МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2020
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2020
Proceedings of the Forty-ninth Spring Conference
of the Union of Bulgarian Mathematicians
2020

ABOUT OpenMP AND SOME COMBINATORIAL ALGORITHMS[*]

Maria Pashinska, Iliya Bouykliev

In the current work, we present Open Multi-Processing programming interface (Open MP) and its capabilities for programming parallel algorithms using CPU. We use an example to illustrate how parallelism is achieved. We examine the accomplished speed-up and present a way of using OpenMP for work with accelerators – an offload model of work.

Introduction. The invention and the mass usage of multi-core processors make parallel computing a viable and preferable option for many algorithms. One way to achieve parallelism using the traditional Central Processing Unit (CPU) is with OpenMP. In the current work, we present basic information about OpenMP and ways to use it for achieving parallelisation using CPU. Then we present an example of the way we have used it in our work for parallelising a computable "heavy" part of an algorithm for determining automorphism group and canonical form of a binary matrix. Here we also present experimental results. The last part of the current work presents information about using OpenMP and accelerators. According to the Avitohol supercomputer user's guide using OpenMP is the more practical way to work with the Intel Xeon Phi's accelerators.

1. OpenMP basics. OpenMP is an API (Application Programming Interface) that supports multi processing programming. It is available since 1997 and is firstly developed by IBM and Intel. In the following years more tech companies such as AMD and Nvidia are included in the project. OpenMP is based on the shared memory systems represented by multi-core CPUs. It is included in most C/C++ compilers (GCC, Clang, CCE, XL, etc) and Fortran (CCE, Flang, XL, etc) compilers and there is no additional software to be installed. Two things needed to use OpenMP are including the library and some compiler directives for achieving parallelisation. The following is the mandatory compiler directive used for OpenMP programming:

```
#pragma omp
```

[*]**2020 Mathematics Subject Classification:** 65Y05, 05E18.
Key words: parallel programming, OpenMP, accelerators.
The research of the second author was supported, in part, by a Bulgarian NSF contract DN02/2-13.12.2016. The research of the first author was supported, in part, by the Bulgarian Ministry of Education and Science by Grant No. DO1-221/03.12.2018 for NCHDC, a part of the Bulgarian National Roadmap on RIs.

The '-fopenmp' flag needs to be included for compiling in the IDE software application. It works on most operating systems and hardware. The main goal is OpenMP programs to be easy to write and to work on every system.

OpenMP works with the fork-joint model. At the start of the program there is only one working thread called *master*. This thread corresponds to one working CPU. When a parallel region (denoted by *#pragma omp parallel*) is reached we generate threads and every thread is given part of the work. Threads work on different CPU cores. Thus work is shared between the cores. We can explicitly tell how many threads to be generated according to what is needed in the algorithm. By default the number of threads is equals to the number of cores. Only the master thread remains at the end of the parallel region. Since we are using shared memory systems threads have access to all the variables declared in the program. There also can be declared private variables visible only by the owner thread. Private variables are destroyed at the end of the parallel regions. There are different ways of using OpenMP for parallel algorithms on the CPU. Here are shown three of the more popular ways:

- Functional parallelism – every thread executes different functions.
- Tasking – units of work are generated dynamically and may be generated and executed by different threads at runtime.
- For cycles – each cycle has a number of iterations. Each thread is assigned a number of iterations of the loop.

In the rest of the article we will only look at the for cycles because it is the most practical and popular type of parallelism in OpenMP. The key part here is the way loop iterations are divided by the threads. This is called *scheduling.* There are 3 types of scheduling – static, dynamic and guided.

- In the static scheduling threads get work in a round robin model. This means that when the iterations are assigned to the number of threads, every thread gets work according to its number. This is best to be used when the amount of the work is the same for each thread. You can explicitly tell how many iterations a thread to get at a time if needed in the algorithm – this is called *chunk size.* The following table shows the difference between this type of division and the default one.

Number of iterations = 1 000 000				
	Thread 1	Thread 2	Thread 3	Thread 4
static (default)	[0, 249 999]	[250 000, 499 999]	[500 000, 749 999]	[750 000, 999 999]
static, chunk = 1	0, 4, 8, . . . , 999 996	1, 5, 9, . . . , 999 997	2, 6, 10, . . . , 999 998	3, 7, 11, . . . , 999 999

- In the dynamic scheduling each thread gets work as it is generated. You can again specify a chunk size. After that the next chunk of iterations is given to the thread that finishes first.
- The guided type of scheduling is a combination of the static and the dynamic ones. Here the work is given at runtime. However chunk size is equal to the number of loops divided by the number of threads. The number of iterations left after the initial divide is again divided by the number of threads. Thus chunk size decreases to better handle load imbalance between iterations.

2. Calculation of pseudo-orbits. Here we present an algorithm used to partition the set of columns of the matrix into subsets containing one or more orbits with respect to automorphism group of the matrix. These subsets are called *pseudo-orbits.* We use this to determine automorphism groups and canonical forms of symmetrical and pseudo-symmetrical structures (Hadamard matrices and combinatorial designs). The concept of pseudo orbits, invariants and how they are used for calculating canonical forms and determining automorphism groups can be found in [4].

Let us consider the following problem:

We have a square binary matrix M_{nxn}. Our task is to calculate an n-dimensional vector A using the given algorithm – for every three different columns i_1 i_2 i_3 we calculate b using the elements of the matrix in the following formula:

$$b(i_1, i_2, i_3) = \sum_{j=1}^{n} (m_{ji_1} + m_{ji_2} + m_{ji_3})^2.$$

For vector A we accumulate $a_i = a_i + b$, where $i = i_1, i_2, i_3$.

The traditional realization of this problem has n^4 iterations. It uses 3 loops for generating i_1, i_2, i_3 and one loop to pass through all rows in the matrix for every different combination. The total number of iterations is $(n * C_3^n)$. There are a few things that we need to consider when trying to parallelise this algorithm. Firstly, OpenMP gives the opportunity to collapse perfectly nested loops (of which we have 3) into one linear loop using *#pragma omp parallel for collapse(it)*, where the variable *it* is the number of the nested loops. The following table shows in practice how this is done. Here k represents the iterations of the linear loop and x is the total number of iterations of j.

$i \in [0, 2]$ and $j \in [0, 3]$				
	Thread 1	Thread 2	Thread 3	Thread 4
(i, j)	(0,0); (0,1); (0,2)	(0,3); (1,0); (1,1)	(1,2); (1,3); (2,0)	(2,1); (2,2); (2,3)
$k = i * x + j$	0, 1, 2	3, 4, 5	6, 7, 8	9, 10, 11

As seen from the table above the total number of iterations must be known before the runtime.This is why using the *collapse* option is not applicable in our case. A more important thing that needs to be considered is the memory access. When using CPU and OpenMP the threads have access to everything in the memory. It turns out that when two threads attempt to read from the same memory there are no issues. However, when more than one thread attempts to write in the shared memory at the same time the result cannot be predicted. This is called *data racing.* This issue can be resolved with the *#pragma omp critical* – only one thread can execute the critical region at a time. This means that a thread reaches critical section it has to wait its turn to write in the shared memory. This is called a *barrier* and significantly slows down the execution. The following is an example of implementing the algorithm with OpenMP and *critical* section.

```
#pragma omp parallel for private (temp) schedule (dynamic)
for (int i=0; i<n-3; i++){
  for (int j=i + 1; j<n–2; j++){
    for (int k=j+1; k<n–1; k++){
```

```
temp = 0;
for(int row = 0; row < n; row++){
int b = mat [row] [i] + mat [row] [j] + mat [row] [k];
temp = temp + b*b;
}
#pragma omp critical {
a [i] = a [i] + temp;
a [j] = a [j] + temp;
a [k] = a [k] + temp;
}}}}
```

To avoid the critical section that slows down the execution we transform the vector to a matrix where the number of columns equals the number of threads (int a [n] [24]). Every thread has id which we can access with the function *omp_get_thread_num()*. This returns an integer – the number of the thread. Thus every thread writes in its own column and the conflict is avoided. After the parallel portion of the program is executed we need to combine the results of all threads. For every row of the result we summarize the columns into the first (0) column of the resulted matrix. This only takes $n * (\textit{thread count})$ and does not result in a significant slowdown.

Another way to achieve parallelism is when you have to apply this algorithm for many matrices. The idea for every thread to get one matrix. In this implementation the time for calculation depends only on the dimensions of the matrices. This implementation is best to apply when the input includes many matrices of the same size. The elapsed time remains constant and is equal to the time needed for sequential realization where the input is one matrix. It does not depend on the number of the matrices if enough threads are available.

We have tested the program on a Linux machine with Intel Xeon processor where the total number of threads is 24. With the second version (no critical clause) we achieve speedup between 2 and 10 times with both gcc and clang compilers. This difference is due to the amount of work for each thread – for small matrices (40×40) the total number of operations is approximately $37^3 * 40$. The first 24 iterations of the first loop will be spread across the threads of the CPU. The thread that cares for $i = 0$ will do approximately $37^2 * 40$ operations and the thread that cares for $i = 23$ will do approximately $14^2 * 40$. It is obvious that the operations cannot be divided symmetrically amongst the threads for this particular matrix. However, for 200×200 matrix the total number of iterations is over 10^9 and after initial division of the iterations there will still be a large number of operations to be divided amongst the threads. In [3] the achieved speedup for matrix multiplication using OpenMP and CPU with 24 cores is only 8 times compared the sequential realization.

As you can see in the following table there is no difference in the speedup for the different compilers – both clang and gcc achieve the same speedup. However, the clang compiler gives better results in general for both sequential and parallel versions. Both here and in [3] the speedup stays constant for bigger matrices.

3. Using OpenMP with accelerators. Another important feature of OpenMP is the ability to program heterogeneous systems. A computer system is heterogeneous when it has different types of devices – one host which is the traditional CPU and one

Matrices dim.		40	60	80	100	120	140	160	200
gcc	s	0.220s	0.170s	0.768s	2.434s	6.244s	14.313s	31.972s	95.401s
	p	0.014s	0.056s	0.152s	0.470s	1.040s	2.231s	3.858s	10.644s
clang	s	0.018s	0.150s	0.674s	2.151s	5.533s	12.869s	27.402s	89.641s
	p	0.009s	0.053s	0.165s	0.440s	0.964s	1.708s	3.178s	8.891s

or more accelerators (computational unit with different structure). There are two different types of accelerators – GPU (graphics processing unit) and MIC (many integrated cores). The Avitohol system has Intel's Xeon Phi accelerators that are easily used with offload model of work. The intention here is to offload computationally heavy part of the program to the accelerator. In OpenMP this is achieved with *#pragma omp target teams*. We implemented the offload model of work with Nvidia Titan Pascal GPU. To accomplish this you need to install additional compiler and libraries. For Nvidia GPUs Cuda is also mandatory. The two compilers, the libraries and the Cuda platform need to be linked in the environment that is used for writing the code and this action is different for the different operating systems. Our results show that OpenMP has limitations when using Nvidia GPUs. Since OpenMP is developed to work on many system the control over the GPU's memory access and thread count is limited or non-existent. There are many researches that compare OpenMP offload model working with GPU with other programming models for GPUs. Comparisons between different implementations of parallel algorithms on CPU and GPU using both OpenMP and CUDA can be found in [1], [2], [3]. The conclusion based on our experience, these and other sources is that for programming GPU it is best to use different platform (OpenCL, Cuda, etc).

4. Conclusion. OpenMP gives easy to implement way for parallel programming. It works on many operating systems and different hardware. The achieved speedup is highly dependant on the algorithm that is to be implemented, the hardware and the division of the work amongst the threads. Different types of accelerators can be used with OpenMP. However, the GPU gives significantly better results when it is programmed with platforms, more specified for the architecture.

REFERENCES

[1] A. Hayashi, J. Shirako, E. Tiotto, R. Ho, V. Sarkar. Performance evaluation of OpenMP's target construct on GPUs. International Journal of High Performance Computing and Networking, **13**, *1* (2019), 54–69.

[2] J. Larkin. OpenMP on GPUs, First Experiences and Best Practices, GPU technology conference, 2018.

[3] K. Thouti, S. R. Sathe. Comparison of OpenMP & OpenCL Parallel Processing Technologies. International Journal of Advanced Computer Science and Applications (IJACSA), **3**, *4* (2012), 56–61.

[4] М. Джумалиева-Стоева. Алгоритми за изследване на комбинаторни структури, Докторска дисертация, София, 2015.

Maria Pashinska
e-mail: mariqpashinska@math.bas.bg
Iliya Bouyukliev
e-mail: iliyab@math.bas.bg
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
P. O. Box 323
Veliko Tarnovo, Bulgaria

ИЗПОЛЗВАНЕ НА OpenMP В НЯКОИ КОМБИНАТОРНИ АЛГОРИТМИ

Мария Пашинска, Илия Буюклиев

В тази статия е представено как приложният програмен интерфейс OpenMP може да се използва за разпаралелване на работата на някои алгоритми, като се използва централният процесор. Разгледан е конкретен пример, чрез който са показани някои от особеностите на OpenMP. Оценено е полученото ускорение и е представен начинът на използване на OpenMP за работа с ускорители.