Proceedings of the International Multiconference on Computer Science and Information Technology pp. 355–362
ISSN 1896-7094
© 2006 PIPS

Parallel Program Execution Anomalies

Jan Kwiatkowski, Marcin Pawlik, Dariusz Konieczny

Institute of Applied Informatics,
Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland
{jan.kwiatkowski, marcin.pawlik, dariusz.konieczny}@.pwr.wroc.pl

Abstract. The parallel runtime depends not only on the utilized algorithm but also on the hardware and software environment properties. Execution time anomalies resulting from unexpected behavior of the hardware and software environment or inexact performance model are well known to every scientist analyzing the performance of the parallel program. An abnormal behavior can usually be observed only for specific input data, execution parameters or hardware configurations. In the paper different sources of anomalies are discussed and the way how the performance analysis can be carried out in their presence is presented.

1 Introduction

One of the most important metrics analyzed during a parallel program evaluation is the time needed for its execution. The parallel runtime depends not only on the utilized algorithm but also on the hardware and software environment properties. The simplifying assumptions frequently imposed on the algorithm sometimes lead to the situation when the standard description method based only on the algorithm computational complexity fails to correctly describe the particular algorithm behavior. The execution time anomalies resulting from unexpected behavior of the hardware and software environment or inexact performance model are well known to every scientist analyzing the performance of the parallel program. This abnormal behavior can usually be observed only for specific input data, execution parameters or hardware configurations. It is easy to overlook, or even omit, these situations and concentrate only on the areas where the program executes as it is expected to. In the paper the opposite approach is followed. The full analysis of anomalies encountered during the evaluation of the parallel algorithm is presented utilizing as the example the Kohonen Self Organizing Map network offline parallelization algorithm chosen from the ones described in [4].

The aim of the paper is to discuss different sources of anomalies and to show how the performance analysis can be carried out in their presence. The structure of the paper is as follows. The second section describes the possible reasons of the abnormal parallel program behavior and presents the technique utilized to analyze the possible sources of anomalies. The next section presents methods of parallelization of the SOM algorithm. In section 4 the performance model of the algorithm is presented. The results of the experiments are discussed in details in section 5. The final section concludes the paper and presents the future work plans.

2 Sources of Execution Time Anomalies

The performance analysis can be carried out analytically or through experiments. One can find that an effective parallel program development cycle, may iterate many times before achieving the desired performance. In parallel programming the goal of the design process is not to optimise a single metrics like for example speed. A good design has to take into consideration a problem specific function of execution time, memory requirements, implementation cost, and others. During analytical or experimental performance evaluation of parallel programs different metrics can be used. The first one is the parallel run time [8]. It is the time from the moment when computation starts to the moment when the last processor finishes its execution. One can find that above definition is ambiguous: what is measured? wall clock time, CPU time or something else. In general we can determine that the parallel run time is composed of three different components: computation time, communication time and idle time, where the last two of them are execution overheads specific for parallel programs execution. Moreover different execution overheads related mainly to operating system can appear during parallel as well as serial programs execution. It means that the wall clock time which capture all of these overheads should be preferred during performance evaluation. Most other measures are related to the parallel execution time.

The second frequently used metric is speedup, which captures the relative benefit of solving given problem using parallel system. There are different speedup definitions. Generally the speedup is defined as the ratio of the time needed to solve the problem on a single processor to the time required to solve the same problem on parallel system with p processors. Depending on the way in which the sequential time is measured we can distinguish absolute, real and relative speedups. In the paper the relative speedup is used with the sequential time defined as a time of executing a parallel program on one of the processors of the parallel computer. Theoretically, speedup can not exceed the number of processors used during program execution, however different speedup anomalies can be observed. In theory a speedup anomaly occurs when the program does not execute in the way predicted by the performance model. Specifically the speedup can be larger than the number of available processors—so called superlinear speedup. There are three different reasons of the above phenomenon:

- algorithm dependant anomaly—caused by the internal algorithm structure, in other words the way how the problem is parallelized, for example in some parallel search algorithms if a search tree contains solutions at different depths, then after distribution of this tree among different processors, the solution can be found by exploring fewer number of nodes,
- hardware dependant anomaly—caused by some specific hardware features, for example by the interconnection network, size of cache, internal memory, etc. The "cache effect" is an example of this anomaly, when a program is executed on a large number of processors, the problem size is reduced and all needed data can be placed in the relatively faster cache memory, it causes the reduction of execution time and efficiency growth even over 1,
- execution environment dependant anomaly—caused by the execution environment, mainly operating system features like scheduling system, for example, when during parallel program execution processors with different performance are used.

The performance metrics presented earlier do not take into account the utilization of processors in the parallel system. The next metric called efficiency of a parallel program is defined as a ratio of speedup to the number of processors. In the ideal parallel system the efficiency is equal to one. In practice efficiency of the parallel systems is between zero and one, however when execution anomalies occurs, the efficiency can be greater than one. The next useful metrics is the scalability of the parallel system [1], [2]. In general it is a measure of its capanility to increase speedup in proportion to the number of processors. There are two ways in which scalability analysis can be carried out: with fixed and scaled problem size. The fixed problem size scalability analysis answers the question: what is the fastest I can solve problem A on computer X [1]. In this case different (mainly hardware dependant) execution time anomalies can occur. In their presence the second approach, scaled problem size analysis can be used. In this case it is checked if the efficiency can be kept at the same level when the problem size and the number of processors is concurrently increased. One can say that a system is scalable when for increasing number of processors and a size of the problem the efficiency is the same. In the paper the influence of these two approaches on existing speedup anomalies is shown. The last metric used in the paper is the computational throughput, defined as the amount of data processed by a single processor in a unit of time. During the throughput analysis only the part of data distributed between the processors with the size directly dependant on their number can be taken into consideration if the remaining data represents the constant time factor.

3 Parallelization of the SOM Algorithm

The Kohonen Self Organizing Maps (SOM) [3] are commonly employed to process large input data but their effective working abilities can be achieved only after a time-consuming process of learning. Hardware requirements needed for usage of even moderate size networks can easily exceed available resources. Parallel algorithms offer the solution to this problem. They divide the process of computations between many independently working small computers or utilize the whole computational power of larger, multiprocessor machines. While offering a possibility to reduce the time needed to create the resulting network, work on bigger data samples or prepare more detailed results, they preserve important properties of sequential SOM implementations.

In the original sequential algorithm the winner search and winners neighbours' weights modification are performed on-line in every learning step. The learning parameters update is run only once after each presentation of all the learning set vectors (at every epoch). The parallel algorithm variants are generally based on division of either the learning set (learning set parallelization) or the network (network parallelization) between the processors [7]. The speedup of parallel implementations utilizing on-line

algorithm is limited by the frequent communication. To lower the communication overhead, the off-line (with weights update performed once at the end of each epoch) algorithm can be employed.

4 Theoretical Performance Model

To analyze the performance of the algorithm its performance model should be constructed. In the paragraphs below the performance model of the single offline algorithm chosen for the further evaluation—the network offline parallelization [4] is presented.

In the algorithm the winner search procedure is divided between p processors by instructing them to find local winners in p times smaller part of the network (the mini-network), approximately reducing the search time by a factor of p. The winner position is determined and remembered in every step and the information transfer followed by neighbors' weights modification is performed at the end of every epoch.

The execution time of the parallel algorithm depends on the communication time and the time of computations performed on each of the utilized processors. In the network offline parallelization algorithm the computation time on the single processor consists of searching the winning neuron and modifying the winner and its surroundings weights. The time of the winner search operations is linearly dependant on the number of neurons. The number of modification depends on the epoch index. The radius of the surroundings changes in accordance to the formula $r(e) = S_a/(S_b + e)$, where S_a, S_b are a priori selected constants used to control the algorithm behavior and e is the index of the current epoch. It can be assumed that for typical algorithm execution every neuron has an equal probability of being the winner. The average number of neurons in the winner surroundings $s(e, n)$ is dependant on the radius of the surroundings $r(e)$ and the network size n. The precise number of neurons in the winner surroundings depends on the position of the winner in the network and the network size but when the assumption of the equal probability of being the winner among the neurons is assumed, $s(e, n)$ has an upper bound of $(2r(e) + 1)^2$. For the number of learning set elements equal N, it can be proved that the number of neurons modified during E epochs is described by the equation $S(e, n) = N \sum_{e=0}^{E-1} s(e, n)$. The shape of the $S(e, n)$ function and the experimental results for the execution parameters utilized for experimental evaluation ($S_a = 16, S_b = 2, E = 16, N = 1152$) is presented on fig. 1.

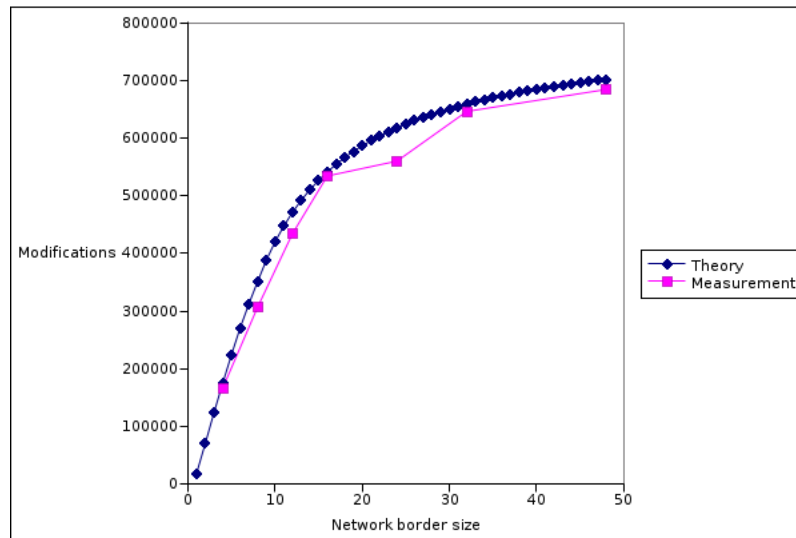

Fig. 1. Number of modifications in network offline parallelization

The runtime of the sequential SOM algorithm T_{seq} depends on the number of neurons n, the time of the elementary search operation τ_c, the number of learning set elements N and the number of epochs E. If the time of modifying a single neuron weights is τ_{sm} and the number of neurons in the winner's surroundings is $s(e)$, the time t_{mod} during which the weights of the winner and its surrounding neurons are modified is given by the equation $t_{mod} = \tau_{sm} \sum_{e=0}^{E} s(e)$. The resulting equation describing the execution time reads $T_{seq} = n\tau_c NE + t_{mod}N$

In the parallel version of the algorithm—the network offline parallelization—the communication phase is present after each epoch. Before starting communication, each processor computes and stores the

vector of winners found for each of the learning set elements. In the communication phase this vector is distributed between processors using reduction operation, resulting in determining one global winner for every learning set element. After this phase each processor modifies the winner and its surroundings weights if they are in its mini-network. The parallel runtime T_p for this algorithm is given by the equation $T_p = (En\tau_c N/p + Et_{mod}N/p) + (Et_r(N,p))$, where $t_r(N,p)$ is a time for communication with reduction for N winners and p processors. The shape of the curve representing the algorithm parallel runtime is presented on fig. 2.

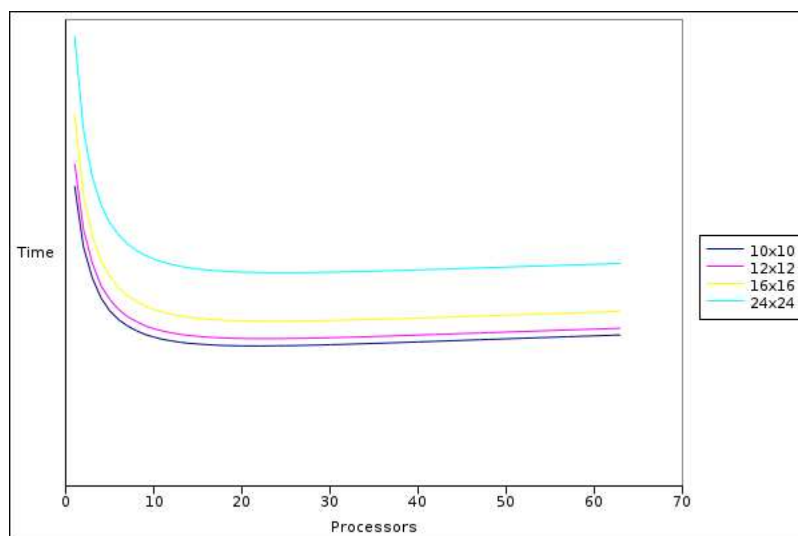

Fig. 2. Network offline algorithm execution time

The number of operations needed for the winner search is constant with respect to the number of processors. With the growth of the number of processors, the number of winners in the mini-network of a single processor decreases. When the communication overhead is small (i.e. for small number of processors) this decrease results in the decrease of the overall execution time. After it reaches its minimum, the raising communication overhead starts to result in the execution time growth.

5 Experimental Evaluation of the Parallel SOM Algorithm

To confirm the correctness of the theoretical analysis presented in section 4 the series of experiments was performed. The tests were executed on the Cumulus cluster [5] consisting of 36 homogenous IBM PC space-shared Sempron 1.7GHz, 512MB RAM nodes. The results presented in the paper are based on the values from the series of at least five test runs. If not stated differently, the averaged values are presented.

In the experiments based on the fixed problem technique for large and moderate SOM network sizes (48x48 and above) the characteristics of application efficiency closely matched the theoretical predictions. For smaller network sizes the superlinear speedup appeared (fig. 3). The parallel execution efficiency largely exceeding 1 suggests that for the number of nodes equal and larger than 8 the utilization of the processor cache speeds up the memory access operations. In the tested algorithm—the network offline parallelization—with the growing number of utilized processors the network size allocated for one processor gets proportionally smaller. In the algorithm implementation a single neuron is represented by the vector of 617 8-byte long fields. The CPUs utilized have 128 kB L1 and 256 kB L2 cache sizes. An analysis of the network size values gathered in the table 1 shows that for the 24x24 network the allocated memory gets close to the cache size when the number of processors reaches 8. For 48x48 network this value is present for 32 processors (thus the final efficiency raise). When 96x96 and larger networks are used, even for the largest tested number of processors, the network size is too big to cause the cache utilization related efficiency growth.

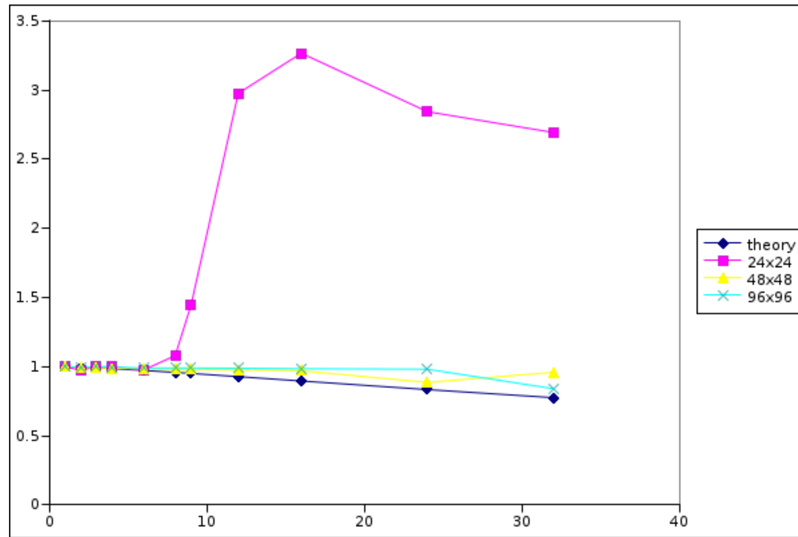

Fig. 3. Efficiency of the parallel SOM algorithm

Table 1. Network size in kB per processor

No. of proc.	Net. size 24x24	Net. size 48x48	Net. size 96x96
1	2776	11106	44424
8	347	1388	5553
32	87	347	1388

5.1 Detailed Execution Efficiency Analysis

During the execution of the parallel SOM algorithm for small network sizes the presence of processor cache utilization based anomalies, shown on the fig. 3, seem to be inevitable. To analyze the impact of these anomalies on the program execution time, the fixed size problem tests were performed for network sizes of 12x12 and 48x24 (fig. 4).

As expected, if smaller network is used the network data size allocated on a single processor is smaller and the “cache effect” appears earlier.

To analyze the impact of the data size on the efficiency, the same experimental results were presented in the more appropriate form (fig. 5). It can be seen that the efficiency for all the network sizes is clearly and in the same way dependant on the data size allocated on a single processor. Analyzing the shape of the curves, three parts can be distinguished. In the first area (0-256 kB) the network data can be fully or almost fully placed in the cache. In this area with the growth of the data size the efficiency growth can be observed. In the second area (256-350 kB) with the data size growth the more data have to be accessed from the main memory and with the growth in the data size efficiency gets lower. Finally in the third area (350+ kB) the data size is too large for the cache utilization to largely influence the execution time and the efficiency is approximately independent on the data size.

5.2 Scalable Problem Size Analyzis

The parallel SOM implementation efficiency shows different characteristics in the three data size areas. To further analyze the program behavior, the scalable problem size technique was employed. The computational throughput, offering the most convenient and descriptive method of the results presentation, was selected as the metric analyzed in the further considerations.

The throughput was measured in kilobytes of the network data processed during one second by a single processor. The tests executed for local data sizes from 19kB to 1234 kB were divided into three parts, representing the three data size areas.

In the first area where the data can be placed in the cache (fig. 6) the throughput clearly depend on the data size and raises with the network size growth (i.e. raises for larger data sizes). This hardware

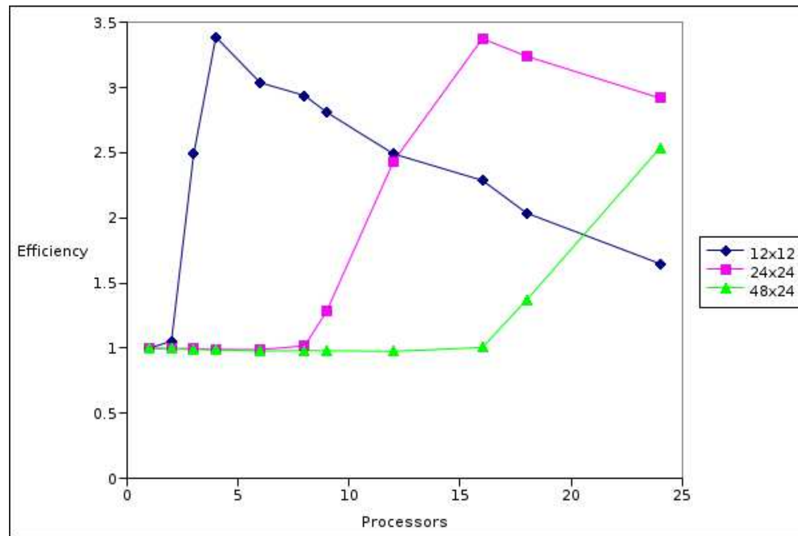

Fig. 4. Execution efficiency in relation to the number of processors

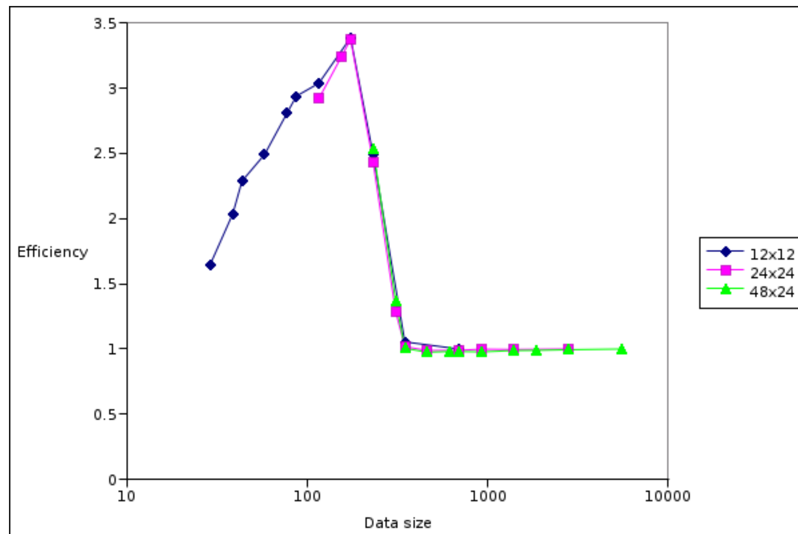

Fig. 5. Execution efficiency in relation to the data size allocated on a single processor

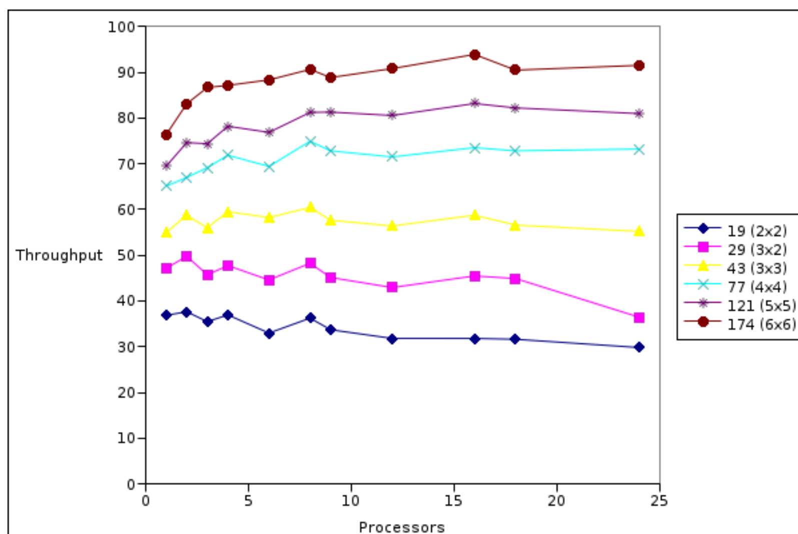

Fig. 6. Throughput in the cache area

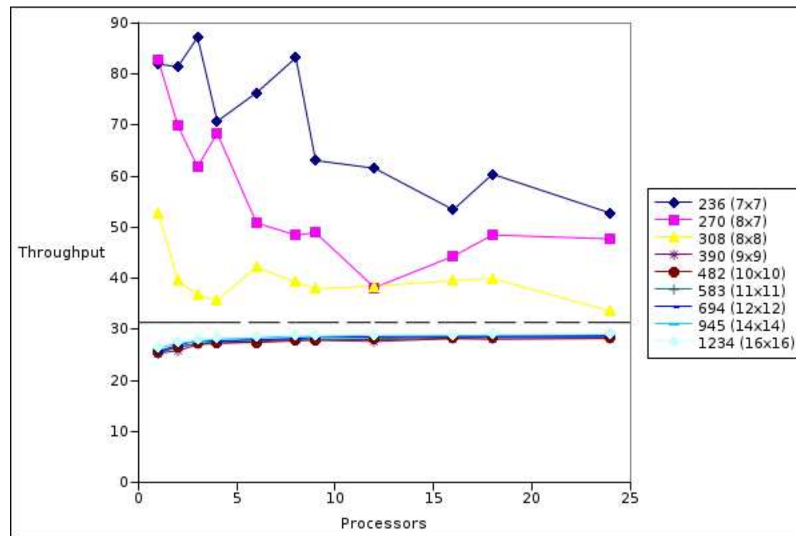

Fig. 7. Throughput at the cache size limits and memory area

dependant anomaly is caused by better cache space utilization, resulting in a higher cache hit ratio and the lower overall memory access time.

In the second data size area, where the data size starts exceeding the cache size (the upper three curves on fig. 7), the throughput gets lower with the data size growth. This behavior is natural since with the data growth the amount of data that must be accessed from the memory raises, resulting in growing number of cache misses. The upper three curves on the figure 7 present the results of a single test run. In this area the software environment behavior can change the program cache utilization efficiency, resulting in the execution environment dependant anomalies. When the results are averaged over the series of experiments, these anomalies are indistinguishable.

The third data size area, representing the data sizes much larger than the cache size, shows much more consistent throughput behavior (the lower six curves on fig. 7). When the detailed behavior is analyzed (fig. 8) it can be seen that like in the cache area the data size growth results in the throughput growth. Although the source of this effect is the same as in the cache area its signifficance is much smaller here because for larger data sizes much smaller is also the influence of the cache utilization. In the memory utilization area, as shown in the analysis presented in section 4, the larger number of processors the smaller is the sum of communication and modification time. This algorithm dependant anomaly constitutes the source of the initial throughput growth when the larger number of processors is utilized.

6 Conclusions and Future Work

In the paper three types of execution anomalies—hardware, algorithm and execution environments dependant, were identified in the parallel SOM algorithm. Utilizing the scalable problem size technique it was possible to describe the dependence between the input data size and the unexpected algorithm behavior. This technique proved also to be a valuable tool that can be utilized to select the correct execution parameters to avoid the hardware dependant anomalies. The limitations of the paper size permitted to present the discussion based only on a single algorithm. In the future works the scalable problem size technique utilization to the analysis of broader class of algorithms, specifically the ones with a nonlinear complexity, together with the application of this technique into the granularity based evaluation [6] and performance prediction methods will be presented.

Acknowklegements. This research was partially supported by by the European Community Framework Programme 6 project DeDiSys, contract No 004152.

References

1. Foster I.: *Designing and Building Parallel Programs,* Addison-Wesley Pub., 1995 (also available at `http://www.mcs.anl.gov/dbpp/text/book.html`).

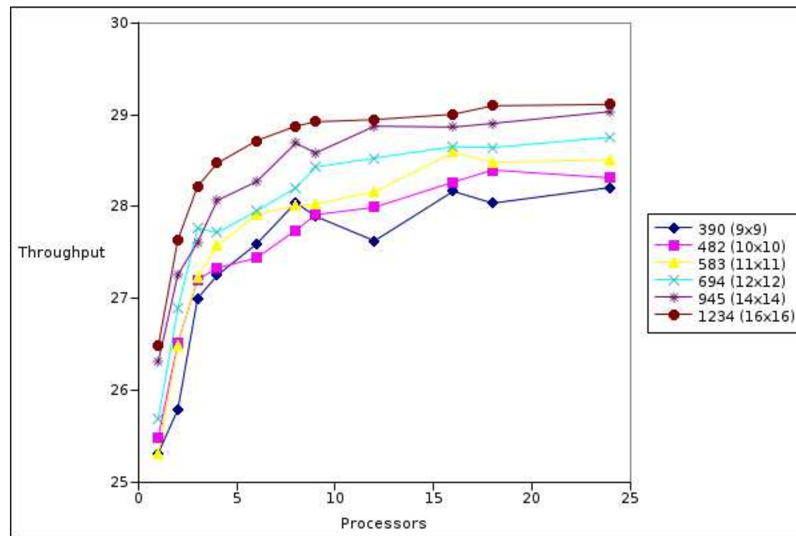

Fig. 8. Throughput in the memory area (details)

2. Grama A., Gupta A., Kumar V.: *Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures,* IEEE Parallel & Distributed Technology, August 1993, pp. 12–21.
3. Kohonen, T.: *The Self-Organizing Map,* Proceedings of IEEE, 1985, vol. 73 pp. 1551–1558.
4. Kwiatkowski J., Pawlik M, Konieczny.D., Markowska-Kaczmar U.: *Performance Evaluation of Different Kohenen Network Parallelization Techniques,* To be published in Proceedings of Parelec'06 Intl. Conf.
5. Kwiatkowski J., Pawlik M, Wyrzykowski R., Karczewski K.: *Cumulus—Dynamic Cluster Available under CLUSTERIX,* Proc. of Cracow Grid Workshop 2005, Cracow 2006, pp. 82-87.
6. Kwiatkowski J.: *Evaluation of Parallel Programs by Measurement of its Granularity,* Proceedings of PPAM'01 International Conference Naleczow, Poland, September 9–12 2001, LNCS, Springer-Verlag Berlin Heidelberg 2002, pp. 145–153.
7. Lawrence R., Almasi G. S., and Rushmeier H. E.: *A scalable parallel algorithm for self organizing maps with applications to sparse data mining problems,* 1999, Data Mining and Knowledge Discovery, vol. III, pp. 171–95.
8. Sahni S., Thanvantri V.: *Performance Metrics: Keeping the Focus on Runtime,* IEEE Parallel & Distributed Technology, spring 1996, pp. 43–56.