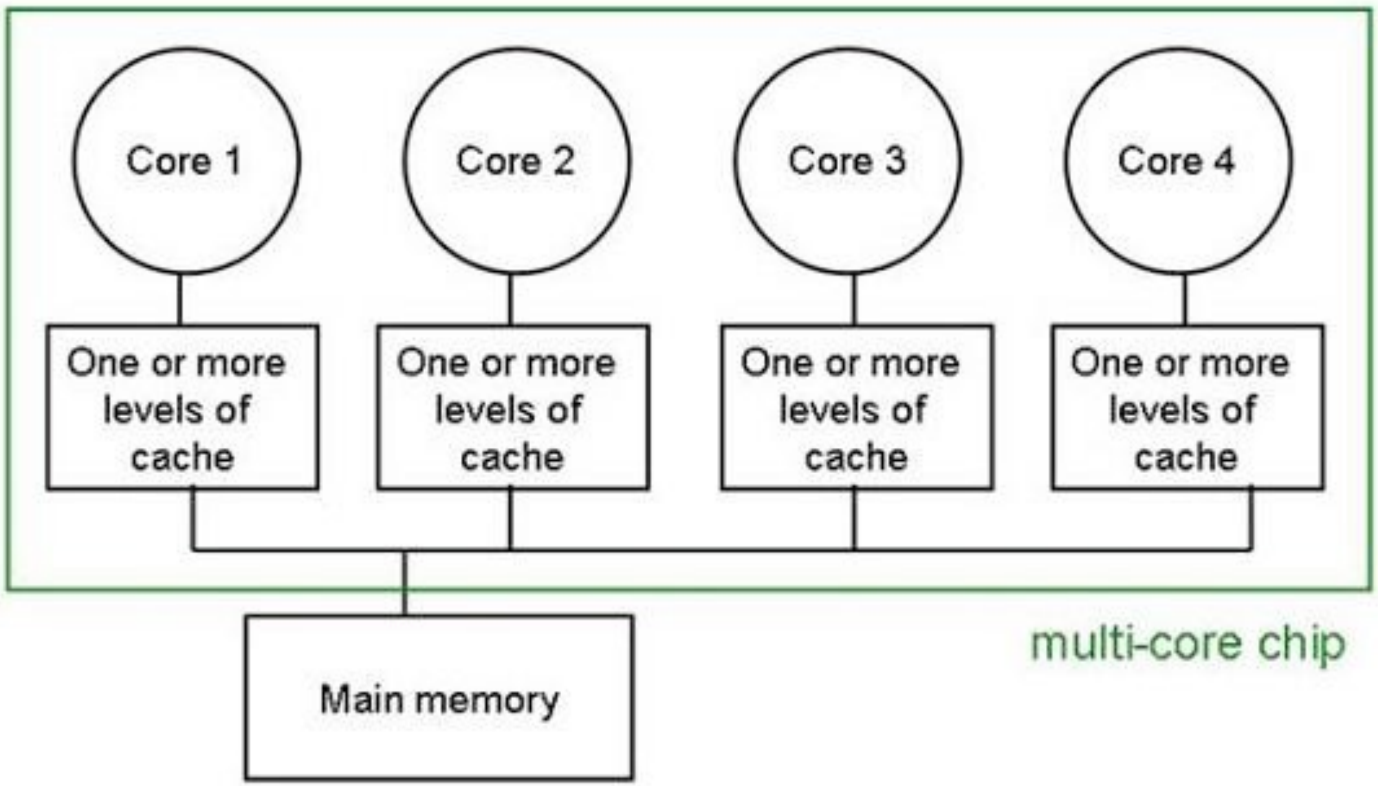
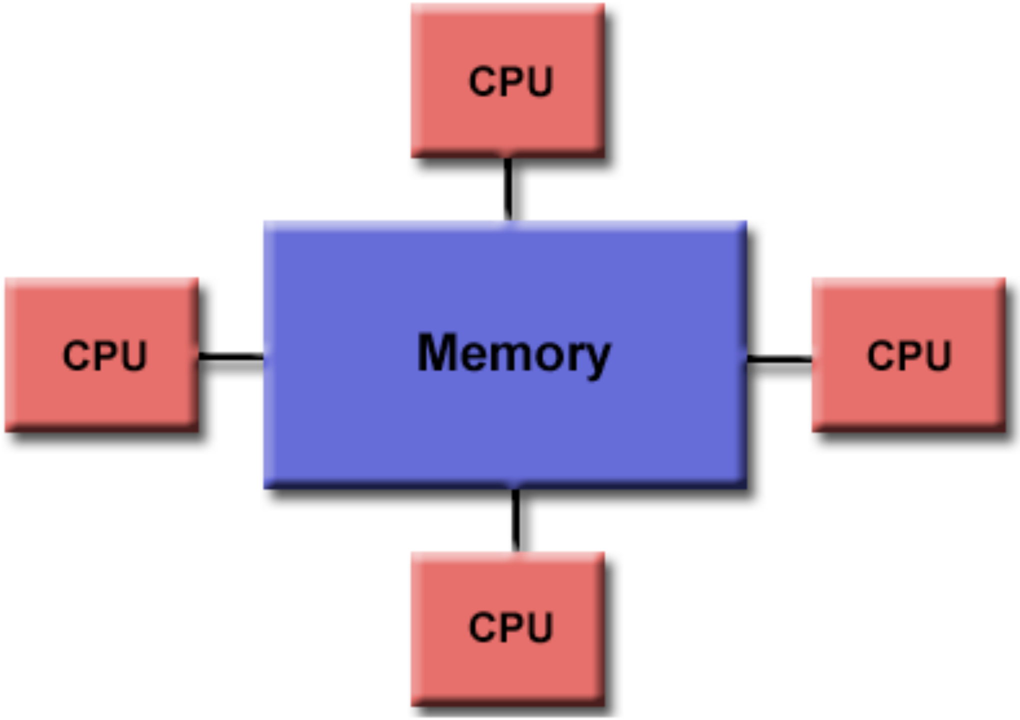
Multicore CPUs

Introduction to OpenMP

Multi-core processor

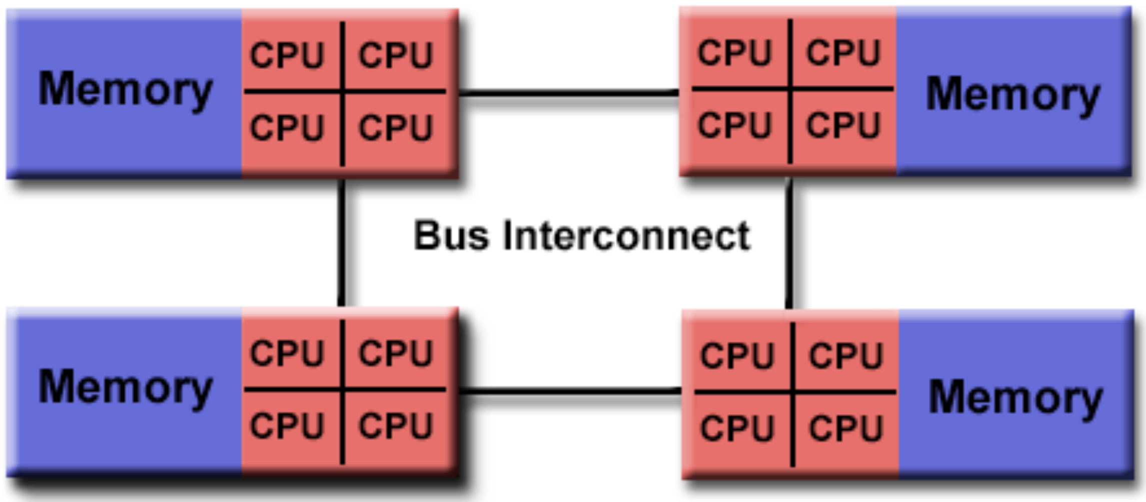


UMA and NUMA



Uniform Memory Access

Non-Uniform
Memory Access



Motivation

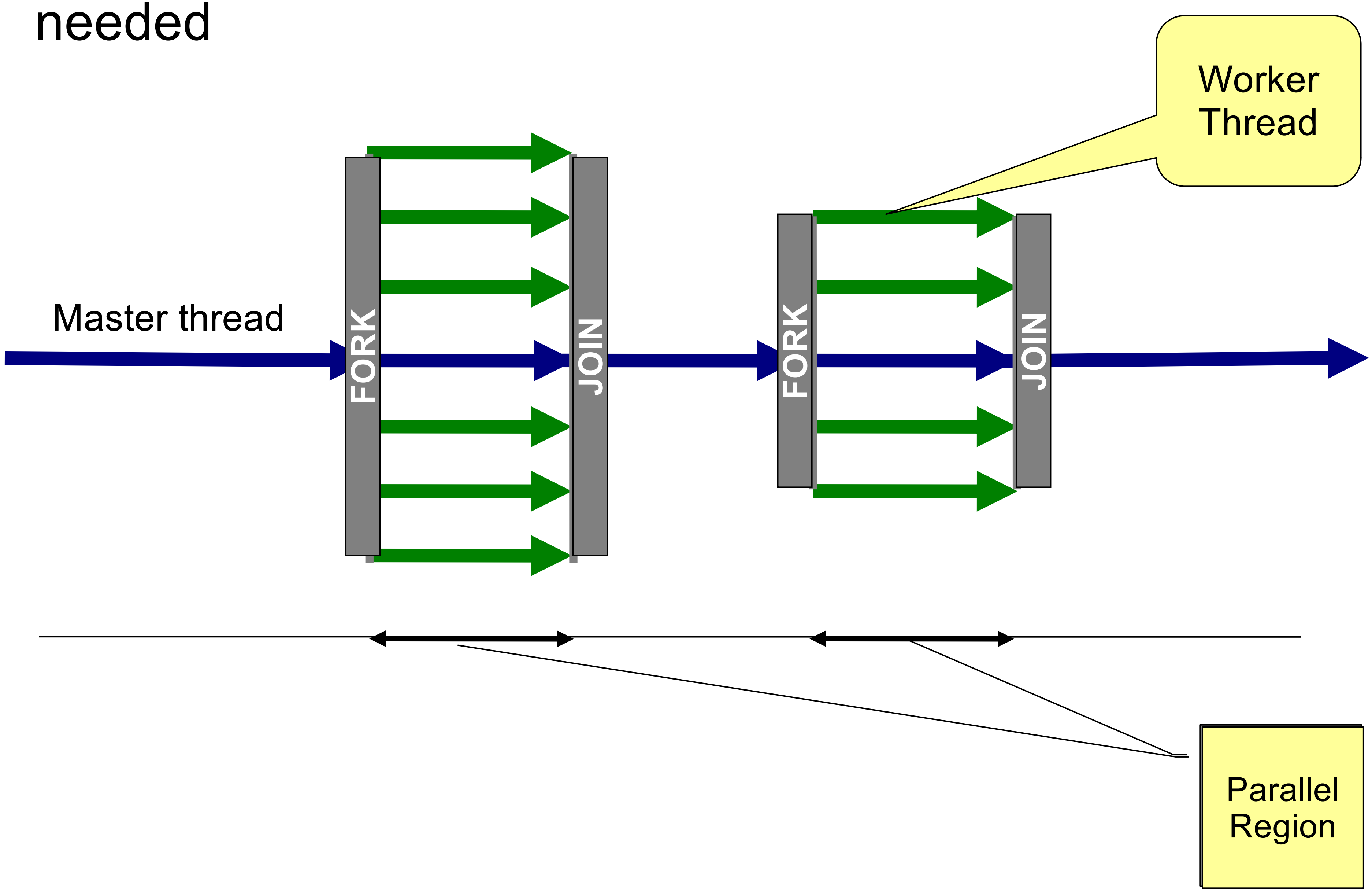
- Parallel machines are abundant
  - Modern processors have 4-68 cores
  - Upcoming processors are multicore – parallel programming is necessary to get high performance
- Multithreading is the natural programming model for SMP
  - All processors share the same memory
  - Threads in a process see the same address space
  - Lots of shared-memory algorithms defined
- Multithreading is (correctly) perceived to be hard!
  - Lots of expertise necessary
  - Deadlocks and race conditions
  - Non-deterministic behavior makes it hard to debug

Motivation: OpenMP

- A language extension that introduces parallelization constructs into the language
- Parallelization is orthogonal to the functionality
  - If the compiler does not recognize the OpenMP directives, the code remains functional (albeit single-threaded)
- Based on shared-memory multithreaded programming
- Includes constructs for parallel programming: critical sections, atomic access, variable privatization, barriers etc.
- Industry standard
  - Supported by Intel, Microsoft, Sun, IBM, HP, etc.
    Some behavior is implementation-dependent
  - Intel compiler available for Windows and Linux

OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed



OpenMP memory model

- Shared memory model
  - Threads communicate by accessing shared variables
- The sharing is defined syntactically
  - Any variable that is seen by two or more threads is shared
  - Any variable that is seen by one thread only is private
- Race conditions possible
  - Use synchronization to protect from conflicts
  - Change how data is stored to minimize the synchronization

OpenMP syntax

- Most of the constructs of OpenMP are pragmas
  - #pragma omp construct [clause [clause] …]
  - An OpenMP construct applies to a *structural block* (one entry point, one exit point)
- Categories of OpenMP constructs
  - Parallel regions
  - Work sharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- In addition:
  - Several omp_<something> function calls
  - Several OMP_<something> environment variables

OpenMP: Parallel Regions

```
double D[1000];
#pragma omp parallel
{
  int i; double sum = 0;
  for (i=0; i<1000; i++) sum += D[I];
  printf("Thread %d computes %f\n",
      omp_thread_num(), sum);
}
```

- Executes the same code several times (as many as there are threads)
  - How many threads do we have? omp_set_num_threads(n)
  - What is the use of repeating the same work several times in parallel? Can use omp_thread_num() to distribute the work between threads.
- D is shared between the threads, i and sum are private

OpenMP: Work Sharing Constructs 2

Sequential code

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

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr, iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Automatic parallelization of the for loop

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

Notes on #parallel for

- Only simple kinds of for loops are supported
  - One signed integer variable in the loop.
  - Initialization: var=init
  - Comparison: var op last, op: <, >, <=, >=
  - Increment: var++, var--, var+=incr, var-=incr, etc.
  - All of init, last, incr must be loop invariant

- Can combine the parallel and work sharing directives:
  #pragma omp parallel for …

#pragma omp shared modifier

- Notify the compiler that the variable is shared

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

- What’s the problem here?

Shared modifier cont'd

- Protect shared variables from data races

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

- Another option: use #pragma omp atomic
  - Can protect only a single assignment
  - Generally faster than critical

#pragma omp reduction

- Syntax: #pragma omp reduction (op:list)
- The variables in "*list*" must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Problems of #parallel for

- Load balancing
  - If all the iterations execute at the same speed, the processors are used optimally
  - If some iterations are faster than others, some processors may get idle, reducing the speedup
  - We don’t always know the distribution of work, may need to re-distribute dynamically
- Granularity
  - Thread creation and synchronization takes time
  - Assigning work to threads on per-iteration resolution may take more time than the execution itself!
  - Need to coalesce the work to coarse chunks to overcome the threading overhead
- Trade-off between load balancing and granularity!

Schedule: controlling work distribution

- schedule(static [, chunksize])
  - Default: chunks of approximately equivalent size, one to each thread
  - If more chunks than threads: assigned in round-robin to the threads
  - Why might we want to use chunks of different size?
- schedule(dynamic [, chunksize])
  - Threads receive chunk assignments dynamically
  - Default chunk size = 1 (why?)
- schedule(guided [, chunksize])
  - Start with large chunks
  - Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize

Controlling Granularity

- #pragma omp parallel if (expression)
  - Can be used to disable parallelization in some cases (when the input is determined to be too small to be beneficially multithreaded)
- #pragma omp num_threads (expression)
  - Control the number of threads used for this parallel region

OpenMP: Data Environment

- Shared Memory programming model
  - Most variables (including locals) are shared by default – unlike Pthreads!

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

  - Global variables are shared
- Some variables can be private
  - Automatic variables inside the statement block
  - Automatic variables in the called functions
  - Variables can be explicitly declared as private.
    In that case, a local copy is created for each thread

Overriding storage attributes

- private:
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
  - Can achieve the same using variables inside { }

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

- firstprivate:
  - Same, but the initial value of the variable is copied from the main copy
- lastprivate:
  - Same, but the last value of the variable is copied to the main copy

```
int idx=1;
int x = 10;
#pragma omp parallel for \
  firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i]==x) idx = i;
}
```

Threadprivate

- Similar to private, but defined per variable
  - Declaration immediately after variable definition. Must be visible in all translation units.
  - Persistent between parallel sections
  - Can be initialized from the master's copy with #pragma omp copyin
  - More efficient than private, but a global variable!
- Example:

```
int data[100];
#pragma omp threadprivate(data)
…
#pragma omp parallel for copyin(data)
for (……)
```

Reduction

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

- How to parallelize this code?
  - sum is not private, but accessing it atomically is too expensive
  - Have a private copy of sum in each thread, then add them up
- Use the reduction clause!
  #pragma omp parallel for reduction(+: sum)
  - Any associative operator must be used: +, -, ||, |, *, etc.
  - The private value is initialized automatically (to 0, 1, ~0 …)

OpenMP Synchronization

```
X = 0;
#pragma omp parallel
X = X+1;
```

- What should the result be (assuming 2 threads)?
  - 2 is the expected answer
  - But can be 1 with unfortunate interleaving
- OpenMP assumes that the programmer knows what (s)he is doing
  - Regions of code that are marked to run in parallel are independent
  - If access collisions are possible, it is the programmer’s responsibility to insert protection

Synchronization Mechanisms

- Many of the existing mechanisms for shared programming
  - Single/Master execution
  - Critical sections, Atomic updates
  - Ordered
  - Barriers
  - Nowait (turn synchronization off!)
  - Flush (memory subsystem synchronization)
  - Reduction (already seen)

Single/Master

- #pragma omp single
  - Only one of the threads will execute the following block of code
  - The rest will wait for it to complete
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region
  - Applicable to parallel for sections
- #pragma omp master
  - The following block of code will be executed by the master thread
  - No synchronization involved
  - Applicable only to parallel sections

Example:

```
#pragma omp parallel
{
   do_preprocessing();
   #pragma omp single
   read_input();
   #pragma omp master
   notify_input_consumed();

   do_processing();
}
```

Critical Sections

- #pragma omp critical [name]
  - Standard critical section functionality
- Critical sections are global in the program
  - Can be used to protect a single resource in different functions
- Critical sections are identified by the name
  - All the unnamed critical sections are mutually exclusive throughout the program
  - All the critical sections having the same name are mutually exclusive between themselves

Atomic execution

- Critical sections on the cheap
  - Protects a single variable update
  - Can be much more efficient (a dedicated assembly instruction on some architectures)
- #pragma omp atomic
  update_statement
- Update statement is one of: var= var op expr, var op= expr, var++, var--.
  - The variable must be a scalar
  - The operation op is one of: +, -, *, /, ^, &, |, <<, >>
  - The evaluation of expr is not atomic!

Ordered

- #pragma omp ordered
  statement
  - Executes the statement in the sequential order of iterations
- Example:

```
#pragma omp parallel for
for (j=0; j<N; j++) {
   int result = heavy_computation(j);
   #pragma omp ordered
   printf("computation(%d) = %d\n", j, result);
}
```

Barrier synchronization

- #pragma omp barrier
- Performs a barrier synchronization between all the threads in a team *at the given point*.
- Example:

```
#pragma omp parallel
{
  int result = heavy_computation_part1();
  #pragma omp atomic
  sum += result;
  #pragma omp barrier
  heavy_computation_part2(sum);
}
```

Controlling OpenMP behavior

- omp_set_dynamic(int)/omp_get_dynamic()
  - Allows the implementation to adjust the number of threads dynamically
- omp_set_num_threads(int)/omp_get_num_threads()
  - Control the number of threads used for parallelization (maximum in case of dynamic adjustment)
  - Must be called from sequential code
  - Also can be set by OMP_NUM_THREADS environment variable
- omp_get_num_procs()
  - How many processors are currently available?
- omp_get_thread_num()
- omp_set_nested(int)/omp_get_nested()
  - Enable nested parallelism
- omp_in_parallel()
  - Am I currently running in parallel mode?
- omp_get_wtime()
  - A portable way to compute wall clock time