Tutorial: Introduction to OpenMP

Stéphane Zuckerman  Haitao Wei  Guang R. Gao

Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark, DE 19716, USA
{szuckerm, hwei, ggao}@udel.edu

Friday, September 26, 2014
Outline

1 Introduction
   - Introduction to OpenMP

2 OPENMP Basics

3 Where to Learn More
Introduction to OpenMP I

The OpenMP Framework

- Stands for Open MultiProcessing
- Three languages supported: C, C++, Fortran
- Supported on multiple platforms: UNIX, Linux, Windows, etc.
  - Very portable
- Many compilers provide OpenMP capabilities:
  - The GNU Compiler Collection (gcc) – OpenMP 3.1
  - Intel C/C++ Compiler (icc) – OpenMP 3.1 (and partial support of OpenMP 4.0)
  - Oracle C/C++ – OpenMP 3.1
  - IBM XL C/C++ – OpenMP 3.0
  - Microsoft Visual C++ – OpenMP 2.0
  - etc.

OpenMP’s Main Components

- Compiler directives
- A functions library
- Environment variables
A Brief History of OpenMP

- Several multi-tasking libraries already existed before OpenMP.
- Most of the time, they were platform-specific (Cray, NEC, IBM, etc.).
- In the 1990’s, symmetric multiprocessing makes a come-back, and shared-memory programming models become fashionable again.
- The Parallel Computing Forum tried to provide a standard, and failed.
- On October 28, 1997, a majority of industrial players and hardware vendors choose OpenMP as an “industrial standard.”
- OpenMP’s specifications belong to the Architecture Review Board (ARB) which is the only official instance to make the spec evolve.
- The current specification is OpenMP 4.0. Most compilers only implement OpenMP 3.x.
The OpenMP Model

- An OpenMP program is executed using a unique process
- Threads are activated when entering a parallel region
- Each thread executes a task composed of a pool of instructions
- While executing, a variable can be read and written in memory:
  - It can be defined in the stack of a thread: the variable is private
  - It can be stored somewhere in the heap: the variable is shared by all threads
Running OpenMP Programs: Execution Overview

OpenMP: Program Execution

- An OpenMP program is a sequence of serial and parallel regions
- A sequential region is always executed by the master thread: Thread 0
- A parallel region can be executed by multiple tasks at a time
- Tasks can share work contained within the parallel region
Running OpenMP Programs: Execution Overview

**OpenMP: Program Execution**

- An OpenMP program is a sequence of serial and parallel regions.
- A sequential region is always executed by the master thread: Thread 0.
- A parallel region can be executed by multiple tasks at a time.
- Tasks can share work contained within the parallel region.
OpenMP Parallel Structures

- Parallel loops

```c
for(i=...; i<N; i+=...)
{
    ...
}
```

Parallel For
OpenMP Parallel Structures

- Parallel loops
- Sections

```
for(i=...; i<N; i+=...) {
    ...
}
```

```
x = a+b;
y = x + c;
function(...);
for(i=...; i<N; i+=...) {
    ...
}
```
OpenMP Parallel Structures

- Parallel loops
- Sections
- Procedures through orphaning

---

*Parallel For*

```
for(i=...; i<N; i+=...) {
    ...
}
```

*Sections*

```
x = a+b;
y = x + c;
function(...);
for(i=...; i<N; i+=...) {
    ...
}
```

*Orphan Procedures*

```
function(...)  
```
OpenMP Parallel Structures

- Parallel loops
- Sections
- Procedures through orphaning
- Tasks
OpenMP Structure I

<table>
<thead>
<tr>
<th>Compilation Directives and Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>They define how to:</td>
</tr>
<tr>
<td>▶ Share work</td>
</tr>
<tr>
<td>▶ Synchronize</td>
</tr>
<tr>
<td>▶ Share data</td>
</tr>
<tr>
<td>They are processed as comments unless the right compiler option is specified on the command line.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fonctions and Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>They are part of a library loaded at link time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Once set, their values are taken into account at execution time</td>
</tr>
</tbody>
</table>
These two programming models are complementary:

- Both OpenMP and MPI can interface using C, C++, and Fortran.
- MPI is a multi-process environment whose communication mode is explicit (the user is in charge of handling communications).
- OpenMP is a multi-tasking environment whose communication between tasks is implicit (the compiler is in charge of handling communications).
- In general, MPI is used on multiprocessor machines using distributed memory.
- OpenMP is used on multiprocessor machines using shared memory.
- On a cluster of independent shared memory machines, combining two levels of parallelism can significantly speed up a parallel program’s execution.
OpenMP: Principles

- The developer is in charge of introducing OpenMP directives.
- When executing, the OpenMP runtime system builds a parallel region relying on the "fork-join" model.
- When entering a parallel region, the master task spawns ("forks") children tasks which disappear or go to sleep when the parallel region ends.
- Only the master task remains active after a parallel region is done.
Creating a Parallel Region: the `parallel` Directive

```c
#pragma omp parallel
{
    /* Parallel region code */
}
```
Data Sharing Clauses

- **shared(...)**: Comma-separated list of all variables that are to be shared by all OpenMP tasks.

- **private(...)**: Comma-separated list of all variables that are to be visible only by their task.
  - Variables that are declared private are “duplicated:” their content is unspecified when entering the parallel region, and when leaving the region, the privatized variable retains the content it had before entering the parallel region.

- **firstprivate(...)**: Comma-separated list of variables whose content must be copied (and not just allocated) when entering the parallel region.
  - The value when leaving the parallel remains the one from before entering it.

- **default (none|shared|private)**: Default policy w.r.t. sharing variables. If not specified, defaults to “shared”.
A First Example: Hello World

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic \
  -O3 -o omp_hello omp_hello.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#ifndef _OPENMP
#define omp_get_thread_num() 0
#endif

int main(void)
{
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("[%d] Hello, World!\n", tid);
    }
    return EXIT_SUCCESS;
}
```

**Figure:** omp_hello.c
Example: Privatizing Variables

examples:$ gcc -std=c99 -Wall -Wextra -pedantic -O3 \
     -o omp_private omp_private.c
omp_private.c: In function `main._omp_fn.0':
omp_private.c:8:11: warning: `a' is used uninitialized [-Wuninitialized]
   a = a + 716.;
   ^
omp_private.c:4:11: note: `a' was declared here
   float a = 1900.0;

#include <stdio.h>
#include <omp.h>
int main() {
    float a = 1900.0;
    #pragma omp parallel default(none) private(a)
    {
        a = a + 716.;
        printf("[\%d]\ta\=\%2f\n", omp_get_thread_num(), a);
    }
    printf("[\%d]\ta\=\%2f\n", omp_get_thread_num(), a);
    return 0;
}

Figure: omp_private.c
Sharing Data Between Threads

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

#ifndef _OPENMP
#define omp_get_thread_num() 0
#endif

int main(void)
{
    int ids[] = {0, 1, 2, 3, 4, 5, 6, 7};
    #pragma omp parallel default(none) shared(ids)
    {
        printf("[%d]\tHello, World!\n", ids[omp_get_thread_num()]);
    }

    return EXIT_SUCCESS;
}
```

Figure: hello2.c
Capturing Privatized Variables’ Initial Values

szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic -O3 -o omp_firstprivate omp_firstprivate.c

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

int main() {
    float a = 1900.0;

    #pragma omp parallel default(none) \ firstprivate(a)
    {
        a = a + 716.;
        printf("a = %f\n",a);
    }

    printf("a = %f\n",a);

    return 0;
}
```

Figure : omp_firstprivate.c
Scope of OpenMP Parallel Regions

When calling functions from a parallel region, local and automatic variables are implicitly private to each task (they belong to their respective task’s stack). Example:

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

void sub(void);
int main(void) {
    #pragma omp parallel default(shared)
    {
        sub();
    }
    return 0;
}

void sub(void) {
    int a = 19716;
    a += omp_get_thread_num();
    printf("a = %d\n", a);
}
```
Parallel Loops

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

int main(void)
{
    #pragma omp parallel
    {
        int n_threads = omp_get_num_threads();
        #pragma omp for
        for (int i = 0; i < n_threads; ++i) {
            printf("[%d]\tHellow, World!\n", i);
        }
    }
}
```

Figure: parallel_for.c

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic -O3 -o omp_for parallel_for.c
examples$ ./omp_for
[1] Hellow, World!
[0] Hellow, World!
```
Parallel Loops: A Few Things to Remember

1. The iterator of a `omp for` loop must use additions/subtractions to get to the next iteration (no `i *= 10` in the postcondition)

2. The iterator of the outermost loop (which directly succeeds to the `omp for` directive) is always private, but not the ones in other nested loops!

3. There is an implicit barrier at the end of the loop. You can remove it by adding the clause `nowait` on the same line: `#pragma omp for nowait`

4. How the iterations are distributed among threads can be defined using the `schedule` clause.
Parallel Loops I
Specifying the Schedule Mode

The syntax to define a scheduling policy is `schedule(ScheduleType, chunksize)`. The final line should like this:

```c
#pragma omp parallel default(none) \ 
        shared(...) private(...) firstprivate(...) 
{
    #pragma omp for schedule(...) lastprivate(...) 
    for (int i = InitVal; ConditionOn(i); i += Stride) 
    { /* loop body */ } 
}
```

// or, all in one directive:

```c
#pragma omp parallel for default(none) shared(...) private(...) \ 
        firstprivate(...) lastprivate(...) 
    for (int i = InitVal; ConditionOn(i); i += Stride) 
    { /* loop body */ } 
```
The number of iterations in a loop is computed as follows:

\[ \text{NumIterations} = \left\lfloor \frac{|\text{FinalVal} - \text{InitVal}|}{\text{Stride}} \right\rfloor + |\text{FinalVal} - \text{InitVal}| \mod \text{Stride} \]

The number of iteration chunks is thus computed like this:

\[ \text{NumChunks} = \left\lfloor \frac{\text{NumIterations}}{\text{ChunkSize}} \right\rfloor + \text{NumIterations} \mod \text{ChunkSize} \]
### Static Scheduling

`schedule(static,chunksize)` distributes the iteration chunks across threads in a round-robin fashion

- **Guarantee:** if two loops with the same “header” (precondition, condition, postcondition, and chunksize for the `parallel for` directive) succeed to each other, the threads will be assigned the *same* iteration chunks
- **By default,** `chunksize` is equal to `OMP_NUM_THREADS`
- **Very useful when iterations take roughly the same time to perform (e.g., dense linear algebra routines)**

### Dynamic Scheduling

`schedule(dynamic,chunksize)` divides the iteration space according to `chunksize`, and creates an “abstract” queue of iteration chunks. If a thread is done processing its chunk, it dequeues the next one from the queue. By default, `chunksize` is 1.

Very useful if the time to process individual iterations varies.
Parallel Loops IV
Specifying the Schedule Mode

Guided Scheduling

`guided,chunksize` Same behavior as `dynamic`, but the chunksize is divided by two each time a thread dequeues a new chunk. The minimum size is one, and so is the default. Very useful if the time to process individual iterations varies, and the amount of work has a “trail”
Parallel Loops I
Specifying the Schedule Mode

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
const double MAX = 100000.;

double sum(const int n) {
    const int id = omp_get_thread_num(); double f = 0.0;
    const int bound = id == 0 ? n*1001 : n;
    for (int i = 0; i < bound; ++i) f += i;
    return f;
}
```

**Figure**: omp_for_schedule.c
Parallel Loops I

Specifying the Schedule Mode

int main(void) {
    printf("MAX_=\_%.2f\n",MAX);
    double acc = 0.0;
    int* sum_until = malloc(MAX*sizeof(int));
    if (!sum_until) perror("malloc"), exit( EXIT_FAILURE );
    for (int i = 0; i < (int)MAX; ++i) sum_until[i] = rand () % 100;
    #pragma omp parallel default(none) shared(sum_until) firstprivate(acc)
    {
        /*
        * Use the OMP_SCHEDULE environment variable on the command
        * line to specify the type of scheduling you want, e.g.:
        * export OMP_SCHEDULE="static" or OMP_SCHEDULE="dynamic,10"
        * or OMP_SCHEDULE="guided,100"; ./omp_schedule
        */
        #pragma omp for schedule(runtime)
        for (int i = 0; i < bound; i+=1) {
            acc += sum( sum_until[i] );
        }
        printf ("[%d]\tMy\_sum\_=_%.2f\n", omp_get_thread_num(), acc);
    }
    free(sum_until);
    return 0;
}

Figure: omp_for_schedule.c
Parallel Loops
Specifying the Schedule Mode: Outputs I

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic \ -O3 -o omp_schedule omp_for_schedule.c

szuckerm@evans201g:examples$ export OMP_NUM_THREADS=4 OMP_PROC_BIND=true

szuckerm@evans201g:examples$ export OMP_SCHEDULE="static"

szuckerm@evans201g:examples$ export OMP_SCHEDULE="static,1"

szuckerm@evans201g:examples$ time ./omp_schedule

MAX = 100000.00
[0] My sum = 41299239778797.00
[1] My sum = 40564464.00
[3] My sum = 40174472.00
[2] My sum = 40502412.00

real 0m11.911s
user 0m11.930s
sys 0m0.004s
```
Parallel Loops
Specifying the Schedule Mode: Outputs I

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic -O3 -o omp_schedule omp_for_schedule.c
szuckerm@evans201g:examples$ export OMP_NUM_THREADS=4 OMP_PROC_BIND=true

szuckerm@evans201g:examples$ export OMP_SCHEDULE="static"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 41299239778797.00
[1] My sum = 40564464.00
[3] My sum = 40174472.00
[2] My sum = 40502412.00
real  0m11.911s
user  0m11.930s
sys  0m0.004s

szuckerm@evans201g:examples$ export OMP_SCHEDULE="static,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 41487115603934.00
[1] My sum = 40266669.00
[3] My sum = 40319644.00
[2] My sum = 40468898.00
real  0m11.312s
user  0m11.356s
sys  0m0.004s
```
Parallel Loops
Specifying the Schedule Mode: Outputs II

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="dynamic,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 166164785868.00
[1] My sum = 55011312.00
[2] My sum = 46974801.00
[3] My sum = 58218664.00
real 0m0.546s
user 0m0.576s
sys 0m0.004s
```
### Parallel Loops

#### Specifying the Schedule Mode: Outputs II

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>export OMP_SCHEDULE=&quot;dynamic,1000&quot;</code></td>
<td>Max = 100000.00</td>
<td>0m0.546s</td>
</tr>
<tr>
<td></td>
<td>[0] My sum = 1661647855868.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] My sum = 55011312.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2] My sum = 46974801.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3] My sum = 58218664.00</td>
<td></td>
</tr>
<tr>
<td><code>time ./omp_schedule</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>export OMP_SCHEDULE=&quot;dynamic,1&quot;</code></td>
<td>Max = 100000.00</td>
<td>0m0.023s</td>
</tr>
<tr>
<td></td>
<td>[1] My sum = 57886783.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] My sum = 76809786053.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2] My sum = 47423265.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3] My sum = 56452544.00</td>
<td></td>
</tr>
<tr>
<td><code>time ./omp_schedule</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 3092266894167.00
[3] My sum = 44855495.00
[2] My sum = 45989686.00
[1] My sum = 40596797.00

real 0m8.437s
user 0m8.452s
sys 0m0.008s

szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 3092266894167.00
[3] My sum = 44855495.00
[2] My sum = 45989686.00
[1] My sum = 40596797.00

real 0m5.401s
user 0m5.438s
sys 0m0.008s
Parallel Loops
Specifying the Schedule Mode: Outputs III

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 3092266894167.00
[3] My sum = 44855495.00
[2] My sum = 45989686.00
[1] My sum = 40596797.00
real 0m8.437s
user 0m8.452s
sys 0m0.008s

szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 17508269385607.00
[1] My sum = 49603788.00
[2] My sum = 40584346.00
[3] My sum = 54438904.00
real 0m5.401s
user 0m5.438s
sys 0m0.008s
```
Parallel Loops

The lastprivate Clause

```c
int main(void) {
    double acc = 0.0; const int bound = MAX;
    printf("[%d]\tMAX=\t%.2f\n",omp_get_thread_num(),MAX);
    int* sum_until = smalloc(MAX*sizeof(int));
    for (int i = 0; i < bound; ++i)
        sum_until[i] = rand () % 100;
    #pragma omp parallel for default(none) shared(sum_until) \ 
        schedule(runtime) firstprivate(acc) lastprivate(acc)
    for (int i = 0; i < bound; i+=1)
        acc += sum( sum_until[i] );
    printf("Value of the last thread to write to acc=\t%.2f\n",acc);
    free(sum_until);
    return EXIT_SUCCESS;
}
```

Figure: omp_for_lastprivate.c
Incrementing a Global Counter I
Racy OpenMP Version

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
unsigned long g_COUNTER = 0;

int
main(void)
{
    int n_threads = 1;
    #pragma omp parallel default(none) shared(n_threads, stdout, g_COUNTER)
    {
        #pragma omp master
        {
            n_threads = omp_get_num_threads();
            printf("n_threads = %d\n", n_threads); fflush(stdout);
        }

        ++g_COUNTER;
    }

    printf("g_COUNTER = %lu\n", g_COUNTER);
    return EXIT_FAILURE;
}
```
szuckerm@evans20lg:examples$ for i in $(seq 100) 
> do ./global_counter ; done | sort | uniq 
n_threads = 4  g_COUNTER = 2 
n_threads = 4  g_COUNTER = 3 
n_threads = 4  g_COUNTER = 4
Incrementing a Global Counter
Using a Critical Section

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

unsigned long g_COUNTER = 0;

int main(void) {
    int n_threads = 1;
    #pragma omp parallel default(none) shared(n_threads, stdout, g_COUNTER)
    {
        #pragma omp master
        {
            n_threads = omp_get_num_threads();
            printf("n_threads = %d\n", n_threads); fflush(stdout);
        }

        #pragma omp critical
        {
            ++g_COUNTER;
        }

        printf("g_COUNTER = %lu\n", g_COUNTER);
    }
    return EXIT_FAILURE;
}
```
Incrementing a Global Counter
Using an Atomic Section

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
unsigned long g_COUNTER = 0;

int main(void) {
    int n_threads = 1;
    #pragma omp parallel default(none) shared(n_threads, stdout, g_COUNTER)
    {
        #pragma omp master
        {
            n_threads = omp_get_num_threads();
            printf("n_threads = %d\t", n_threads); fflush(stdout);
        }
    
        #pragma omp atomic
        ++g_COUNTER;
    }
    printf("g_COUNTER = %lu\n", g_COUNTER);
    return EXIT_FAILURE;
}
```
critical Directive

`#pragma omp critical [(name)]`

Guarantees that only one thread can access the sequence of instructions contained in the (named) critical section. If no name is specified, an “anonymous” name is automatically generated.

atomic Directive

`#pragma omp atomic`

Guarantees the atomicity of the *single* arithmetic instruction that follows. On architectures that support atomic instructions, the compiler can generate a low-level instruction to ensure the atomicity of the operation. Otherwise, *atomic* is equivalent to *critical*.
Synchronization in OpenMP II

### barrier Directive

`#pragma omp barrier`

All threads from a given parallel region must wait at the barrier. All parallel regions have an implicit barrier. All `omp for` loops do too. So do `single` regions.

### single Directive

Guarantees that a single thread will execute the sequence of instructions located in the `single` region, and the region will be executed only once. There is an implicit barrier at the end of the region.
**master Directive**

Guarantees that only the master thread (with ID = 0) will execute the sequence of instructions located in the `single` region, and the region will be executed only once. There is NO implicit barrier at the end of the region.

**nowait Clause**

`nowait` can be used on `omp for`, `single`, and `critical` directives to remove the implicit barrier they feature.
OpenMP 3.x brings a new way to express parallelism: tasks.

- Tasks must be created from within a single region.
- A task is spawned by using the directive `#pragma omp task`.
- Tasks synchronize with their siblings (i.e., tasks spawned by the same parent task) using `#pragma omp taskwait`. 
Case Study: Fibonacci Sequence

We’ll use the Fibonacci numbers example to illustrate the use of tasks:

```c
/**
 * brief Computes Fibonacci numbers
 * param n the Fibonacci number to compute
 */
u64 xfib(u64 n) {
    return n < 2 ? // base case?
        n // fib(0) = 0, fib(1) = 1
    : xfib(n-1) + xfib(n-2);
}
```

<table>
<thead>
<tr>
<th></th>
<th>Average Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential - Recursive</td>
<td>196051726.08</td>
</tr>
</tbody>
</table>

**Table**: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz
#ifndef UTILS_H_GUARD
#define UTILS_H_GUARD
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include "rdtsc.h"

static inline void fatal(const char* msg) { perror(msg), exit(errno); } 
static inline void sfree(void* p) { if (p) { *(char*)p = 0;} free(p); } 
static inline void* scalloc(size_t nmemb, size_t size) {
    void* p = calloc(nmemb,size); if (!p) { fatal("calloc"); } return p;
} 
static inline void* smalloc(size_t size) {
    void* p = malloc(size); if (!p) { fatal("malloc"); } return p;
}
static inline void usage(const char* progname) {
    printf("USAGE:\n%s positive_number\n", progname); exit(0);
}
void u64_measure(u64 (*func)(u64), u64 n, u64 n_reps, const char* msg);
void u64func_time(u64 (*func)(u64), u64 n, const char* msg);
#endif // UTILS_H_GUARD
#ifndef COMMON_H_GUARD
#define COMMON_H_GUARD
#include "utils.h" // for smalloc(), sfree(), fatal(), scalloc(), ...
#define FIB_THRESHOLD 20

typedef uint64_t u64; typedef uint32_t u32; typedef uint16_t u16;
typedef uint8_t u8; typedef int64_t s64; typedef int32_t s32;
typedef int16_t s16; typedef int8_t s8;

u64 xfib(u64); u64 trfib(u64,u64,u64);
u64 trFib(u64); u64 sfib(u64);
u64 memoFib(u64); u64 memofib(u64,u64*);

void* mt_memofib(void*); u64 mt_memoFib(u64);
void* mtfib(void*); u64 mtFib(u64);

u64 oFib(u64); u64 ofib(u64);
u64 o_memoFib(u64); u64 o_memofib(u64,u64*);
#endif /* COMMON_H_GUARD */
#ifndef MT_H_GUARD
#define MT_H_GUARD
#include <pthread.h>
typedef struct fib_s { u64 *up, n; } fib_t;
typedef struct memofib_s { u64 *up, *vals, n; } memofib_t;
static inline pthread_t* spawn(void* (*func)(void*), void* data) {
    pthread_t* thread = smalloc(sizeof(pthread_t)); int error = 0;
    do {
        errno = error = pthread_create(thread, NULL, func, data);
    } while (error == EAGAIN);
    if (error) fatal("pthread_create");
    return thread;
}
static inline void sync(pthread_t* thread) {
    int error = 0; void* retval = NULL;
    if ( (errno = ( error = pthread_join(*thread, &retval) ) ) )
        fatal("pthread_join");
    sfree(thread);
}
#endif // MT_H_GUARD
#include "common.h"
#include "mt.h"

void* mtfib(void* frame) {
    fib_t* f = (fib_t*) frame;
    u64 n = f->n, *up = f->up;
    if (n < FIB_THRESHOLD)
        *up = sfib(n), pthread_exit(NULL);
    u64 n1 = 0, n2 = 0;
    fib_t frame1 = { .up = &n1, .n = f->n-1 },
        frame2 = { .up = &n2, .n = f->n-2 };
    pthread_t *thd1 = spawn(mtfib,&frame1),
        *thd2 = spawn(mtfib,&frame2);
    sync(thd1); sync(thd2);
    *up = n1+n2;
    return NULL;
}

u64 mtFib(u64 n) {
    u64 result = 0;
    fib_t f = { .up = &result, .n = n };
    (void)mtfib(&f);
    return result;
}
Computing Fibonacci Numbers: Code II
Naïve Pthread and OpenMP

```c
#include "common.h"
#include <omp.h>

u64 ofib(u64 n) { u64 n1, n2;
    if (n < FIB_THRESHOLD) return sfib(n);
    # pragma omp task shared(n1)
    n1 = ofib(n-1);
    # pragma omp task shared(n2)
    n2 = ofib(n-2);
    # pragma omp taskwait
    return n1 + n2;
}

u64 oFib(u64 n) { u64 result = 0;
    # pragma omp parallel
    {
        # pragma omp single nowait
        { result = ofib(n); } // parallel
    }
    return result;
}
```
### Table: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz

<table>
<thead>
<tr>
<th></th>
<th>Average Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential - Recursive</td>
<td>196051726.08</td>
</tr>
<tr>
<td>Parallel - PThread</td>
<td>2837871164.24</td>
</tr>
<tr>
<td>Parallel - OpenMP</td>
<td>17707012.14</td>
</tr>
</tbody>
</table>
```c
#include "common.h"
#include "mt.h"

void* mt_memofib(void* frame) { memofib_t* f = (memofib_t*) frame;
    u64 n = f->n * vals = f->vals, *up = f->up;
    if (n < FIB_THRESHOLD) *up = vals[n] = sfib(n), pthread_exit(NULL);
    if (vals[n] == 0) { u64 n1 = 0, n2 = 0;
        memofib_t frame1 = {.up=&n1,.n=f->n-1,.vals=vals},
            frame2 = {.up=&n2,.n=f->n-2,.vals=vals};
        pthread_t *thd1 = spawn(mt_memofib,&frame1),
                  *thd2 = spawn(mt_memofib,&frame2);
        sync(thd1); sync(thd2);
        vals[n] = n1 + n2; }
    *up = vals[n], pthread_exit(NULL);
}

u64 mt_memoFib(u64 n) { u64 result = 0;
    u64* fibvals = scalloc(n+1,sizeof(u64));
    fibvals[1]=1; memofib_t f={.up=&result,.n=n,.vals=fibvals};
    (void)mt_memofib(&f);
    return result;
}
```
#include "common.h"
#include <omp.h>

u64 o_memofib(u64 n, u64* vals) {
    if (n < FIB_THRESHOLD) return sfib(n);
    if (vals[n] == 0) {
        u64 n1 = 0, n2 = 1;
        #pragma omp task shared(n1,vals)
        n1 = o_memofib(n-1,vals);
        #pragma omp task shared(n2,vals)
        n2 = o_memofib(n-2,vals);
        #pragma omp taskwait
        vals[n] = n1 + n2;
    }
    return vals[n];
}

u64 o_memoFib(u64 n) {
    u64 result=0, *fibvals=calloc(n+1,sizeof(u64));
    #pragma omp parallel
    {
        #pragma omp single nowait
        { fibvals[1] = 1; result = o_memofib(n,fibvals); }
    }
    return result; }
#include "common.h"

u64 memofib(u64 n, u64* vals) {
    if (n < 2)
        return n;
    if (vals[n] == 0)
        vals[n] = memofib(n-1,vals) + memofib(n-2,vals);
    return vals[n];
}

u64 memoFib(u64 n) {
    u64* fibvals = calloc(n+1,sizeof(u64));
    fibvals[0] = 0; fibvals[1] = 1;
    u64 result = memofib(n,fibvals);
    sfree(fibvals);
    return result;
}
**Computing Fibonacci Numbers: Code IV**

Memoization using Serial, Pthread and OpenMP

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential - Recursive</td>
<td>196051726.08</td>
</tr>
<tr>
<td>Parallel - PThread</td>
<td>2837871164.24</td>
</tr>
<tr>
<td>Parallel - OpenMP</td>
<td>17707012.14</td>
</tr>
<tr>
<td>Parallel - PThread - Memoization</td>
<td>2031161888.56</td>
</tr>
<tr>
<td>Parallel - OpenMP - Memoization</td>
<td>85899.58</td>
</tr>
<tr>
<td>Sequential - Memoization</td>
<td>789.70</td>
</tr>
</tbody>
</table>

**Table:** Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz
#include "common.h"

u64 trfib(u64 n, u64 acc1, u64 acc2) {
    return n < 2 ?
        acc2 :
        trfib( n-1, acc2, acc1+acc2);
}

u64 trFib(u64 n) { return trfib(n, 0, 1); }

#include "common.h"

u64 sfib(u64 n) {
    u64 n1 = 0, n2 = 1, r = 1;
    for (u64 i = 2; i < n; ++i) {
        n1 = n2;
        n2 = r;
        r = n1 + n2;
    }
    return r;
}
## Computing Fibonacci Numbers: Code II
When Serial is MUCH Faster Than Parallel

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential - Recursive</td>
<td>196051726.08</td>
</tr>
<tr>
<td>Parallel - PThread</td>
<td>2837871164.24</td>
</tr>
<tr>
<td>Parallel - OpenMP</td>
<td>17707012.14</td>
</tr>
<tr>
<td>Parallel - PThread - Memoization</td>
<td>2031161888.56</td>
</tr>
<tr>
<td>Parallel - OpenMP - Memoization</td>
<td>85899.58</td>
</tr>
<tr>
<td>Sequential - Memoization</td>
<td>789.70</td>
</tr>
<tr>
<td>Sequential - Tail Recursive</td>
<td>110.78</td>
</tr>
<tr>
<td>Sequential - Iterative</td>
<td>115.02</td>
</tr>
</tbody>
</table>

**Table:** Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz
Learning More About Multi-Threading and OpenMP

Books (from most theoretical to most practical)
- Herlihy and Shavit, *The Art of Multiprocessor Programming*
- Rünger and Rauber, *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*
- Kumar, *Introduction to Parallel Computing*
- Chapman, Jost, and Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*

Internet Resources
- “The OpenMP® API specification for parallel programming” at openmp.org
  - Provides all the specifications for OpenMP, in particular OpenMP 3.1 and 4.0
  - Lots of tutorials (see http://openmp.org/wp/resources/#Tutorials)

Food for Thoughts
- Lee, “The Problem with Threads” (available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf)
Boehm, Hans-J. “Threads Cannot Be Implemented As a Library”. In: *SIGPLAN Not.* 40.6 (June 2005), pp. 261–268. ISSN: 0362-1340. DOI: 10.1145/1064978.1065042. URL: http://doi.acm.org/10.1145/1064978.1065042.


