FIAS Frankfurt Institute for Advanced Studies

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

High-Performance Computer Architectures Practical Course

Multiple Instructions Multiple Data (MIMD) Instructions and OpenMP

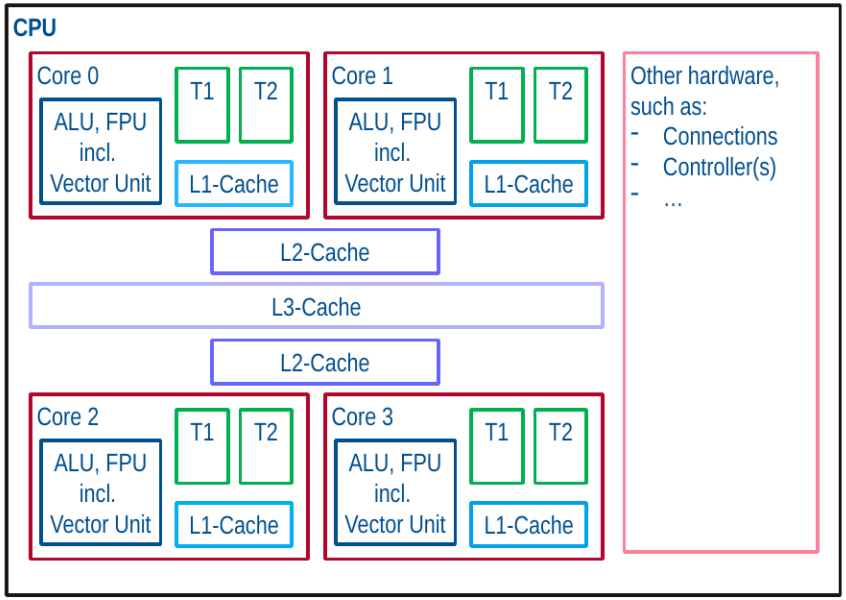
Prof. Dr. Ivan Kisel
Robin Lakos
Akhil Mithran
Oddharak Tyagi

June 3, 2025

Multi-Core, Multi-Thread and Vector Units

Parallelization with CPUs is possible by using

- vector units
  - in a core that supports SIMD instructions
- multiple threads handled on
  - single core (physical vs. logical cores)
  - multi-core
- multiple cores
  - in a single CPU
  - multiple CPUs (multi-socket mainboards) or
  - multiple machines (network)



The new focus is on distributing subtasks between CPUs or CPU cores.

Tools for Multi-Threading

- The programming language itself (in C++, for instance, `std::thread`)
- Other libraries:
  - Open Multi-Processing (OpenMP)
  - Message Passing Interface (MPI)
  - Open Computing Language (OpenCL)
  - (Intel's) Thread Building Blocks (iTBB)
  - PThreads
  - ...

OpenMP

OpenMP is an API for shared-memory multi-process programming in C/C++/Fortran

- supports many processor architectures
- supports most operating systems (Linux, UNIX, Windows, AIX, Solaris, OS X)

Import of library: `#include <omp.h>`
Compiler flag: `-fopenmp`

OpenMP is applied by using pre-processor directives that have the following format:

```
#pragma omp <construct> [clause ...]
```

Fork-Join Parallelism

Sequential program execution: 1 thread (Master Thread)

Parallel program execution with fork-join parallelism:

- Fork: Master thread creates new thread(s) that solve(s) subtasks in parallel
- Join: Synchronization and termination of created threads

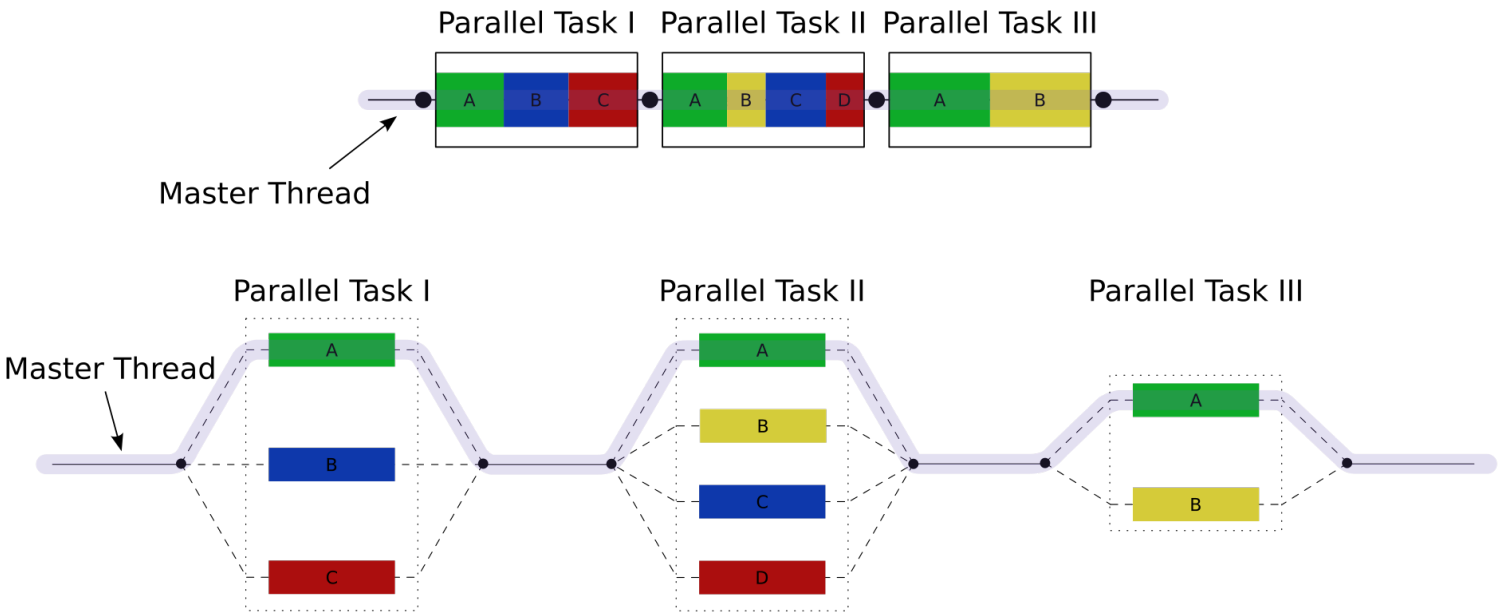


Figure: Fork-Join Parallelism.
https://en.wikipedia.org/wiki/Fork%E2%80%93join_model#/media/File:Fork_join.svg (A1-w:en:File:Fork_join.svg)

OpenMP – Useful Functionality

There are several functions that might be helpful when working with OpenMP:

- `omp_set_num_threads()`
- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_get_max_threads()`
- `omp_in_parallel()`
- `omp_get_num_procs()`

More information here: https://www.openmp.org/resources/refguides/

OpenMP – Example 1 – Parallel Region

```c
#include <omp.h>

int main() {
    // set num_threads globally
    omp_set_num_threads(4);
    // declare parallel region
    #pragma omp parallel
    {
        ...
    }
}
```

```c
#include <omp.h>

int main() {
    // declare parallel region using 4
    ↪ threads
    #pragma omp parallel num_threads(4)
    {
        ...
    }
}
```

OpenMP – Example 2 – For-Loops

```
// Without work sharing:
int arr[N];
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nThreads = omp_get_num_threads();
    int iStart = id * N / nThreads;
    int iEnd = (id + 1) * N / nThreads;
    if (id == nThreads - 1) iEnd = N;
    for (int i = iStart; i < iEnd; i++) {
        arr[i] = 2 * arr[i];
    }
}
```

```
1  // With work sharing:
2  int arr[N];
3  #pragma omp parallel for
4  for (int i = 0; i < N; i++) {
5      arr[i] = 2 * arr[i];
6  }
```

Reduction-Statement

The reduction statement combines the values of variables from multiple threads into a single result using a specified operator (e.g., sum, product) while ensuring thread safety.

Example:

```
// calculate average value of arr[N]
float avg = 0.f;

#pragma omp parallel for reduction(+:avg)
for (int i = 0; i < N; i++) {
    avg += arr[i];
}

avg /= N;
```

Thread-Synchronization

- **High level:**
  - **critical**
  - atomic
  - barrier
  - ordered
- **Low level:**
  - flush
  - locks

```
1  #pragma omp parallel
2  {
3      for(int i=0; i<N; i++) {
4          #pragma omp critical
5          {
6              f(...); // only called by 1
                   ↪  thread at a time
7          }
8      }
9  }
```

Thread-Synchronization

- **High level:**
  - critical
  - **atomic**
  - barrier
  - ordered
- **Low level:**
  - flush
  - locks

```
1  #pragma omp parallel
2  {
3      float X[100];
4      double tmp;
5      ...
6      #pragma omp atomic
7      X[i] += tmp; // overwrite protection
8  }
```

A critical region ensures exclusive access to a block of code by one thread at a time, while an atomic operation enforces thread safety for a single, low-level memory operation with less overhead.

Thread-Synchronization

- **High level:**
  - critical
  - atomic
  - **barrier**
  - ordered
- **Low level:**
  - flush
  - locks

```
1   #pragma omp parallel
2   {
3       ...
4       #pragma omp barrier // explicit barrier
5       ...
6       #pragma omp for
7       for(int i=0; i<N; i++) {
8           ...
9       } // implicit barrier here
10
11      #pragma omp nowait
12      #pragma omp for
13      for(int i=0; i<N; i++) {
14          ...
15      } // no implicit barrier, due to "nowait"
16  }
```

Thread-Synchronization

- **High level:**
  - critical
  - atomic
  - barrier
  - **ordered**
- **Low level:**
  - flush
  - locks

```
1  // Enforce correct order of execution
2  #pragma omp parallel for ordered
3  for(int i=0; i<N; i++) {
4      #pragma omp ordered
5      {
6          // code that needs to be executed in
           ↪  order
7      }
8  }
```

Thread-Synchronization

- High level:
  - critical
  - atomic
  - barrier
  - ordered
- **Low level:**
  - **flush**
  - locks

```
1  // Enforce threads to write most recent
   ↪  values to memory, so other threads can
   ↪  access it
2  #pragma omp parallel
3  {
4      ...
5      #pragma omp flush
6      ...
7  }
```

Thread-Synchronization

- High level:
  - critical
  - atomic
  - barrier
  - ordered
- **Low level:**
  - flush
  - **locks**

```
1  #include <omp.h>
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      omp_set_num_threads(4);
8      omp_lock_t lck;
9      omp_init_lock(&lck);
10
11     #pragma omp parallel
12     {
13         omp_set_lock(&lck);
14         cout « "Hello World" « endl;
15         omp_unset_lock(&lck);
16     }
17
18     omp_destroy_lock(&lck);
19 }
```

Limitations of Thread-Synchronization

While OpenMP provides various thread synchronization mechanisms, their use can negatively impact runtime performance.

For instance, if a parallel region contains only operations within a critical section, it effectively behaves like a sequential region, but with additional overhead introduced by OpenMP's parallelization setup.

Therefore, synchronization should be minimized, and parallel regions should be designed to maximize independent work across threads to achieve better performance.

Data Sharing

- `shared` – variable is shared between threads within the area
- `private` – each thread creates a private, uninitialized copy of the variable
- `firstprivate` – like private, but initialized with the value of the master thread
- `lastprivate` – like private, but the last value will be saved to a global variable
- `default (private or none)` – sets default for all variables

Data Sharing – Example

```cpp
#include <omp.h>
#include <iostream>

using namespace std;

int main() {
    omp_set_num_threads(4);
    int tmp = 0;

    #pragma omp for private(tmp)
    for(int i=0; i<N; i++) {
        tmp += i; // tmp not initialized!
    }

    #pragma omp parallel for firstprivate(tmp)
    for(int i=0; i<N; i++) {
        tmp += i; // tmp initialized with 0
    }
}
```

Schedulers and Chunks

- `schedule(static, chunk)`
  - All iterations are split into ”packages” of size chunk and each thread handles its ”package”
- `schedule(dynamic, chunk)`
  - Each thread grabs a ”package” of size chunk from a queue and processes it. If it is finished, it grabs the next available non-distributed package.
- `schedule(guided, chunk)`
  - Each thread grabs a ”package” from a queue and processes it. If it is finished, it grabs the next available non-distributed package. Here, the size of the ”packages” is shrinking down to size chunk over time.
- `schedule(runtime)`
  - Schedule and chunk size taken from the `OMP_SCHEDULE` environment variable

Schedulers and Chunks

`schedule(static, chunk)`

- All iterations are split into ”packages” of size chunk and each thread handles its ”package”

```
1  #pragma omp parallel for schedule(static,
   ↪  4)
2  for (int i = 0; i < N; i++) {
3      // Each thread processes its chunk
4      process(arr[i]);
5  }
```

Schedulers and Chunks

`schedule(dynamic, chunk)`

- Each thread grabs a ”package” of size chunk from a queue and processes it. If it is finished, it grabs the next available non-distributed package.

```
1  #pragma omp parallel for
   ↪  schedule(dynamic, 4)
2  for (int i = 0; i < N; i++) {
3      // Threads grab next available chunk
       ↪  dynamically
4      process(arr[i]);
5  }
```

Schedulers and Chunks

`schedule(guided, chunk)`

- Each thread grabs a ”package” from a queue and processes it. If it is finished, it grabs the next available non-distributed package. Here, the size of the ”packages” is shrinking down to size chunk over time.

```
1  #pragma omp parallel for schedule(guided,
   ↪  4)
2  for (int i = 0; i < N; i++) {
3      // Thread packages shrink in size
       ↪  over time
4      process(arr[i]);
5  }
```

Schedulers and Chunks

`schedule(runtime)`

- Schedule and chunk size taken from the `OMP_SCHEDULE` environment variable

```
1  #pragma omp parallel for
   ↪  schedule(runtime)
2  for (int i = 0; i < N; i++) {
3      // Scheduling controlled by
       ↪  OMP_SCHEDULE
4      process(arr[i]);
5  }
```