Parallelization Programming Techniques: Benefits and Drawbacks

Goran Martinovic
Faculty of Electrical Engineering
J.J. Strossmayer Univ. of Osijek
Osijek, Croatia
e-mail: goran.martinovic@etfos.hr

Zdravko Krpic
Faculty of Electrical Engineering
J.J. Strossmayer Univ. of Osijek
Osijek, Croatia
e-mail: zdravko.krpic@etfos.hr

Snjezana Rimac-Drlje
Faculty of Electrical Engineering
J.J. Strossmayer Univ. of Osijek
Osijek, Croatia
e-mail: snjezana.rimac@etfos.hr

***Abstract*-In engineering program implementations, there is always a need for more computer resources, apart from which, many computer resources are still unused. The most common resource demanding applications are applications which require a lot of processor power or RAM space. Today, advancing technology offers processing power in grids, clusters, multi-core CPUs, cloud computers, or even graphics processing units. Thus, given all this computing power, smart and efficient utilization of these systems is needed. All these necessities and mentioned facts laid foundation for parallel programming. One of the major issues in parallel programming is reconfiguring of the existing applications to work on a parallel system; not just to work, but to work faster and more efficiently. In this paper some of the most common parallelizing methods will be presented using MPI on the Croatian National Grid Infrastructure (CRO-NGI), as well as their advantages in terms of cost-effectiveness and simplicity.**

Keywords: computational grid, load balancing, MPI, parallel computing.

I. INTRODUCTION

The complexity, data requirements and processing in scientific researches, such as visualization and modeling in various scientific branches continue to increase. Problems in medicine, weather prediction, global climate modeling, complex stress calculations in mechanics, etc. are good examples of computer intensive applications. Historically, the computational power of computer resources has not been able to keep pace with this increase and for this reason, parallel computer systems (PCS) were developed. Not every resource intensive problem can be solved in decent time manner on simple mainstream computers, as shown in [3] and [10]. Single-processor systems and single core processor computers by themselves are getting time consuming in running these applications, and are causing major drawbacks of developing such applications. As resource consumption by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors [14]. Today, there are various types of parallel computing systems, like clusters, grids, distributed systems, multi-core and many-core processors and a recent concept – cloud computing systems, all based on spreading use of parallel algorithms. Nowadays parallel computers are very common in research facilities as well as companies all over the world and they are used extensively for complex computations. Some of the most powerful supercomputers are made with over 10,000 processors, and are capable of reaching over 1 petaFLOPS. Careful and effective parallel programming is the only way to bridle such enormous computing power. Massive migration to parallel systems causes that still many applications need to be adjusted for the use on these systems by means of two main options: recoding or writing the code in parallel from scratch. Sometimes, these procedures are not as intuitive as one may think, because not all tasks can be parallelized.

In this paper different parallelizing methods and techniques will be mentioned. A couple of experiments will be used to support or to undermine these aforementioned facts and myths about parallel programming. Finally, efficiency of parallel computer systems will be questioned, and their advantages and disadvantages will be compared to serial systems. Section 2 describes basic aspects of parallelization. The concept of automatic parallelization is briefly explained in Section 3. Prior to experimental results given in Section 5, Section 4 presents formerly known benefits of parallel programming in order to experimentally confirm or reject them. Section 6 presents planned work, and Section 7 brings noted conclusions.

II. PARALLELIZATION METHOD IN USE

Many parallelization tools and compilers are already available and yet many more are in development. Two basic modes of parallelization are automatic parallelization and manual parallelization. Automatic parallelization techniques are mainly the tools for real-time systems and scientific computing industry [1], while manual parallelization allows maximum application optimization for parallel execution and performances gain. Interesting concepts are parallelization compilers, which turn codes parallel and make the application parallel at runtime by which they can be perceived as automatic parallelization techniques [12]. Manual parallelization techniques are issued by programmers. The latter techniques are based on speculative parallelization [8], parallelization of loops [16], dynamic data parallelization [2], load balancing, thread pipelining,

 ISBN: 978-1-61208-106-9

data access partitioning [7] and others. Parallel programming techniques, parallel platforms and parallel programming tools are in constant development.

The main goal of this paper is to point out some problems that appear by optimization of sequentially written programs for execution on parallel platforms, and presenting various parallelization methods. Various data structures, loops and iterations can be parallelized efficiently without rewriting the whole program code. Another important issue is load balancing, the goal of which is to gain a higher throughput, and to reduce the user-perceived latency, especially in the case of high network traffic or a high request rate causing the network to be bottlenecked, or a high computational load, [11]. Load balancing is hardly achieved on, for example, cloud computing systems, because of the high system heterogeneity, especially network. These systems are overwhelmed with inconsistency and are prone to many changes in the matter of seconds, and that is why these systems have to use standalone load balancing appliances or content switches.

The first step of parallelizing an application is to find the most resource/time intensive part of the program (algorithm). If this part cannot be made parallel, little can be done for speeding up the application. The next step, step two, is to determine whereas parallel parts of the code are data independent, and remove this dependency if possible. Then there comes step three, i.e., determining a method which will be used in parallelization of kernels. These steps alone comprise several sub-steps, which are chosen according to the problem at hand. Several concepts will be introduced in this paper for parallelization: data partitioning, loop parallelization and functional decomposition, which are given in detail in [2] and [9]. Other concepts are briefly presented in Table 1.

Data parallelization (or data decomposition) is based on parallelizing data, e.g., dividing large databases, matrices, vectors and other data types into small chops often adjusted to be processed on the nodes of parallel systems. Data can be divided equally or in some other manner (load balancing or adaptive parallelization, [11]). A negative impact of dividing data in that manner is conspicuous on computer grid systems with nodes interconnected with the network, because the network is the slowest subsystem in inter-application communication. Other important issue to be confronted with is reducing communication between processes. There are three different types of algorithms based on communication frequency: coarse-grained, fine-grained and embarrassingly parallel. There are various methods of reducing user perceived latencies, which are derived from inter-process communication. Another shortcoming can be seen in heterogeneous systems in which data should be divided in accordance with available computer resources, see [10]. In heterogeneous systems the best performance gain would be noticed if there is a system monitoring current resource availability and sending information about available resources back to the application, which in turn sends appropriate portions of data to computer nodes, as shown in [10]. These systems exist in cloud computing systems, however, with only limited functionality. There is a number of possible data partitioning, first when portions of data are sent to nodes, and second when a copy of the whole data is sent to all computer nodes, as in Figure 1, the latter using more communication, so it has to be tested thoroughly which system benefits from this approach. Figure 2 shows which data are computed by which node.

```
myRank = MPI::COMM_WORLD.Get_rank();
nProc = MPI::COMM_WORLD.Get_Size();

for(i=((CONST/nProc)*myRank);i<(CONST/nProc)+(
(CONST/nProc)*myRank);i++) {
   for (j=0;j<DIMV;j++) {
      if ((maxi[11] < optiMatrix[i][j]) &&
(matrix[i][j][16] !=0)) {
         maxi [0] = I;
         maxi[3] = j;
         maxi [11] = optiMatrix [i][j];
      }
   }
}
```

Figure 1. Example of data parallelization

Node A	Node B	Node C

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T

Figure 2. Visual representation of data division amongst parallel nodes

Dividing data in programming is tightly coupled with loop parallelization, because data containers are generally accessed by loops, so a condition deciding which data is going to which node should be put in the loop initialization and termination. Loops are often parallelized by dividing the number of iterations equally, if possible, amongst computer nodes, as in [3], [9], [13], and [16]. Data division is prone to data dependency, and by that care must be taken when preparing data for parallel distribution. Loops can be made parallel only if iterations are not data dependent. In other words, Bernstein's conditions [4] have to be fulfilled.

The third concept is based on functional decomposition, described in [4] and [9]. This is done by dividing the program into functional blocks independent of each other at the time of execution, e.g., one does not require data of the other. This is explained by the Church-Rosser property [4], which holds that the arguments to a pure function can be evaluated in any order or in parallel, without changing the result. A negative side of functional decomposition can be seen if parallel computations differ in the execution time a lot, because the slowest one determines the total execution time of this parallel part assuming that a computation result must be provided before processes move on to the next

 ISBN: 978-1-61208-106-9

program block, except in applications with active load balancing. If the latter is not the case (computation results are processed independently), the algorithm is called *embarrassingly parallel*. Embarrassingly parallel algorithms are often used in cloud computing systems.

Main program			
Declarations and initializations Common procedures			
Node 1	**Node 2**	**…**	**Node n**
Search the DATA for value "A"	Search the DATA for value "B"	…	Search the DATA for value "N"

Figure 3. Example of a pseudo-code of data parallelization for searching for different values in the same data set

A living example is searching for two or more values in a data matrix, then the first node searches for one number and the second node searches for the other (if all nodes are assumed to have access to all data), as in Figure 3.

Computer nodes intercommunication is generally done by sending some signal in a preconfigured way by using some routines. This type of parallel programming is called Message Passing. Message passing applications communicate over a high speed network in distributed computing systems, or over high speed buses in shared memory computers, so the program can hold sufficient cohesiveness.

III. AUTOMATIC PARALLELIZATION

Given the example of [1], [3], [5] and [14], in order to remove the burden from a programmer to manually rewrite sequential codes for parallel execution, many new methods are introduced as an attempt to solve this problem automatically. They often comprise compilers which "know" how to parallelize a certain program code. A vast majority of automatic parallelization compilers are developed for FORTRAN, such as the Vienna Fortran compiler, the Paradigm compiler, the Polaris compiler, the SUIF compiler, and some of the concepts independent of the programming language, such as commutativity analysis [14]. Automatic parallelization is the ultimate goal for parallel programming, as it removes the programmer from the parallelizing part in coding the application, thus making parallel applications production faster and more efficient. Every automatic parallelization concept has been done only with limited success. Despite poor progress, automatic parallelization has been intensively researched for the past few decades, and a lot of work is still dedicated to it.

An interesting concept for automatic parallelization is presented in [14], which is called commutativity analysis. It aggregates both data and computation into larger grain units. It then analyzes computation at this granularity to discover when pieces of computation commute (i.e., generate the same result regardless of the order in which they are executed). If all of the operations required to perform a given computation commute, the compiler can automatically generate a parallel code. Some sources also describe various hybrid approaches, such as in [9]. Parallelization methodologies are expanded in Table 1, which gives additional information about other main parallelization techniques, advantages and disadvantages. Even with many methods for automatic parallelization, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for a complex program analysis and the unknown factors (such as the input data range) during compilation. Automatic parallelization combined with cloud computing systems in near future will probably serve as self-sufficient parallel systems, which will bring high performance computing to every computer user connected to web.

IV. BENEFITS OF PARALLEL PROGRAMMING

It would be expected of parallel programs to have the execution time cut in proportion with the number of nodes, as opposed to sequential programs. However, if this fact is analyzed more thoroughly, there is always some portion of code which cannot be parallelized, and that portion must be taken into account. This issue was addressed by Amdahl's and Gustafson's law [3]. Some researchers noticed that even with Gustafson's law, which suggests that it is beneficial to build a large-scale parallel system as the speedup can grow linearly with the system size, there are some physical constraints which do not allow many applications to scale up and meet the time bound constraint. In practice, that constraint is often of physical nature in the form of memory limitation.

To sum it all up, in [15] authors propose a memory bounded speedup model. Amdahl's law is a special case of a *memory bounded speedup model*. This model is greatly applicable to multi-core systems and GPU cores, which represent home shared memory systems. Embarrassingly parallel applications are not affected by Amdahl's law, at least not to such a great extent, and that is why these applications can easily be run on clouds. These applications often comprise independent tasks, which are then executed on different computer nodes, without a need for any type of communication, or data dependencies, except at the beginning of a code. With all added up, parallel applications have many benefits from today's parallel systems. Most of parallelizing methods can be run on almost all existing parallel systems. There are limitations, but with constant research in this field, the number of limitations decreases. For example, cloud computing systems can serve as parallel platform only for applications that are easily parallelized. In all other cases this is nearly impossible, because cloud system is too hard to handle resource-wise, which is caused by the high system heterogeneity. With parallel system properties in mind, it is easy to classify given parallelizing methods from Table 1 in correspondence with these platforms. Every existing application can be more or less parallelized; it is the question of cost and time-effectiveness, a parallel system on which the application is run, time bound constraints on application and application environment which method will be used.

 ISBN: 978-1-61208-106-9

V. EXPERIMENTAL SETUP

In order to visualize given facts into real appliances benefits and speedup, two experiments will be shown. In the first experiment, there are two multi-criteria optimization algorithms, whose performance will be compared. In the second algorithm, a parallel image processing algorithm is tested in different environments and with a different setup.

A. Experiment 1: Multi-criteria optimization algorithms (PMCO1 and PMCO2)

The first algorithm, Parallel Multi-Criteria Optimization 1 (PMCO1) is an example of computing large data in parallel with a small amount of communication between processes. It uses the approach described in [10]. The system makes a decision based on different preferences amongst options in a large data set. There are databases containing various system parameters, and prior to program execution it is necessary to extract data from these databases, which are formed by plain text files. PMCO1 reads different databases in different computer nodes.

TABLE I. PARALLELIZATION MODELS COMPARED

Parallel Program Design	Parallelization technique	Positive features	Negative features
Manual Parallelization	**Shared Memory**	No need for communication between tasks	Difficult data locality management
	Threads	Fine program granularity and efficient platform utilization	Not reusable, errors affect whole process
	Data Parallel Techniques	Large performance increase, error on one data "chunk" rarely affect other data	No performance increase if the data is not independent, need for task communication
	Message Passing	Universality, Data locality management, Easy debugging	Programmer manages memory placement and communication occurrence
	Hybrid	Combination of the techniques above	Combination of two or more parallelization techniques can greatly reduce their disadvantages
Automatic Parallelization i.e. parallelizing compilers (pre-processors)	**Fully Automatic**	Parallelization without programmer, fast parallel code generation, computer aided parallelization cost-effectiveness analysis	Can produce wrong results, application performance can be actually degraded, much less flexible than manual techniques, if the code is too complex parallelization cannot occur
	Programmer Directed	Usage of compiler directives, better parallelization management	

B. Experiment 1: Multi-criteria optimization algorithms (PMCO1 and PMCO2)

The first algorithm, Parallel Multi-Criteria Optimization 1 (PMCO1) is an example of computing large data in parallel with a small amount of communication between processes. It uses the approach described in [10]. The system makes a decision based on different preferences amongst options in a large data set. There are databases containing various system parameters, and prior to program execution it is necessary to extract data from these databases, which are formed by plain text files. PMCO1 reads different databases in different computer nodes. This parallelism is based on function level parallelism (FLP). The second algorithm, Parallel Multi-Criteria Optimization 2 (PMCO2) is a parallel algorithm which reads all databases in every node, and serves the analysis of the computational part of the program. PMCO2 is mainly a data parallel model, but it uses a hybrid approach, explained in [2], comprising FLP and data parallelism, and for the purpose of illustration, its execution time is divided into reading data and computation. In PMCO2, every node has access to all data and there is a significant process communication time, but communication takes place rarely. PMCO2 approach enables all nodes to read only a portion of data, regardless of the fact that they contain the whole database. On the other hand, PMCO1 has more frequent communications between processes, because nodes contain only a portion of data. These small communications can make a great deal if the database is very large as in Table 3. In a small data set, PMCO1 tends to have better performance (Table 2). PMCO2, though, has one more drawback, which is memory-wise, considering the fact that every node holds all data. So if the data is too large, it would not be possible to run the program based on PMCO2, whereas on PMCO1 it would be possible but slow.

C. Experiment 2: Parallel Image Processing Algorithm (PIPA)

The second experiment deals with an image processing algorithm, which does some basic pixel manipulation on the grayscale satellite images in different sizes. The application was run in parallel on the Croatian National GRID infrastructure on 4 and 8 nodes, as well as on two different CRO-NGI installations (ETFOS, located at the Faculty of Electrical Engineering, University of Osijek and SRCE (University Computing Center in Zagreb). What can be easily seen is the difference in performance, as well as the impact of different architectures and operating systems on the application execution time. The application was also run on a PC with a dual-core processor. In that way it can be analyzed whether parallel programming and execution on parallel systems can be justified by taking performance into main consideration. Figure 4 shows dependency of the application execution time on image size by different operating systems, numbers of nodes, numbers of processor cores and image sizes.

 ISBN: 978-1-61208-106-9

TABLE II. ALGORITHMS PMCO 1 AND 2 PERFORMANCE TEST (A SMALL DATA SET)

		4 – node computer grid time (s)	6 – node computer grid time (s)
PMCO 1	Data read	1.79	0.78
	Computation	3.77	8.00
	Total	5.56	8.78
PMCO 2	Data read	12.24	15.46
	Computation	0.20	0.15
	Total	12.44	15.61
Program execution time difference (s)		-6.88	-6.83

TABLE III. ALGORITHMS PMCO 1 AND 2 PERFORMANCE TEST (A LARGE DATA SET)

		4 – node computer grid time (s)	6 – node computer grid time (s)
PMCO 1	Data read	9.89	3.18
	Computation	261.67	206.27
	Total	271.56	209.44
PMCO 2	Data read	42.60	55.75
	Computation	24.21	17.23
	Total	66.81	72.98
Program execution time difference (s)		204.75	136.46

Image size affects performance most, which is expected, because image size grows almost exponentially.

Also, the number of grid nodes and processor cores, which can be distinguished in Figures 5 and 6, has a great impact on performance. Figure 7 shows that careful multi-core parallel programming can lead to a significant performance boost, which is almost 90% of the increase, with doubling the number of cores. Multi-core processors show their true strength when loaded with applications optimized for multi-core execution. Also, multi-core platforms do not have one major drawback which grids and clusters have, and that is a relatively large process communication time in message passing applications. Figure 9 shows a process communication impact on application performance. It clarifies what was mentioned before; i.e., the grid suffers from great performance loss when using too much of communication between processes. Furthermore, this is more expressive in public computational grids and cloud systems, whose networks are always under some load, leading an application expected to finish faster to finish slower, waiting for processes to finish their communication. On the other hand, there is a communication between processes run on multiple cores onto one processor, whose process communication time can be safely ignored in performance analysis. This issue is a problem of its own and part of future work based on heterogeneity modeling. But not to be confused, grids offer a big advantage compared to multi-core processors. They can have much more nodes, which in turn can have multi-core processors themselves, and modern commercial multi-core processors can have only up to 8 cores, so it is up to the application which platform should be used in its execution. Another example are clouds, which are hybrid parallel systems and offer various performance advantages.

More performance boost can be obtained by increasing the number of computer nodes executing the application, as shown in Figures 4 to 7. Usage of ETFOS installation of CRO-NGI lowers process communication a bit (because of a lighter network load), and decreases execution time significantly.

VI. FUTURE WORK

Further work will be based on implementing more parallel platforms such as GPGPUs (General-purpose computing on graphics processing units) in NVidia CUDA (Compute Unified Device Architecture) and ATI Stream [6]. This work will tend to put these parallel newcomers into performance tables of other parallel systems. Another research is covering the role of cloud computing systems as parallel systems with great computing power. Image processing algorithms will be thoroughly reworked. Due to image size limitations, new image processing algorithms will be used with support for color images. There are other algorithms being developed; each of them will put different aspects of parallel systems onto test. There is also an idea to provide a mathematical proof for the exact limit of parallelization efficiency, and profitability of using parallel systems opposed to other parallel and non-parallel systems.

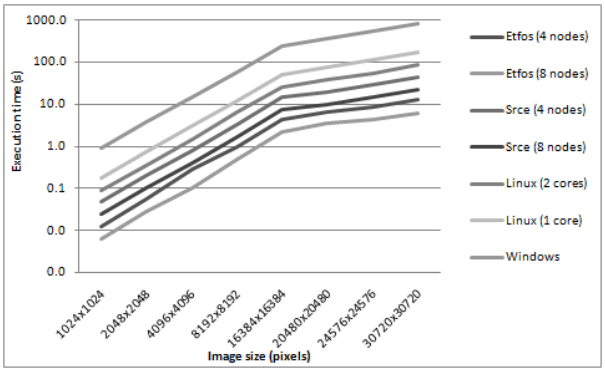

Figure 4. Performance gain

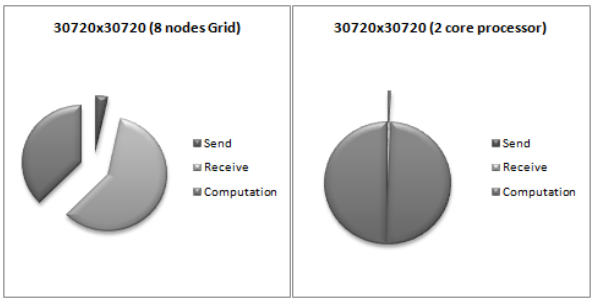

Figure 5. Communication impact on the overall performance

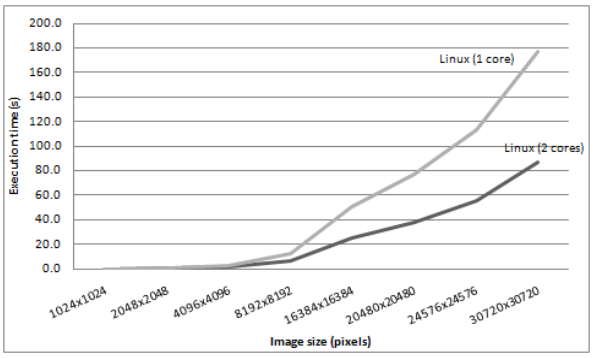

Figure 6. Performance with a different number of processor cores

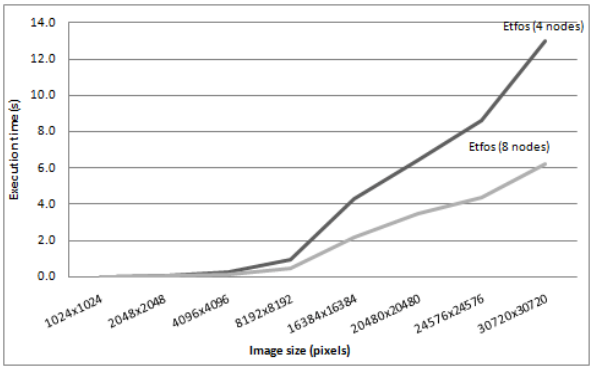

Figure 7. Performance with a different number of computer nodes

VII. CONCLUSION

The latest technologies give many opportunities when it comes to execution of demanding applications. More recent parallel systems such as computational grids, clusters, multi-core systems, Massively Parallel Processors systems and Graphics Processing Units are platforms that offer much more computer resources than standard PCs, and their ideas and technologies are slowly making their way to desktop computers. The best examples are multi-core computers, which share some backbone principles with parallel systems. Many existing applications are made sequential, but sometimes with just few changes they can be made parallel, therefore reducing their executing time and other time demands. Parallelizing methods are chosen in accordance with the type of parallel systems they are run on. Many parallelizing methods are presented today, and their ultimate goal is to produce a fully automatic parallelizing system, which can be used as any other today's programming language and system. The aforepresented experiments prove that parallel programming is the present and the future of all scientific research, not only in computer science, but in other researches as well. Parallel programming brings an enormous performance advantage, possibilities such as ability to process larger data and to calculate more complex mathematics and statistics. But this is only possible if these systems, their potential and their drawbacks are understood well. In order to make use of squeezing most out of these systems, careful and wise resource usage is needed, as well as efficient programming and work distribution. There are many other parallelization concepts, many of which are still being in their infant phase, but there is no doubt that research involved in them will produce more efficient, scalable and simple parallel applications and mechanisms. Such techniques involve function and block level parallelization, automatic parallelization, parallelization techniques for heterogeneous systems, speculative parallelization models, loop based parallelization techniques, and many more. Undoubtedly, parallel applications bring more performance, more options, and remove barriers for many research branches, bringing the overall sky-high progress in computing technology. It is obvious that in future, by combining these and similar methods, most advanced parallelization techniques will be created and parallel computing will be driven into the mainstream.

ACKNOWLEDGMENT

This work was supported by research project grant No. 165-0362980-2002 from the Ministry of Science, Education and Sports of the Republic of Croatia.

REFERENCES

[1] B. Armstrong and R. Eigenmann, Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries, Proc. 2008 Int. Conf. Parallel Processing, Portland, OR, USA, Sept. 8-12, 2008, pp. 279-286.

[2] D. Banerjee and J. C. Browne, Complete Parallelization of Computations: Integration of Data Partitioning and Functional Parallelism for Dynamic Data Structures, Proc. 10th Int. Parallel Processing Symp., Honolulu, HI, USA, Apr. 15-19, 1996, pp. 354-360.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua, Automatic Program Parallelization, Proc. of the IEEE, Vol. 81, No. 2, 1993, pp. 211-243.

[4] J. Błażewicz, K. Ecker, B. Plateau, and D. Trystram, Handbook on Parallel and Distributed Processing, Springer, 2000, pp. 96-173.

[5] U. Bondhugula, et al., Towards Effective Automatic Parallelization for Multi-core Systems, Proc. 22nd IEEE Int.

 ISBN: 978-1-61208-106-9

Symp. Parallel and Distributed Processing, Miami, FL, USA, Apr. 14-18, 2008, pp. 1-5.

[6] G. Chen, G. Li, S. Pei, and B. Wu, High Performance Computing Via a GPU, Proc. IEEE Int. Conf. on Information Science and Engineering, Shanghai, China, Dec 26-28, 2009, pp. 238 - 241.

[7] M. Chu, R. Ravindran, and S. Mahlke, Data Access Partitioning for Fine-grain Parallelism on Multi-core Archs., Proc. IEEE/ACM Int. Symp. on Microarchitectures, Chicago, IL, USA, Dec. 1-5, 2007, pp. 369-380.

[8] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, Copy or Discard Execution Model for Speculative Parallelization on Multi-cores, 41st Ann IEEE/ACM Int. Symp. on Microarchitectures, Lake Como, Italy, Nov. 8-12, 2008, pp. 330-341.

[9] K. A. Kumar, A.K. Pappu, K.S. Kumar, and S. Sanyal, Hybrid Approach for Parallelization of Sequential Code with Function Level and Block Level Parallelization, Proc. IEEE Int. Symp. Parallel Computing in Electrical Engineering, Bialystok, Poland, Sept. 13-17, 2006, pp. 161-166.

[10] G. Martinović, L. Budin, and Ž. Hocenski, Static-Dynamic Mapping in Heterogeneous Comp. Environ., Proc. IEEE Symp. Virtual Environments, Human-Computer Interfaces and Measurement Systems, Lugano, Switzerland, July 27-29, 2003, pp. 32-37.

[11] G. Martinovic, Resource Management System for Computational Grid Building, IEEE Int. Conf. Systems, Man, and Cybernetics, San Antonio, TX, USA, Oct. 11-14, 2009, pp. 1312-1316.

[12] T. Nakatani and K. Ebcioglu, Making Compaction-Based Parallelization Affordable, IEEE Trans. Parallel Distributed Systems, Vol. 4, No. 9, 1993, pp. 1014-1029.

[13] V. Purnell, P. H. Corr, and P. Milligan, A Novel Approach to Loop Parallelization, Proc. IEEE Proc 23rd Euromicro Conf., Budapest, Hungary, Sept. 1-4, pp. 272-277.

[14] M. C. Rinard and P. Diniz, Commutativity Analysis: A New Technique for Automatically Parallelizing Serial Programs, 10th Int. Parallel Processing Symp., Honolulu, HI, USA, April 15-19, 1996, pp. 14-14.

[15] X.-H. Sun and Y. Chen, Reevaluating Amdahl's Law in the Multi-core Era, J. Parallel. Distrib. Comput., Vol. 70 (2010), pp. 183-188.

[16] C. Xu and V. Chaudhary, Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences, IEEE Trans. Parallel and Distributed Systems, Vol. 12, No. 5, 2001, pp. 433-450.

 ISBN: 978-1-61208-106-9