Parallel Programming with OpenMP

Luis Miguel Pinho

July 2022

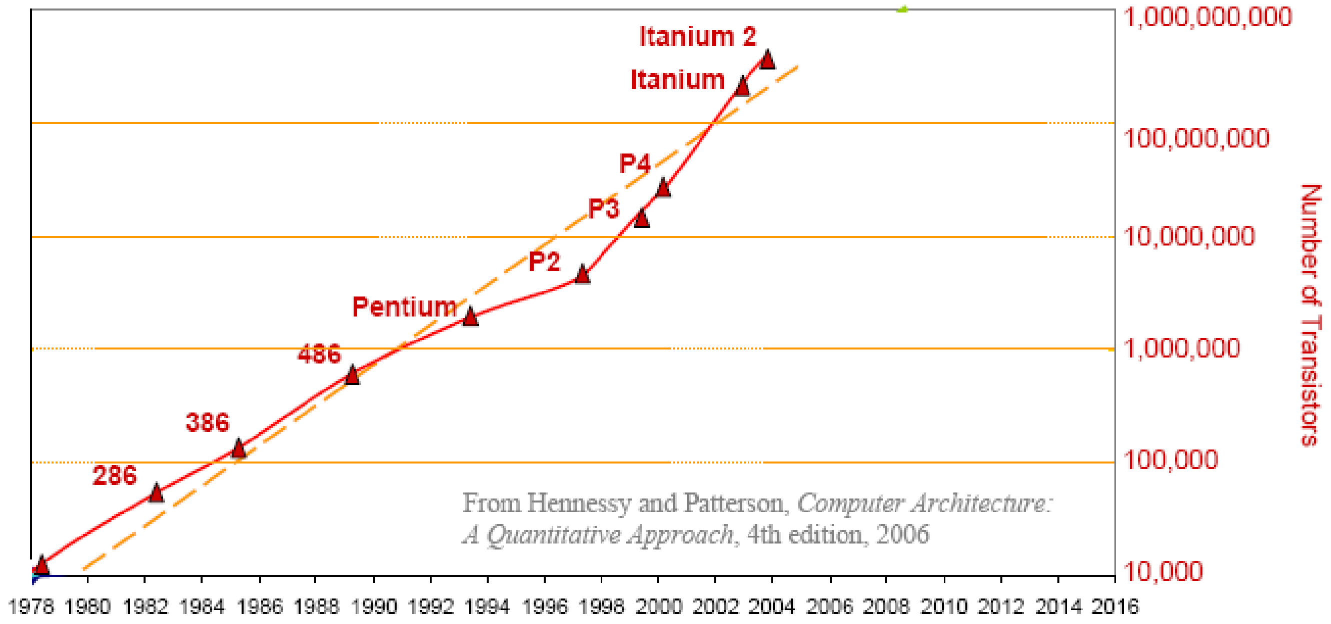
Instituto Superior de **Engenharia** do Porto

Preliminary reading

- This module requires a basic understanding of computer architecture and the C programming language. Previous knowledge of concurrency is recommended.
- Although not required, preliminary analysis of the following online resources help with understanding the concepts that will be discussed in the classes:
  - Learning openmp eBook, https://riptutorial.com/ebook/openmp
  - OpenMP in Small Bites, https://hpc-wiki.info/hpc/OpenMP_in_Small_Bites

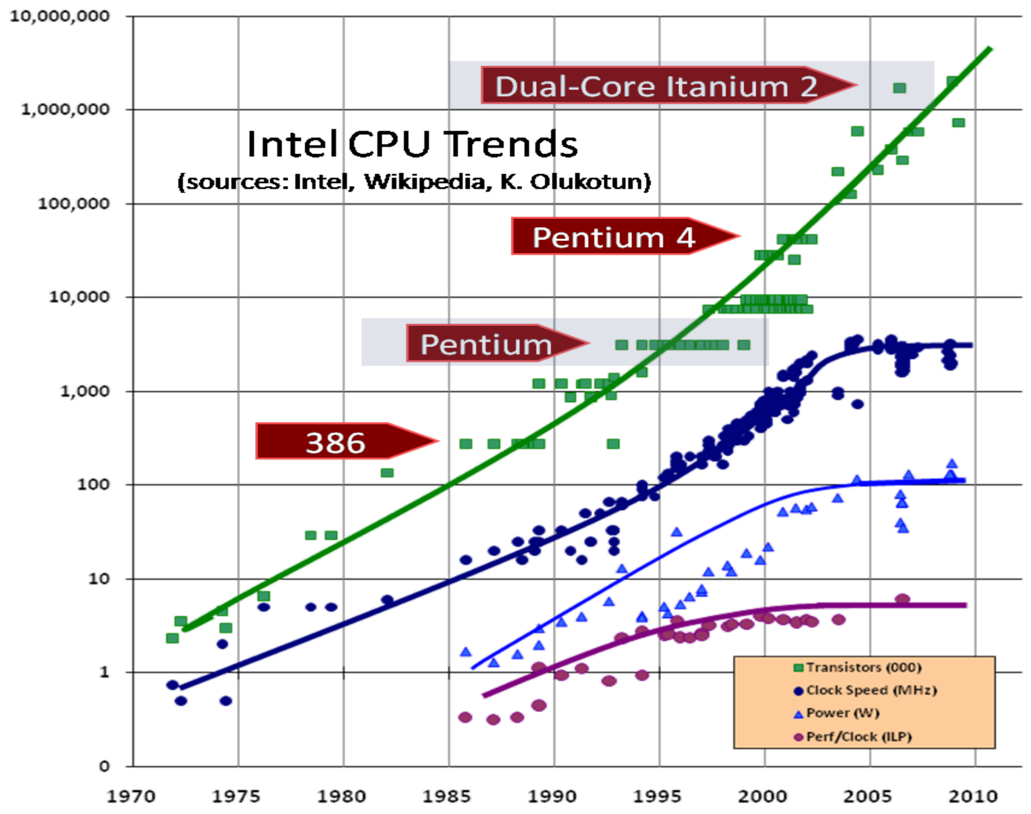
Importance of Parallelism

- Moore's Law: The quantity of transistors that can be placed on an integrated circuit has doubled approximately every two years (1965, revised 1975).



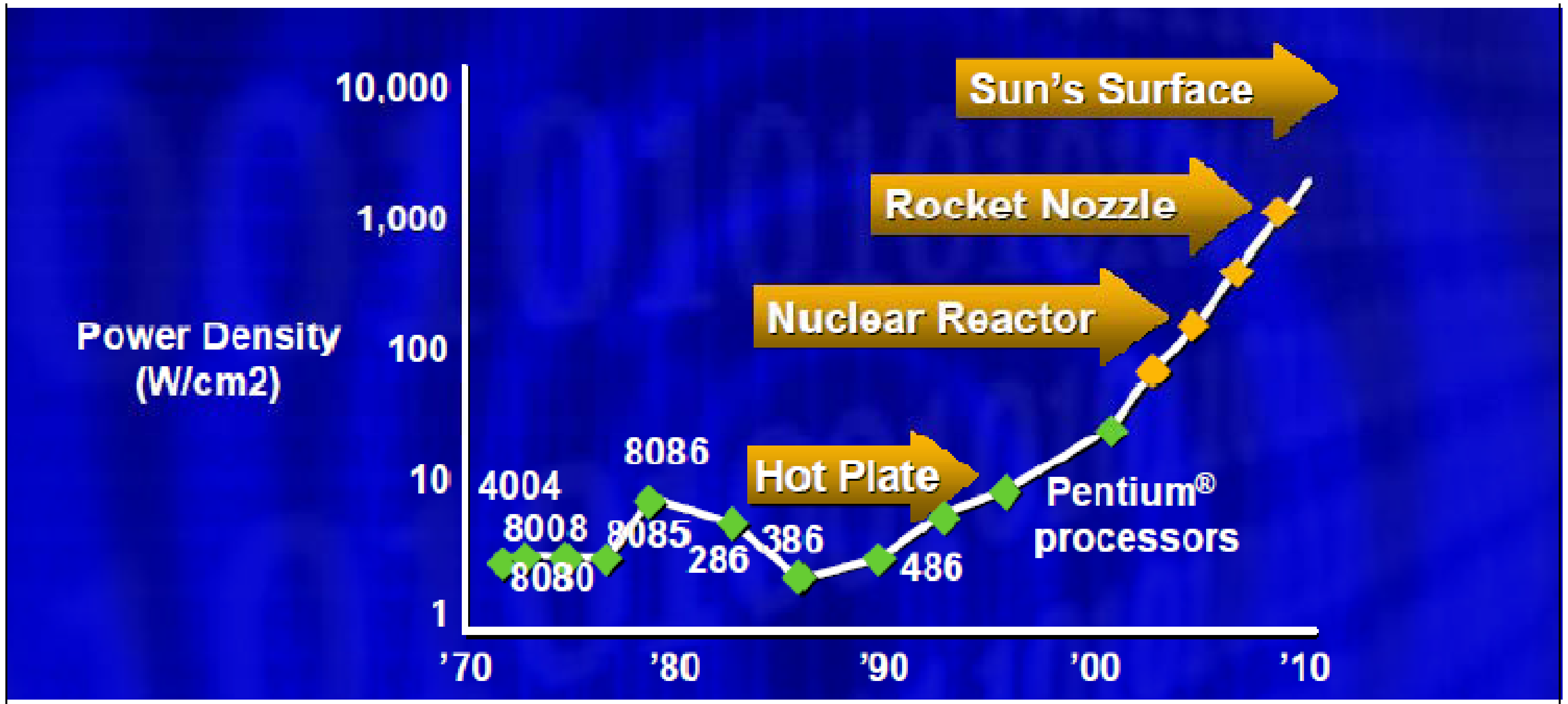
Importance of Parallelism

- At the same time processors were increasingly faster
  - Until ~2005



Importance of Parallelism

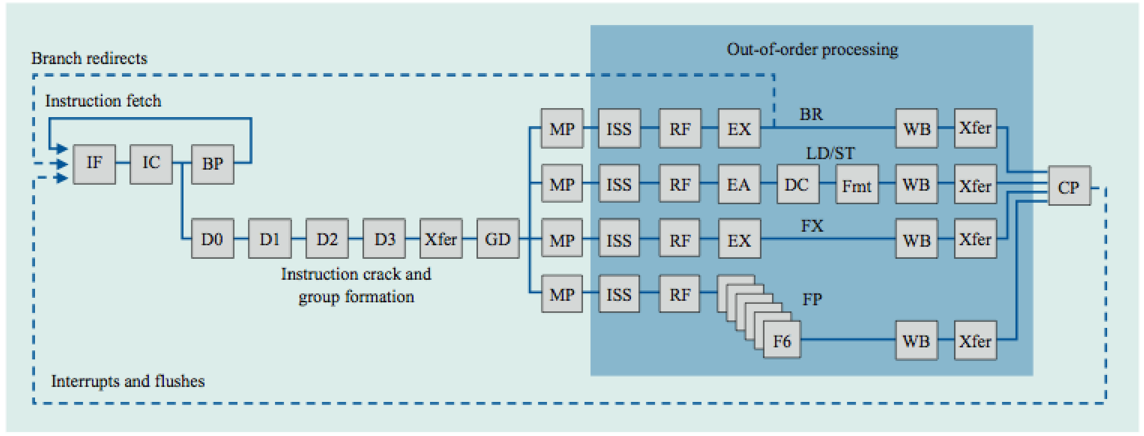
- The Power Wall



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Importance of Parallelism

- Memory wall
  - Access times to memory are another wall
  - Memory technology is also changing
    - Fast is expensive
  - Power efficiency is also an issue
- Instruction-level parallelism (ILP) wall
  - In the ’80s: expansion
    - Improvement in performance: 50%/year
    - Implicit parallelism (pipeline processor: 10 CPI -> 1 CPI)
  - The ’90s: diminishing returns
    - Smaller improvement in implicit parallelism
      - out-of-order issue, branch prediction: 1 CPI -> 0.5 CPI
  - performance below expectations



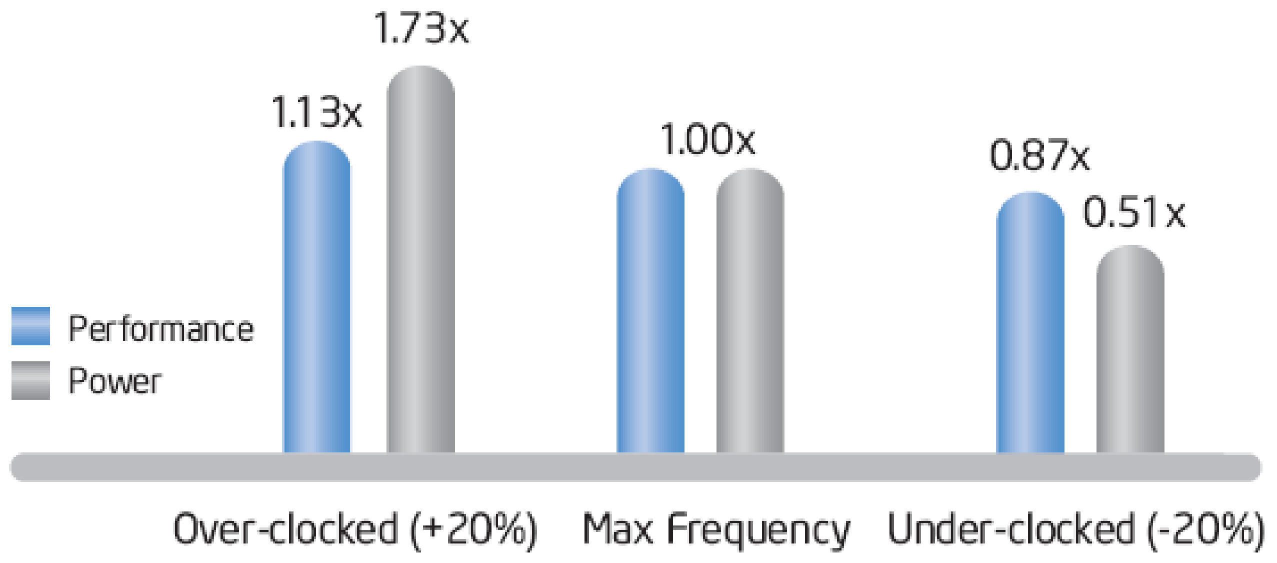
Importance of Parallelism

- Power wall
  - Cannot clock processors faster
- Memory wall
  - Performance is dominated by memory access times
- Instruction-level Parallelism (ILP) wall
  - No more extra work to keep functional units busy

- But the IT industry depends on increased performance
  - Software developers also
  - Solution was to parallelize CPUs

Importance of Parallelism

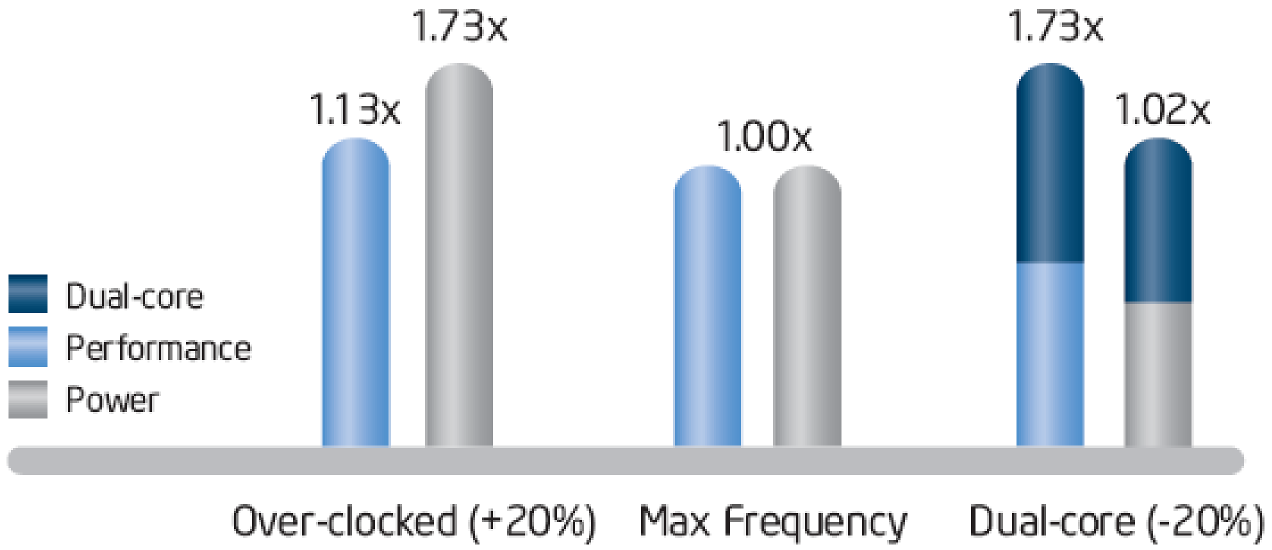
- Relation between speed and performance



Source: Intel® Multi-Core Processors - Making the Move to Quad-Core and Beyond

Importance of Parallelism

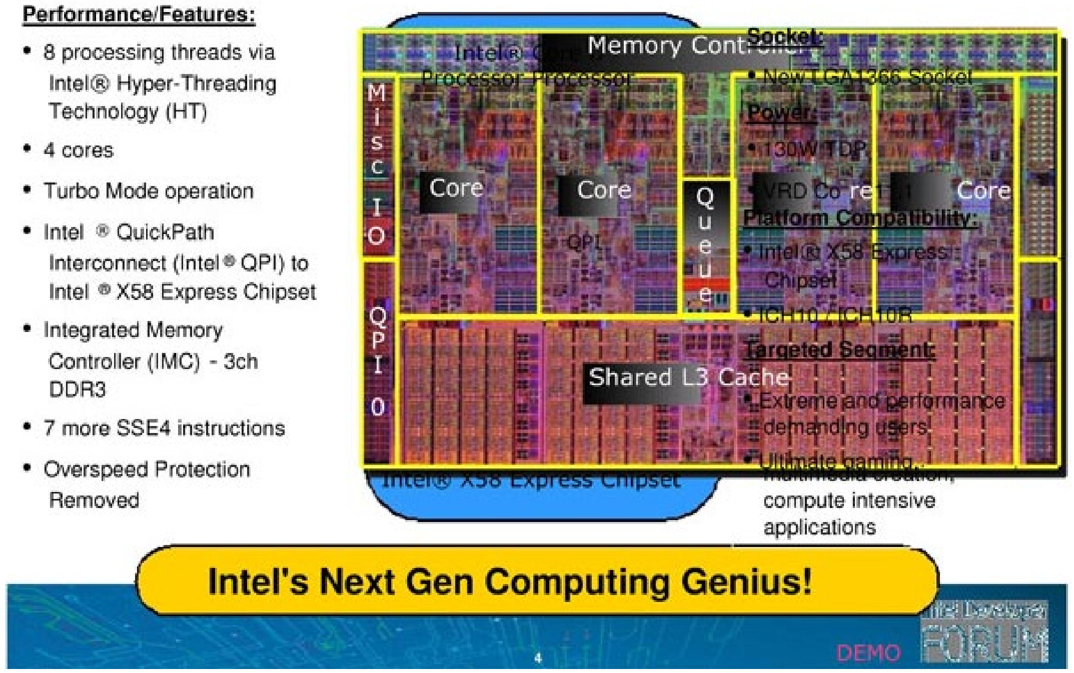
- Relation between speed and performance



Source: Intel® Multi-Core Processors - Making the Move to Quad-Core and Beyond

Importance of Parallelism

- Intel Core i7



Performance/Features:

- 8 processing threads via Intel® Hyper-Threading Technology (HT)
- 4 cores
- Turbo Mode operation
- Intel ® QuickPath Interconnect (Intel ® QPI) to Intel ® X58 Express Chipset
- Integrated Memory Controller (IMC) - 3ch DDR3
- 7 more SSE4 instructions
- Overspeed Protection Removed

Intel® Core Processor

Socket:

- New LGA1366 Socket

Power:

- 130W TDP
- VRD 11.1

Platform Compatibility:

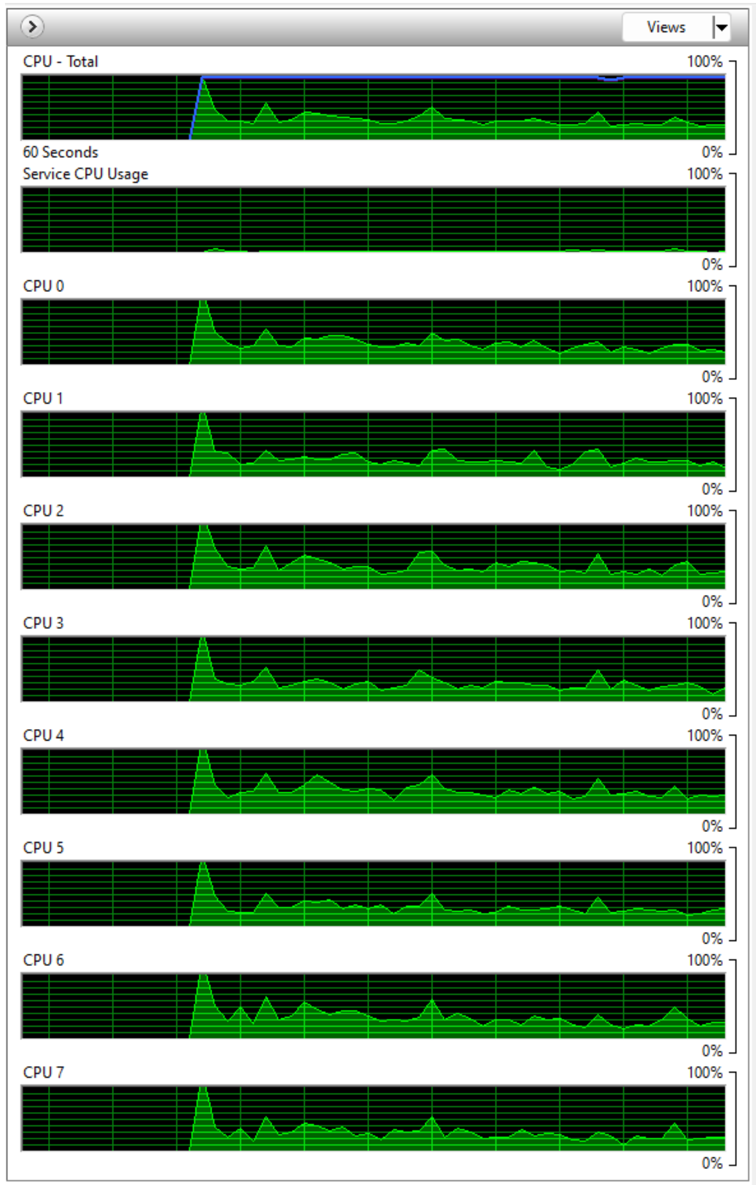
- Intel® X58 Express Chipset
- ICH10 / ICH10R

Targeted Segment:

- Extreme and performance demanding users
- Ultimate gaming, multimedia creation, compute intensive applications

Intel® X58 Express Chipset

Intel's Next Gen Computing Genius!



Importance of Parallelism

- During decades, programmers had the processing power for free
  - HW designers had to cope with increasing requirements for processing
  - SW engineers using higher-level concurrency and parallel oblivion tools

- 2005: The Free Lunch Is Over
  - A Fundamental Turn Toward Concurrency in Software, Herb Sutter, Microsoft

- HW becomes flexible
  - HW designers can focus on parallelizing simpler cores
  - And SW engineers need to adapt

Importance of Parallelism

- Programmers are now unaware of architectures
  - Solid boundary between Hardware and Software
  - Programmers don’t have (want) to know anything about the processor
- High level languages abstract away the processors
  - Java bytecode, .Net IL, is machine independent
- Moore’s law does not require the programmers to know anything about the processors to get good speedups
  - Programs are oblivious of the processor (work on all processors)
- A program written in ’70 using C still works and is much faster today
  - This abstraction provides a lot of freedom for the programmers

Importance of Parallelism

- The Third Software Crisis
  - Problem: Sequential performance is left behind by Moore's law
- Needed continuous and reasonable performance improvements
  - to support new features
  - to support larger datasets
- While sustaining portability, malleability and maintainability without unduly increasing complexity faced by the programmer
  - critical to keep-up with the current rate of evolution in software
- **We need to start thinking parallel, and we need better tools**
  - Era of programmers not caring about what is under the hood is over
    - A lot of variations/choices in hardware, many with performance implications
    - Understanding the hardware will make it easier to make programs get high performance
  - But: If program is too closely tied to the processor cannot port or migrate
    - Not as bad as assembly programming, but need to be aware of underlying mapping to the architecture

Importance of Parallelism

- Implicit approaches
  - ILP, OoO, TLP
  - Hyperthreading
  - Parallelizing compiler
- Explicit approaches
  - Processes
  - Threads
  - Light-weight threads (tasks)
- Implicit approaches
  - Specify what to parallelize, not how (if possible)

Focus of the course

Limits of Parallelism

- Amdhal’s Law: If P is the proportion of a program that can be made parallel, and (1 − P) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{(1-P)+\frac{P}{N}}$$

- If in my program 80% is parallelizable, then
  - If N= 2   speedup is 1,67
  - If N= 8   speedup is 3,33
  - If N= 16 speedup is 4
  - If N= 32 speedup is 4.44

Limits of Parallelism

- Gustafson’s law:

$$S(P) = P - \alpha.(P-1)$$

- Gustafson's law addresses the shortcomings of Amdahl's law, which does not scale the availability of computing power to do more work as the number of machines increases.
- Gustafson's Law proposes that programmers set the size of problems to use the available equipment to solve problems within a practical fixed time.
  - Therefore, if faster (more parallel) equipment is available, larger problems can be solved whilst current ones take the same time.
  - Amdahl's law: fixed workload - the sequential part of a program does not change with respect to number of processors; the parallel part is evenly distributed by the processors.
  - Gustafson's law: increase computation will lead to increased workload, so higher speedup

Decomposition of problems

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time, and/or
  - To solve bigger problems, and/or
  - To achieve better solutions
- The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Decomposition of problems

- Flow
  - Find concurrent activities in the problem
  - Structure the algorithm to exploit parallelism
  - Implement the algorithm in a suitable programming environment
  - Execute and tune the performance of the code on a particular system

Finding Parallelism in Problems

- Identify a decomposition of the problem into sub-problems that can be solved simultaneously
  - A task decomposition that identifies tasks for potential concurrent execution
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering the groups to satisfy temporal constraints
  - An analysis on the data sharing patterns among the concurrent tasks
  - A design evaluation that assesses of the quality the choices made in all the steps

Decomposition issues

- Finding and exploiting parallelism often requires looking at the problem from a non-obvious angle
  - Contrary to many common problem solving strategies
- Dependences need to be identified and managed
  - The order of execution may change the result
    - Obvious: One step feeds result to the next steps
    - Subtle: accuracy may be affected by ordering steps that are logically parallel with each other
- Performance can be reduced by
  - Parallel overhead
  - Load imbalance
  - Inefficient data sharing
  - Contention on critical resources such as memory access
- Shared memory parallel programming still dominant in multicore/multiprocessor systems

Decomposition issues

- Focus on the longest running parts of the program first
  - be realistic about possible speedups
  - different parts may need to be parallelized with different techniques
- Understand the different resource requirements of a program computation, communication, and locality
  - Consider how data accesses interact with the memory system:
  - will the computation done on additional cores pay for the data to be brought to them?
- The performance of one thread is highly dependent on what other threads are doing
  - It's important to understand the h/w of the machine, and the techniques it uses
- For modest numbers of cores, it may be possible to identify large granularity tasks, with good data locality
  - With modest numbers of cores, beware of overheads

Decomposition issues

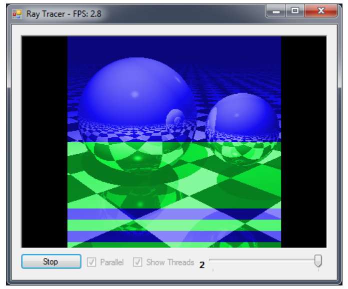
- Data parallelism
  - divide data among the processors.
  - processors must exchange boundary information at regular intervals.
- Task parallelism
  - divide tasks among the processors.
  - sometimes can be “embarrassingly parallel” with very little inter-process communication required.

Data-based decomposition

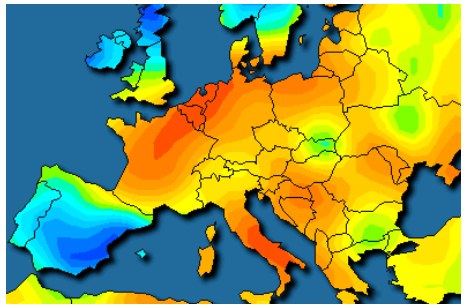
- The most compute intensive parts of many large problem manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner.
  - This is what data decomposition is optimized for.
- The simpler abstraction to use when trying to parallelize
  - In order to decompose the problem domain, take the entire set of data and break it into smaller, discrete portions, or chunks.
  - Then work on each chunk in the data set in parallel.
- The data decomposition should lead to
  - Efficient data usage by tasks within the partition
  - Few dependencies across the tasks that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics

Data intensive problems

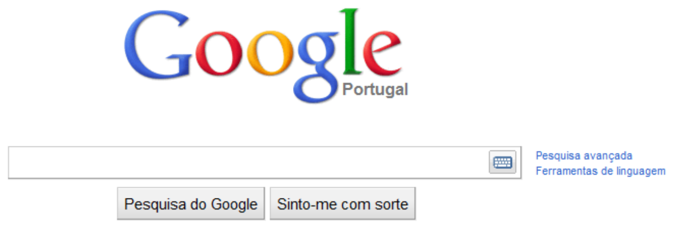
- Rendering in 3D
  - e.g., ray tracing



- Modeling of complex physical systems
  - e.g., weather prediction



- Analysis of massive datasets
  - e.g., web search

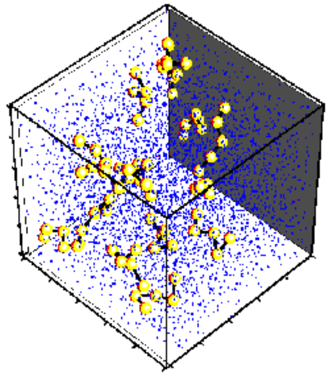


Task-based decomposition

- Focused on the individual tasks that need to be performed instead of focusing on the data
  - In order to decompose by tasks, we need to think about our algorithm in terms of discrete operations, or tasks, which can then later be parallelized.
- In practice, more difficult than data decomposition.
  - Look at what the algorithm actually does, and how it performs its actions.
  - Analyze the steps of the algorithm and determine whether there are any constraints in terms of shared data or ordering.
  - There is no silver bullet approach.
- But, many large problems can be naturally decomposed into tasks
  - The number of tasks used should be adjustable to the execution resources available.
  - Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.
  - Tasks should maximize reuse of sequential program code to minimize effort.
- Data and Task Decomposition are not mutually exclusive.
  - Often, the two approaches coexist while trying to parallelize an application.
  - Decompose the problem based on data, then further decompose the processing of each element of data based on tasks.

Task intensive problems

- Dynamic Systems
  - e.g., molecular dynamics

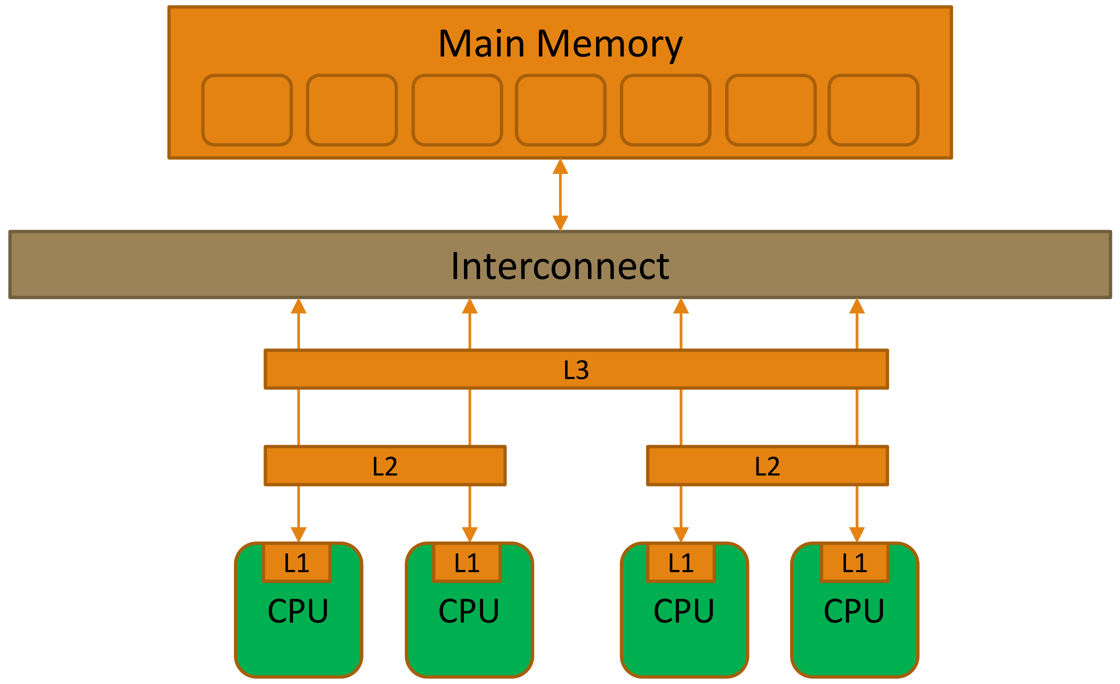


- Wavefront computation
  - e.g., Computer Generated Holography

- Any problem with a dynamic data set

Shared Memory Systems (again)

- In this course, we will focus on shared memory systems
  - All cores can access the same physical memory of the system
  - Although it is usually not that simple … later we will discuss the impact of caches

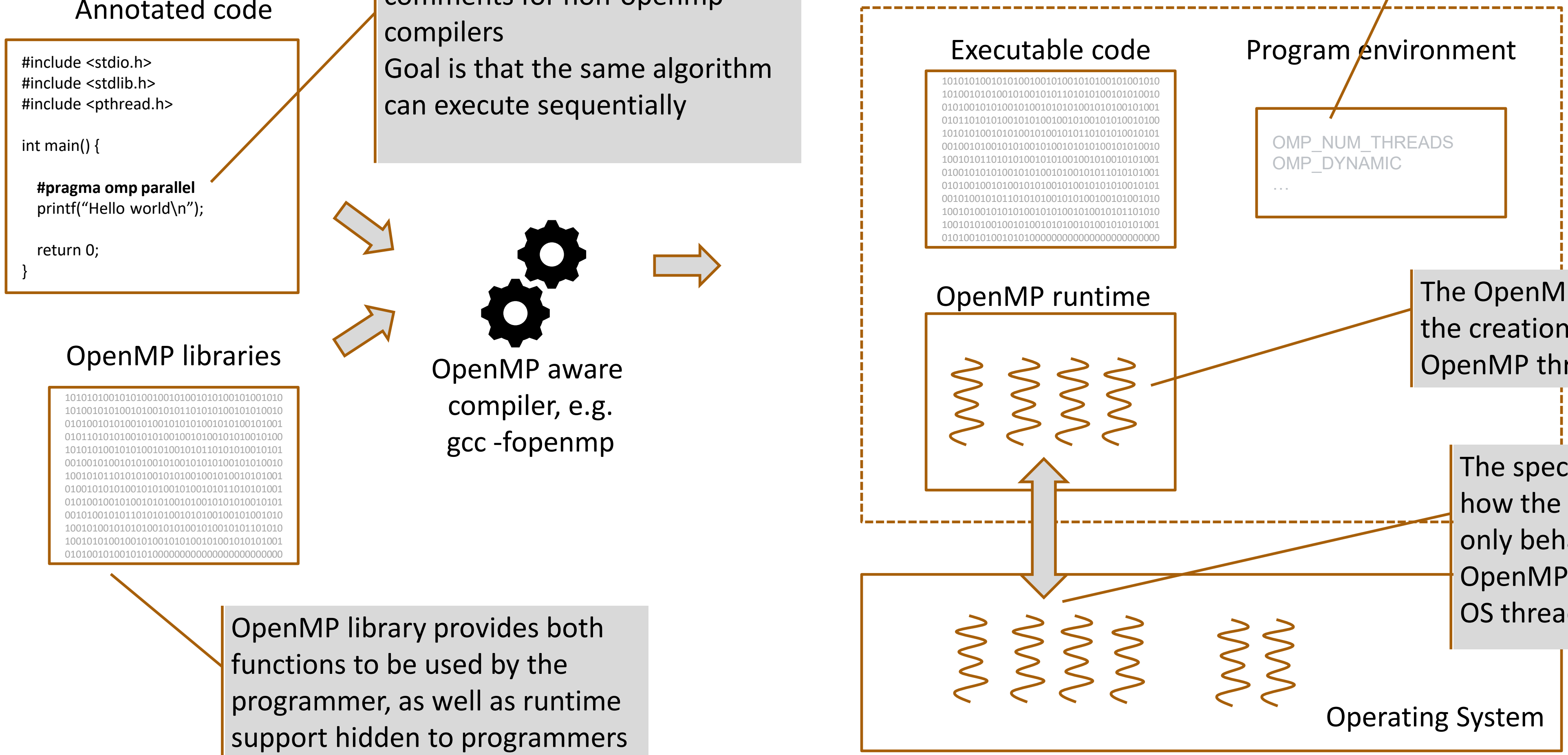


- We will not deal with heterogenous systems or distributed memory systems

OpenMP Basics

- OpenMP (www.openmp.org) is a specification from a consortium (OpenMP Architecture Review Board)
  - Membership includes the main companies and academic organizations in the high-performance computing domain
- The specification provides
  - an annotation-based (compiler directives) approach for specifying parallelism
  - the required libraries, and
  - runtime environment for the execution of parallel computation.
- The focus is portability across shared memory systems, programmed in C/C++ or Fortran
- It is one of the most common parallel programming models in the high-performance computing domain, being increasingly used in embedded computing systems

OpenMP Basics



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main() {

   #pragma omp parallel
   printf("Hello world\n");

   return 0;
}
```

Annotation in code are comments for non-openmp compilers
Goal is that the same algorithm can execute sequentially

OpenMP library provides both functions to be used by the programmer, as well as runtime support hidden to programmers

Some parameters (e.g. number of parallel threads) can be specified in the environment context

The OpenMP runtime manages the creation and execution of the OpenMP threads

The specification does not specify how the runtime is implemented, only behavior. This includes how OpenMP threads are mapped to OS threads (usually 1-to-1)

OpenMP Basics

- Basic OpenMP is based on the concept of parallel regions and fork-join computation

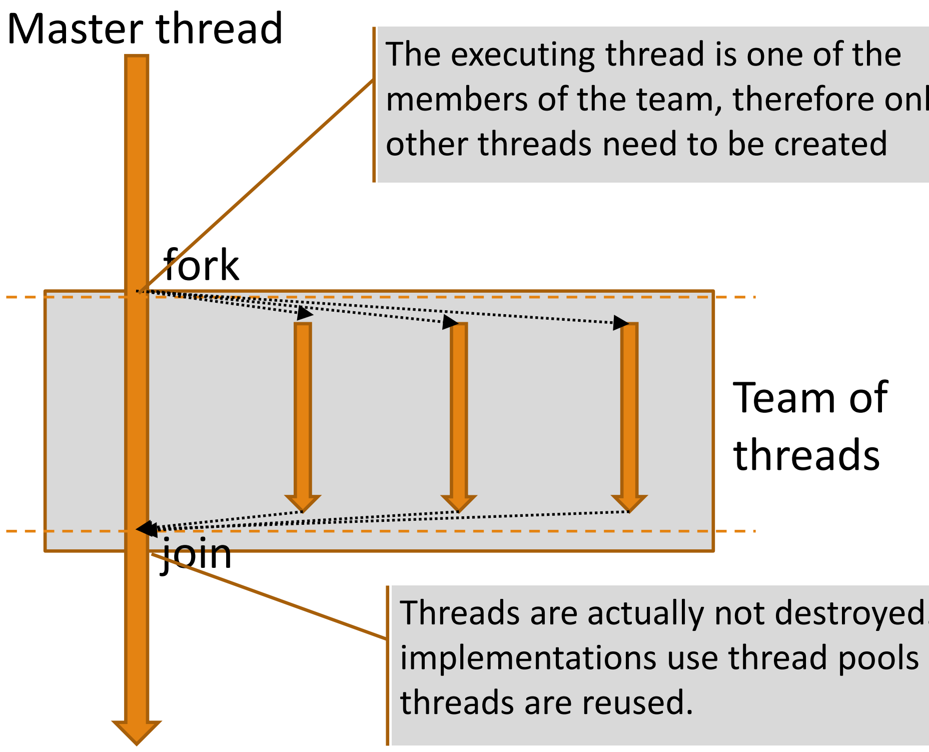
```
int main() {
   // single thread
   #pragma omp parallel
   {
      // parallel region
   }
   // single thread
   return 0;
}
```

Initially only one thread exists

When a parallel region (a block of code) is declared in the program, the runtime creates a team of threads to execute the region (execution forks)

All threads execute the code

At the end of the parallel region all threads need to join before the computation continues with only the initial thread



The executing thread is one of the members of the team, therefore only 3 other threads need to be created

Threads are actually not destroyed. OpenMP implementations use thread pools so threads are reused.

OpenMP Basics

#include <omp.h> required due to omp_get_thread_num

Team of 4 threads (default is number of logical cores)

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

int main() {

    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        printf("Hello world! I am thread %d\n", myid);
    }
    printf("Hello world! I am the main thread \n");

   return 0;
}
```

Order of execution inside the parallel region is arbitrary

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 3
Hello world! I am thread 0
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am the main thread
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 0
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 1
Hello world! I am the main thread
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 1
Hello world! I am thread 0
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am the main thread
```

"Hello world! I am the main thread" always last – it is after the parallel region

Each thread is assigned a unique id, between 0 (always the master) and number of threads minus 1

OpenMP Basics

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

int main() {

   #pragma omp parallel
   {
      int myid = omp_get_thread_num();
      if(myid == 0)
         printf("Hello world! I am the main thread (thread 0) \n");
      else
         printf("Hello world! I am thread %d\n", myid);
   }


  return 0;
}
```

Order of execution inside the parallel region is arbitrary

"Hello world! I am the main thread" is now printed as one of the team

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp
user@ubuntu:~/omp$ ./a.out
Hello world! I am the main thread (thread 0)
Hello world! I am thread 3
Hello world! I am thread 1
Hello world! I am thread 2
user@ubuntu:~/omp$ ./a.out
Hello world! I am the main thread (thread 0)
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am thread 1
user@ubuntu:~/omp$ ./a.out
Hello world! I am the main thread (thread 0)
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 1
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am the main thread (thread 0)
Hello world! I am thread 1
```

OpenMP Basics

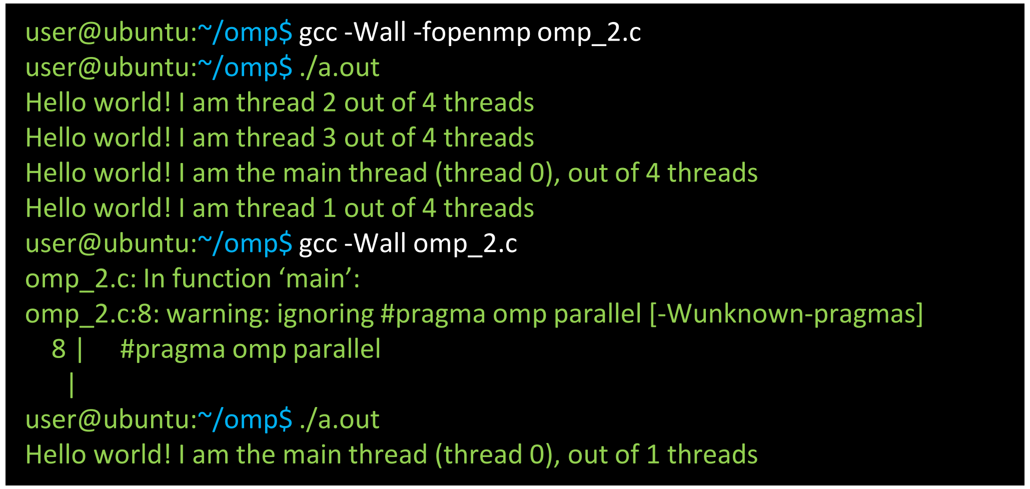
- Remember the pthreads equivalent

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

void* hello_world(void* id) {
  long myid = * (long*) id;
  printf("Hello world! I am thread %ld\n", myid);
  return NULL;
}

int main() {
  long id1 = 1, id2 = 2;
  pthread_t thread_id1, thread_id2;

  pthread_create(&thread_id1, NULL, hello_world, &id1);
  pthread_create(&thread_id2, NULL, hello_world, &id2);

  printf("Hello world! I am the main thread \n");

  pthread_join(thread_id1, NULL);
  pthread_join(thread_id2, NULL);

  return 0;
}
```

OpenMP Basics

```
#include <stdio.h>
#ifdef _OPENMP
   # include <omp.h>
#endif

int main() {

   #pragma omp parallel
   {
      #ifdef _OPENMP
         int myid = omp_get_thread_num();
         int tcount = omp_get_num_threads ( );
      #else
         int myid = 0;
         int tcount = 1;
      #endif
      if(myid == 0)
         printf("Hello world! I am the main thread (thread %d), out of %d threads \n", myid, tcount);
      else
         printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
   }

   return 0;
}
```

There are limits to compiler ignoring parallelism, if using openmp library calls



```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_2.c
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 2 out of 4 threads
Hello world! I am thread 3 out of 4 threads
Hello world! I am the main thread (thread 0), out of 4 threads
Hello world! I am thread 1 out of 4 threads
user@ubuntu:~/omp$ gcc -Wall omp_2.c
omp_2.c: In function ‘main’:
omp_2.c:8: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
    8 |     #pragma omp parallel
      |
user@ubuntu:~/omp$ ./a.out
Hello world! I am the main thread (thread 0), out of 1 threads
```

Sharing Data

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

#define SIZE 1000
int main() {
   long vector[SIZE];
   long long sum;

   for(int i = 0; i < SIZE; i++)
            vector[i] = i + 1;

   sum = 0;
   #pragma omp parallel
   {
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
            sum += vector[i];
   }

   printf("Average = %f\n", ((double)sum)/SIZE);
   return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3.c
user@ubuntu:~/omp$ ./a.out
Average = 313.976000
user@ubuntu:~/omp$ ./a.out
Average = 304.760000
user@ubuntu:~/omp$ ./a.out
Average = 160.530000
user@ubuntu:~/omp$ ./a.out
Average = 347.125000
```

As usual, we are assuming SIZE to be multiple of N_THREADS.

Variables my_index, n_threads and i are declared inside the block, therefore they are local to the block, and to the thread

Variable sum is being updated by several threads in parallel, there is a race condition

Mutual Exclusion

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

#define SIZE 1000
int main() {
    long vector[SIZE];
    long long sum;

    for(int i = 0; i < SIZE; i++)
             vector[i] = i + 1;

    sum = 0;
    #pragma omp parallel
    {
        int my_index = omp_get_thread_num();
        int n_threads = omp_get_num_threads ( );
        for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
             #pragma omp critical
             sum += vector[i];
    }

    printf("Average = %f\n", ((double)sum)/SIZE);
    return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3.c
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
```

Mutual exclusion: OpenMP provides a directive that guarantees that the following block/line is executed by only one thread at a time

However, this also means that there will be SIZE accesses to the shared variable, one at a time -> **this is a sequential program**

Side note: this looks like Python, but there is no relation with indentation. In C, if there are no brackets, the line is the block of code. And #prama is a compiler directive, not an instruction

Mutual Exclusion

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

#define SIZE 1000
int main() {
   long vector[SIZE];
   long long sum;

   for(int i = 0; i < SIZE; i++)
            vector[i] = i + 1;

   sum = 0;
   #pragma omp parallel
   {
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      long long local_sum = 0;
      for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
         local_sum +=vector[i];

      #pragma omp critical
      sum += local_sum;
   }

   printf("Average = %f\n", ((double)sum)/SIZE);
   return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3.c
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
user@ubuntu:~/omp$ ./a.out
Average = 500.500000
```

Working with local variables allows reducing contention.

Only update shared variables when actually needed!

Mutual Exclusion

```c
#include <stdio.h>
#include <omp.h>
#define SIZE 1000000

int main() {
   long vector[SIZE];
   long long sum;

   for(int i = 0; i < SIZE; i++)
             vector[i] = i + 1;

   sum = 0;
   double start = omp_get_wtime();
   #pragma omp parallel
   {
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
           #pragma omp critical
           sum += vector[i];
   }
   double end = omp_get_wtime();

   printf("Average = %f\n", ((double)sum)/SIZE);
   printf("Work took %f seconds\n", end - start);

   return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3.c
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.105183 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.099528 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.075229 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.096032 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.093436 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.075797 seconds
```

Mutual Exclusion

```
#include <stdio.h>
#include <omp.h>
#define SIZE 1000000

int main() {
   long vector[SIZE];
   long long sum;
   for(int i = 0; i < SIZE; i++)
             vector[i] = i + 1;

   sum = 0;
   double start = omp_get_wtime();
   #pragma omp parallel
   {
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      long long local_sum = 0;
      for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
         local_sum +=vector[i];

      #pragma omp critical
      sum += local_sum;
   }
   double end = omp_get_wtime();

   printf("Average = %f\n", ((double)sum)/S
   printf("Work took %f seconds\n", end - s

   return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3b.c
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.003022 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.002093 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.003789 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.003210 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.001869 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.001909 seconds
```

3-5 times faster

Mutual Exclusion

```c
#include <stdio.h>
#include <omp.h>
#define SIZE 1000000

int main() {
   long vector[SIZE];
   long long sum;

   for(int i = 0; i < SIZE; i++)
            vector[i] = i + 1;

   sum = 0;

   double start = omp_get_wtime();
   int my_index = 0;
   int n_threads = 1;
   for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
      sum += vector[i];

   double end = omp_get_wtime();

   printf("Average = %f\n", ((double)sum)/SIZE);
   printf("Work took %f seconds\n", end - start);
   return 0;
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp omp_3c.c
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.006381 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.004043 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.002905 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.006388 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.002935 seconds
user@ubuntu:~/omp$ ./a.out
Average = 500000.500000
Work took 0.006408 seconds
```

Sequential in some cases almost similar to parallel. There isn’t much work to do, and there is the overhead of parallel execution

Clauses

- In OpenMP directives can have clauses, which either change or define behavior of the directive

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

#define SIZE 1000
int main() {
   long vector[SIZE];
   long long sum;

   for(int i = 0; i < SIZE; i++)
            vector[i] = i + 1;
   sum = 0;
   #pragma omp parallel num_threads(2)
   {
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      long long local_sum = 0;
      for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++)
         local_sum +=vector[i];

      #pragma omp critical
      sum += local_sum;
   }

   printf("Average = %f\n", ((double)sum)/SIZE);
   return 0;
}
```

The num_threads clause requests to OpenMP the parallel region to be a team of X threads (2 in this case)

There are other ways to control the number of threads in a team:
- OMP_NUM_THREADS environment variable
- omp_set_num_threads library routine

The actual number of threads may depend on availability and parallel parallel regions. More later.

Reduction Clauses

- The pattern of calculating local values and later reducing to a single value is very common
  - OpenMP allows to specify these reductions through clauses, which define the operation and the reduction variable
    - The operator is a binary operation (such as addition or multiplication), needs to be associative
    - The reduction is repeatedly applying the same operator to a sequence of operands in order to get a single result.
    - The intermediate results are stored in a local copy of the reduction variable (this needs to exist outside of the parallel region)
    - OpenMP implicitly deals with
      - Creating thread private variables and Initialization with the neutral element of the variable type
      - The mutual exclusion to update the shared variable

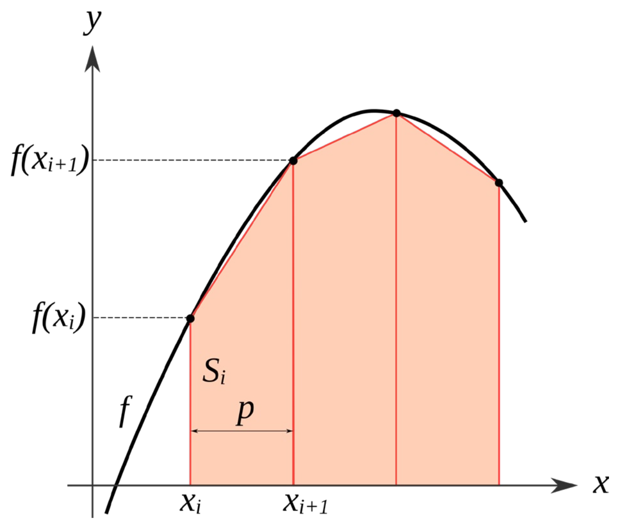
```
#pragma omp parallel num_threads(4) reduction(+:sum)
{
    for(int i = 0; i < N; i++)
        sum +=vector[i];
}
```

Programmer is responsible to guarantee that the reduction operation is the correct one for the algorithm

You can compare the efficiency of reduction with implementation of explicitly using a local sum

Hands-on

- Exercise: parallelize the trapezoidal rule to calculate the integral



$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k .$$

Source: wikipedia

```
double integral (double a, double b, int n){
  double x, p, sum=0, integral;

  p=fabs(b-a)/n;

  for(int i=1; i<n; i++){
    x = a + i * p;
    sum=sum + func(x);
  }

  integral=(p/2)*(f(a)+f(b)+2*sum);

  return integral;
}
```

Worksharing constructs: Parallel loops

- A worksharing construct distributes the execution of the corresponding region among the members of the parallel team
- Probably the most used construct is the parallel loop
  - Iterations of (one or more) associated loops will be executed in parallel by the threads in the team of the parallel region

```
#pragma omp parallel
{
        int i;
        #pragma omp for
        for (i=0;i<N;i++){
                // iterations
        }
}
```

Iterations of the loop will be split among the threads. Needs to be followed by an for loop conforming to a **canonical form**

Each thread gets its own i variable, even if declared before

Iterations should be independent (more later), and program cannot rely on any specific order between iterations

Worksharing constructs: Parallel loops

```
for(i=0;i< N;i++) {
        a[i] = a[i] + b[i];
}
```

Sequential code

```
#pragma omp parallel
{
      int my_index = omp_get_thread_num();
      int n_threads = omp_get_num_threads ( );
      for(int i = my_index*N/n_threads; i < (my_index+1)*N/n_threads; i++)
          a[i] = a[i] + b[i];
}
```

Explicit thread handling in parallel region

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

Parallel region with a worksharing loop

Worksharing constructs: Parallel loops

- This is usually the first approach to try to speedup a program through parallelization
  - Find time consuming loops in the code
  - Make iterations independent
  - Add a parallel region with a worksharing loop

```
int i, j, A[SIZE];
j = 5;
for (i=0;i< SIZE; i++) {
        j +=2;
        A[i] = func(j);
}
```

```
int i, j, A[SIZE];
#pragma omp parallel for
for (i=0;i< SIZE; i++) {
        int j = 5 + 2*(i+1);
        A[i] = func(j);
}
```

Add a parallel loop (both directives can be collapsed in one pragma)

Change calculation of j to make iterations independent

Hands-on

- Implement a parallel implementation of the π approximation with the Leibniz series

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}.$$

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
#include <stdio.h>

int main ()
{
   int i; double pi, sum = 0.0;
   static long num_steps = 1000000;
   double factor = 1.0;

   for (i=0;i< num_steps; i++){
      sum = sum + r/(2*i+1);
      factor = -factor;
   }
   pi = 4.0 * sum;
   printf("Pi = %.10f", pi);
}
```

Worksharing constructs: Parallel loops

- It is possible to control how iterations are mapped on threads through the “schedule” clause
  - schedule(static [,chunk])
    - Map fixed blocks of iterations of size “chunk” to each thread
    - If chunk is not specified, the iteration size is divided in chunks approximately equal in size, and one per thread (~SIZE/N_THREADS)
      - Different implementations may approximate differently
    - Almost no runtime overhead
    - Should be used when computation time is similar per iteration and predictable by the programmer
  - schedule(dynamic[,chunk])
    - Each thread fetches “chunk” iterations off a queue until all iterations have been handled
    - If chunk is not specified, it defaults to 1
    - More runtime overhead, allows to cope with varying computation time
  - schedule(guided[,chunk])
    - Similar to dynamic, but the size of the block starts large and shrinks down to size “chunk” as the calculation proceeds
    - The size of each chunk is proportional to the number of unassigned iterations divided by the number of threads
    - If chunk is not specified, it defaults to 1
    - Allows to reduce runtime overhead (sometimes)

Worksharing constructs: Parallel loops

- It is possible to control how iterations are mapped on threads through the “schedule” clause
  - schedule(runtime)
    - Type of schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library)
    - Allows the schedule to be changed during execution
  - schedule(auto)
    - Schedule is left up to the runtime to choose
    - Can be completely different, implementations can provide some “learning” approaches (theoretically)

Worksharing constructs: Parallel loops

- For nested loops, it is possible to parallelize any or several of the loops
  - But need to carefully plan parallel regions, so placement of the worksharing construct impacts the efficiency and speedup of the program
- For perfectly nested rectangular loops it is possible to collapse into a single loop (length NxM) and parallelize the collapsed loop with the collapse clause

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
        for (int j=0; j<M; j++) {
                .....
        }
}
```

Number of loops to collapse, starts from outer loop

Worksharing constructs: Sections

- The Sections construct allows to specify different code for each thread
  - Allows for irregular parallel programming

```
#pragma omp parallel num_threads(3)
{
   #pragma omp sections
   {
      #pragma omp section
      {
         int myid = omp_get_thread_num();
         int tcount = omp_get_num_threads ( );
         printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
      }
      #pragma omp section
      {
         int myid = omp_get_thread_num();
         int tcount = omp_get_num_threads ( );
         printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
      }
   }
}
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp sect.c
user@ubuntu:~/omp$ ./a.out
Hello world! I am thread 2 out of 3 threads
Hello world! I am thread 1 out of 3 threads
```

Note that one of threads of the team is not used (but exists)

Worksharing constructs: Sections

- The Sections construct allows to specify different code for each thread
  - Allows for irregular parallel programming

```
#pragma omp parallel sections num_threads(3)
{
      #pragma omp section
      {
         int myid = omp_get_thread_num();
         int tcount = omp_get_num_threads ( );
         printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
      }
      #pragma omp section
      {
         int myid = omp_get_thread_num();
         int tcount = omp_get_num_threads ( );
         printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
      }
 }
```

Again, both parallel and sections directives can be collapsed

Hands-on

- Implement a recursive Fibonnaci function using sections
- Change the Fibonacci function to cut parallelism after a certain number of levels
  - controlled by a static flag

Data Scoping Clauses

- OpenMP defines default storage attributes for data
  - Variables are either Shared (all threads see the same variable) or Private (each thread has its own)
  - Using C, variables are SHARED if
    - They have a wider scope than the parallel region (they were declared before)
    - They are static
    - They are dynamically allocated
  - Using C, variables are PRIVATE if
    - Declared inside the parallel region
    - Declared in a function called from a parallel region (these are in stack, therefore they need to be private to the thread)
- The default storage attributes can be changed by specific clauses
  - The change applies to the specific variables listed and only in the scope of the construct that has the clause
  - Or the default behavior can be changed with the default clause

Data Scoping Clauses

- shared (only available in the parallel construct)
  - The variables in the list are shared between the threads
  - The variable needs to exist when the clause is reached, therefore it is shared by default
  - The clause is useful when the default is changed
- private
  - Each thread has a local private copy of the variable
  - The variable is uninitialized, and it is not copied back to the original
- firstprivate
  - Each thread has a local private copy of the variable, which is initialized with the value of the shared variable when the clause was reached
- lastprivate
  - Each thread has a local private copy of the variable, the value of the last sequentially-equivalent value is propagated back to the original

Data Scoping Clauses

```
void wrong() {
        int tmp = 0;

        #pragma omp parallel for private(tmp)
        for (int j = 0; j < 1000; ++j)
                tmp += j;

        printf("%d\n", tmp);
}
```

Each thread has its own tmp variable, which is not initialized

Outside of the parallel region, the original tmp variable is still 0

```
int tmp = 0;

void wrong() {

        #pragma omp parallel for private(tmp)
        func();

        printf("%d\n", tmp);
}
```

```
extern int tmp;
void func() {
        for (int j = 0; j < 1000; ++j)
                tmp += j;
}
```

Not clear which tmp was used in this case, so result is unpredictable

Data Scoping Clauses

```
void wrong() {
        int tmp = 0;

        #pragma omp parallel for firstprivate(tmp)
        for (int j = 0; j < 1000; ++j)
                tmp += j;

        printf(“%d\n”, tmp);
}
```

Each thread has its own tmp variable, which is correctly initialized at 0

Outside of the parallel region, the original tmp variable is still 0

```
void wrong() {
        int tmp = 0;

        #pragma omp parallel for firstprivate(tmp) lastprivate(tmp)
        for (int j = 0; j < 1000; ++j)
                tmp += j;

        printf(“%d\n”, tmp);
}
```

Each thread has its own tmp variable, which is correctly initialized at 0

tmp is defined as the value at the “last sequential” iteration (i.e., for j=999)

Still not quite correct

Data Scoping Clauses

- default
  - Allows to change the default for variables in the clause
  - In C, only available default(shared) or default(none)
  - default(none) is a good way to guarantee safety – it is only possible to access global variables which are explicitly shared
  - default(shared) is the default, except for tasks (later)

```
int main() {

   int A, B;

   #pragma omp parallel default(none) shared(A)
   {
      printf("%d\n", A);
      printf("%d\n", B);
   }
   return 0;
}
```

Compiler error:
error: ‘B’ not specified in enclosing ‘parallel’

More Data Scoping

- A complementary directive exists to make global variables private to a thread
  - Each thread gives its own set of global variables, with initial values undefined
- Different from private
  - With private clause, global variables are hidden
  - threadprivate preserves global scope within each thread
  - Parallel regions must be executed by the same number of threads for global data to persist (and thread with the same id accesses the same private global variable across parallel regions)
- Threadprivate variables can be initialized using copyin clause or at time of definition.

```
int global[SIZE];
#pragma omp threadprivate(global)
int index = 0;
#pragma omp threadprivate(index)

int main(){
        for(int i= 0; i<100; i++) global[i] = i+2;
        #pragma omp parallel copyin(global)
         {
                // Each thread gets a copy of global and index, with the value initialized
         }
}
```

Hands-on

- Identify the printed values of the following code

```
int main()
{
        int A=1, B=1, C=1;
        #pragma omp parallel private(B) firstprivate(C)
        {
                #pragma omp single
                 {
                        printf("A = %d\n", A);
                        printf("B = %d\n", B);
                        printf("C = %d\n", C);
                        A = 0; B = 0; C = 0;
                }
        }
        printf("\nA = %d\n", A);
        printf("B = %d\n", B);
        printf("C = %d\n", C);
        return 0;
}
```

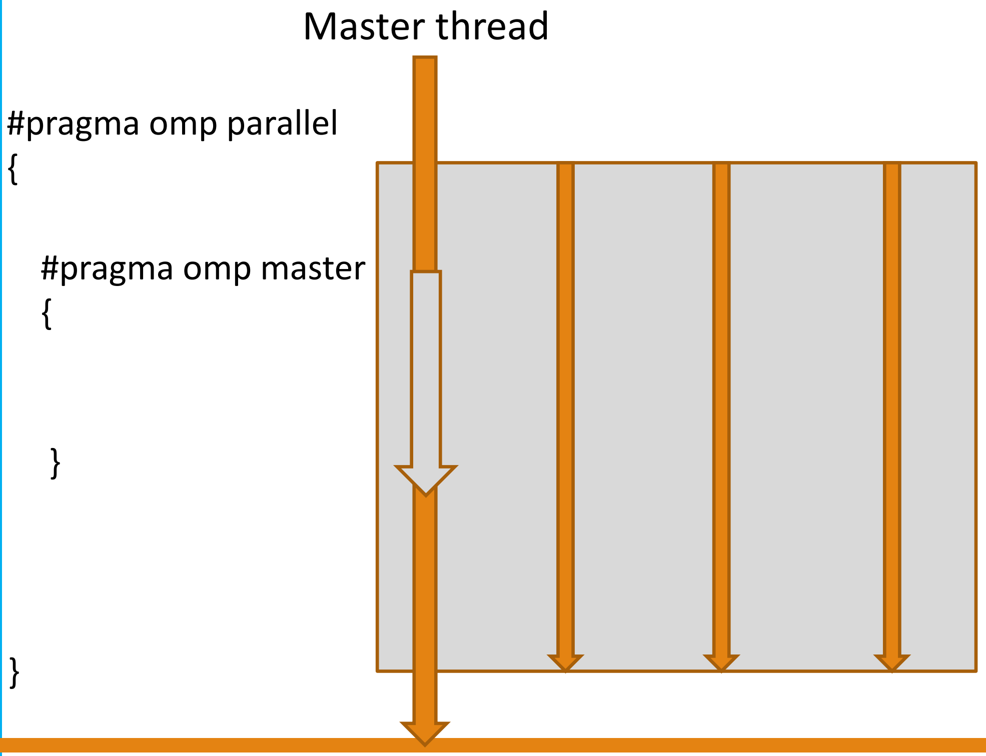
Hands-on

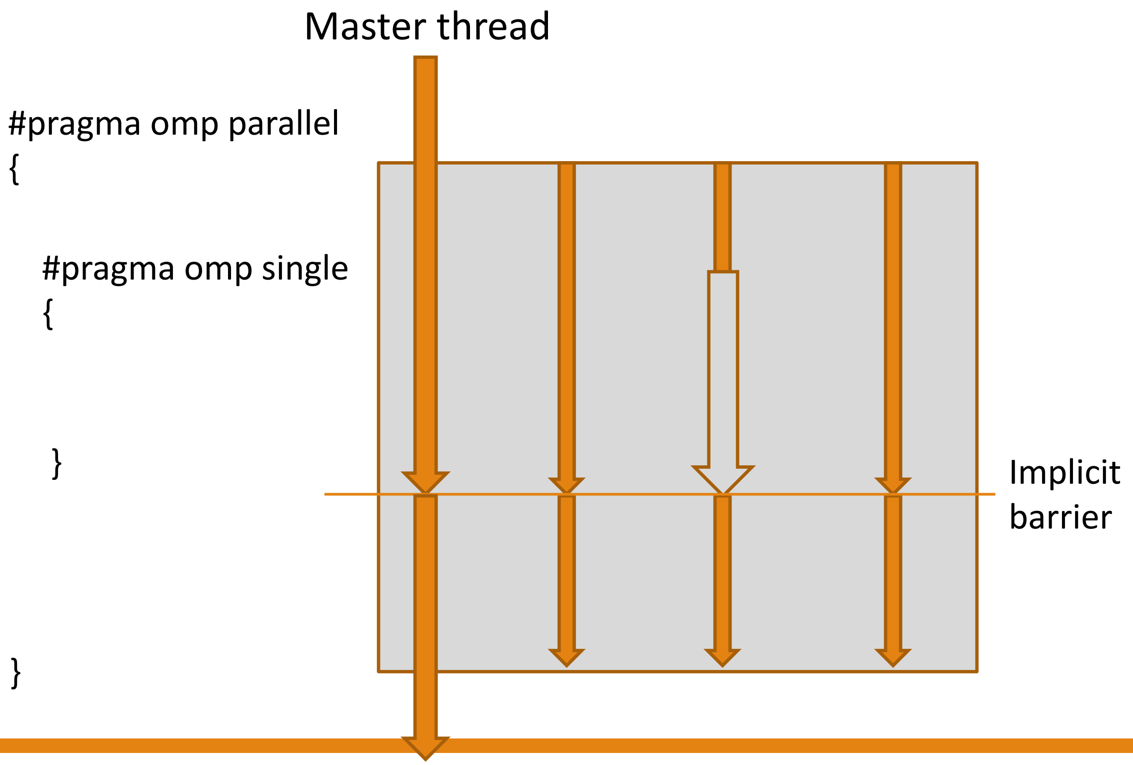
- Identify the storage attributes of each variable in the following code

```
int counter = 0;
#pragma omp threadprivate(counter)

void func(int *a, int n){
        int i, j, m = 3;
        #pragma omp parallel for
        for(i=0; i<n; i++) {
            int k = m;
            for (j=1; j<=5; j++)
                func2(&a[i], &k, j);
        }
}

void func2(int *x, int *y, int z){
    int ii;
    static int cnt;
    cnt++;
    counter++;
    for(ii=1; ii<z;ii++)
        *x = *y + z;
}
```

Synchronization: Master and Single

- In the examples we have used explicitly the id of the master/primary thread with an if construct
  - But we can use directives to identify
    - Master/masked: only the master/primary thread will execute the block
    - Single: only one thread will execute the block, and all threads must wait for the block to be terminated (there is an implicit barrier)





Synchronization: Master and Single

- In the examples we have used explicitly the id of the master/primary thread with an if construct
  - But we can use directives to identify
    - Master/masked: only the master thread will execute the block
    - Single: only one thread will execute the block, and all threads must wait for the block to be terminated (there is an implicit barrier)

```
long counter  = 0;
long prev_counter  = 0;
#pragma omp parallel reduction(+:counter)
{
   int my_index = omp_get_thread_num();
   int n_threads = omp_get_num_threads ( );

   for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++){
      if(vector[i] > threshold)
          #pragma omp atomic
          counter++;
      #pragma omp master
       if(counter - prev_counter > 100){
          prev_counter = counter;
          printf("Found more 100 threshold violations\n");
       }
   }
}
```

atomic is similar to critical, enforcing mutual exclusion. In its basic form works only updating a memory location (the update is atomic), with simple operators (++,--,+=,-=,…)

The if block will be executed only by the master thread

OpenMP 5.2
#pragma omp masked filter(0)

Synchronization: Master and Single

- In the examples we have used explicitly the id of the master thread with an if construct
  - But we can use directives to identify
    - Master: only the master thread will execute the block
    - Single: only one thread will execute the block, and all threads must wait for the block to be terminated (there is an implicit barrier)

```
#pragma omp parallel reduction(+:counter)
{
   #pragma omp single
   {
      counter  = 0;
      prev_counter  = 0;
    }

   int my_index = omp_get_thread_num();
   int n_threads = omp_get_num_threads ( );

   for(int i = my_index*SIZE/n_threads; i < (my_index+1)*SIZE/n_threads; i++){
      if(vector[i] > threshold)
           #pragma omp atomic
           counter++;
```

Only one thread will initialize the variables. All threads need to wait for initialization

Synchronization: Barriers

- In an OpenMP program there are implicit barriers, where all threads need to synchronize
  - At the end of a parallel region
  - At the end of a single directive
  - At the end of worksharing constructs (parallel loops and sections)
- Barriers have overhead
  - Need to count threads, with a shared counter, and suspend/resume threads
  - Therefore, single and worksharing implicit barriers can be suppressed with the nowait clause

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i=0; i<N; i++)
        process(Vector_A[i]);
    #pragma omp for
    for(int j=0; j<N; j++)
        process(Vector_B[i]);
}
```

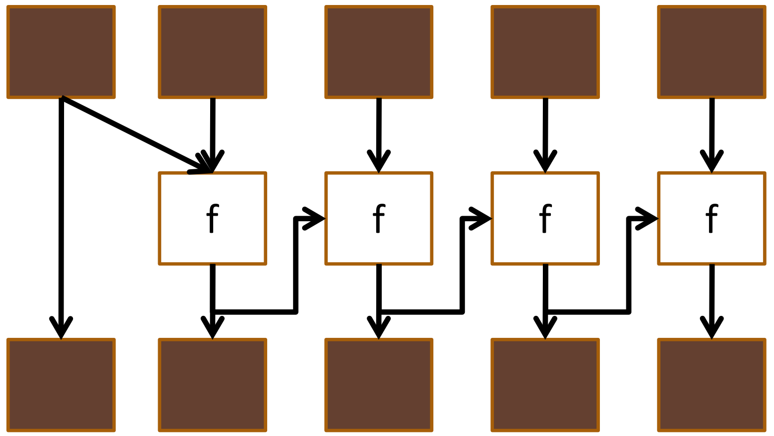
If a nowait clause is added to the loop, the implicit barrier at the end is surpressed, so threads can continue with B as soon as finishing with A

Threads that finish processing Vector_A in the first loop, need to wait all threads to finish processing Vector_A, before starting processing Vector B

Synchronization: Barriers

- It is also possible to explicitly set a barrier in a parallel region
  - All threads of the region need to reach the barrier, before any can proceed
  - Useful to build algorithms that work in steps

```
a[0] = b[0];
for (int i = 1; I < n; i++) {
    a[i] = f(a[i-1], b[i]);
}
```



```
a = { 1, 3, 5, 3, 4, 2, 3, 4}
for (int i = 1; i < n; i++) {
    a[i] = a[i-1] + a[i];
}
Write(a);
```

```
{ 1, 4, 9, 12, 16, 18, 21, 25}
```

Synchronization: Barriers

Doing more work than sequential, but depending on the cost of each individual operation, it may be faster when parallel

```
int main() {

  long vector[SIZE];
  long intermediates[NUM_THREADS];

  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int my_index = omp_get_thread_num();
    int n_threads = omp_get_num_threads ( );
    int it;
    for(it = my_index*SIZE/n_threads; it < (my_index+1)*SIZE/n_threads - 1; it++)
      vector[it + 1] = vector[it + 1] + vector[it];
    intermediates[my_index] = vector[it];

    #pragma omp barrier

    #pragma omp single
    {
      int total = intermediates[0];
      for (int i = 1; i < n_threads; i++) {
                int prevTotal = total;
                total = total + intermediates[i];
                intermediates[i] = prevTotal;
      }
    }

    if(my_index != 0)
      for(it = my_index*SIZE/n_threads; it < (my_index+1)*SIZE/n_threads; it++)
        vector[it] = vector[it] + intermediates[my_index];
  }
  return 0;
}
```

First phase: calculate in parallel the partial accumulators

Wait that all threads have calculated the accumulators

Second phase, accumulate the accumulators, only a single thread performs the calculation
There is an implicit barrier at the end of the single

Final phase, all threads except first update the values with the intermediate accumulators

Hands-on

- Change the prefix-sum example (previous slide) to use parallel loops

Synchronization: Locks

- Revisiting critical sections, assume you are creating a histogram out of values in an array

```
int create_histogram (int* array, int* hist)
{
        for(int i = 0; i < HIST_SIZE; i++)
                hist[i] = 0;

        #pragma omp parallel for
        for(int i = 0; i < ARRAY_SIZE; i++)
        {
                if(array[i]<0 || array[i]>HIST_SIZE-1)
                        return -1;
                #pragma omp critical
                hist[array[i]] ++ ;
        }
        return 0;
}
```

We need to protect the parallel updates of the histogram variable

But now the function is if sequential …

Synchronization: Locks

- OpenMP includes locks that can be used for synchronization
  - Similar to pthread mutexes
  - Operate on lock variables, using library functions
  - Like mutexes, threads need to hold the lock to be able to unlock it

```
#include <omp.h>
omp_lock_t lock;

int main()
{
        omp_init_lock(&lock);
        #pragma omp parallel
        {
                //…
                omp_set_lock(&lock);
                // protected code
                omp_unset_lock(&lock);
        }

        omp_destroy_lock(&lock);
}
```

The lock variable (opaque)

Lock needs to be initialized

Critical section inside set_lock/unset_lock

Lock should be destroyed

Synchronization: Locks

- A new histogram

```
int create_histogram (int* array, int* hist)
{
        omp_lock_t locks[HIST_SIZE];
        for(int i = 0; i < HIST_SIZE; i++){
                omp_init_lock(&locks[i]);
                hist[i] = 0;
        }

        #pragma omp parallel for
        for(int i = 0; i < ARRAY_SIZE; i++){
                if(array[i]<0 || array[i]>HIST_SIZE-1)
                        return -1;  // locks were not destroyed
                omp_set_lock(&locks[array[i]]);
                hist[array[i]] ++ ;
                omp_unset_lock(&locks[array[i]]);
        }

        for(int i = 0; i < HIST_SIZE; i++)
                omp_destroy_lock(&locks[i]);
        return 0;
}
```

Only contention if accesses the same position of the histogram

Going further

- Nested parallel regions
  - If within a parallel region, a new parallel region construct is reached, a new region is created
  - If nested parallelism is not enabled, the new team will be only the thread that the encountered the new region

```
void report_num_threads(int level)
{
   #pragma omp single
   printf("Level %d: number of threads in the team - %d\n", level, omp_get_num_threads());
}
int main()
{
   omp_set_nested(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return 0;
}
```

Changing to anything different than zero

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp nested.c
user@ubuntu:~/omp$ ./a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp nested.c
user@ubuntu:~/omp$ ./a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

Going further

- Number of threads in a parallel region
  - It is possible that the number of threads in a parallel region is less than what is requested
    - E.g. when nested parallelism is not enabled
  - The OpenMP runtime maintains a pool of threads, which are shared to be assigned among all parallel regions
    - This means that there might be less threads available than those requested
  - Even if there are more threads available than requested, OpenMP allows for the runtime to manage dynamically the number of threads
    - And assign less than requested
  - It is possible to disable dynamic management of threads
    - omp_set_dynamic(0);
    - This means that if there are threads available, the parallel region gets the requested number of threads
    - In case it is nested, if nested is enabled
- Conclusion: do not assume the number of threads
  - If the algorithm requires, check the number at runtime

Going further

- The memory model of OpenMP is based on relaxed consistency
  - Threads may have a temporary view of the memory (non-coherent caches, registers, scratchpads, compilers optimizations, …)
  - It is not possible to trust the value of a shared variable
    - It may be updated in parallel and the update still not visible
    - Or the ordering of read-writes in different variables
  - It is possible to force the consistency of the memory
    - A barrier forces all threads to make memory consistent
      - Applies to all threads of the team
    - The flush directive forces one thread (the one encountering the flush) to make its temporary view of the memory consistent
      - A flush operation is implied by OpenMP at entry/exit of parallel regions, implicit and explicit barriers, entry/exit of critical regions, lock set or unset
        - but not at entry to worksharing regions or entry/exit of master regions

Going further

- OpenMP does not provide thread conditional synchronization
  - Programmers have to explicitly manage it, with spinlocking
  - Makes programming of irregular applications significantly difficult

Mitigated by the OpenMP tasking model, next week!

```
double *A, sum, runtime;
int numthreads, flag = 0, flg_tmp;
A = (double *)malloc(N*sizeof(double));
#pragma omp parallel sections
{
        #pragma omp section
        {
                fill_rand(N, A);
                #pragma omp flush
                #pragma atomic write
                flag = 1;
                #pragma omp flush (flag)
        }
```

```
#pragma omp section
{
        while (1){
                #pragma omp flush(flag)
                #pragma omp atomic read
                flg_tmp= flag;
                if (flg_tmp==1) break;
        }
        #pragma omp flush
        sum = Sum_array(N, A);
}
```

Going further

- OpenMP Programs can introduce concurrency induced errors
  - Race conditions, deadlocks, livelocks
  - The same problems as described in the previous module
  - The specification guides on developing correct programs (data scoping, threads and barriers), and compilers are increasingly better in enforcing it, but
    - You can step outside of the "safer" model (e.g. using locks instead of critical for performance)
    - You can implement incorrect (from a specification point of view) programs, which the compiler does not detect

```
#pragma omp parallel
 {
     int x = 1;
     #pragma omp single
     x = 2;

     if(x==2)
    {
       #pragma omp barrier
       //…
    }
 }
```

Incorrect according to specification, but GCC does not detect it. This program deadlocks.

  - The use of static analysis tools, and model checking, are fundamental here

Advanced reading

- OpenMP Application Programming Interface, Version 5.2 November 2021, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
- OpenMP API 5.2 Examples, https://www.openmp.org/wp-content/uploads/openmp-examples-5-2.pdf
- Introduction to OpenMP - Tim Mattson (Intel), series of lectures, https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
- The OpenMP Common Core - Making OpenMP Simple Again, Timothy G. Mattson, Yun (Helen) He and Alice E. Koniges, MIT Press, ISBN 9780262538862, 2019
- Klemm, Michael and Cownie, Jim. High Performance Parallel Runtimes: Design and Implementation, Berlin, Boston: De Gruyter Oldenbourg, 2021. https://doi.org/10.1515/9783110632729

Credits

- Version 1.0, Luis Miguel Pinho, with inputs from:
  - A “Hands-on” Introduction to OpenMP, Tim Mattson, https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf
  - L.M. Pinho, Parallel Computing, Parallel Programming, Parallel Programming Techniques and Patterns, Computação Avançada, 2011
  - P. Pacheco, “An Introduction to Parallel Programming”, 1st Edition, Morgan Kaufmann 2011, ISBN 9780080921440, companion materials, https://booksite.elsevier.com/9780123742605/