Advances and Applications in Mathematical Sciences
Volume 18, Issue 8, June 2019, Pages 775-787

PARALLEL COMPUTING OF MATRIX MULTIPLICATION IN OPEN MP SUPPORTED CODEBLOCKS

HARI SINGH, DINESH CHANDER and RAVINDARA BHATT

[1,3]Jaypee University of Information Technology
Solan, Himachal Pradesh, India
Email: hari.singh@juit.ac.in
ravindara.bhatt@juit.ac.in

Panipat Institute of Engineering & Technology
Panipat, Haryana
Email: me.dinesh17@gmail.com

Abstract

Matrix multiplication is a very popular and widely used operation in linear algebra. It has a number of application areas such as Graph Theory, Numerical Algorithms, Signal Processing and Digital Control. A lot of researchers have implemented and analyzed parallel computation of matrix multiplication in a number of parallel computing platforms such as Messaging Passing Interface (MPI) in distributed memory architecture; Open Multiprocessing (Open MP) shared memory architecture and Compute Unified Device Architecture (CUDA). In this paper, parallel computation of matrix multiplication in Open MP (OMP) has been analyzed with respect to evaluation parameters execution-time, speed-up, and efficiency. The experimental results validate the high performance gained with parallel processing OMP as compared to the traditional sequential execution of matrix multiplication.

1. Introduction

Though, the storage, processing speed, communication speed and, other computer hardware and software resources are technologically advanced and easily available, but, they fail to meet the current complex application requirement. The parallel computation has become the need of the day. Generally, the execution-time of a program is the amount of time taken to execute all the instruction of the program with some average instruction

2010 Mathematics Subject Classification: 15-XX.
Keywords: matrix multiplication, open multi processing.
Received February 21, 2019; Accepted March 22, 2019

execution time. With the advancement in technology, the increased processor speed has been able to improve/decrease the execution-time. However, there is some limitation to increasing the processor speed; it cannot be increased indefinitely due to more power required to run the processor and subsequently more heat generated that cannot be disposed off by the heat sink [6].

Parallel computing is about using many processors or multi-core CPU simultaneously to execute a program or multiple computational threads [11, 12]. Though, multi-core processors are widely available, but, the parallel programming is not that much popular among its users to harness the available multi-cores. Multi-core technology means more than one core inside a single chip. This allows multiple instructions of a program to be executed in parallel at the same time. Thread level parallelism (Multithreaded processors) executes multiple threads on multiple-cores in parallel and improves processor performance. A core is a part of the processor that performs read, execute, and write operations. One significant advantage of using Open MP is that the same source code can be used with Open MP compliant compilers and normal compilers as the Open MP commands and directives remain hidden to normal compilers.

Open MP is typically used for exploiting loop-level parallelism; it can be used to establish coarse grain parallelism, potentially leading to less overhead. The primary motivation for adopting new programming paradigms is increased capability, efficiency and ease of programming. Adding MPI to Open MP programs allow users to run on larger collections of processors. Pure shared memory machines are limited in number of processors. Adding message passing can increase number of processors that are available for a job. Adding Open MP to MPI programs can also increase efficiency and also increase capability. With Open MP, there is no implicit copying of data.

The following are the evaluation parameters in parallel computation of a task.

(1) Execution time: The time difference between the starting (when execution is started) and ending of the process (when the execution is completed and results are obtained).

(2) Speed-up: it is the ration of execution time of a single core processor to

the multi-core processor. The speed-up is linear if it is equal to the number of processors. It is poor if it is less than the number of processors. Theoretically speed-up cannot be greater than the number of processors.

(3) Efficiency: It is the ratio of speed-up to the number of processors. In a parallel model, increasing the number of processors improves the performance but this trend does not continue when we keep on increasing the number of processors. After a time, adding more cores or processors becomes inefficient. So, mathematically, higher the efficiency, more cores or processors can be added. Theoretically, efficiency cannot exceed 100%.

The organization of the paper is as follows. Section 2 discusses the related work of matrix multiplication on Open MP. Section 3 describes the fork-join processing model of multithreading used in Open MP. Section 4 describes the matrix multiplication algorithm in Open MP. Section 5 presents an experimental evaluation and results. Section 6 concludes the paper.

2. Related Work

This section relates the work done by various researchers for matrix multiplication in Open MP domain. Study and evaluation of execution time of matrix multiplication is performed using Open MP on a single, dual and multi-core processor. Open MP standard exploits parallelism in a shared memory architecture with its multi-threading. A speed-up of around 3.6 was obtained for various matrix sizes with Open MP [1].

In another paper, the authors compared the various HPC techniques, such as MPI, Open MP, Hybrid Open MP and Pthread for matrix multiplication of two matrices of size ranging from 16x 16 to 8192x8192 and multi-processors ranging from 2-64 processors. The techniques are compared on the performance of parallel model with respect to execution-time, speed-up and efficiency. It was observed that the performance of parallel models (MPI, Hybrid Open MP and Open MP) remains almost comparable to the sequential model for smaller sizes of matrix. This is due to the fact that the parallel model requires initializing some libraries and executing extra code to manage the parallel code. Further, the performance of Open MP is low as compared to the MPI and Hybrid Open MP parallel models for similar number of multi-core processors or the size of input matrices. The effect of the compilers on

the Open MP performance is also observed with gcc (4.2) 64 cores and gcc (6.1) 8 cores for the parameter- execution-time. The performance of 64 cores is found better [2].

A related work showed the performance (Execution-time and Speed-up) gain achieved using Open MP parallel programming model over the sequential programming on dual-core and quad-core processors processor architectures for Merge Sort and Floyd's algorithm [3].

In another paper, the authors identified the areas where multithreading in Open MP causes performance bottlenecks that causes thread serialization, such as critical sections, barriers, imbalanced amount of work in the parallel region, etc. Approaches such as synchronization, thread management, task scheduling and memory access are discussed to reduce execution-time and other overheads to improve performance. As a program that spends much time in bottlenecks or serialization looses a significant speed-up that could have been achieved through parallelization. A matrix multiplication program in Open MP has been used for it [4].

In some research papers, matrix multiplication on dual-core 2.0 GHz processor with two threads on Open MP was analyzed for parameters speed-up, efficiency and execution time. The experimental results show better performance of the parallel model than the serial model. It was also observed that parallelism should be adopted beyond a certain problem size [5, 7, 8, 10].

In another paper, the authors proposed a manual data distribution approach in Open MP in contrast to the automatic data distribution approach of Open MP for computing matrix addition and multiplication. The proposed approach is found better than the default Open MP parallel model and the sequential model [9].

In this paper, a parallel matrix multiplication in Open MP is run on a quad-core processor in order to further validate the performance gain achieved using parallel processing over the traditional sequential processing. We have tried to omit instructions that force sequential region in program execution. Here, the input is auto generated as opposed to the user supplied input [1]. Random generator function rand () causes sequential read [5] and malloc () call to dynamic memory allocation also causes sequential execution [6]. Other papers do not clearly specify the input and computing approaches used [2, 4, 7].

3. Fork-Join Processing Model in Open MP

The Open MP operates on fork-and join model of parallel execution. All Open MP programs start as a single process called a master thread. Master thread executes sequentially until a parallel region is encountered. At this point, the master threads forks into a number of parallel region threads. The instructions in the parallel region are then executed by this team of worker threads. At the end of parallel region, the threads synchronize and join to become the single master thread again. The whole idea is presented in Figure 3.1.

In the matrix multiplication algorithm, presented in section 4, matrices reading and their multiplication is done in a parallel region. Two matrices $A[\,][\,]$ and $B[\,][\,]$ are read in parallel and a third matrix that would store the multiplication result of the two input matrices is initialized to 0. These tasks are performed in loops. For large size input matrices, two loops for reading $A[\,]$ and $B[\,]$ and one loop for initializing $C[\,][\,] = 0$, there is a need to specify the chunk size. Similarly, the matrix multiplication also requires a number of iterations. The chunk size in Open MP divides the iterations into chunks and these chunks are assigned to the available threads in circular order. If no chunk size is specified then Open MP divides iterations into chunks of equal size (chunk size = Number of iterations/Number of threads). It distributes at most one chunk to each thread.

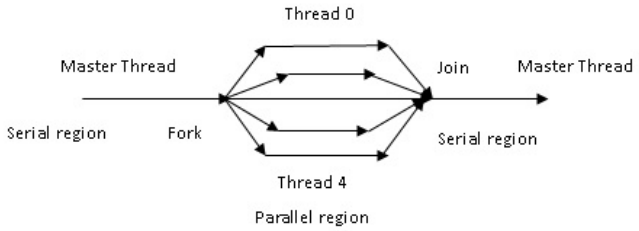

Figure 3.1. Fork-Join Processing Model in Open MP.

4. Open MP parallel Matrix Multiplication Algorithm

The following parallel matrix multiplication algorithm is implemented in Code Blocks 13.12, having support for Open MP.

Algorithm: Open MP parallel Matrix Multiplication

1. Start with int main (intargc, char *argv[]). Declare Matrix A[NRA][NCA], B[NCA][NCB], C[NRA][NCB], nthreads and Chunk_Size. NRA, NCA, NCB stands for the number of rows in Matrix A, number of columns in Matrix A and number of columns in Matrix B, respectively. These are macro defined variables. Initialize Chunk_Size.

2. Start parallel region explicitly

```
#pragma omp parallel shared(A,B,C,nthreads, Chunk_Size) private(tid,i,j,k).
{
tid = omp_get_thread_num();
    if (tid == 0)
    {
nthreads = omp_get_num_threads();
    }
```

Where A,B,C represents matrices; nthreads keeps a count on the number of threads in use with the help of a thread identifier variable *tid*.

1. Initialize matrices in parallel

```
 #pragma omp for schedule (static, chunk)
    for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
    a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
    for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
    b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
    for (i=0; i<NRA; i++)
```

```
for (j=0; j<NCB; j++)
c[i][j]= 0;
2. Perform matrix multiplication in parallel
 #pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
  {
for(j=0; j<NCB; j++)
for (k=0; k<NCA; k++)
c[i][j]= C[i][j]+a[i][k] * b[k][j];
}
3. End parallel region created in Step 1 with a closing bracket
}
4. Get the output matrix printed
for (i=0; i<NRA; i++)
 {
for (j=0; j<NCB; j++)
printf("%4.2f   ", c[i][j]);
printf("\n");
 }
}
```

5. Experimental Evaluation and Results

The parallel matrix multiplication program was run on a machine with processor: Intel(R) Core(TM) i5-6500 CPU@ 3.20 GHz, Memory 8GB RAM, Operating System – Windows 10, 64-bit. This processor has 4-cores and and by-default 4-threads. However, we have taken up to 10 threads to analyze the execution time.

The parallel matrix multiplication program in CodeBlocks 13.12, having support for Open MP, was run for square output matrix of sizes of (5x5) 25, (10x10) 100, (20x20) 400, (50x50) 2500, (100x100) 10000, (200x200) 40000, (250x250) 62500, (275x275) 75625, (500x500) 250000, (1000x1000) 1000000, (2000x2000) 4000000, (4000x4000) 16000000. However, it is not mandatory to take only square matrices. As per the matrix multiplication requirement, the number of columns of the first matrix should be equal to the number of rows of the second matrix. Any input matrices $A[x][y]$ and $B[x][y]$ that validates the matrix multiplication rules can be taken. –gomp.dll and –pthreads-3C.dll libraries are included in the configuration process in Code Blocks.

5.1. Parallel vs sequential execution with varying data size

During this experimental run chunk size was varies with values Chunk Size (CS)= 10, 20, 30, 40, and 50 keeping the number of threads up to 50. Apart from running the program for these different CS values, one sequential version of the program without OMP support was also run. It was observed that the parallel version in OMP is beneficial if the problem size is significantly large, the benefit of parallel processing cannot be achieved. This thing is evident from the Table 5.1 and Figure 5.1. There is no significant gain in execution time up to an output matrix size of 10000. The observations for running the parallel program with different chunk sizes do not make any noticeable difference in execution time.

Speed-up represents the difference in execution time of a single-core and multi-core processors.

Speed-up=Execution time single-core / Execution time multi-core

The Speed-up is evaluated in Table 5.1, 2nd last column, and is also shown in the Figure 5.2. Here, Speed-up is calculated by taking various problem sizes for a single-core and multi-core processor. The maximum Speed-up of 2.30 is obtained for an output matrix size of 4000x4000. The average Speed-up is 1.605. The results show that the Speed-up > 1 in all cases, that shows a gain in execution time using a multi-core processor. Theoretically, a Speed-up<1 shows a poor parallelism and Speed-up can never exceed the number of processor value.

Efficiency shows the performance of the system by adding more cores or processors to the system. Sometimes it is not efficient to add more processors. It shows the amount of speed or performance gained on adding more cores or processors.

*Efficiency= (Speed-up/Number of cores or processors)*100*

The efficiency is evaluated in Table 5.1, last column, and is also shown in the Figure 5.3. Here, Efficiency is calculated by taking various problem sizes for a single-core and multi-core processor. The average efficiency is approximately 40.15%. The maximum efficiency is gained for an output matrix size of 4000x4000, it is 57.51 %. However, efficiency rate almost stabilizes after a matrix size of 1000x1000.

5.2. Parallel execution with varying data size and number of threads

The parallel matrix multiplication program for different number of threads was observed in OMP for a fixed chunk size = 10. The number of threads (nt) was kept/controlled as 4, 10, 25, 50 for the output matrix sizes of 25, 100, 400, 2500, 10000, 40000, 62500, 75625, 250000, 1000000, 4000000, and 16000000. It was observed that there is no considerable difference in execution time up to an output matrix size of 250x250, but as the matrix size is increased to 275x275 then a slightly better performance is seen with $nt = 4$. The decreasing order of performance is with $nt = 50, nt = 25, nt = 10$ and $nt = 4$. Due to the limitation of memory size with the computer taken in the experiment, results could not be produced for output matrix of higher order. The system started crashing for a matrix size of 4000x4000; results for the matrix size of 2000x2000 and 4000x4000 were taken with much difficulty. This observation is shown in Table 5.2 and Figure 5.4.

Table 5.1. Execution time analysis of a parallel matrix multiplication program in OMP for different chunk sizes and a serial program without OMP.

Output matrix size	Execution Time (Seconds) with OMP					Execution Time (Seconds) without OMP	Speed-up (at CS=10)	Efficiency (at CS=10)
	CS=10	CS=20	CS=30	CS=40	CS=50	Sequential Program		
5x5	0.11	0.082	0.064	0.068	0.079	0.119	1.08	27.04
10x10	0.124	0.108	0.072	0.1	0.08	0.133	1.07	26.81
20x20	0.148	0.144	0.124	0.133	0.125	0.177	1.19	29.89
50x50	0.446	0.523	0.441	0.428	0.461	0.54	1.21	30.26
100x100	1.217	1.163	1.225	1.219	1.258	1.5	1.23	30.81
200x200	4.082	4.096	4.137	4.144	3.976	6.357	1.56	38.93
250x250	6.376	6.371	6.461	6.389	6.4	10.232	1.60	40.11
275x275	7.452	7.074	7.417	6.982	7.45	12.313	1.65	41.30
500x500	28.34	28.02	27.15	25.85	24.26	56.589	1.99	49.91
1000x1000	87.39	86.26	84.14	82.37	80.76	185.473	2.12	53.05
2000x2000	1275.28	1267.45	1252.36	1231.75	1208.56	2865.752	2.24	56.17
4000x4000	4127.43	4076.57	4002.37	3994.76	3878.47	9495.964	2.30	57.51

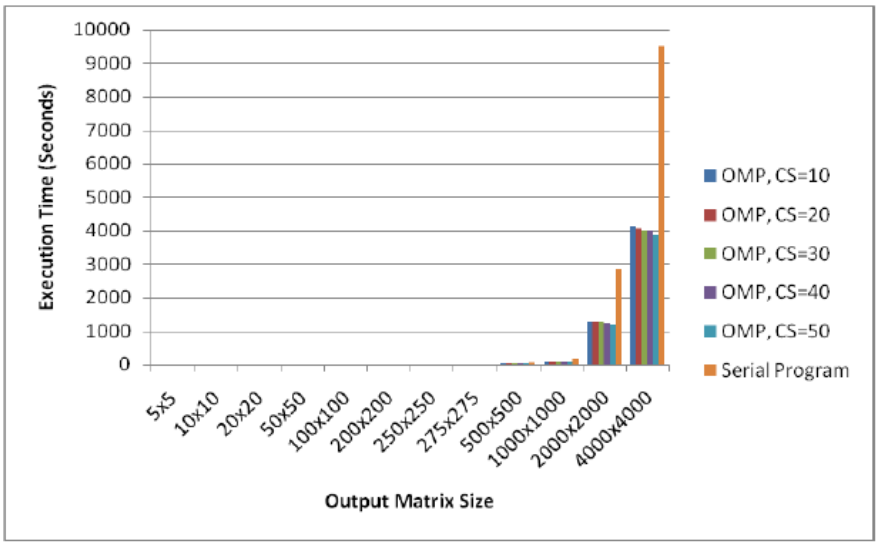

Figure 5.4. Execution time analysis of a parallel matrix multiplication program in OMP for different number of threads.

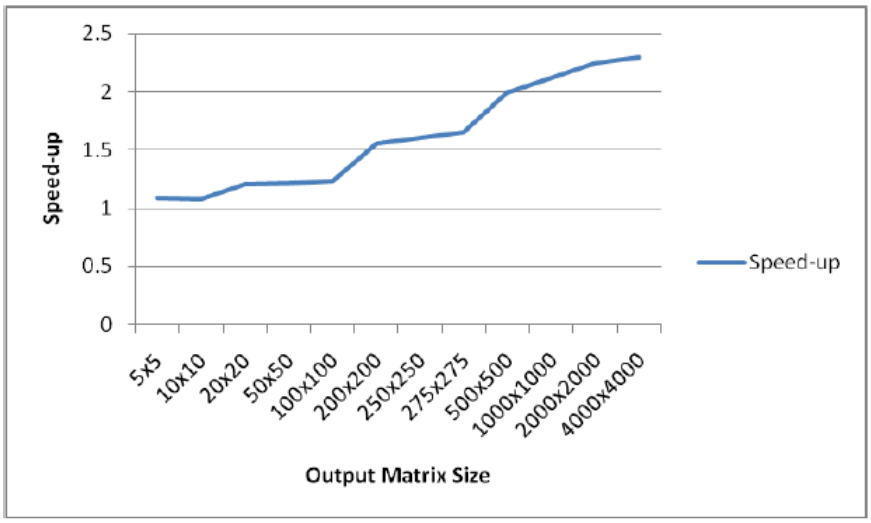

Figure 5.2. Speed-up evaluation for a single-core vs Core(TM) i5-6500 CPU.

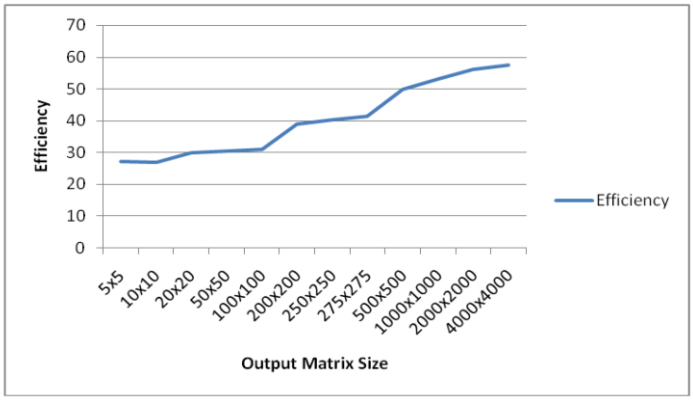

Figure 5.3. Efficiency evaluation for a single-core vs Core(TM) i5-6500 CPU.

Table 5.2. Execution time analysis of a parallel matrix multiplication program in OMP for different number of threads.

Output matrix size	Execution time (seconds) with OMP			
	nt=4	nt=10	nt=25	nt=50
5x5	0.079	0.078	0.078	0.078
10x10	0.08	0.08	0.082	0.081
20x20	0.125	0.124	0.125	0.125
50x50	0.461	0.454	0.451	0.449
100x100	1.258	1.121	1.078	0.992
200x200	3.976	3.276	3.082	2.882
250x250	6.4	6.012	5.8	5.376
275x275	7.45	7.163	6.763	6.452
500x500	24.26	22.539	20.373	17.34
1000x1000	80.76	76.647	71.243	64.39
2000x2000	1208.56	1178.549	1109.564	1012.28
4000x4000	3878.47	3767.645	3665.352	3441.43

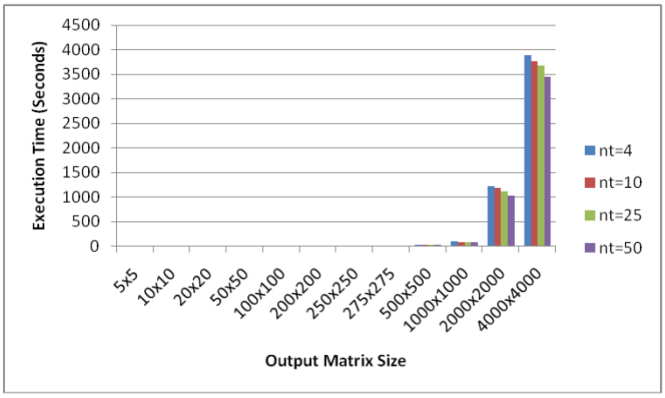

Figure 5.4. Execution time analysis of a parallel matrix multiplication program in OMP for different number of threads.

6. Conclusion and future scopes

The parallel programming in OMP is beneficial only when the input problem size is significantly larger. For smaller size problems, it is better to go with sequential programming. OMP encourages the parallel execution of the program and efficiently utilizes the multi-core processors in the present generation CPUs. The work discussed in this paper does not observe fully the pattern of execution times with high values of chunk sizes and number of threads. It is due to the unavailability of the high configuration machines. Similarly, Speed-up and Efficiency are computed on a single-processor and one multi-core processor. Speed-up and efficiency need to be observed in a variety of multi-processors. We would like to extend the analysis with various multi-processor machines, such as having 8-cores, 16-cores, etc. We would also like to test the execution times on various other complex programs and on different voluminous data.

References

[1] Yogesh Singh Rathore and Dharaminder Kumar, Performance evaluation of matrix multiplication using Open MP for single dual and multi-core machines, IOSR Journal of Engineering 4(1) (2014), 56-59.

[2] Ali A. Alabboud, Sazlinah Hasan, Nor Asilawati Abdul Hamid and Ammar Y. Tuama, Performance evaluation of MPI approaches and Pthread in Multi-core systems, Journal of Engineering and Applied Sciences 12(3) (2017), 609-616.

[3] Baydaa Mohammed Saeed Mustafa and Waseem Ahmed, Parallel algorithm performance analysis using Open MP for multi-core machines, International Journal of Advanced Computer Technology 4(5) (2015), 28-32.

[4] Vibha Rajput and Alok Katiyar, Proactive bottleneck performance analysis in parallel computing using Open MP, Int. J. Adv. Stud. Comp. Sci. Engg. 2(5) (2013), 46-53.

[5] Yajnaseni Dash, Sanjay Kumar and V. K. Patle, Evaluation of performance on Open MP parallel platform based on problem size, International Journal of Modern Education and Computer Science 6 (2016), 35-40.

[6] Alina Kiessling, An Introduction to Parallel Programming with OpenMP, 2009, https://www.roe.ac.uk/ifa/postgrad/pedagogy/2009_kiessling.pdf

[7] S. N. Sheshappa, Maximize execution performance analysis of parallel algorithm over sequential using Open MP, International Research Journal of Engineering and Technology 3(9) (2016), 438-442.

[8] C. K. Balaji and S. Kartheeswaran, Analyzing the matrix multiplication performance in shared memory processor under multi-core architecture using Open MP, International Journal of Pure and Applied Mathematics 119(15) (2018), 3249-3256.

[9] Mohammed Faiz Aboalmaaly, Ali Abdul Razzaq Khudher, Hala A. Albaroodi and Sureshwaran Ramadass, Performance analysis between explicit schedulinmg and implicit scheduling of parallel array-based domain decomposition using Open MP, Journal of Engineering Science and Technology 9(5) (2014), 522-532.

[10] Ashwini M. Bhugal, Parallel computing using Open MP, International Journal of Computer Science and Mobile Computing 6(2) (2017), 90-94.

[11] M. Mathews and J. P. Abraham, Automatic code parallelization with open mp task constructs. In 2016 International Conference on Information Science (ICIS) (pp. 233-238). (2016, August), IEEE.

[12] H. J. Plum, A. Krechel, S. Gries, B. Metsch, F. Nick, M. A. Schweitzer and K. Stüben, Parallel algebraic multigrid, In Scientific Computing and Algorithms in Industrial Simulations (pp. 121-134). Springer, Cham. (2017).