HPS
julian.kunkel@gwdg.de

Julian Kunkel

OpenMP

Parallelism within nodes

Learning Objectives

- Describe the features of OpenMP and it’s parallelization strategy
- Create simple programs in C that demonstrate OpenMP features
- Parallelize smaller sections of existing code using OpenMP

Table of contents

Motivation

- Problems exist where Shared-Memory is required or beneficial
- Development of dedicated share-memory architectures is still ongoing
- Number of processor for such systems continuously increases
- But hardware specific code is not portable
- MPI might be too difficult

OpenMP

Open Specifications for **M**ulti **P**rocessing

- Consists of
  - Specification of pragmas (Hints) for the compiler describing parallelizable sections
  - Lightweight API to inquire and control parallelization
  - Compiler extension that translates code into parallelized (multi-threaded) version
- Therefore, a bit different everywhere
- Specified for Fortran and C
  - Also works with C++

OpenMP

Not a Magic Spell

- Meant for Shared Memory Systems
- Can be combined with MPI
- Does no magic! You have to
  - Sync IO access on your own
  - Lock memory on your own
  - Avoid deadlocks on your own
- Latest specification:
  `OpenMP 5.2` from 09.11.21

Open MP Components

- The C-API consists of 3 parts
  - Compiler Directives
    ```
    #pragma omp parallel default(shared) private(beta,pi)
    ```
  - A Library
    ```
    #include <omp.h>
    int omp_get_num_threads(void)
    ```
  - Environmental Variables
    ```
    export OMP_NUM_THREADS=8
    ```
- Compile with:
  - `gcc -fopenmp foobar.c`
  - `icc -no-multibyte-chars -qopenmp foobar.c`

Hello World

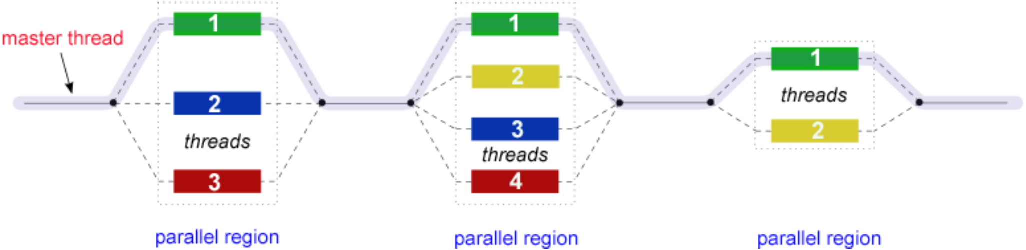
hello-openmp.c

```
#include <omp.h>

main ()  {
        #pragma omp parallel
        printf("Hello World");
}
```

Output

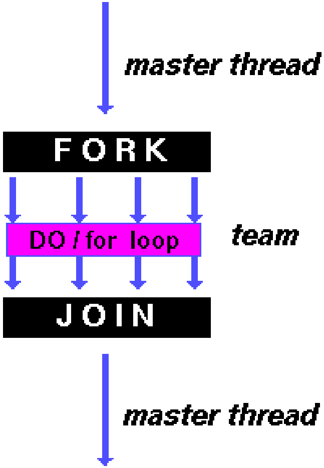
```
$ ./hello-openmp

Hello World
Hello World
Hello World
Hello World
```

Fork-Join Model



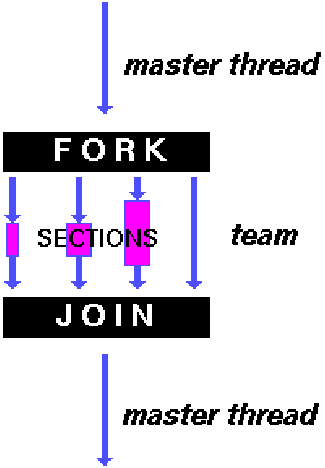
- Start with a single thread
  - *Master thread*
  - or *thread 0*
- More threads created at runtime
- Barrier at the end of parallel region
  - Additional threads are closed
  - Master thread continues

Work Sharing I



- Single Instruction Multiple Data (SIMD)
- Example:
  - Add fixed number to vector
- Easy to parallelize

Work Sharing II



- Multiple Instructions Multiple Data (MIMD)
- Different tasks (and code!) for different threads in the parallel section
- Hard to parallelize

Communication and Data Space I

- Communication via Shared Variables
- Master thread
  - Execution context during entire runtime
- Worker threads
  - Execution context only during parallel regions

Communication and Data Space II

- Variables are categorized in
  - *Shared*
  - *Private*
- Default is *Shared*
  - Good practice to always specify
- Simplifies coding special attributes
  - e.g., *Reduction*

Communication and Data Space III

- Shared variables
  - All thread access same memory address
  - Common way to communicate
- Private variables
  - One copy for each thread
  - Value **undefined** at beginning and end of parallel region

Synchronization

When using shared variables

- Avoid concurrent writes!
- One thread might read while another writes
- State at end of parallel region unclear
- Memory cache can be used to avoid conflicts
  - *Flush*-directive synchronizes memory

Multiple Threads Example

thread-number.c

```
#include <omp.h>
main ()  {
  int nthreads, tid;
  /* do something in parallel: */
  #pragma omp parallel private(tid)
  {
        /* Obtain and print thread id */ 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 */
}
```

Omp Directives Example

omp-directives.c

```
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
  #pragma omp for schedule(dynamic)
  for (i=0; i < N; i++) { c[i] = a[i] + b[i]; }
}  /* end of parallel section */

/* only the master does printf */
#pragma omp master
{
  for(i=0;i<N;i++) {printf("c[%d] = %f\n",i,c[i]);}
}
```

Omp Directives

- *schedule* - defines how to distribute tasks
  Scheduler: static/dynamic/guided/runtime/auto
- *nowait* - Do not synchronize threads afterward (e.g., flush)
- *ordered* - Iterations must be done in same order as in serial

omp-ordered.c

```c
#pragma omp parallel for ordered
for (i=0; i < N; i++){
  // do heavy stuff

  #pragma omp ordered
  c[i] = a[i] + b[i];

  // more heavy stuff
}  /* end of parallel section */
```

OpenMp Directives

- Parallization
  - *for*
  - *parallel*
  - *sections*
  - *single*
  - *task*
  - ...
- Synchronization
  - *barrier*
  - *critical*
  - *master*
  - *atomic*
  - ...
- Data space
  - *threadprivate*

Directives Syntax

C/C++

```
#pragma omp directive [clause [[,] clause ...] ...]
  //Structured Block
```

Fortran

```
!$OMP directive [clause[[,] clause ...] ...]
  ! Structured Block
!$OMP END
```

Important Clauses for Data Space

`#pragma omp parallel ...`

- *private* (*var1,var2,var3*)
- *shared* (*var1,var2,var3*)
- *default* (*shared/none*)
  - *private* is not allowed here!
- *reduction* (*operator:var1*)
  - *var1* is (implicitly) thread private and aggregated via operator at the end

omp-clauses.c

```c
#pragma omp parallel default(shared) private(i) reduction(+:result)
{
  #pragma omp for schedule(static,chunk)
  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
} // end omp parallel
printf("Final result= %f\n",result);
```

Undefined Variables

- *private* variables
  - Undefined at start and end of parallel region
- *firstprivate(list of variables)*
  - Initializes *private* variables with value prior to region

Important Library Functions

- omp_in_parallel()
- omp_get_num_threads()
- omp_get_thread_num()
- omp_set_num_threads()
- omp_get_num_procs()
- omp_get_wtime()
- omp_get_wtick()
- omp_ini_lock()
- omp_set_lock()
- omp_unset_lock()
- omp_test_lock()
- omp_destroy_lock()

Time Measurement

`double omp_get_wtime(void);`

- Returns time in seconds since a fixed arbitrary time in the past
- Temporal resolution may be limited due to OS architecture
- Elapsed time calculated as difference between two calls

timing.c

```
#pragma omp parallel
{
  // ...
  #pragma omp single nowait
  start = omp_get_wtime();
  // ... code of interest
  #pragma omp single nowait
  end = omp_get_wtime();
  // ...
} // end of parallel section
printf("time in seconds: %lf\n", end - start);
```

OpenMP Loop Parallelization I

- Strength of OpenMP!
- Each thread handles a subset of iterations
- Should be SIMD - Beware of dependencies
- Clauses: Schedule, Order, ...

C
```c
#pragma omp for (+clauses)
  for(...)
```

- Only affects directly subsequent `for` loop

OpenMP Loop Parallelization II

- *omp parallel* - Create parallel section
- *omp for* - Use existing threads to process loop
  - Must be in *omp parallel* region
  - Only affects the very next `for` loop
- *omp parallel for* - Both in one line

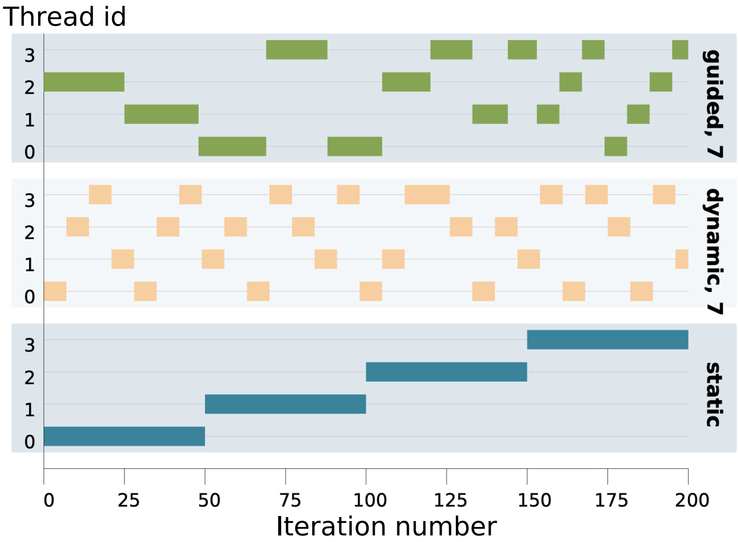
C

```c
#pragma omp parallel for collapse(3)  // collapse will flatten multiple loops
for(int l=0; l<10; ++l) {
  /∗ no code allowed here ∗/
  for(int j=0; j<4; ++j) {
        /∗ no code allowed here ∗/
        for(int k=0; k<5; ++k){
          foo[l][j][k] = 0;
} } }
```

OpenMP Loop Parallelization III

- *omp parallel for ordered*
  - Threads process `for` loop iterations ordered as if sequentially
- *omp parallel for schedule*
  - Hint how iterations should be distributed among threads
  - *static* - Same chunk size
  - *dynamic* - Give out chunks on request (controlled with *chunk*)
  - *guided* - Chunk size decreases with iterations
  - *runtime* - Using environmental variables
  - *auto* - Let compiler and runtime decide

Loop Scheduling



Parallel Sections

- Useful for MIMD operations
- *omp parallel sections* - to start several regions
  - Otherwise *omp section* for each region
- Each section is executed by one thread!
- Good for small tasks
- Order of execution is not defined

Parallel Sections Example

parallel-sections.c

```
for (i=0; i < N; i++) {
  a[i] = i * 1.5;  b[i] = i + 22.35;
}
#pragma omp parallel shared(a,b,c,d) private(i)
{
  #pragma omp sections        // you might use "nowait"
  {
    #pragma omp section
    for (i=0; i < N; i++)
          c[i] = a[i] + b[i];
        #pragma omp section
        for (i=0; i < N; i++)
          d[i] = a[i] * b[i];
  }  /* end of sections */
}  /* end of parallel section */
```

Important Directives

`#pragma omp directive`

- *master* - Only executed by master thread
- *critical* - Only one thread allowed at a time
- *barrier* - Wait for all threads to reach this point
- *flush* - Synchronize shared memory of all threads
    - Implicitly done at *barrier*, *for*, *critical*, *parallel* ...

Other Workload Distribution

- *omp single* - Only one thread executes block, used in parallel section
  - Useful for I/O operations
- *omp critical* - Only one tread at a time executes block
  - Useful to avoid data races
- *nowait* - Allow threads to pass by without waiting on each other

Code within/without Parallel Sections I

parallel-1.c

```
int my_start, my_end;

void work(){ /* my_start and my_end are undefined */
  printf("My subarray is from %d to %d\n", my_start, my_end);
}
int main(int argc, char* argv[]){
#pragma omp parallel private(my_start, my_end)
  {
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(), omp_get_num_threads());
    my_end = get_my_end(omp_get_thread_num(), omp_get_num_threads());
    work();
  }
}
```

Code within/without Parallel Sections II

Solution 1: Variables as parameters

parallel-2.c

```c
int my_start, my_end;

void work(int my_start, int my_end){
  printf("My subarray is from %d to %d\n", my_start, my_end);
}
int main(int argc, char* argv[]){
  #pragma omp parallel private(my_start, my_end)
  {
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(), omp_get_num_threads());
    my_end = get_my_end(omp_get_thread_num(), omp_get_num_threads());
    work(my_start, my_end);
  }
}
```

Code within/without Parallel Sections III

Solution 2: Using *omp threadprivate*

parallel-3.c

```
int my_start, my_end;
#pragma omp threadprivate(my_start, my_end)
void work(){
  printf("My subarray is from %d to %d\n", my_start, my_end);
}
int main(int argc, char* argv[]){
  #pragma omp parallel
  {
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(), omp_get_num_threads());
    my_end = get_my_end(omp_get_thread_num(), omp_get_num_threads());
    work();
  }
}
```

Exercise

- Simple to more complex tasks
- Use the online OpenMP specification!
- Questions without coding are to test your understanding
- Use OpenMP for more problems
  - e.g., Calculate π

References

- https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html
- https://www.openmp.org/spec-html/5.2/openmp.html
- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- https://gcc.gnu.org/wiki/Graphite/Parallelization
  https://hpc-tutorials.llnl.gov/openmp/