DOI: 10.17951/ai.2016.16.1.50

ANNALES
UNIVERSITATIS MARIAE CURIE-SKLODOWSKA
LUBLIN – POLONIA

VOL. XVI, 1 SECTIO AI 2016

A LOW-COST MULTICOMPUTER FOR SOLVING THE RCPSP

Grzegorz Pawiński [1], Krzysztof Sapiecha [1]

[1] Department of Computer Science,
Kielce University of Technology

KEY WORDS: RCPSP, multicomputer, distributed processing model

ABSTRACT: In the paper it is shown that time necessary to solve the NP-hard Resource-Constrained Project Scheduling Problem (RCPSP) could be considerably reduced using a low-cost multicomputer. We consider an extension of the problem when resources are only partially available and a deadline is given but the cost of the project should be minimized. In such a case finding an acceptable solution (optimal or even semi-optimal) is computationally very hard. To reduce this complexity a distributed processing model of a metaheuristic algorithm, previously adapted by us for working with human resources and the CCPM method, was developed. Then, a new implementation of the model on a low-cost multicomputer built from PCs connected through a local network was designed and compared with regular implementation of the model on a cluster. Furthermore, to examine communication costs, an implementation of the model on a single multi-core PC was tested, too. The comparative studies proved that the implementation is as efficient as on more expensive cluster. Moreover, it has balanced load and scales well.

1. INTRODUCTION

Resource allocation, called the Resource-Constrained Project Scheduling Problem (RCPSP), attempts to reschedule project tasks efficiently using limited renewable resources minimising the maximal completion time of all activities [3 - 5]. A single project consists of *m* tasks which are precedence-related by finish-start relationships with zero time lags. The relationship means that all predecessors have to be finished before a task can be started. To be processed, each task requires a human resource *(HR)*. The resources are limited to one unit and therefore have to perform different tasks sequentially. RCPSP is an NP- hard problem. In most cases, branch-and-bound is the only exact method which allows the generation of optimal solutions for scheduling rather small projects (usually containing less than 60 tasks and not highly constrained) within acceptable computational effort [1, 5]. Results of the Hartmann and Kolisch [8] investigation showed that the best performing heuristics were the GA of Hartmann [7] and the SA procedure of Bouleimen and Lecocq [2]. Their latest research revealed that the forward-backward improvement technique applied to X-pass methods, metaheuristics or other approaches produces good results and that the most popular metaheuristics were GAs and TS methods.

In our previous works, cost-efficient project management based on a critical chain (CCPM) was investigated. The CCPM is one of the newest scheduling techniques [19]. It was used to solve a variant of the RCPSP. A goal of the management was to allocate resources in order to minimise the project total cost and complete it in a given time. A sequential metaheuristic from Deniziak [6] was adapted to take into account specific features of human resources participating in a project schedule. The research showed high efficiency of this adaptation for resource allocation [12]. An extension of the problem, where *HRs* are only partially available since they may be involved in many projects, was also investigated [14]. The research proved that the adaptation is efficient but the minimization was still time consuming and would require accelerating to cope with bigger real-life problems

Our latest research showed that the algorithm has got an inherent parallelism. Hence, a distributed processing model for solving the extension of the RCPSP was developed and tested on a regular PCs [13]. It gave a time of scheduling even 10 times smaller than the sequential processing. Therefore, in this research we present a new implementation of the model, on a low-cost multicomputer built from PCs connected through a local network Furthermore, we compare it with regular implementation of the model on a cluster and show that it may be just as efficient, but not so expensive what might limit its practical value.

The next section of the paper contains a brief overview of related work. Motivation for the research is given in section 3. An implementation of the distributed processing model for the algorithm is presented in section 4. Evaluation of the implementation in both distributed and parallel environments is given in section 5. The paper ends with conclusions.

2. RELATED WORK

Researchers studied the problem and suggested their own solutions which can be divided into exact procedures and heuristics. Branch and bound methods are an example of the exact procedures (see e.g. [3], [4]). In [11] another method, a tree search algorithm, was presented. It is based on a new mathematical formulation that uses lower bounds and dominance criteria. An in-depth study of the performance of the latest RCPSP heuristics can be found in [10]. Heuristics described by the authors include X-pass methods, also known as priority rule-based heuristics, classical metaheuristics, such as Genetic Algorithms (GAs), Tabu search (TS), Simulated annealing (SA), and Ant Colony Optimisation (ACO). Non-standard metaheuristics and other methods were presented as well. The former consist of local search and population-based approaches, which have been proposed to solve the RCPSP. The authors investigated a heuristic which applies forward-backward and backward-forward improvement passes. For detailed description of the heuristic schedule generation schemes, priority rules, and representations refer to [8].

The effectiveness of scheduling methods can be further improved using parallel processing. Some implementations of parallel TS [15 17] and SA [18] algorithms for different combinatorial problems have already been proposed. The most common one is based on dividing (partitioning) the problem such that several partitions could be run in

parallel and then merged. Parallelism in GAs can be achieved at the level of single individuals, the fitness functions or independent runs [21, 22]. All of the parallel approaches fall into three categories: the first uses a global model, the second uses a coarse-grained (island) model and the third uses a fine-grained (grid, cellular) model [20]. In the global model, a master process manages the whole population by assigning subsets of individuals to slave processes. In the island model a population is divided into sub-populations that are evolved separately. During evolution, some individuals are exchanged periodically between them. In the grid model a population is represented as a network of interconnected individuals where only neighbors may interact. It was observed that parallel GAs (PGAs) usually provide better efficiency than sequential ones [20]. The same parallel approaches can be applied for ACO. In [23] five strategies of parallel processing are described, which are mainly based on the well-known master/slave approach [24].

3. MOTIVATION

The sequential algorithms are time consuming, what considerably limits their usefulness. Speeding up the calculations would be desirable for project managers because it may allow managing complex projects in acceptable time. Parallel models offer the advantage of reducing the execution time and give an opportunity to solve new problems which have been unreachable in case of sequential models. The most popular parallel strategies are based on master/slave approach [24] with centralized management of distributing tasks and gathering results. The master can efficiently coordinate the system, avoiding potential conflicts before they take place, and react on failures of the slaves. However, global gathering and re-broadcasting of large configurations can be time-consuming. Costs of synchronization between slaves have to be considered, also. Some slaves may have to wait for completing other tasks, which is necessary to retain data integrity. More-over, the master is the weakest point of the system. The system will slow down if the master cannot handle incoming requests. If the master crashes, the whole system will also crash. Another problem is load imbalance caused by unpredictable processing time of each slave. Summarizing, the gain coming from parallelization of the algorithm may be significantly reduced.

From our research it also follows that parallel processing could reduce efficiently the amount of the time consumed by the metaheuristic algorithm [13]. Usually, such reduction requires a use of a cluster and hence is expensive what may limit its popularity. The key idea to overcome this inconvenience is to make use of multi-core architecture of low-cost PCs, instead of the cluster. Such a multi-multi computer is cheap, easily assembled and might be very useful for practical reasons. However, it should be proven that the implementation is as efficient as on the cluster, and that it has balanced load and scales well.

4. OPTIMIZATION ALGORITHM

The metaheuristic algorithm starts with the initial point and searches for the cheapest solution satisfying given time constraints. The initial schedule is generated by greedy procedures that try to find a resource for each task basing upon to the smallest increase of the project duration or the project total cost. It is a suboptimal solution which the algorithm tries to enhance. In each pass of the iterative process, the current project

schedule is being modified in order to get closer to the optimum. In the first add stage a new HR which is not in the schedule is attached to it. Tasks of HRs which have already been engaged in the schedule are moved to the HR but only when a positive gain is achieved. Afterwards, if there are HRs without allocated tasks, they are removed from the schedule. The best schedule goes to the next stage

and the proceeding is repeated until no more free HRs are available. In the second rem stage all tasks allocated to the HR are moved onto other HRs, still remaining in the schedule, but only when a positive gain is achieved. Then again, HRs without allocated tasks are removed from the schedule. Finally, the best project schedule coming from all stages is chosen. The iterative process is repeated for every resource from the resource library until no improvement can be found. At the very end, project tasks may be shifted right to the latest feasible position into their forward free slack by means of As Late As Possible (ALAP) schedule.

4.1. Distributed processing model

The distributed processing model is shown in Figure 1.

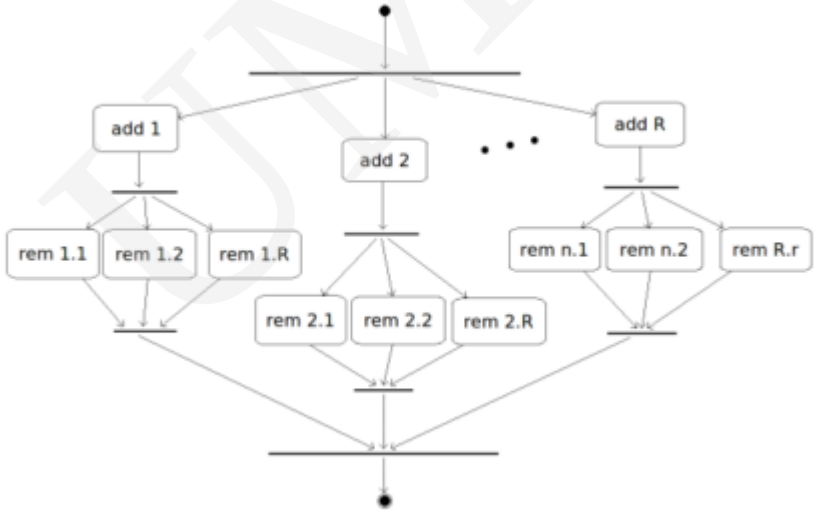

Figure 1 Distributed processing model

In general, there are $R \cdot (1 + R_r)$ schedule modifications that have to be calculated, where R is the number of *HRs* and R_r is the number of *HRs* that have left after particular *add* stage. However, not all of them can be performed at the same time. At the beginning, only R attempts to add a new *HR* to the schedule may be calculated. Each of the add stages could be performed simultaneously. Afterwards, if any of them is finished, R_r attempts in the *rem* stage may be started. The attempts to move all tasks from each of *HRs* may also be calculated separately. Thus, the maximal number of simultaneous modifications is $R \cdot R_r$, when all the add stages finish at the same time. The process iteration ends after finishing all of the second stages.

4.2. Implementation of the model

The distributed processing model (Figure 2) was implemented in Java. One application, which is a *tasks dispatcher (D)*, manages a pool of threads responsible for communication with other *worker* applications, located on remote computers.

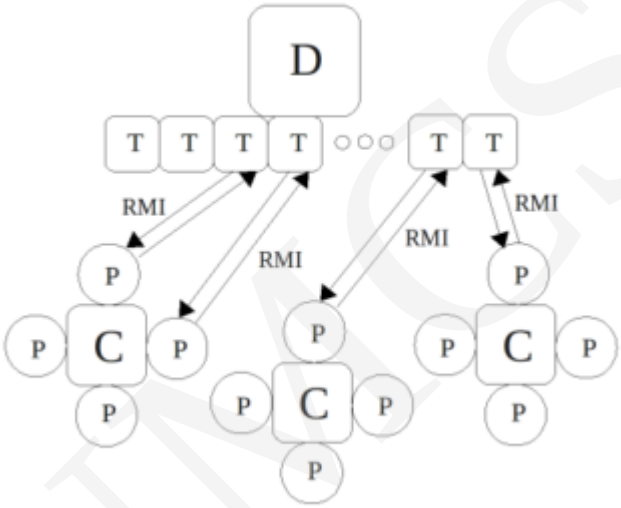

Figure 2 Implementation of the distributed processing model

(D - tasks dispatcher, T - thread, C - remote computer, P - process, RMI - remote method invocation)

At the beginning, *workers* notify the *dispatcher* about their readiness to execute tasks. The tasks dispatcher creates a new thread for each worker and joins it to the pool. The pool contains as many threads as needed, but will reuse previously constructed threads when they are available. On the remote computers, *workers* run as independent processes, what makes them available for direct communication. Therefore, the *tasks dispatcher* may uniformly split the computational tasks, so as to workload could easily be balanced. Each remote computer runs as many processes as the number of processor cores, in order to use the whole computing power of multi-core machines. During executing an iteration of the algorithm, the *tasks dispatcher* sends schedule modification requests to the first free worker. To this end, it uses Remote Method Invocation (RMI) for communication. If a *worker* is not responding, it will be removed from the pool and the request will be sent to another free *worker*. *Workers* receive project data and the searching parameters so as to invoke a method, in order to perform the *add* or the *rem* stage. Afterwards, results of modifications are sent back to the *dispatcher* and then the thread can be reused. Synchronization occurs at the end of each of the iterations because all the *rem* stages have to be finished in order to choose the best schedule.

5. COMPARATIVE STUDIES

The efficiency of the algorithm described in the paper was estimated on 100 randomly generated project plans containing from 30 to 60 tasks, and from 8 to 16 HRs with random data. Each project plan was scheduled several times and results were averaged. Tasks in the project plan may have at most 4 precedence relationships with probability 0,35. They can be easily scheduled because they have few predecessors or none. If the probability of inserting the precedence relationships were lower, the project plan would contain mostly unconnected tasks. On the other hand, tasks with two or more predecessors significantly decrease the search space. In each project, resource availability was reduced by allocating 30 tasks from PSPLIB, developed by Kolisch and Sprecher [9]. The set with 30 non-dummy activities currently is the hardest standard set of RCPSP-instances for which all optimal solutions are known [4]. However, we considered an extension of RCPSP where resources have already got their own schedule and a cost of the project, but not the project duration, should be minimized. So even though we take the project instances from PSPLIB, the results cannot be compared. The initial schedule was generated by two greedy procedures mentioned at the beginning of section 4.

Implementation of the distributed model was run on two distributed systems:

- multicomputer built from PCs ($Cluster_{PCs}$) that comprises 10 multi-core computers with Intel Core i5-760 Processor (8M Cache, 2,80 GHz) and 2 GB of RAM memory, connected via a Gigabit Ethernet TCP/IP local network,
- regular *cluster* that comprises 1 head node with Intel Xeon E5410@2,33GHz, 16GB of RAM memory and 10 processing nodes with Intel Xeon E5205@1,86GHz, 6GB of RAM memory, connected via a Gigabit Ethernet TCP/IP local network.

Furthermore, to examine communication costs, an implementation of the model on a single multi-core PC was tested, too.

5.1. Tests which examine implementation of the model in distributed environments

The algorithm scalability depends on the number of *HRs* because it is related to the number of schedule modifications. The number of independent requests, and consequently the need for *workers*, increases along with the increase of the number of *HRs*. Influence of changing the number of *workers* on the computation time towards the number of tasks is shown in Figure 3. In both distributed environments, the computation time significantly falls as the number of *workers* grows. Decline is particularly visible when only few *workers* are used. Finally, the computation time exceeds its minimum, no matter how many *workers* is used. In both environments, also the increase of the number of tasks influences the drop of the scheduling time. However, the cluster, despite slower CPUs, copes better along with the increase of the number of tasks. In the cluster, the growth of the scheduling time in more complex projects is slower, especially when only few workers are used. In general, a reduction of the computation time looks similar in both environments. It is worth noticing that, the computation time was reduced even to 6% of sequential computation time for the project with 60 tasks and 12 HRs (Figure 3b, left column).

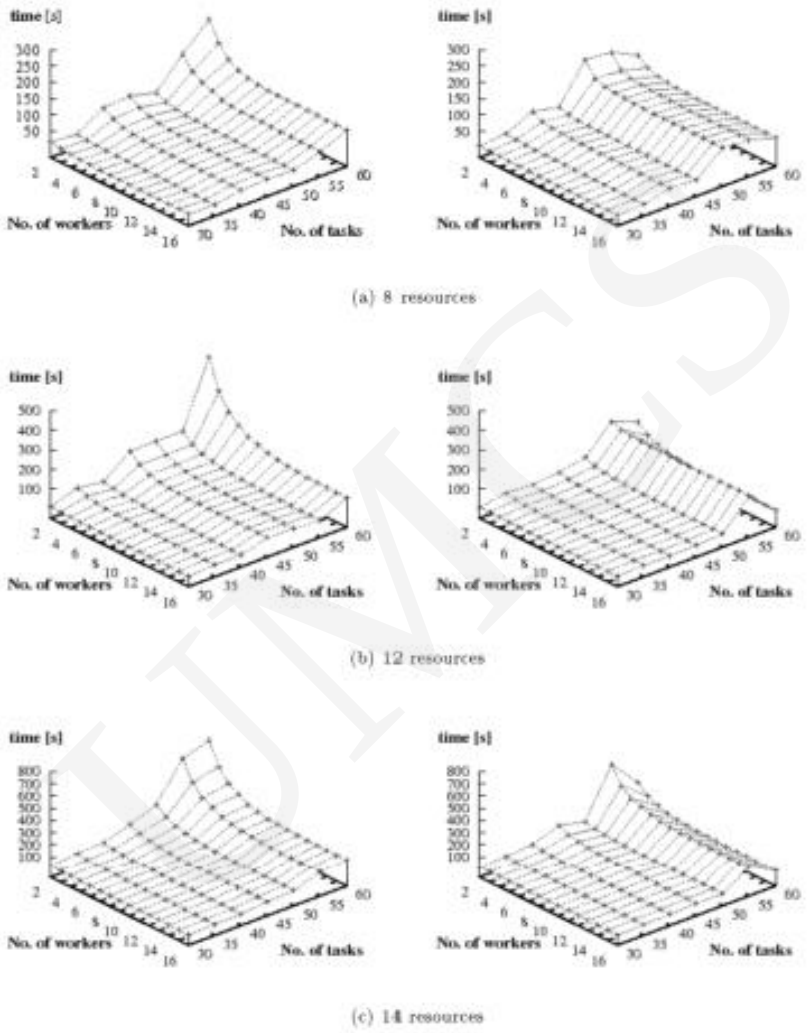

Figure 3 Computation times compared with the number of workers for constant number of *HRs* (left column – *Cluster*$_{PCs}$, right column – the*Cluster*)

A CPU usage in *Cluste*$_{rPCs}$ during scheduling of a project with 35 tasks and 16 *HRs* was examined (Figure 4). The CPU usage was monitored every 50 ms and the reads were averaged at the end of calculations. More frequent reads could influence the processor load. The number of *HRs* was chosen so that enough simultaneous attempts were provided to make workers busy. PCs were running 4 *workers* each (one *worker* was assigned to every core).

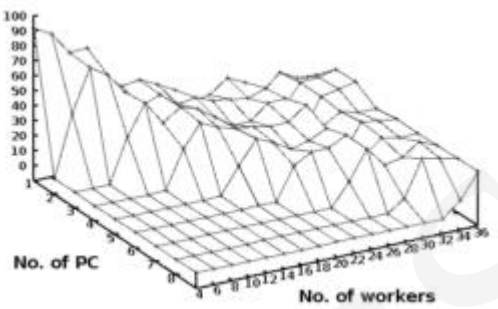

Figure 4 CPU usage in $Cluster_{PCs}$ [%]

Figure 4 illustrates how the schedule modification requests spread over the available PCs. CPU usage on PC #1 is almost 100% but only when 4 *workers* are used. If the number of *workers* increases, the load is balanced by the use of the other PCs. The distributed algorithm scales well because the computational tasks may be uniformly splitted among *workers*. Summing up the cores usage (counted in 100%), it grows from 3,7 cores for 4 *workers* to 9,48 cores for 36 *workers*. The total core usage together with the tasks dispatcher was 10,02. Hence, the scheduling time was reduced 10 times by the use of 40 cores on 10 PCs.

5.2. Tests which examine the in uence of the communication cost on algorithm performance

Distributed tests were executed in order to examine how the network latency influences the algorithm performance. To that end, 4 *workers* were run on the *ClusterPCs* that comprises 2 multi-core PCs and compared with 4 *workers* on 2 processing nodes in the cluster and 4 workers on a single PC (so called *LocalPC*). All workers were using RMI for communication. At first, the number of modification requests was counted with respect to the number of resources and the number of tasks (Table 1).

Table 1 The number of modification requests

	No. task		
No. resources	30	35	40
10	634	755	480
12	765	930	869
14	1009	694	1492
16	1412	1412	1564

The number of requests increases as the number of resources increases and varies along with the increase of the number of tasks. However, the more requests are sent, the greater will be the impact of communication cost on the performance. The average scheduling time for a project with 30 tasks is shown in Table 2.

Table 2 Average time of transferring data between the tasks dispatcher and workers for a project with 30 tasks [ms] (Remote - workers located on 2 remote computers, Local - workers located on the same machine, Res_{num} - No. resources).

	No. tasks											
Res_{num}	*cluster*			$Cluster_{PCs}$			$Local_{PC}$			*Threads*		
	2	3	4	2	3	4	2	3	4	2	3	4
10	5587	3922	2949	3961	2603	1846	3355	2261	1963	2058	1350	996
12	7242	4825	3827	4660	3191	2300	4016	2876	2494	2419	1684	1311
14	9677	6427	5137	6187	4213	3042	5190	3718	3260	3128	2163	1671
16	12911	8548	6555	7745	5311	4010	6730	4787	4173	4371	3016	2360

It is clear that the scheduling time decreases when the number of workers grows. Yet, the decline is very low between 3 and 4 workers in the $Local_{PC}$ because computer resources start to be overloaded when 4 workers and the tasks dispatcher run on the same machine. On average, the $Local_{PC}$ is about 13% faster than the corresponding $Cluster_{PCs}$ (for less than 4 *workers*), due to low communication costs. On the other hand the $Cluster_{PCs}$ is better when the number of *workers*exceeds the number of processor cores. It is also not limited to the number of workers. But even the usage of 4 workers reduced the scheduling time by 54% in the ClusterPCs and by 48% in the cluster, in the project with 30 tasks and 10 HRs. However, the reduction ratio in the former decreases along with the increasing number of resources and does not change in the latter. It means that the cluster copes better than PCs also with the increase of the number of resources.

The average time of transferring data between the tasks dispatcher and 3 workers is shown in Table 3. It increases when the number of tasks increases because more data needs to be transferred. It also increases when the number of resources increases due to increased number of requests that the tasks dispatcher has to handle.

Table 3 Average time of transferring data between the tasks dispatcher and workers for a project with 30 tasks [ms] (*Remote* - workers located on 2 remote computers, *Local* - workers located on the same machine, Re_{snum} - No. resources).

	No. tasks											
Res_{num}	*cluster*			$Cluster_{PCs}$			$Local_{PC}$			*Threads*		
	30	35	40	30	35	40	30	35	40	30	35	40
10	5,62	6,41	6,22	5,84	6,29	6,87	3,33	3,36	3,72	0,24	0,48	0,5
12	5,62	6,41	6,22	5,96	6,76	7,23	3,34	3,63	3,89	0,2	0,25	0,44
14	5,66	5,66	6,29	6,06	7,03	7,48	3,38	3,78	4,03	0,14	0,29	0,37
16	5,77	5,72	6,31	6,49	6,73	7,33	3,49	3,8	4,13	0,24	0,33	0,48

Yet, the increase of the time is much faster in the ClusterPCs, than in the cluster. Consequently, the data transfer in the ClusterPCs gets slower in the projects with more than 35 tasks and 10 HRs. On average, the data transfer is about 2,2 times slower in the ClusterPCs than within a single multi-core PC. On a single machine, it may be further reduced to less than 0,5 ms by the use of threads instead of processes in LocalPC (so called Threads). Threads are much lighter than processes and share the process'

resources. Thus, even if only one multi-core machine is available, the scheduling time with the use of 4 workers may be reduced by about 47%. The scheduling time on a single machine with the use of 4 threads is relevant to the scheduling in ClusterPCs on 2 multi-core PCs with 4 workers on each. But still, if the need for workers is greater, the ClusterPCs is better. Moreover, running more threads than 5 on a 4-core processor is not so efficient. Comparison results of time needed to transfer data between the tasks dispatcher and 3 workers, averaged from all attempts, are shown on Figure 5.

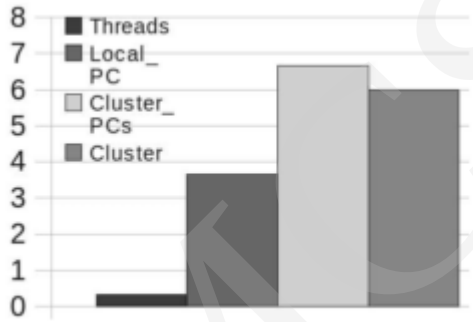

Figure 5 Comparison results of time needed to transferring data between the tasks dispatcher and 3 workers averaged from all attempts [ms]

6. CONCLUSIONS

In the research, a distributed model was used in order to reduce the computation time for a solution of the RCPSP when resources are partially available. An implementation of the model on a multicomputer built from PCs was tested and compared with regular implementation of the model on a *cluster*. The tasks dispatcher and *workers* were connected through a local network and were using RMI for communication. The *tasks dispatcher* was using multithreading for spreading and gathering data while, at the same time, *workers* were calculating different schedule modifications and sending back the results. The *workers* were run on remote computers as independent processes and hence did not have to be synchronized. *Workers* were gathered in a pool managed by the *tasks dispatcher* and were available for a direct use. The best efficiency was obtained when there were as many processes running as the number of computer cores. Hence, the more cores inside the computer, the more *workers* can run on it and fewer PCs are needed. Consequently, the more *workers* the shorter the computation time, but only when there is enough work to do for the *workers*. Too few workers cannot handle rapidly growing calculation requests after the first stage of the algorithm. The maximum number of workers depends on the number of *HRs* because it is related to the number of schedule modifications Thus, the project scheduling cannot be speed up if there is a lot of resources and not enough workers and vice versa.

The research showed that the multicomputer built from multi-core PCs may be successfully used for reduction of the scheduling time. Obtained results are comparable with the cluster. In both environments the reduction of time looks similar. However, the cluster copes better along with the increase of the number of tasks and the number of resources. In the cluster the communication cost is lower than in the $Cluster_{PCs}$, in the

projects with more than 35 tasks and 10 *HRs*. On a single machine, the scheduling time is about 13%, faster than through a local network (for less than 4 *workers*) due to lack of the network latency. It can be further reduced by about 47% by the use of threads instead of processes. However, the computer resources start to be overloaded when the tasks dispatcher and more than 3 processes or more than 5 threads run on the same 4-core processor. Therefore, the $Cluster_{PCs}$ outperforms the $Local_{PC}$ when more than 3 workers and the usage of threads when more than 7 *workers* are used.

The experimental results showed that the distributed model is well-balanced. The computational tasks are uniformly splitted among *workers*. If the number of *workers* increases, the load spreads over the available PCs. The distributed algorithm scales well, adjusting to the number of workers. Moreover, if any of the *workers* crashes, its task will be taken over by another worker and the proceeding will be continued. Various complexities of the projects were tested. However in each, the scheduling time was significantly reduced by the distributed calculations, even up to 6% of sequential time. In comparison to the sequential computing, the number of used cores (counted in 100%) was 10 times higher, during scheduling of a project with 30 tasks and 16 *HRs* by 36 *workers*.

LITERATURE

[1] Alcaraz J., Maroto C., *A robust genetic algorithm for resource allocation in project scheduling*, Annals of Operations Research 102 (2001): 83 109.
[2] Bouleimen K., Lecocq H., *A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple modes version*, European Journal of Operational Research 149 (2003): 268 28,
[3] Brucker, P., Knust, S., Schoo, A., Thiele, O., *A branch-and-bound algorithm for the resource-constrained project scheduling problem*, European Journal of Operational Research 107 (1998):272 - 288.
[4] Demeulemeester, E. L., Herroelen, W. S., *New benchmark results for the resource-constrained project scheduling problem*, Management Science 43 (1997): 1485 - 1492.
[5] Demeulemeester, E. L., Herroelen, W. S., *Project Scheduling*. A Research Handbook, Springer (2002).
[6] Deniziak, S., *Cost-efficient synthesis of multiprocessor heterogeneous systems*, Control and Cybernetics 33 (2004): 341 355.
[7] Hartmann, S., *A Competitive Genetic Algorithm for Resource Constrained Project Scheduling*, Naval Research Logistics 45 (1998): 733 750.
[8] Hartmann, S., Kolisch, R., *Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem*, European Journal of Operational Research 127 (2000): 394 407
[9] Kolish R., Sprecher A., *PSPLIB - A project scheduling library*, European Journal of Operational Research 96 1996: 205-216.
[10] Kolisch R., Hartmann S., *Experimental investigation of heuristics for resource-constrained project scheduling: An update*, European Journal of Operational Research 174 (2006): 23 - 37.
[11] Mingozzi, A., Maniezzo, V., Ricciardelli, S., Bianco, L., *An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation*, Management Science 44 (1998): 714 - 729.

[12] Pawiński G., Sapiecha K., *Resource allocation optimization in Critical Chain Method*, Annales Universitatis Mariae Curie-Sklodowska sectio Informaticales 12(1) (2012): 17 - 29.

[13] Pawiński G., Sapiecha K., *Cost-efficient Project Management Based on Distributed Processing Model*, Proceedings of the 21th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society (2013): 157 - 163.

[14] Pawiński G., Sapiecha K., *Cost-efficient project management based on critical chain method with partial availability of resources*, Control and Cybernetics 43 (2014): 95 109.

[15] Niar, S., Freville, A., *A parallel tabu search algorithm for the 0-1 multidimensional knapsack problem*, Proceedings of the 11th International Parallel Processing Symposium (1997): 512- 516.

[16] Randall, M. Abramson, D., *A General Parallel Tabu Search Algorithm for Combinatorial Optimization Problems*, Proceedings of the 6th Australasian Conference on Parallel and Real Time Systems, Cheng, W. and Sajeev, A. (eds), Springer-Verlag (1999): 68-79.

[17] He,Y., Qiu, Y., Liu, G. Lei, K., *A parallel adaptive tabu search approach for traveling salesman problems*, Proceedings of 2005 IEEE International Conference on Natural Language Processing and Knowledge Engineering (2005): 796 801.

[18] Malek M., Guruswamy M., Pandya M., Owens H., *Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem*, Annals of Operations Research 21 (1989): 59 84.

[19] Steyn, H., *An investigation into the fundamentals of critical chain project management*, International Journal of Project Management 19 (2000): 363-369.

[20] Tomassini M., *Parallel and distributed evolutionary algorithms: A review*, In P. Neittaanmki K. Miettinen, M. Mkel and J. Periaux, editors, Evolutionary Algorithms in Engineering and Computer Science, J. Wiley and Sons, Chichester (1999): 113 133.

[21] Koza J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, Sixth printing (1998).

[22] Koza J.R., Keane M.A., Streeter M.J., Mydlowec W., Yu J.,Lanza G., *Gentic Programming IV*, Kluwer Academic Publishers (2003).

[23] Randall M., Lewis A., *A Parallel Implementation of Ant Colony Optimization*, Journal of Parallel and Distributed Computing 62(9) (2002): 1421 1432.

[24] Foster, I., *Designing and building parallel programs: concepts and tools for parallel software engineering*, Addison Wesley: Reading, MA (1995).