OpenMP

Open Multi Processing

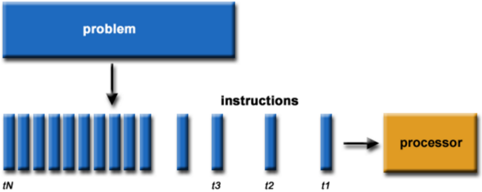
Philipp Quach

Seminar “Effiziente Programmierung”
University of Hamburg

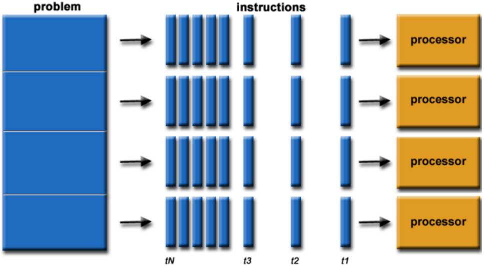
December 15th 2016

Content

- Simpler code is serial
  - One instruction at a time
  - executed one after the other
  - run on a single machine
- Performant code should be parallelized
  - concurrent execution



[Bar]



[Bar]

Introduction to OpenMP

- Supports C, C++ and Fortran
- Comes with the compiler
- Programmer directed
- High-level

Low vs high-level approach

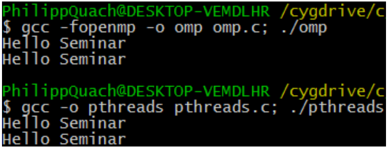
PThreads (low-level)

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

void* say_hello(void* data)
{
    char *str;
    str = (char*)data;
    printf("%s\n",str);
}

void main()
{
    pthread_t t1,t2;
    pthread_create(&t1,NULL,say_hello,"Hello Seminar");
    pthread_create(&t2,NULL,say_hello,"Hello Seminar");
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
}
```

Low vs high-level approach

OpenMP (high-level)

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

void main()
{
    #pragma omp parallel num_threads(2)
    printf("Hello Seminar\n");
}
```



Syntax

- Preprocessor directive begins with #pragma omp
- Followed by a specification as to what feature is being applied
- The parallelism is applied to the block of code following the preprocessor directive

```
foo(){
        #pragma omp <command specifier>
        {
                //some block of code that runs parallel
        }
}
```

- !$OMP <COMMAND SPECIFIER> in Fortran
- unknown pragmas are ignored by the compiler

omp.h

- #include <omp.h>
- provides many helpful functions
  - e.g. omp_get_thread_num()
- not required to run OpenMP code

Parallel construct

- #pragma omp parallel

```c
int main(void){
    #pragma omp parallel
    printf("hello Seminar\n");

    return EXIT_SUCCESS;
}
```

- Creates a team of n threads
- n usually depends on the number of cpu cores unless specified otherwise
- Parallelized block is executed once by every thread

```
PhilippQuach@DESKTOP-VEMDLHR /cygdrive/c/Users/PhilippQuach/Documents/OpenMP
$ gcc -fopenmp -o omp omp.c; ./omp
Hello Seminar
Hello Seminar
Hello Seminar
Hello Seminar
```

num_threads

- #pragma omp parallel num_threads(int)
- alternative: omp_set_num_threads(int) from omp.h

```c
int main(void){
    #pragma omp parallel num_threads(3)
    printf("hello Seminar\n");

    return EXIT_SUCCESS;
}
```

- Let's you specify the number of threads to be created

```
PhilippQuach@DESKTOP-VEMDLHR /cygdrive/c/Users/PhilippQuach/Documents/OpenMP
$ gcc -fopenmp -o omp omp.c; ./omp
hello Seminar
hello Seminar
hello Seminar
```

Parallel if

- #pragma omp parallel if(bool)

```c
int main(void){
    #pragma omp parallel if(0)
    printf("hello Seminar\n");

    return EXIT_SUCCESS;
}
```

- parallelizes only if the boolean within the if clause is true

```
PhilippQuach@DESKTOP-VEMDLHR /cygdrive/c/Users/PhilippQuach/Documents/OpenMP
$ gcc -fopenmp -o omp omp.c; ./omp
hello Seminar
```

For construct

- #pragma omp for

```c
int main(void){
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for
        for(int n=0; n<10; ++n)
        {
          printf(" %d", n);
        }
    }

    return EXIT_SUCCESS;
}
```

- Each thread of the active team handles a different part of the loop

```
PhilippQuach@DESKTOP-VEMDLHR /cygdrive/c/Users/PhilippQuach/Documents/OpenMP
$ gcc -fopenmp -o omp omp.c; ./omp
 0 5 1 6 2 7 3 8 4 9
```

Parallel for

- #pragma omp parallel for

```
int main(void){
    #pragma omp parallel for
    for(int n=0; n<10; ++n)
    {
        printf(" %d", n);
    }

    return EXIT_SUCCESS;
}
```

- Combines #pragma omp parallel and #pragma omp for into one line
- Creates a team of threads and assigns each thread a part of the loop

Schedule

- static (default), dynamic, auto, guided, runtime

```c
...//2 active threads
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
 0 5 1 6 2 7 3 8 4 9
```

```c
//2 threads
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
 0 3 1 4 2 5 6 9 7 8
```

Ordered

```
//2 threads
#pragma omp for ordered schedule(static)
for(int n=0; n<10; ++n) {
    printf(" %d", n);
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
 0 5 1 6 2 7 3 8 4 9
```

```
//2 threads
#pragma omp for ordered schedule(static)
for(int n=0; n<10; ++n) {
    #pragma omp ordered
    printf(" %d", n);
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
 0 1 2 3 4 5 6 7 8 9
```

Nested loops and the collapse clause

```c
//2 threads
#pragma omp for
for(int n=0; n<3; ++n) {
        for(int m=0; m<2; ++m) {
                printf("(%d%d)",n,m);
        }
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
(00)(20)(01)(21)(10)(11)
```

```c
//2 threads
#pragma omp for collapse(2)
for(int n=0; n<3; ++n) {
        for(int m=0; m<2; ++m) {
                printf("(%d%d)",n,m);
        }
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
(00)(11)(01)(20)(10)(21)
```

Sections

```c
//3 threads
#pragma omp sections
{
            {
                        printf("a ");
            }
            #pragma omp section
            {
                        printf("b1 ");
                        printf("b2 ");
            }
            #pragma omp section
            {
                        printf("c ");
            }
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp;./omp ;./omp ;./omp ;./omp ;./omp
a b1 c b2
b1 a c b2
a b1 c b2
b1 a c b2
c a b1 b2
a c b1 b2
```

Shared, unshared variables

- shared: One variable shared by all threads (default)
- private: Each thread has their own variable of this name

```
int main(void){
        int m,l=0;
        #pragma omp parallel for num_threads(2) private(l) shared(m)
        for(int n=0; n<10;n++) {
                l++;
                m++;
                printf("(%d,%d)",l,m);
        }
}
```

```
$ gcc -fopenmp -o omp omp.c;./omp
(1,1)(103625,2)(2,3)(103626,4)(3,5)(103627,6)(4,7)(103628,8)(5,9)(103629,10)
```

Firstprivate

```
int main(void){
        int m,l=0;
        #pragma omp parallel for num_threads(2) firstprivate(l)
        ↪  shared(m)
        for(int n=0; n<10;n++) {
                l++;
                m++;
                printf("(%d,%d)",l,m);
        }
}
```

```
$ gcc -fopenmp -o omp omp.c;./omp
(1,1)(1,2)(2,3)(2,4)(3,5)(3,6)(4,7)(4,8)(5,9)(5,10)
```

Offloading

- Execution also on other hardware than the computers CPU

```
#pragma omp target device(device_number)
{
    //executed on the device with the number specified
}
```

- omp.h provides helpful methods e.g. to set a default device or find out device numbers

Atomic

```c
int count = 0;
#pragma omp parallel num_threads(100)
{
        //#pragma omp atomic
        count++;
}
printf("Number of threads: %d\n", count);
```

Not atomic:

```
$ gcc -fopenmp -o omp omp.c;./omp ;./omp;./omp
Number of threads: 98
Number of threads: 100
Number of threads: 99
```

Reduction

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

int main()
{
        int count = 0;
        #pragma omp parallel num_threads(100) reduction(+:count)
        {
                count++;
        }
        printf("Number of threads: %d\n", count);
        return 0;
}
```

Critical

```c
#pragma omp parallel num_threads(2)
{
        if(omp_get_thread_num() == 0){
                #pragma omp critical(loop)
                for(int n = 0; n < 5; n++) printf("a");
        } else {
                #pragma omp critical(loop)
                for(int n = 0; n < 5; n++) printf("b");
        }
}
```

```
$ gcc -fopenmp -o omp omp.c; ./omp
aaaaabbbbb
```

Barrier

```c
#pragma omp parallel num_threads(2)
{
        if(omp_get_thread_num() == 1)
        {
                for(int n = 0; n < 10; n++) printf("n ");
        }
        #pragma omp barrier
        printf("\npast the barrier");
}
```

With barrier:

```
$ gcc -fopenmp -o omp omp.c;./omp
n n n n n n n n n n
past the barrier
past the barrier
```

Without barrier:

```
$ gcc -fopenmp -o omp omp.c;./omp
n
past the barriern n n n n n n n n
past the barrier
```

Nowait

```
#pragma omp parallel num_threads(2)
{
        #pragma omp for nowait
        for(int n = 0; n < 10; n++){
                printf("%d", omp_get_thread_num());
                if(omp_get_thread_num() == 1) printf("");
        }
        printf("\ndone with the loop");
}
```

With nowait:

```
$ gcc -fopenmp -o omp omp.c;./omp
01001001
done with the loop11
done with the loop
```

Without nowait:

```
$ gcc -fopenmp -o omp omp.c;./omp
0100100111
done with the loop
done with the loop
```

Requirements

- Compiler supporting OpenMP
- Set compiler flag for OpenMP e.g. -fopenmp
  - Produces serial code otherwise
- Link the runtime library libgomp-1.dll

Additional compilation

Additionally to the usual compilation:

- Reads omp directives and checks for correctness
- Substitution:
  - Replace sections by Do- and For-constructs
  - Implicit to explicit barrier
- Handles memory
- Applies some optimization
- Creates multithreading code from omp constructs
- Outlines parallel region to function

Transformed code example

Original:

```
void main(){
        #pragma omp parallel
        {
                #pragma omp for
                for( i = 0; i < n; i++ ){...}
        }
}
```

Transformed:

```
void outlined(...){
        tid = ompc_get_thread_num();
        ompc_static_init(tid, lower,upper,incr,.);
        for( i = lower;i < upper;i += incr ){ ... }
        ompc_barrier();
}

void main(){
        __ompc_fork(...,&outlined,...);
}
```

Parallel Overhead

- Time spent coordinating threads etc.
  - Initializing threads
  - Terminating threads
  - Coordination such as synchronization
- Aim: Minimize overheads

Speedup

- OpenMP uses a thread-pool
  - Threads are created once
  - Once done with their work, return to dock
  - Then wait for new work
- Speedup over serial code can vary strongly
- $Speedup(P) = \frac{T_{Serial}(P)}{T_{Elapsed}(P)} = \frac{1}{\frac{f}{P} - f + 1 + O_P \cdot P}$ (simplified)
- $Efficiency(P) = \frac{Speedup(P)}{P}$

Bad usage makes it worse

```c
#pragma omp parallel for private(j)
for(i=0;i<=100000;i++)
{
        for(j=0;j<=100000;j++)
        {
                #pragma omp atomic
                a++;
        }
}
printf("%lld", a);
```

Serial:

```
$ gcc -o omp omp.c; time ./omp
10000200001
real     0m24.882s
```

Parallel:

```
$ gcc -fopenmp -o omp omp.c; time ./omp
2618243971
real     0m39.045s
```

Thread-save:

```
$ gcc -fopenmp -o omp omp.c; time ./omp
10000200001
real     3m49.735s
```

Example speedup

Table: Matrix-Vector-Product

Size	Serial time	Parallel Time	Speedup
10000*10000	0.10	0.03	2.95
30000*30000	1.01	0.23	4.33
40000*40000	1.88	0.39	4.73

[App14]

- Execution with 4 cores, 8 logical processors/threads

Optimization

- Minimize Overheads
- Load balance: Threads should have similar runtime
- Thread-Safety causes waiting time
- Don't parallelize in inner loops
- Maximize parallel regions
- The ordered construct is slow
- Optimize barrier and nowait usage
- Avoid memory conflicts

How good is OpenMP

Pro:

- Target audience: general-purpose application programers
  - portability, maintainability, convenience
- Highly effective for simple loop based code

Contra:

- Too narrow for complexer code structures
- Doesn’t optimize for the specific hardware the code runs on

Summary

- OpenMP is easy to use
  - Parallelize by adding a few lines
  - Not necessary to rewrite existing code
- Not a substitution of low-level APIs
- Although high level, the many features allow for flexible control
- Possible speedup depends on hardware
- Poor parallelization may even slow down, optimize well!

Sources I

Appentra.
A widely-used algebraic code: Parallel computation of matrix-vector product.
http://www.appentra.com/parallel-computation-of-matrix-vector-product/, 2014.
[Online; accessed 8-December-2016].

Blaise Barney.
Introduction to Parallel Computing.
https://computing.llnl.gov/tutorials/parallel_comp/.
[Online; accessed 8-December-2016].

Lei Huang Barbara Chapman.
How OpenMP is Compiled.
https://iwomp.zih.tu-dresden.de/downloads/OpenMP-compilation.pdf.
[Online; accessed 8-December-2016].

Keld Helsgaun.
How to Get Good Performance by Using OpenMP.
http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf, 2010.
[Online; accessed 8-December-2016].

Sources II

Timothy G. Mattson.
How Good is OpenMP.
Hindawi Publishing Corporation, 2003.

Microsoft.
atomic.
https://msdn.microsoft.com/de-de/library/8ztckdts.aspx, 2016.
[Online; accessed 8-December-2016].

Joel Yliluoma.
Guide into OpenMP: Easy multithreading programming for C++.
http://bisqwit.iki.fi/story/howto/openmp/, 2007.
[Online; accessed 8-December-2016].