Introduction to OpenMP

Dov Kruger

Contents

1 Introduction to OpenMP

1.1 What is OpenMP?

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++,Fortran, and other languages. It provides a simple and flexible interface for developing parallel applications, enabling efficient use of multi-core processors.

Advantages: it runs on most major platforms, including Linux, Windows, and MacOS, it uses standard C/C++ compilers, and it is supported by most major vendors (Intel, AMD, NVIDIA, IBM, Fujitsu, etc.)

Disadvantages: OpenMP relies on shared memory so we have the same bottlenecks as we saw in C++, but at least the code is generated automatically.

OpenMP also supports devices like GPUs and other accelerators, but this is mediated through the compiler, OpenMP does not support them directly.

1.2 History and Evolution

OpenMP was introduced in 1997 as a standard for parallel programming. It has evolved over the years with multiple versions, each adding new features and improvements. The latest version includes support for tasking, SIMD (Single Instruction, Multiple Data), and improved support for accelerators, making it more versatile and powerful.

Major versions:

- 1.0: 1997 FORTRAN
- 2.0: 2002 C/C++
- 2.5: 2005 C++/Fortran
- 3.0: 2008 Tasks
- 4.0: 2013 SIMD
- 5.2: 2021

1.3 Why Use OpenMP?

OpenMP simplifies the process of writing parallel code. It allows developers to add parallelism to existing code with minimal changes. It is widely supported by compilers and provides a portable solution for parallel programming. OpenMP also offers a range of constructs for parallel regions, work sharing, synchronization, and tasking, making it easier to optimize performance and scalability.

2 Basic Concepts

2.1 Parallel Regions

Parallel regions are blocks of code executed by multiple threads simultaneously. They are defined using `#pragma omp parallel` in C/C++ and `!$OMP PARALLEL` in Fortran.

2.2 Work Sharing Constructs

Work sharing constructs distribute code execution among threads. Examples include `#pragma omp for` in C/C++ and `!$OMP DO` in Fortran, used to parallelize loops.

By default variables within the block being parallelized are local to the thread. Variables declared outside the block are shared. There can be subtle problems, both with performance and correctness. That is why we learned first to writ the code manually.

2.3 Synchronization Constructs

Synchronization constructs manage access to shared resources and ensure correct execution order. Examples include `#pragma omp critical` in C/C++ and `!$OMP CRITICAL` in Fortran, defining critical sections of code that only one thread can execute at a time.

3 OpenMP in Programming Languages

3.1 C/C++

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

int main() {
    omp_set_num_threads(4);     // Set the number of threads

    #pragma omp parallel        // Parallel region
    {
        int thread_id = omp_get_thread_num();
        std::cout << "Hello from thread " << thread_id << std::endl;
    }
    return 0;
}
```

3.2 Fortran

```
program hello
    use omp_lib
    integer :: thread_id

    ! Set the number of threads
    call omp_set_num_threads(4)

    ! Parallel region
    !$OMP PARALLEL PRIVATE(thread_id)
    thread_id = omp_get_thread_num()
    print *, 'Hello from thread ', thread_id
    !$OMP END PARALLEL
end program hello
```

4 Building OpenMP Programs

4.1 Compiler Flags

To compile OpenMP programs in C++, use the following compiler flags:

- `-g`: This flag includes debugging information in the compiled program. Example:

  ```
  g++ -g your_program.cpp -o your_program
  ```

- `-O2`: This flag enables optimization for faster code execution. Example:

  ```
  g++ -O2 your_program.cpp -o your_program
  ```

- `-fopenmp`: This flag enables OpenMP support. Example:

  ```
  g++ -fopenmp your_program.cpp -o your_program
  ```

- `-fopenmp-simd`: This flag enables OpenMP SIMD support. Example:

  ```
  g++ -fopenmp-simd your_program.cpp -o your_program
  ```

- `-mavx2`: This flag enables the use of AVX2 SIMD instructions. Example:

  ```
  g++ -mavx2 your_program.cpp -o your_program
  ```

- `-mavx512`: This flag enables the use of AVX-512 SIMD instructions. Example:

  ```
  g++ -mavx512 your_program.cpp -o your_program
  ```

4.2 Linking Libraries

OpenMP programs require linking with the OpenMP library, which is handled automatically with the `-fopenmp` flag in the compiler command. Example:

```
g++ -g -O2 -fopenmp-simd -mavx2 -mavx512 your_program.cpp -o your_program
```

5 OpenMP Initialization

A number of OpenMP functions are used to control or get information about the parallel environment. These include:

- `omp_get_num_threads` (and set)
- `omp_get_thread_num` (and set)
- `omp_get_max_threads` (and set)
- `omp_get_thread_limit` (and set)
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_in_parallel`

The numnber of threads used is set by default to the number of cores, but can be overridden by the environment variable `OMP_NUM_THREADS` and the function `omp_set_num_threads`. The limit on the number of threads is a cap to prevent requesting more threads than the system can effiently support. You can also set this variable. with the environment variable `OMP_THREAD_LIMIT` and the function `omp_set_thread_limit` though this is not supported by all compilers (gcc 12 does not). OpenMP will not prevent you from starting more threads than can be efficiently supported.

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

int main() {
    /*
      default number of threads is the number of cores
      overridden by  environment variable OMP_NUM_THREADS
      overridden by direct call to omp_set_num_threads();
    */
    omp_set_num_threads(4); // manually set number of threads = 4
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
```

```
        int max_threads = omp_get_max_threads();
        int thread_limit = omp_get_thread_limit();
        cout << "Thread ID: " << thread_id << ", Number of threads: " << num_thr
            << ", Max threads: " << max_threads << ", Thread limit: " << thread_
    }
    cout << (omp_in_parallel()) ? "in parallel region\n" : "not in parallel regi

    omp_set_dynamic(1); // Enable dynamic adjustment of number of threads
    cout << "Dynamic adjustment: " << omp_get_dynamic() << endl;
    return 0;
}
```

6 Basic: Parallel Blocks

The basic building block of OMP is a parallel block. The rules:

- There is only one way in and out of a parallel block.
- Barriers are automatically inserted at the end of the block.
- Blocks must be structured

Example: An OpenMP parallel Block

```
#include <omp.h>
#include <unistd.h>
#include <iostream>
using namespace std;

void f() {
    sleep(3);
    #pragma omp critical
    {
        cout << "in thread: " << omp_get_thread_num() << ", total threads: " <<
    }
}

int main() {
    #pragma omp parallel
    {
        f();
    }
    return 0;
}
```

Here is an example of a parallel block that does not follow the rules:

```
if(count==1) goto more;
#pragma omp parallel
{
more: do_big_job(id);
if(++count>1) goto done;
}
done: if(!really_done()) goto more;
```

7 Parallel Loops

7.1 Example: multiplying two arrays

Multiplying two arrays element-wise using OpenMP. The array is shared by all threads. It is essential to partition it so that each thread works on a different part. The loop variable is private to each thread. So is any variable declared inside the block.

```
void multiply_arrays(const int a[], const int b[], int c[], int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}
```

Declaring variables private to each thread:

```
#include <omp.h>
#include <iostream>
using namespace std;

void multiply_arrays(const float a[], const float b[], float c[], int n) {
    float temp = (a[0] + b[0]) * (a[0] - b[0]);
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i] - temp;
    }
}
```

7.2 Reduction

Reduction is used to perform a reduction operation (e.g., sum) on variables in parallel.

```
float dot_product(const float a[], const float b[], int n) {
    float dot = 0.0;

    #pragma omp parallel for reduction(+:dot_product)
```

```
    for (int i = 0; i < n; i++) {
        dot += a[i] * b[i];
    }
    return dot;
}
```

Reduction can also be used to multiply, though usually the result would be too big, these numbers would have to be very close to 1 for this not to overflow:

```
double product(const float a[], int n) {
    double prod = 1;
    #pragma omp parallel for reduction(*:prod)
    for (int i = 0; i < n; i++) {
        prod *= a[i];
    }
    return prod;
}
```

7.3 Reduction Using SIMD Instructions

SIMD instructions are used to perform the same operation on multiple data points in parallel. As we have seen, it is possible for the computer to saturate memory, so how much performance gain threading and vectorization achieve is very much dependent on how memory-intensive the code is, and whether there is enough computation to keep the processors busy while they are waiting for memory.

```
float dot_product_simd(const float a[], const float b[], int n) {
    float dot = 0.0;

    #pragma omp parallel
    {
        float local_dot = 0.0;

        #pragma omp for simd reduction(+:local_dot)
        for (int i = 0; i < n; i++) {
            local_dot += a[i] * b[i];
        }
        #pragma omp critical
        {
            dot += local_dot;
        }
    }
    return dot;
}
```

7.4 Code Generated by SIMD Reduction

```
void multiply_arrays(const float a[], const float b[], float c[], int n) {
    #pragma omp parallel for simd safelen(8)
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}
```

```
_Z15multiply_arraysPKfS0_Pfi:
    .cfi_startproc
    endbr64
    ...
.L21:
    vmovups (%rdi,%rax), %ymm1
    vmulps (%rsi,%rax), %ymm1, %ymm0
    vmovups %ymm0, (%rdx,%rax)
    addq $32, %rax
    cmpq %rax, %rcx
    jne .L21
    ...
    ret
```

You really have to be careful though, I noticed that for reduction my compiler was not generating AVX2 instructions, but SSE (128 bit). So don't just assume the code is right, this really requires checking the assembly code.

7.5 Local Variables

Each thread may need its own local variables. Here is an example where each thread has its own local sum.

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

int main() {
    int n = 1000;
    int a[n], b[n];
    int global_sum = 0;

    // Initialize arrays
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = i * 2;
    }
```

```
    // Parallel region with local variables
    #pragma omp parallel
    {
        int local_sum = 0;

        #pragma omp for
        for (int i = 0; i < n; i++) {
            local_sum += a[i] + b[i];
        }

        #pragma omp critical
        {
            global_sum += local_sum;
        }
    }

    printf("Global sum: %d\n", global_sum);

    return 0;
}
```

7.6 Example: Matrix multiplication

```
void matmult(const float a[], const float b[], float c[], int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        #pragma omp simd safelen(8)
        for (int j = 0; j < n; j++) {
            c[i * n + j] = 0;
            for (int k = 0; k < n; k++) {
                c[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
}
```

7.7 HW: Gravity Simulator

8 Limits of OpenMP's Understanding

8.1 Data Races

8.2 False Sharing

8.3 Nested Parallelism

8.4 SIMD Instructions Not Supported

Will not do sorting using min/max (too specialized) Any other examples of special purpose use of AVX/AVX-512 that OpenMP does not support?

9 Advanced Topics

9.1 Nested Parallelism

9.2 Tasking

9.3 Memory Model

9.4 Genering Code to run on GPUs

We will be covering CUDA and HIP OpenMP has support for GPUs, but it does not directly generate GPU code Example: OpenMP 5.0 has a HIP backend