Schriftliche Ausarbeitung

OpenMP

vorgelegt von

Philipp Quach

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Software-System-Entwicklung
Matrikelnummer:	6706421

Betreuer:	Konstantinos Chasapis

Hamburg, 03.03.2017

Abstract

Open Multi-Processing (short: OpenMP) is a library for the programming languages C, C++ and Fortran that can be used to write parallel programs. Among the libraries that share this purpose, OpenMP stands out due to the simplicity of applying it to already existing, working code.
This paper gives an Introduction to OpenMP, then for the main part explains the most important features, displays a "behind the scenes" over the compilation process and talks about performance. Code examples are in C.

Contents

1 Introduction **4**
1.1 Parallel Programming 4
1.2 Introduction to OpenMP 5

2 Features **7**
2.1 Parallel Construct 7
2.2 Loops 8
2.2.1 For construct 8
2.2.2 Scheduling 8
2.2.3 Nested loops 9
2.3 Sections 10
2.4 Shared, unshared variables 11
2.5 Offloading 11
2.6 Thread-Safety 12
2.7 Synchronization 13

3 Compilation **15**

4 Performance **17**

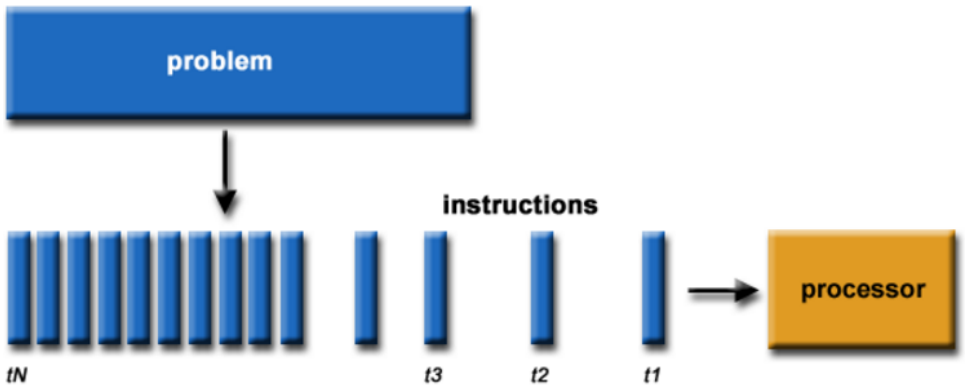
5 Conclusion **19**

Bibliography **20**

1 Introduction

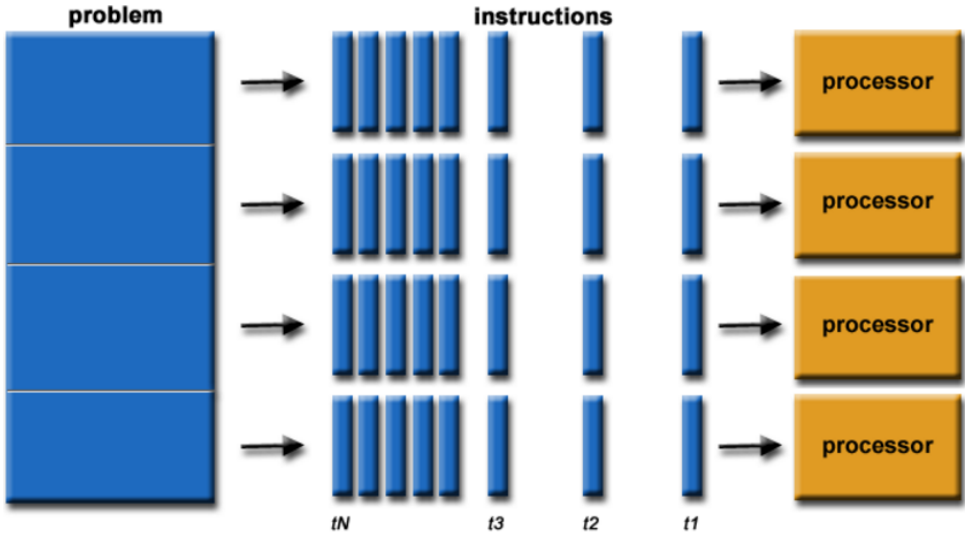
1.1 Parallel Programming

When writing code in order to solve a problem, we usually define instructions that we execute one after the other with only one processor. Code of this kind is called serial code.



[Bar]

In the case of parallel programming however, we split the problem into sub-problems and define a set of instructions for each of those. Each of those sets of instructions can be executed on a different processor. Having multiple processors working together to solve the problem can make the program a lot faster.



[Bar]

1.2 Introduction to OpenMP

OpenMP works with the programming languages C, C++ and Fortran. It can be seen as a downside that it works only with those, however these are languages that are known to produce very fast code. Hence aiming for high performance in terms of speed may lead you to choose one of these languages anyway, so you may as well see it as an advantage that it works with all three. OpenMP comes with the compiler, so there is no need to download a software package for it, just a version of a compiler that supports it. One such compiler would be the GCC-compiler, used to compile the code examples in this paper.
OpenMP uses a programmer-directed approach, which means that the programmer is in charge of defining which parts of the code are to be parallelized. The opposite approach would be automatic parallelization, which would mean that the programmer himself wouldn't need to change any part of his serial code in order to create the parallel version. As of today there is no working solution to the automatic approach. In fact OpenMP works on a very high level compared to other libraries and keeps the workload on the programmer relatively low.
The following code uses the PThreads-library, one that works on a lower level:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void* say_hello(void* data)
5  {
6      char *str;
7      str = (char*)data;
8      printf("%s\n",str);
9  }
10
11 void main()
12 {
13     pthread_t t1,t2;
14     pthread_create(&t1,NULL,say_hello,"Hello Seminar");
15     pthread_create(&t2,NULL,say_hello,"Hello Seminar");
16     pthread_join(t1,NULL);
17     pthread_join(t2,NULL);
18 }
```

In PThreads it is necessary to write many steps of the parallelization manually: Creating the thread-variables in line 13, starting the threads in lines 14 and 15, joining these back into the main algorithm in lines 16 and 17 and passing a starting-function as well as other parameters for each thread-creation to work with.
Looking at the following OpenMP code we will see in what way exactly OpenMP works on a higher level:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main()
5  {
6      #pragma omp parallel num_threads(2)
7      printf("Hello Seminar\n");
8  }
```

Line 6 is the only line that OpenMP adds to the original, serial code. Both, the PThreads and the OpenMP code, print "Hello Seminar" twice into the console by creating two threads and having both threads print it once. However, it is easy to note that the OpenMP code is a lot simpler. While PThreads requires you to embed the parallelization deeply into your code, OpenMP leaves the original, serial code completely intact, adding only short preprocessor directives.

The syntax of OpenMP looks as follows:

```
1  foo(){
2          #pragma omp <command specifier>
3          {
4                  //some block of code that runs parallel
5          }
6  }
```

The preprocessor directive begins with "#pragma omp", followed by a specification of the OpenMP feature that is to be applied. The feature specified is then being applied to the block of code following the preprocessor directive, starting with "{", ending with "}". An advantage of the pragma statement is that unknown pragmas are being ignored by the compiler. Hence you can run OpenMP code serial by simply not telling your compiler to recognize the pragma. Unknown pragma would mean that if the compiler came across the "#pragma omp" not recognizing what "omp" is, then it would ignore it; if however it would recognize the "#pragma omp" but come across a broken command specifier then it wouldn't interpret that as unknown and ignore, but recognize it as an error, since the compiler knows what "#pragma omp" means, knowing what command specifiers would be valid.

One optional header that can be included is <omp.h>. It provides many helpful functions to work with such as omp_get_thread_num() used to return the ID of a thread.

2 Features

OpenMP includes many different features for various purposes, this chapter will explain the most important of them one by one.

2.1 Parallel Construct

```
1 int main(void){
2     #pragma omp parallel
3     printf("hello Seminar\n");
4 
5     return EXIT_SUCCESS;
6 }
```

The parallel construct is the construct that creates threads and is hence fundamental in that it's always the feature that is called first. The number of threads that it creates is dependant on the number of cpu cores that are available on the executing computer. Each created thread executes the block of code following exactly once. For instance a machine with 2 cores creating 4 logical processors via hyper-threading would print "Hello Seminar" 4 times if it were to run aboves code.

Instead of creating the default number of threads based on your hardware you may specify the number of threads that are to create by using the num_threads construct.

```
1 int main(void){
2     #pragma omp parallel num_threads(3)
3     printf("hello Seminar\n");
4 
5     return EXIT_SUCCESS;
6 }
```

It is also possible to restrict the creation of threads by a condition. The following code would be serial as the boolean passed into the if statement is false. Only if the condition is fulfilled does the parallel construct go into effect.

```
1 int main(void){
2     #pragma omp parallel if(0)
3     printf("hello Seminar\n");
4 
5     return EXIT_SUCCESS;
6 }
```

2.2 Loops

Parallelizing loops is a big strong point of OpenMP. They appear very frequently and OpenMP handles them very efficiently in terms of simplicity and performance.

2.2.1 For construct

```
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;
}
```

The for construct divides the affected loop into equally sized portions for each active thread to work on. In the program above thread 0 would handle the iterations with n = 0 to n = 4 while thread 1 would handle those with n = 5 to n = 9. The console output of the program would be "0 5 1 6 2 7 3 8 4 9" with thread 0 starting with the printout of 0 while thread 1 starts with that of 5 and then both threads advancing their portion of the loop at similar speed. It is not necessarily the case that thread 0 happens to be faster than thread 1, the printout could possibly end up being "5 0 6 1 7 2 8 3 9 4".

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

    return EXIT_SUCCESS;
}
```

Constructs can be combined with each other for shorter code. The above first creates threads with the parallel construct and then parallelizes the loop using the for construct.

2.2.2 Scheduling

The for construct divides the loop into equally sized portions and in the case of 2 threads just cuts the loop in half, but there is a construct for better control over how the loop is

supposed to be divided: The schedule construct. There is different available options to schedule including static which is the default, dynamic, auto, guided and runtime.

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

Aboves schedule(dynamic, 3) for instance causes the loop to be divided in steps of 3. Thread 0 doing the iterations from n = 0 to n = 2, thread 1 doing n = 3 to n = 5, thread 0 doing n = 6 to n = 8 and thread 1 doing n = 9. So thread 0 prints "0 1 2 6 7 8" while thread 1 prints "3 4 5 9", hence the output of aboves code could be "0 3 1 4 2 5 6 9 7 8".

There is oftenly the requirement that instructions are to be executed in a certain order which could be destroyed by parallelism.

```
//2 threads
#pragma omp for ordered schedule(static)
for(int n=0; n<10; ++n) {
    doSomethingInParallel();
    #pragma omp ordered
    doSomethingOrdered();
}
```

The ordered construct allows one to define a block of code in which the serial order is maintained while the rest of the for loop is running parallel. In aboves example the iteration with n = 5, which thread 1 starts with, would finish doSomethingInParallel() before the iteration with n = 4, which is the last one that thread 0 executes, would finish it. Coming across the "#pragma omp ordered" the iteration with n = 5 would halt its progress and wait for the iterations with n = 0 to n = 4 to execute doSomethingOrdered() before doing it itself.

2.2.3 Nested loops

Having loops inside of loops requires some special direction as to which thread does which iterations.

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

The above simply divides the outer loop letting thread 0 do the iterations with n = 0 and n = 1 while thread 1 does the iteration with n = 2. The output would be "(00)(20)(01)(21)(10)(11)". This is an uneven workload distribution as thread 0 prints 4 of the tuples while thread 1 only prints 2 of them.

Ideally, both threads would do 3 of the printouts. The collapse clause can be used to let OpenMP consider both loops when dividing into the 2 parts.

```
//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);
        }
}
```

With the above thread 0 would print (00) up to (10) while thread 1 would print (11) up to (21), creating a printout of "(00)(11)(01)(20)(10)(21)" that shows even distribution.

2.3 Sections

Defining sections is another way to create parallelism.

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

In the above, thread 0 would print "a", thread 1 would print "b1" and "b2" and thread 2 would print "c".

2.4 Shared, unshared variables

Threads can share variables with each other or have their own versions of them. OpenMP defines "shared" variables as variables that are used by all threads together and "private" variables as those that each thread has its own version of.

```
1 int main(void){
2         int m,l=0;
3         #pragma omp parallel for num_threads(2) private(l) shared(m)
4         for(int n=0; n<10;n++) {
5                 l++;
6                 m++;
7                 printf("(%d,%d)",l,m);
8         }
9 }
```

One console output of aboves code happened to be "(1,1)(103635,2)...". The peculiar number 103635 occurred because the private variable l was first created and initialized in line 2 and kept being used for thread 0, however it was never copied into the second version of l that thread 1 is using for the parallel block.

Private variables must be initialized for each thread. In order to do this the firstprivate statement can be used instead of private.

```
1 int main(void){
2         int m,l=0;
3         #pragma omp parallel for num_threads(2) firstprivate(l)
          ↪  shared(m)
4         for(int n=0; n<10;n++) {
5                 l++;
6                 m++;
7                 printf("(%d,%d)",l,m);
8         }
9 }
```

Firstprivate copies the original value of l into all the other ls that the other threads are using. The console output in this case would be "(1,1)(1,2)(2,3)(2,4)(3,5)(3,6)(4,7)(4,8)(5,9)(5,10)". We can see that together both threads increment the shared variable m up to 10 while each thread only increases their version of l up to 5.

2.5 Offloading

Offloading can be used in order to execute code on other hardware than the "main" computers CPU. One example for possible usage would be to specify the device number of the device that the code is to be executed on:

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

2.6 Thread-Safety

Having multiple threads access and write inside of shared memory at the same time can lead to erroneous results. OpenMP provides multiple ways of ensuring thread-safety.

One such way is the usage of the atomic clause.

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

In the above the variable count is incremented once by each of the 100 threads, hence the final value should be 100. However due to many many threads accessing that variable at the same time they would oftenly get in each others way and not produce the 100 as the final value. If the commented out atomic clause would be in effect it would ensure that the threads don't intervene with each other and the final value would be the correct 100.

Another solution is the usage of the reduction clause.

```
#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;
}
```

The reduction clause causes count to be treated as a private variable so that each thread only accesses their own copy of count. The values of the many count variables are then summed up together into the final value for the shared variable. Instead of summing the values up a different operation could be specified. The reduction clause is oftenly faster than the atomic clause but requires the programmer to specify the operation that is being applied in the end.

Another construct to ensure thread-safety is the critical construct. The critical construct defines critical regions of different names and ensures that no more than 1 thread enters a critical region of the same name.

```
#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");
        }
}
```

In the above we have 2 critical regions of the same name “loop”. The first accessed by thread 0 printing “a”, the later accessed by thread 1 printing “b”. Once the faster thread has reached its loop the slower thread can’t enter its own loop until the faster one has finished printing. Hence the console output would be either “aaaaabbbbb” or “bbbbbaaaaa”.

2.7 Synchronization

```
#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");
}
```

The barrier directive causes all threads to wait at the barrier until every thread has reached the barrier. Without the barrier the output could be “n \npast the barrier n n n n n n n n \npast the barrier” while it would be “n n n n n n n n \npast the barrier \npast

the barrier" if the barrier was in effect with thread 0 waiting for thread 1 to finish the loop.

There is an implicit barrier at the end of parallel blocks, for blocks and sections. In order to remove the implicit barrier the nowait directive can be used.

```
#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");
}
```

The above wouldn't print "done with the loop" before both threads would have finished the loop if the nowait directive wasn't there. With the nowait directive however the thread that finishes the loop first would go ahead and print it.

3 Compilation

In order to compile the code using OpenMP one requires a compiler that supports OpenMP, fulfill other dependencies such as linking the runtime library libgomp-1.dll on windows and set the compiler flag for OpenMP e.g. -fopenmp for GCC. If the compiler flag isn't set it will ignore the "#pragma omp" and produce serial code instead.

Additionally to the usual compilation process there is extra work that the compiler needs to do when compiling an OpenMP program. The compiler needs to read omp directives ie. the pragmas and check them for correctness. It needs to transform sections to Do- and For-constructs and turn implicit barriers to explicit ones. It needs to handle extra memory for all the different threads and the memory each of those is using. It transforms the omp directives into multi-threading code ie. transforming the code with pragmas into something that looks more similar to the PThread code. Furthermore parallel regions are being outlined into functions. And last but not least it is performing optimization wherever it can detect it.

One example for the code transformation:

```
1  void main(){
2          #pragma omp parallel
3          {
4                  #pragma omp for
5                  for( i = 0; i < n; i++ ){...}
6          }
7  }
```

The above OpenMP code would internally be transformed into something like the below:

```
1  void outlined(...){
2          tid = ompc_get_thread_num();
3          ompc_static_init(tid, lower,upper,incr,.);
4          for( i = lower;i < upper;i += incr ){ ... }
5          ompc_barrier();
6  }
7
8  void main(){
9          __ompc_fork(...,&outlined,...);
10 }
```

The entire parallel region is put into an outlined function which is forked to in the main function, the implicit barrier is turned into an explicit one creating the function call in line 5 and the omp directives are turned into the “usual” code.

4 Performance

Performance increase in terms of speed is the big motivation to even use parallel programming. Important to note is that using parallelization doesn't only yield positive effects. There is a time cost that comes along with having to coordinate the parallel processes. Time is spent initializing threads, terminating threads and coordinating them e.g. in means of synchronization or maintaining their memory. That additional cost is called the parallel overhead.

The speedup over the serial version of the code can vary strongly. The simplified formula for speedup and average efficiency of each processor are as follows:

$$Speedup(P) = \frac{T_{Serial}(P)}{T_{Elapsed}(P)} = \frac{1}{\frac{f}{P} - f + 1 + O_P \cdot P}$$
$$Efficiency(P) = \frac{Speedup(P)}{P}$$

P is the number of processors. f is the fraction of the code that is being parallelized, $f = 0$ would mean that the code is serial while $f = 1$ would mean a perfectly parallel application. $O_P \cdot P$ is the parallel overhead with O_P being a constant percentage which is simplified as that percentage may change, higher O_P means higher overhead. Assuming the perfect, but unrealistic case, of having $O_P = 0\%$ and $f = 1$ we can see that the maximal speedup is P ie. P-times as fast:

$$\frac{1}{\frac{1}{P} - 1 + 1 + 0 \cdot P} = \frac{1}{\frac{1}{P}} = P$$

This would mean that each processor is running at 100% efficiency with every processor doing as much work just as fast as the single processor in the serial version.

As for OpenMP in particular, one upside worth pointing out is that OpenMP uses a thread-pool. A thread-pool means that threads are only created once and once a thread has finished its work it will return to that pool waiting for new work. This is efficient.

On a critical note however, if the parallelization is done poorly then the performance may even decrease.

```
1 #pragma omp parallel for private(j)
2 for(i=0;i<=100000;i++)
3 {
```

```
4           for(j=0;j<=100000;j++)
5           {
6                   #pragma omp atomic
7                   a++;
8           }
9   }
10  printf("%lld", a);
```

The above for instance took 24 seconds to execute in serial, 39 seconds in parallel while non-thread-save and 3 minutes and 49 seconds in the parallel and thread-save version.

Below is a table of the well done parallelisation of a matrix-vector-product calculation with 4 cores and 8 logical processors. The bottom row shows a speedup of 4.73 while the theoretical maximum given 0% overhead and perfect parallelization would be 8.

Table 4.1: 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]

In order to optimize performance there is multiple things that can be payed attention to. One general aim is to minimize the overhead which can be achieved by different means. One big time consumer is parallelizing inner loops. "#pragma omp for" should be used at the most outer loop possible, if used on an inner loop the overhead of the parallelization will be cause for each iteration of the loops outside of the parallelized loop. Another good practice is to maximize parallel regions e.g. instead of using "parallel for" multiple times you would rather use the parallel construct once and then only use the for construct multiple times, since each time the parallel construct is called new threads will be initialized. Memory conflicts cause overhead as well so these should be avoided.

Then other than reducing overheads it will help a lot to increase the efficiency of the threads which is achievable in multiple ways. First of all there is the load balancing problem which states that each thread should have about the same amount of workload. Having one thread finish early and then waiting for the rest of the threads isn't efficient. It is better to have all threads working all the time and finishing their task simultaneously. Another way to increase thread performance is to optimize the usage of barriers and the nowait directive. Then generally thread-safety and the ordered construct are very slow since both require threads to wait for one another, thread-safety should only be ensured when necessary.

5 Conclusion

How good is OpenMP? On the upside OpenMP is highly convenient given that it is very simple to apply and doesn't require one to rewrite ones serial code. That makes the target-audience of OpenMP the general-purpose application programmer. It is unusual to worry about performance before implementing functionality, optimization is usually done somewhere near the end of a project. Having to rewrite the entire code in order to turn an application into a parallel program would cost a lot of time, if it weren't for OpenMP. With OpenMP most of what you do is to just add the pragma statements in front of already existing code parts. OpenMP is also very efficient for parallelizing loops.

On the downside, OpenMP is too narrow for complexer code structures. OpenMP is not very efficient when it comes to making different threads do different things. OpenMP also doesn't optimize for the specific hardware the code is running on.

Summarizing, OpenMP is very easy to use, requiring you to only add a few lines which makes it unnecessary to rewrite code, but it is no substitution for lower-level APIs. For a more performance oriented audience a lower-level API may prove to be more pleasing given the possibility of finer tuning, but for the average programmer OpenMP will prove to be very convenient. Nevertheless, OpenMP has many features that allow flexible control. In terms of performance the possible speedup is hardware dependant and it is necessary to optimize the parallelization well, otherwise it may even yield negative effects.

Bibliography

[App14] 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].

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

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

[Hel10] 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].

[Mat03] Timothy G. Mattson. *How Good is OpenMP*. Hindawi Publishing Corporation, 2003.

[Mic16] Microsoft. atomic. https://msdn.microsoft.com/de-de/library/8ztckdts.aspx, 2016. [Online; accessed 8-December-2016].

[Yli07] Joel Yliluoma. Guide into OpenMP: Easy multithreading programming for C++. http://bisqwit.iki.fi/story/howto/openmp/, 2007. [Online; accessed 8-December-2016].