A New Design Paradigm for a Parallel Uniform Block Distribution (UBD) Algorithm

Mohammed Faiz Aboalmaaly
Department of Computer Techniques Engineering, Alsafwa University College
mo.abomaali@outlook.com

ABSTRACT

In several areas of parallelism, the novelty of parallel algorithms is formed by decomposing the algorithm's sequence either on the basis of data or tasks without touching the internal peculiarity of the original algorithms (serial algorithm). Parallel hardware has experienced significant growth in recent years and is readily affordable, as for today; multicore hardware now exists in the vast majority of low-cost digital devices. At the same pace, parallel libraries have demonstrated a noticeable improvement and ease in utilisation. These facts play a vital role in significantly changing the way of designing an algorithm. In this paper, a new design paradigm for a parallel uniform block distribution (UBD) algorithm is proposed by taking advantage of the capability of the parallel libraries during the design phase of the parallel algorithm, rather than making parallelisation as an improvement stage. In particular, the main contribution of this work utilises a new mathematical calculation that uses the thread ID as a variable to explicitly decompose the data of a matrix (array) of one or multiple dimensions among several threads in parallel. Experimental results show a considerable speedup compared to the serial version of the algorithm and comparable results to the original OpenMP implementation.

Keywords: Shared memory architecture, OpenMP, Domain decomposition, Parallel efficiency.

الخلاصة

في عدة مجالات من البرمجة بالتوازي، يتم تشكيل حداثة الخوارزميات المتوازية من خلال تحليل تسلسل خوارزمية إما على أساس البيانات أو المهام دون لمس الخصوصية الداخلية للخوارزميات الأصلية (الخوارزمية التسلسلية). شهدت الأجهزة متعددة النواة نموا كبيرا في السنوات الأخيرة، وبأسعار معقولة ، كما هي الأن. الأجهزة متعددة النوى موجودة الآن في الغالبية العظمى من الأجهزة الرقمية منخفضة التكلفة. على نفس الوتيرة، قد أثبتت المكتبات موازية تحسن ملحوظ وسهولة في الاستخدام. هذه الحقائق تلعب دورا حيويا في تغيير كبير في طريقة تصميم الخوارزمية. في هذه الورقة، يتم أقتراح نموذج تصميم جديد لخوارزمية موازية توزيع كتلة موحدة من خلال الاستفادة من قدرة المكتبات الموازية أثناء مرحلة تصميم خوارزمية موازية، بدلا من جعل البرمجة بالتوازي كمرحلة التحسن. المساهمة الرئيسية لهذا العمل يتم عن طريق عملية حسابية رياضية جديد يستخدم معرف النواة كمتغير لتتحلل بشكل واضح ضمن بيانات المصفوفة (مجموعة) من بعد واحد أو متعددة الأبعاد. النتائج التجريبية للعمل يظهر تسريع كبير مقارنة مع الإصدار التسلسلي للخوارزمية وقابلة للمقارنة مع معيار المعالجة المتعددة المعروف.

الكلمات المفتاحية :- بنية الذاكرة المشتركة ،المعلجات المتعددة مفتوحة المصدر ، تقسيم المجال، كفاءة البرمجة بالتوازي.

1. INTRODUCTION

In a large class of scientific computations, the data domains are located in a one-, two-, or three-dimensional space. This class includes, but is not limited to, linear algebra kernels, image rendering algorithms, particle-in-cell simulations, databases, and many more. For faster processing, parallel and distributed computing has been employed to this class of computations by decomposing the data among processors. Because the parallel efficiency is evaluated by the time required for the last processor to finish its task, it is fundamentally important to provide a load-balanced distribution for the computational workloads such that each processor receives an equal workload to achieve a good efficiency. For a multidimensional space of data, it has been proven that finding the optimal distribution, formally known as generalised block distribution (GBD), is NP-complete (Aspvall *et.al.*, 2001); hence, several approximation algorithms have been proposed to find near-optimal solutions to this problem. However, for numerous time-

critical algorithms, especially those that show high uniformity in the computational workload of data, it is more preferable to develop a distribution with an emphasis on an equivalent distribution [i.e., a uniform block distribution (UBD)] with less or no overhead, rather than distributing the data with exactly equal workloads.

In GBD, also known as rectangular partitioning (RP), there are several different families that are commonly studied. Figure 1 shows three common families for two-dimensional space: (a) arbitrary, allowing any partitioning into rectangular tiles, (b) hierarchical, obtained through recursive cuts, and (c) *pxq* partitioning (Muthukrishnan and Suel, 2005).

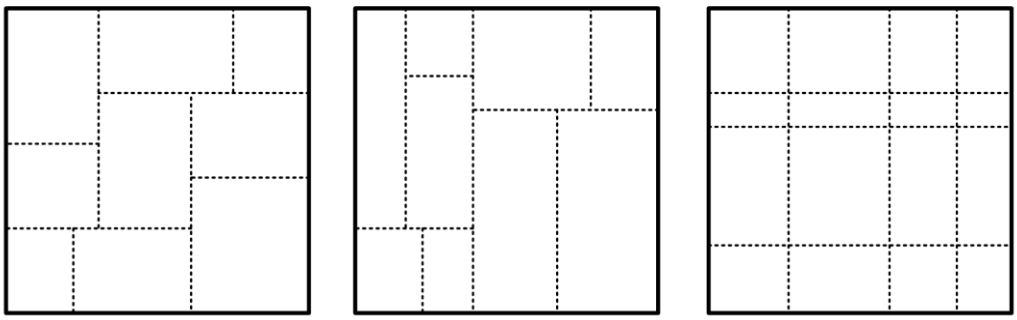

Figure 1: Families of GBD.

However, there are some good reasons why *pxq* partitioning is preferred in certain scenarios. In the context of parallel computing, *pxq* partitioning results in a very simple communication pattern because every node has exactly *2N* horizontal and vertical neighbours, where *N* represents the number of dimensions.

From an algorithmic point of view, GBD as well as several other algorithms were not developed with architecture in mind. In other words, there was no explicit consideration of the continuing changes and development in hardware architecture, such as parallel architecture, even though most of these developed algorithms were proposed to take advantage of such architecture.

The Co-exploration between algorithm and architecture (CEAA) (Gwo-Giun *et.al.*, 2009) is a new design paradigm for algorithms, which takes into account the current trends of the continuous development in computing architecture, particularly parallel architecture, during the design phase of algorithms. Unfortunately, this term has not been significantly utilised among several classes of algorithms such as the class of GBD. In contrast, the design phase of many algorithms has lacked the facilities introduced by parallel libraries and parallel application programming interfaces (APIs) that have shown a significant improvement over the years.

As a matter of fact, a parallel architecture such as multicore architecture is forming the vast majority, if not the only, architecture equipped among digital devices that are currently in the market. Two and four cores are becoming common, even in handheld devices such as smart phones. Fundamentally, this number of cores is expected to increase over time. Hence, we believe that now is the right time to explicitly consider the trends of architecture and software libraries during the design phase of a parallel algorithm. However, we believe that such designs for inherently parallel algorithms

should not generate a conflict when the parallel algorithm is examined sequentially. In this paper, such a consideration is applied by designing a parallel algorithm for a non-overlapping UBD algorithm that explicitly assigns each part of the domain to one processing element in parallel.

2. BACKGROUND

The GBD problem has been intensively studied in the literature (Gaur *et.al.,* 2002, Saule *et.al.,* 2011, Manne and Sørevik, 1996). Applications of the problem include various parallel sparse-matrix computations, compilers for high-performance languages, particle-in-cell computations, video and image compression, and simulations associated with communication networks. All of the literature studies have stated that finding an optimal solution to the GBD problem is NP-complete, even for simple computational problems such as the sum of numbers in a block. For example, image-compression data are stored in memory as a two-dimensional array of pixels; from a GBD point of view, the optimal distribution is when all blocks (sub-images) require the same computational resources (same complexity) for compression. In video coding, the same scenario is repeated but with several successive images (frames). However, the per-frame GBD in video coding (video compression) and other application areas will became a bottleneck owing to the increase in frame rate along with the frame resolution, especially in time-critical situations such as real-time. As an alternative, the UBD will probably lead to a better execution time.

The UBD is a special case of *pxq* partitioning in which all blocks (in some cases half of a border's block will have a different size) have the same size regardless of the different workloads of these blocks (see Figure 2). Despite its simplicity, UBD has also lacked consideration of CEAA and parallel software facilitations. UBD utilises the recursive coordinate bisection (RCB) algorithm to recursively decompose the data which prevents parallelisation from being applicable.

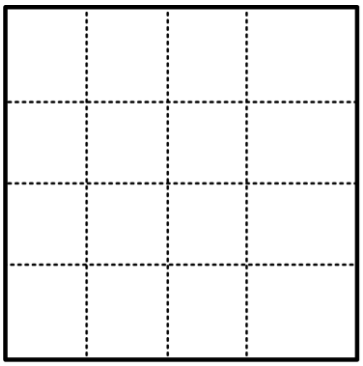

Figure 2: Uniform block distribution.

It is clear that in order to effectively take advantage of a parallel architecture such as a multicore or distributed architecture, one should use the facilities provided for such architectures in the design phase of a particular algorithm. As an example, OpenMP (Pacheco, 2011) and POSIX threads (in short, Pthreads) (2011) are examples of a language extension and parallel library, respectively, to explore parallelism on shared memory architectures. Both have rich features that facilitate easy parallel implementation.

OpenMP is one of the most common language extensions used for parallel computing on shared memory architectures, and it is categorised as cross-platform,

scalable, and easy to use. OpenMP is a collection of compiler directives, library functions, and environment variables that can be used to specify shared-memory parallelism in FORTRAN, C, and C++ programs. OpenMP uses the fork-join model of parallel execution. Although this fork-join model is useful for solving a variety of problems, it is somewhat tailored for large array-based applications (Board, 2008). Pthreads, which is a standard application program interface that could potentially be implemented on many different systems for multi-threaded support, also has wide adoption in several parallel algorithms. The main difference between OpenMP and Pthreads is that adding OpenMP directives in OpenMP to explore parallelisation is performed without significant code modification, while a minor to major code modification is required in Pthreads to match the syntax of this interface.

3. THE PROPOSED PARALLEL UBD ALGORITHM

The main theme behind the idea of this work is how to effectively consider the features of parallel libraries in the design phase of an algorithm owing to the fact that most, if not all, digital devices are equipped with more than one processing core.

The problem covered in this work is the UBD. The choice of this problem was because the usefulness of such straightforward partitioning in applications required the fast decision of a partitioning problem, even if the partitioned domains are not exactly equal in terms of their workload. Real-time multimedia services such as video conferencing, video on demand, IPTV, and many more are a class of applications that require such speed in partitioning. However, we believe that the proposed algorithm can suit any algorithm that deals with array data-type.

3.1 Preliminaries

We assume that R is N-dimensional array; N is any integer number, where $N > 0$. m is a positive integer number such that $m > 0$, which represents the number of partitions that R will be partitioned into. We refer to m as the number of blocks even if R was one-dimensional. We are interested in m because m represents the number of processors (threads) in our algorithm, which is the usual case in parallel processing, where each block is assigned to one processor for processing. In the design of our algorithm, we have moved a step backward when we consider m to represent the number of partitions, as the number of partitions is represented by pxq partitions (the case of two-dimensional space) in most GBD problems. In our algorithm, m is equal to p, pxq, and $pxqxr$ for one-, two-, and three-dimensional space, respectively.

3.2 Block Size (Dimensions)

The proposed parallel algorithm first determines N and m. Then, the algorithm defines a number of integer variables equal to the value of N. If $N = 1$, only one variable (D_1) will be declared, while if $N = 2$, two variables (D_1 and D_2) will be declared and so on. These variables will set to 1 as initial values. After this initialisation, if $N > 1$, m will be used as an input for a prime factorisation function to find its factors $(f_1, f_2, f_3, \ldots\ldots, f_x)$. If $x > N$, a reduction function is proposed to reduce the number of these factors ($f_{D1}, f_{D2}, \ldots., f_{DN}$) to N factors. However, if $x \leq N$, there is no need to reduce the factors. The condition among these factors after reduction is that $\sum f_{Di}$ is the lowest among other possible reductions. For example, if the factors were 2, 2, 3, and 5, which are the factors of 60, and $N = 3$, then the reduced factors are 4, 3, and 5, **(Total 12),** where 4 comes from multiplying 2 by 2 and not 2, 2, and 15 (3 × 5) (**Total 19)** nor 2, 5, and 6 (2 × 3) **(Total**

13).

Once the factor reduction step is finalised, the values of the variables (D_1, D_2, ……, D_N) will be changed by these reduced factors in descending order. As in the above example, D_1 will be equal to 5, D_2 will be equal to 4, and D_3 will be equal to 3. However, if the number of factors was less than N, then the values of some of the variables (D_{x+1}, ……, D_N) will remain 1, as initialised at the beginning of the algorithm.

Finally, the new values of (D_1, D_2,……, D_N) will be the dominators for their analogous dimensions to determine the size of each block. As for three-dimensional space, *width*, *length*, and *height* are used for representation. The block's width will be equal to **width/D_3 + width%D_3**, the block's length will be equal to **length/D_2 + length%D_2**, and the block's height will be equal to **length/D_1 + length%D_1**.

3.3 Parallel UBD

In order to explicitly assign each block to one processor for processing, we need to utilise the facilitations of the parallel libraries. In a parallel environment, each processing unit has a unique identification number called the *thread ID* or *processor ID*. In fact, most, if not all, of the parallel libraries as well as parallel APIs have a built-in facility to retrieve the thread ID within a parallel region. OpenMP, Pthread, MPI, and many more support such identification by creating a variable to store this ID for all threads/processors within a parallel region. In a parallel environment with m processors, the group of *thread IDs* will start from 0 to (m – 1). The *thread ID*, the new values of (D_1, D_2, ……, D_N), and block dimensions will form a new mathematical calculation. The idea of this calculation is based on using the integer division (/) and integer modulus (%) operations to explicitly assign each block to one thread. As illustrated in the following sections, the problem has been divided on the basis of the dimensions of the data space into one, two, and three dimensions.

3.4 One-Dimensional Domain

In terms of a one-dimensional R (width), N will be equal to 1. Hence, only one variable will be declared, D_1, and will be initialised to 1. From an algorithmic point of view, one-dimensional data can be iteratively traced using one loop only, particularly a *for* loop. Because N = 1, there is no need to factorise m, and D_1 will be assigned the value of m. The next step is to determine the block's width. The block width (B_w) is equal to **width(R)/m + width(R)%m**. Once B_w is determined, the parallel distribution of R will be implemented using the formula in Figure 3 that will run in parallel.

```
TID = Retrieve the ThreadID()
FOR i = (TID%D1)Bw to  (TID%D1 + 1)Bw
Do F(R[i])
```

Figure 3: Parallel UBD of one-dimensional space.

Where in Figure 3, *TID* is the thread ID in a parallel region [0 to (m – 1)]. This parallel loop will be run at once among all threads. Each thread will utilise the value of its *TID*. However, in some cases, the thread with the largest value of *TID* (m – 1) will go over the length of that dimension when the length of one or more of the dimensions is not divisible by the number of blocks of that particular dimension, as shown in Figure 4. In order to make sure that no such memory error can happen, a *min* function will be used. Therefore, the final equation for the one-dimensional parallel UBD is illustrated in the formulation in Figure 5.

Figure 3: Thread accessing memory beyond the array size.

```
TID = Retrieve the ThreadID()
FOR i = (TID%D1)Bw to  min ( (TID%D1 + 1)Bw, width)
Do F(R[i])
```

Figure 5: Parallel UBD of one-dimensional space with the memory accessing treatment.

3.5 Two-Dimensional Domains

For a two-dimensional *R* (width, length), *N* will be equal to 2, and in this case, two variables will be created (D_1 and D_2) and initialised to 1. The next step is to factorise *m* because $N > 1$. If the number of factors of *m* is greater than the number of dimensions, a factor reduction algorithm will be employed to reduce the factors to two reduced factors. The value of these two factors will substitute for D_1 and D_2. Similar to the case of one-dimensional space, the next step is to determine the dimension of the blocks. In this case, B_w will be equal to $\mathbf{width(R)/D_2 + width(R)\%D_2}$ and the block's height (B_l) will be equal to $\mathbf{length(R)/D_1 + length(R)/D_1}$.

In two-dimensional space, we note that in order for each thread to access one block for processing, we need to use the integer division (/) and integer modulus (%) operations, rather than using only integer division, as in the case of one-dimensional space. Figure 6 shows the mathematical formulation embedded in two *for* loops.

```
TID = Retrieve the ThreadID()
FOR i = (TID/D1)Bw to min (((TID/D1) + 1 )Bw  , width)
For j = (TID%D1)Bh to min (((TID/D1) + 1 )Bh  , length)
  Do F(R[i,j])
```

Figure 6: Parallel UBD of two-dimensional space with the memory accessing treatment.

3.6 Three-Dimensional Domains

a three-dimensional *R* (width, length, height) such as a cube, *N* will be equal to 3. Therefore, the number of created variables is three (D_1, D_2, and D_3). Similar to the previous two examples, the next step is to factorise *m.* If the number of factors is more than three, then a factor reduction algorithm will be used to reduce the factors to three. The values of these reduced factors will substitute for the values of D_1, D_2, and D_3. As expected, the next step is to determine the block's dimensions (width, length and height). B_w and B_l are determined using the same equations for two-dimensional space, while the block's height (B_h) is equal to $\mathbf{height\ (R)/D_1 + height\ (R)\%D_1}$.

Once all the block dimensions are determined, Figure 7 shows how a three-dimensional space is uniformly distributed using a three *for* loops.

```
TID = Retrieve the ThreadID()
For i= (TID/(D1xD2))Bw to min(((TID/(D1xD2) + 1)Bw ,width)
```

```
For j= ((TID/D1)%D2)Bl to min(((TID/D1)%D2) +1)Bl, length)
For k= (TID%D1)Bh to  min ( (TID%D1 + 1)Bh, height)
  Do F(R[i,j,k])
```

Figure 7: Parallel UBD of three-dimensional space with the memory accessing treatment.

3.7 Inter-Subdomain Communication

In many applications, the interface between blocks (block boundaries) receives special attention. As an example, in block-based video and image coding, the blocks' boundaries will enter a coding stage called a deblocking filter. The purpose of this stage is to remove artefacts around the block's boundaries. Hence, a partitioning algorithm needs to ensure the lowest possible length for such boundaries to provide better algorithm efficiency. In our algorithm, such a requirement has been achieved by adding the condition that $\sum D_i$ must be the lowest among the other conditions. Figure 8 shows an example where a two-dimensional array is partitioned into 30 blocks by assuming that the values of x and y are comparable. Figure 3 (a) shows the optimal partitioning where the interface (borders) of the blocks is less (4y + 5x) compared to Figure 3 (b) (9y + 2x) and Figure 3 (c) (14y + x).

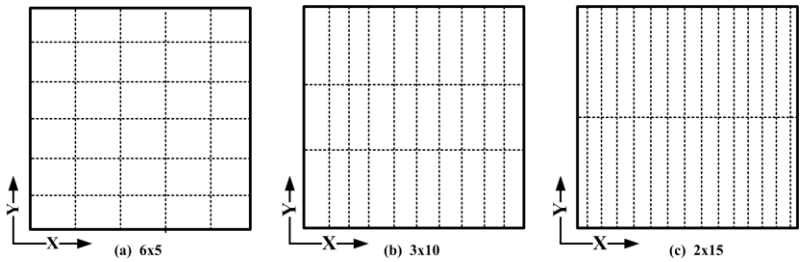

Figure 4: Different scenarios of 2D array partitioning.

4. WORKING EXAMPLES

In order to validate the proposed parallel algorithm, we need to propose working examples with different input parameters. Table 1 summarises the scenarios that cover all possible cases for one-, two-, and three-dimensional space. The purpose is to know whether the proposed parallel GBP algorithm will correctly distribute the input space into m blocks. When $N = 1$, three possible cases are proposed, $m = 1$ (no partitioning), $m = 2$ (even number), and $m = 3$ (odd number). When $N = 2$, similar scenarios are proposed in addition to the scenario of $m = 4$, which shows the partitioning along both dimensions. Finally, for $N = 3$, the scenarios cover the partitioning among one, two, and three dimensions with an additional scenario ($m = 27$), which is an odd number of partitions for all dimensions.

Table 1: Scenarios of one-, two-, and three-dimensional parallel array partitioning

Number of Dimensions (*N*)	Number of Blocks (*m*)	Input Size
One Dimension	1	100
	2	100
	3	100
Two Dimensions	1	100x100100 × 100
	2	100 × 100
	4	100 × 100
	9	100 × 100
Three Dimensions	1	100 × 100 × 100
	2	100 × 100 × 100
	4	100 × 100 × 100
	8	100 × 100 × 100
	27	100 × 100 × 100

5. IMPLEMENTATION, PARALLEL EVALUATION, AND EXPERIMENTAL RESULTS

The main reason from employing parallelism is to finish a particular programming task in a shorter time. Therefore, this section will show how and when the proposed parallel algorithm has outperformed the serial algorithm with regard to different array sizes and a different number of blocks. As the correctness of the proposed algorithm has been mathematically proven, the parallel evaluation will emphasise different array sizes and a different number of blocks to show the speedup and scalability of the proposed parallel algorithm. Regardless of the number of dimensions, the total numbers of pixels in each experiment were 10^3, 10^4, 10^5, and 10^6, while the number of blocks (thread) was limited to 2, 4, or 8 owing to hardware availability.

We have used the OpenMP standard to implement our parallel algorithm. However, Pthreads or any other parallel library can employ the proposed algorithm as long as those libraries have a built-in facility to retrieve the thread ID within a parallel region. The OpenMP style using C++ language for three, two, and one dimensions is shown in Figure 9.

```
#pragma omp parallel shared (R) num_threads (nthreads)
{
int TID =  omp_get_thread_num();

for (int i = (TID/(D1 * D2)) * BW ; i < min (  ((TID/(D1 * D2)) + 1) * BW , WIDTH);
i++)
for (int j = ((TID/D1)%D2) * BL; j <min (  ((TID/D1)%D2 +1) * BL , lENGTH); j++)
for (int k = (TID%D1) * BH; k < min (  ((TID%D1) + 1) * BH, HEIGHT) ; k++)
run_over(R);
}
```

(A) OpenMP parallelisation of GBD for 3D space

```
#pragma omp parallel shared (R) num_threads (nthreads)
{
int TID =  omp_get_thread_num();
for (int i = (TID/D1) * BW2D; i < min (   ((TID/D1) + 1) * BW2D , WIDTH2D) ; i++)
for (int j = (TID%D1) * BL2D; j <min (  ((TID%D1) + 1) * BL2D, lENGTH2D); j++)
run_over(R);
}
```

(B) OpenMP parallelisation of GBD for 2D space

```
#pragma omp parallel shared (R) num_threads (nthreadsD)
{
int TID =  omp_get_thread_num();
for (int i = (TID%D1) * BW1D; i < min (    ((TID%D1) + 1) * BW1D , WIDTH1D);
i++)
run_over(R);
}
```

(C) OpenMP parallelisation of GBD for 1D space

As shown in the Pseudo above ,we have used *#pragma omp parallel* instead of *#pragma omp parallel for*, although the latter is designed to be used with a *for* loop, because the loop is explicitly partitioned into blocks among threads. Therefore, *#pragma omp parallel* is used, which is designed to duplicate the code between brackets and pass a copy of this code to each individual thread. *num_threads* is an OpenMP clause used to set the number of threads in a thread team. *omp_get_thread_num* is an OpenMP function used to return the thread number of the thread executing within its thread team. Finally, *run_over* is a simple polynomial function of the form of $aR + b$, where a and b are constants.

The hardware environment was an Intel® Core™ i7-4960HQ Processor (4 cores, 8 threads) and 4 GB of RAM, and the software platform was Microsoft Visual C++ 2010 Professional. Results were evaluated with regard to the serial version of the *run_over* function as well as the original OpenMP version. Moreover, in order to provide a more reliable evaluation environment, the number of processors has been explicitly configured to suit the number of threads in each case. Time measurement was in milliseconds. Figure 11 shows the execution time for one-, two-, and three-dimensional spaces for different sizes and a different number of threads. In Figure 11-14, **P** refers to the original parallel (see pseudo code below), while **NP** refers to the new parallel paradigm proposed in this work.

```
#pragma omp parallel for private (j,k) shared (R) num_threads (nthreads)
for (i = 0; i < width; i++)
for (j = 0; j < length; j++)
for (k = 0; k < height; k++)
run_over(R);
```

(A) Typical OpenMP parallelisation for 3D space

```
#pragma omp parallel for private (j) shared (R) num_threads (nthreads)
for (i = 0; i < width; i++)
```

```
    for (j = 0; j < length; j++)
    run_over(R);
```

(B) Typical OpenMP parallelisation for 2D space

```
#pragma omp parallel for shared (R) num_threads (nthreads)
for (i = 0; i < width; i++)
run_over(R);
```

(C) Typical OpenMP parallelization for 1D space

Figure 5: Typical OpenMP parallelisation (row-wise).

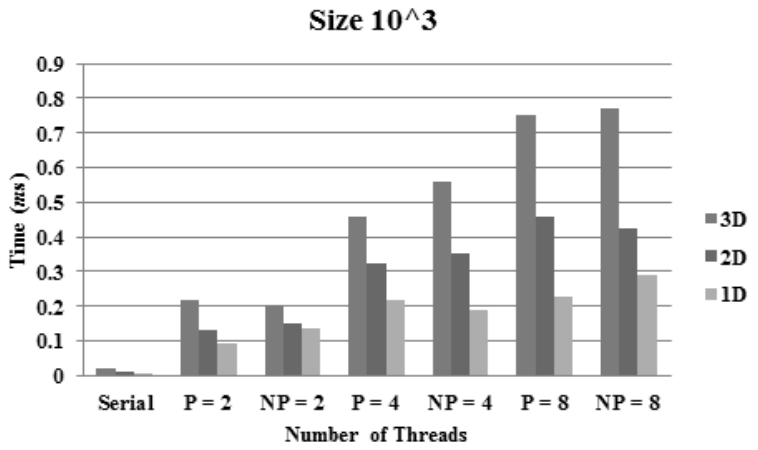

Figure 6: Execution times of the typical OpenMP parallelisation and the proposed UBD, size = 10^3

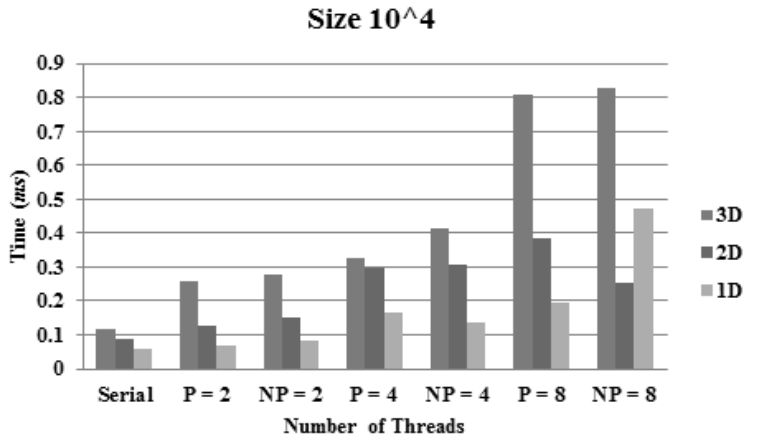

Figure 7: Execution times of the typical OpenMP parallelisation and the proposed UBD, size = 10^4

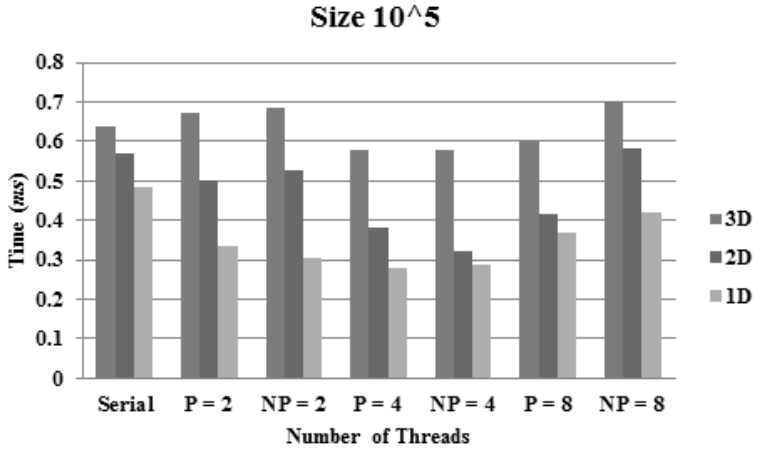

Figure 8: Execution times of the typical OpenMP parallelisation and the proposed UBD, size = 10^5

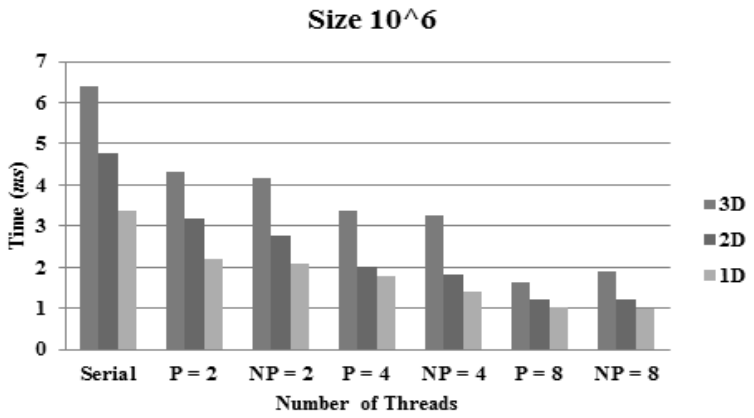

Figure 9: Execution times of the typical OpenMP parallelisation and the proposed UBD, size = 10^6

As can be observed in Figures 11-14, the when the size of the space is relatively small (10^3), the serial version is faster than the parallel version. Moreover, more time is needed compared to the serial version when more threads are created. This is due to the overhead caused by thread creation and termination. This observation changes when the size of the space becomes bigger (10^5); as illustrated in some experiments, the parallel implementations outperform the serial version in terms of time. Obvious parallel efficiency is observed when the size of the data is 10^6. In this scenario, the task is executed faster when more threads are created. This is due to the fact that the parallel granularity is quite sufficient for each thread to provide a considerable speedup.

Moreover, encouraging results are observed between the parallel time of the proposed parallel algorithm and the original implementation of OpenMP. The results of the proposed parallel algorithm exhibit comparability, and there was no significant time difference for a particular case although the proposed algorithm overcomes several limitations of the original OpenMP style, such as multidimensional partitioning (explicit nested parallelism).

6. CONCLUSION AND FUTURE DIRECTIONS

This work has presented a new parallel algorithmic design approach for the block partitioning problem. The main theme of this work is represented by the employment of the facilities in the parallel library in the design phase of UBD algorithm. Experimental results demonstrate a considerable parallel speedup and efficiency compared to the serial version of UBD. The contribution of this work represents an example of static scheduling where data are partitioned regardless of their workload. This approach is good with data that has a comparable intra-workload. However, the proposed parallel algorithm can be extended to support dynamic scheduling by adjusting the size of the partitioned blocks to reach a near-optimal distribution because the optimal distribution of this problem is NP-complete. Moreover, the number of blocks can be a prime number for particular cases. For the one-dimensional space there is no problem, but for two or more dimensions, there will be a problem, as it is impossible to find two numbers or more in which the result after multiplication is a prime number. Hence, it is possible to subtract a number from the prime number so that it becomes an even number. By doing such a step, it will be possible to find two or more numbers where the result of multiplying them will result in that even number, and a multidimensional distribution can be made. Finally, several optimisations can be incrementally introduced because the work of this paper represents an algorithmic-level contribution.

REFERENCES

Aspvall, B.; M. Halldórsson, and F. Manne, 2001 "Approximations for the general block distribution of a matrix," *Theor. Comput. Sci.*, vol. 262, pp. 145–160.

Gaur, D. R. ; T. Ibaraki, and R. Krishnamurti, 2002. "Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem," *J. Algorithms*, vol. 43, pp. 138–152.

Gwo-Giun, L.; C. Yen-Kuang, M. Mattavelli, and E. S. Jang, 2009 "Algorithm/ architecture co-exploration of visual computing on emergent platforms: Overview and future prospects," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, pp. 1576–1587.

Manne, V and T. Sørevik, 1996. "Partitioning an array onto a mesh of processors, in *Applied Parallel Computing Industrial Computation and Optimization*, vol. 1184, J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, Eds. Heidelberg: Springer Berlin, pp. 467–477.

Muthukrishnan ,V., and T. Suel, 2005 "Approximation algorithms for array partitioning problems," *J. Algorithms*, vol. 54, pp. 85–104.

OpenMP Application Program Interface Available: http://www.openmp.org/mp-documents/spec30.pdf.

Pacheco ,P. S., 2011 "Shared-memory programming with OpenMP," in *An Introduction to Parallel Programming*, P.S. Pacheco, Ed. Elsevier: Morgan Kaufmann, Ch. 5, pp. 209–270.

Pthreads (POSIX Threads),V., , 2011 in *Encyclopedia of Parallel Computing*, vol. 2092, D. Padua, Ed., New York: Springer, p..

Saule, E. ; E. O. Bas, x, atalyu, and U.V. Rek, 2011. "Partitioning spatially located computations using rectangles," *IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, Alaska, pp. 709–720.