Open Multi-Processing

Katarzyna Bułat, TCS Jagiellonian University

Overview

- What is OpenMP
- Design
- OpenMP in C++
- Cons

What is OpenMP

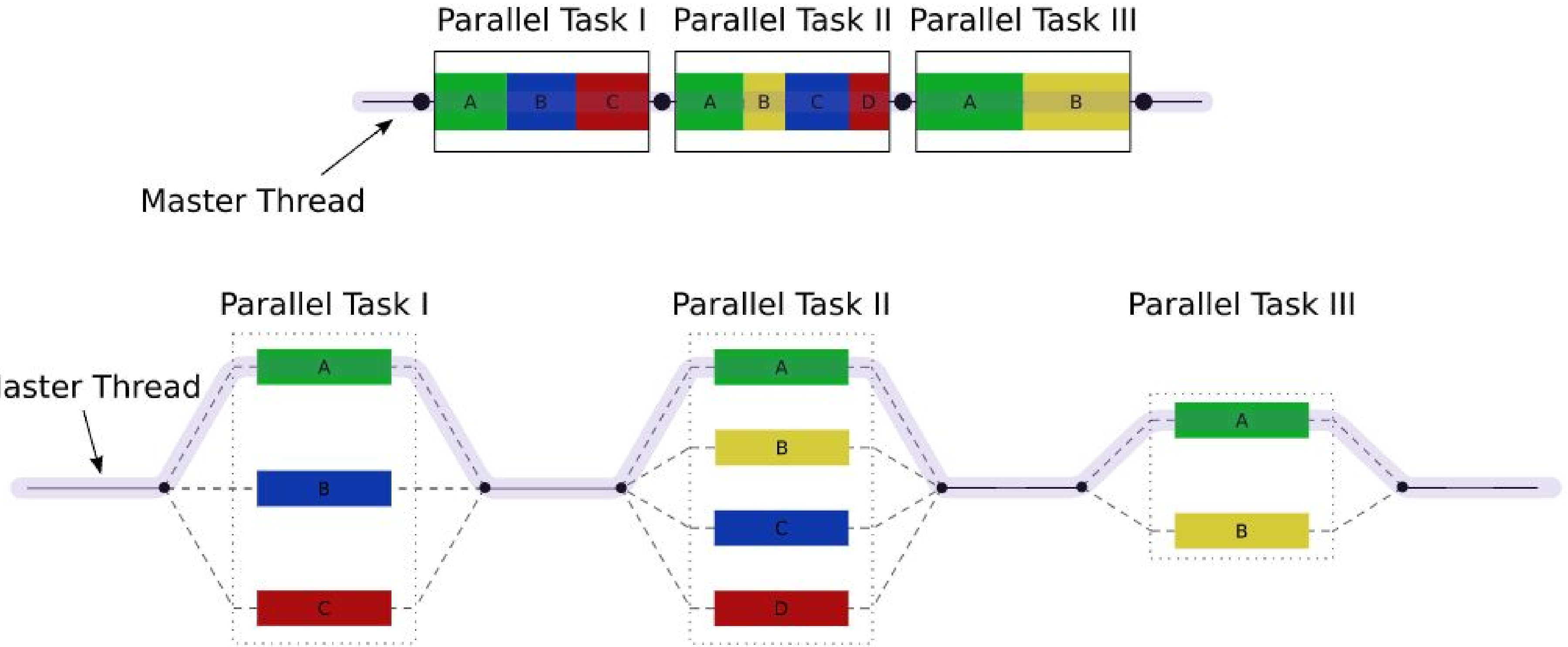
- API supporting **multi-platform shared memory multiprocessing** programming
- set of **compiler directives**, **libraries** and **environment variables** that influence run-time behavior
- **portable, scalable** model defined by major computer hardware and software vendors
- **simple and flexible** interface for developing **parallel** applications

Distributed systems

Combination of OpenMP for parallelism within the nodes and Message Passing Interface (MPI) for parallelism between the nodes

Design

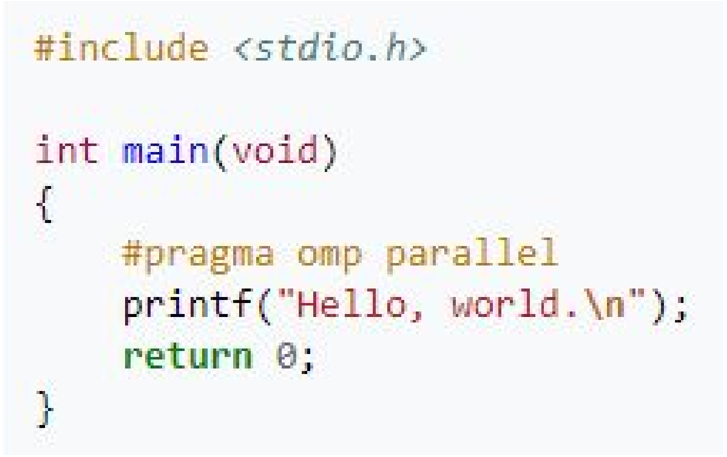
- multithreading
- master thread forking and assigning tasks to slave threads
- slave threads working concurrently and independently
- work-sharing constructs
- section to be executed in parallel marked with compiler directive


Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
A
B
C
D
A
B
Master Thread
Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
A
B
C
D
A
B

OpenMP in C++

- OpenMP functions are included in a header file labelled omp.h in C/C++
- Use flag -fopenmp to compile using GCC
- Core elements of OpenMP are the constructs for:
    - thread creation
    - work sharing
    - data-environment management
    - thread synchronization
    - user-level runtime routines and environment variables

Thread creation

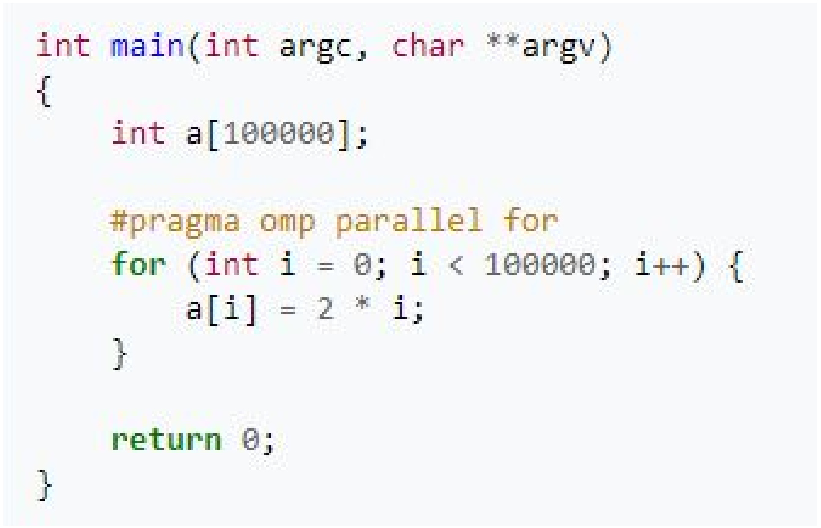


```c
#include <stdio.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Work sharing constructs

omp for or omp do, sections, single, master



```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Data sharing attribute clauses

- **shared** - accessible by all threads simultaneously
- **private** - each thread have a local copy
- **default** - set by programmer (shared or none)

Synchronization clauses

- **critical** - code block executed by only one thread at a time
- **atomic** - the memory update (write, or read-modify-write) in the next instruction will be performed atomically
- **ordered** - the structured block is executed in the order in which iterations would be executed in a sequential loop
- **barrier** - each thread waits until all of the other threads of a team have reached this point
- **nowait** - specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish

Scheduling clauses

- **schedule**(type, chunk) - this is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:
  - **static** - all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter chunk will allocate chunk number of contiguous iterations to a particular thread.
  - **dynamic**: here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter chunk defines the number of contiguous iterations that are allocated to a thread at a time.
  - **guided**: a large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter chunk

IF control

- this will cause the threads to parallelize the task only if a condition is met

Initialization

- **firstprivate -** the data is private to each thread, but initialized using the value of the variable using the same name from the master thread
- **lastprivate -** the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop.
- **threadprivate -** the data is a global data, but it is private in each parallel region during the runtime

Data copying

- **copyin** - similar to firstprivate for private variables, threadprivate variables are not initialized, unless using copyin to pass the value from the corresponding global variables
- **copyprivate** - used with single to support the copying of data values from private objects on one thread (the single thread) to the corresponding objects on other threads in the team.

Reduction

- **reduction**(operator | intrinsic : list): the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable

Cons

- risk of introducing difficult to debug synchronization bugs and race conditions
- runs efficiently only in shared-memory multiprocessor platforms
- requires a compiler that supports OpenMP
- no support for compare-and-swap
- reliable error handling is missing

Overview

- What is OpenMP
- Design
- OpenMP in C++
- Cons

Thank you for your attention