



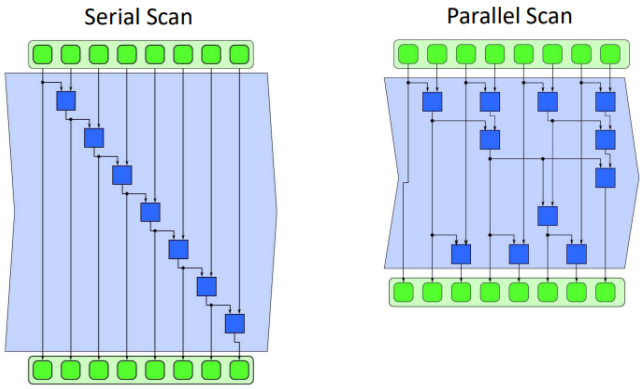
Parallel Control Patterns: Scan

Computes all partial reductions of a collection.
For every output in a collection, a reduction of the input up to that point is computed

It was used in the practice of the sum vector.

- Apply some parallelism at the first level
- Apply corrections later on

To compute a value I need to perform an operation on the previous one.



Parallel Control Patterns: Recurrence

More complex version of map, where the loop iterations can depend on one another.

Map 2 sequences and evaluate the distances. Wavefront parallelism

- Once I have computed one front, I can compute in parallel the next front.

Example: Smith Waterman

GPU Computing and OpenACC

The CPU is the Central Processing Unit and it is used for primary calculations.
The GPU is the Graphics Processing Unit and it is an auxiliary processor designed for parallelizable problems. It has many cores but a small memory.

- Make the same operation many times, for each pixel, for example.

The idea is that some of the computations are going to be delivered from the CPU to the GPU.

Steps:

1. Setup inputs on the host (CPU-accessible memory)
2. Allocate memory for outputs on the host
3. Allocate memory for inputs on the GPU
4. Allocate memory for outputs on the GPU
5. Copy inputs from host to GPU
6. Start GPU kernel → Computations at the GPU are known as kernel executions (each program that is executed in the GPU is known as kernel)
7. Copy output from GPU to host

Latency: Delay that takes place between the start of the operation and the end.

- GPU large latency → A simple operation takes a lot of time
- CPU short latency

The GPU is useful not because of its latency but because of its huge parallelism because it has a lot of cores.

- A lot of throughput

OpenACC

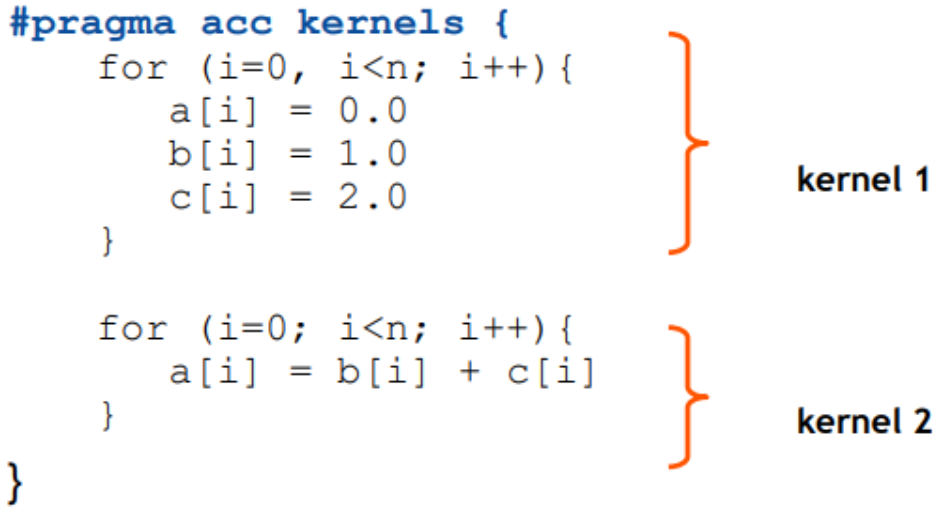
To program an application that will run in a GPU, we must use OpenACC.

When using the directive:

#pragma acc kernels

We are telling the compiler that we want to parallelize this loop and the compiler will do it automatically.

We request that each loop executes as a separate kernel on the GPU. Compiler will be responsible for parallelization.



This looks easy, since the compiler does everything, even the movement of data. **Why don't we just throw kernels in front of every loop? Better yet, why doesn't the compiler do this for me?**

There are two general issues that prevent the compiler from being able to just automatically parallelize every loop:

- Data Dependencies in Loops
- Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results and reasonable performance

Data Management

#pragma acc data [clause]

List of the possible clauses:

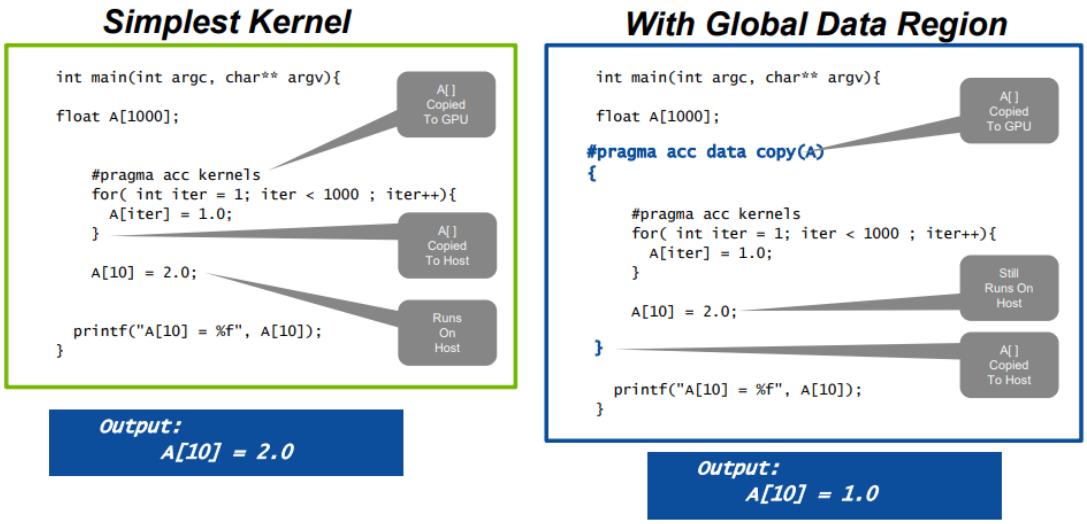
- **Copy(list):** Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

- **Copyin(list):** Allocates memory on GPU and copies data from host to GPU when entering region.

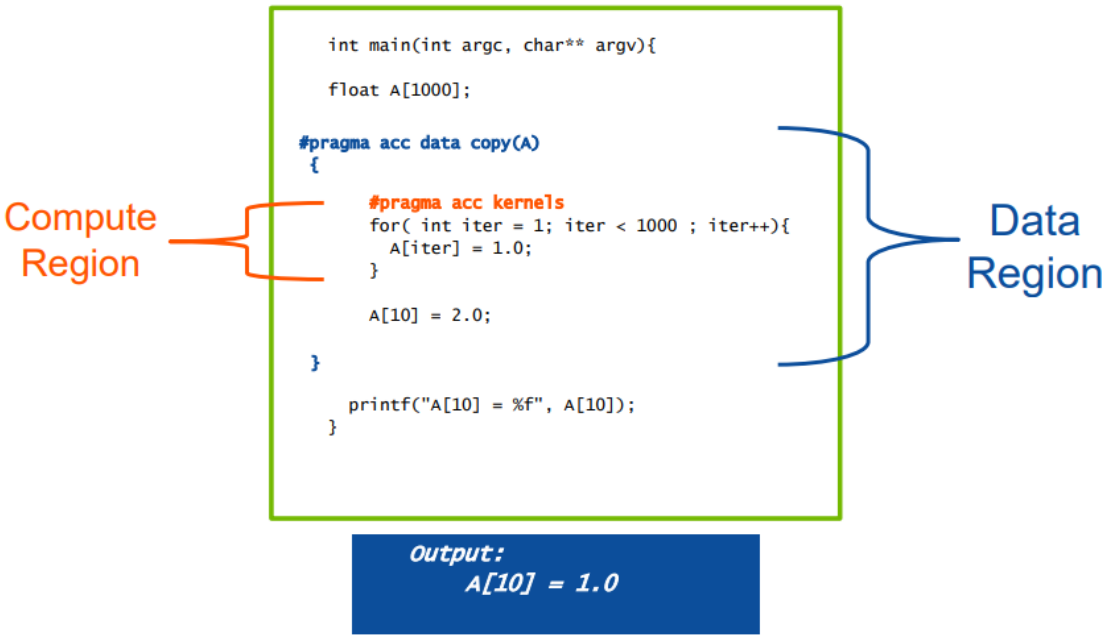
- **Copyout(list):** Allocates memory on GPU and copies data to the host when exiting region.

- **Create(list):** Allocates memory on GPU but does not copy.

Why in the first case the output is 2 and in the second case the output is 1?



Because in the second case the value of A[10] is overwritten.



The parallel directive

So far we've concentrated on the KERNELS directive. The KERNELS directive tells the compiler I may want the loops in a section of code parallelized, but leaves the final decision to the compiler. But what if I know I want the loop to be parallelized? Can I take more control?

#pragma acc parallel loop

You are specifically saying that you want to parallelize that part of the code. In the kernel case, everything is decided by the compiler.

We are responsible if there are data dependencies…