On Optimum Multi-installment Divisible Load Processing in Heterogeneous Distributed Systems

Maciej Drozdowski[1,⋆] and Marcin Lawenda[2]

[1] Institute of Computing Science, Poznań University of Technology,
ul.Piotrowo 3A, 60-965 Poznań, Poland
Maciej.Drozdowski@cs.put.poznan.pl
[2] Poznań Supercomputing and Networking Center,
ul.Noskowskiego 10, 61-704 Poznań, Poland
Marcin.Lawenda@man.poznan.pl

Abstract. In this paper we study multi-installment divisible load processing in heterogeneous distributed systems. Divisible loads are computations which can be divided into parts of arbitrary sizes, and these parts can be processed independently in parallel. In order to reduce the waiting time during the parallel computation initialization phase, load is sent to the processors in multiple small installments. In a heterogeneous system the sizes of the installments should be adjusted to the communication, and computation capabilities of the processors. We propose two algorithms that gear the load chunk sizes to different communication and computation speeds. The first one is an optimization branch and bound algorithm. The second algorithm is based on genetic search. The running times of both methods and the quality of the genetic algorithm solutions are compared. Then, we use these algorithms to analyze features of the scheduling problem solutions.

Keywords: scheduling and load balancing, divisible load, multi-installment processing, heterogeneous systems, optimization algorithms.

1 Introduction

Divisible loads are computations which can be divided into parts of arbitrary sizes, and these parts can be processed independently in parallel. This means that the grain of parallelism is small, and there are no data dependencies. The sizes of the load parts should be adjusted to the speeds of communication and computation such that processing finishes in the shortest possible time. Examples of real divisible applications include, among others, distributed searching for patterns in text, audio, graphic files, database and measurement processing, data retrieval systems, some linear algebra algorithms, and simulation. Surveys of the divisible load theory can be found in [4, 6, 11].

⋆ This research has been partially supported by the Polish State Committee for Scientific Research.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 231–240, 2005.

Communication delays constitute an important part of the processing time in all distributed algorithms. To reduce the initial waiting for the data, and for initialization of the computations, load is sent in multiple small chunks rather than in a single long message. This way of divisible load distribution and execution is called *multi-installment processing* [3, 6, 8, 12]. In the earlier publications certain assumptions were usually made on the structure of the schedule. For example, messages of equal size were sent to processors in a round-robin fashion [6, 8, 12]. It has been shown [12] that this way of multi-installment processing reduces the length of the schedule in a homogeneous system at most $\frac{e-1}{e}$ times. Unequal load chunk size partitioning has been also proposed [3, 6, 13], but with a tacit assumption that the set of used processors, and their activation sequence are given and fixed. Furthermore, it was assumed that there are no idle times, neither in the communication nor in the computations [3, 4, 6, 13]. However, to our best knowledge, the problem of multi-installment divisible load processing with unequal chunk sizes adjusted to the communication and computation speeds, with selection of the set of exploited processors, and selection of their activation sequence is open. The goals of this paper are twofold: to propose algorithms for the multi-installment divisible load processing including selection of the set and sequence of processors, and to study influence of the system parameters on the quality of the scheduling problem solutions.

The rest of this paper is organized as follows. In Section 2 we formulate the multi-installment divisible load scheduling problem for heterogeneous systems. In Section 3 two algorithms are proposed: an optimization branch-and-bound algorithm, and a heuristic genetic algorithm. The results of computational experiments are presented and discussed in Section 4.

2 Problem Formulation

We will use the word *processor* to denote a processing element with CPU, memory, and communication link. In divisible load model it is classically assumed that initially some volume of load V (e.g. a file with data to be processed) resides on a processor P_0 called *originator*. The originator sends the load to its neighbors for remote processing. Each of the neighbors intercepts some part of the received load, and immediately starts computations related with the received load. The rest of the load is retransmitted to the still inactive neighbors. In this work we assume a star interconnection (a.k.a. a single level tree). In the star network the originator is located in the star center (or the root of the single level tree), and is connected to a set $P_1, \ldots, P_m$ of processors which perform computations. All communications involve the originator, and there are no direct communications between processors $P_1, \ldots, P_m$. For simplicity of presentation we assume that originator is communicating only. Otherwise, the computing ability of the originator can be represented as an additional processor. Each processor P_i is defined by the following parameters: A_i - computing rate (reciprocal of computing speed), C_i - communication rate (reciprocal of bandwidth), S_i - communication startup time. A_i, C_i can be expressed, e.g., in seconds per byte, and S_i can be

expressed in seconds. Computing x units (e.g. bytes) of load on processor P_i takes xA_i units of time. Sending the same amount of load to P_i lasts $S_i + xC_i$. We assume that memory sizes of the processors are sufficiently big and do not influence the construction of the schedule. To simplify the mathematical model we assume that the results returning time is negligible. This simplification is not limiting generality of our considerations because result gathering can be included in the model (see e.g. applications [2, 5, 7, 12]). The computations start only after receiving the whole message with load. We assume that processors have independent communication hardware which allows for simultaneous communication and computations on the previously received load.

To reduce the initial waiting for the load, and for the start of the computations, load is sent to processors in multiple small chunks rather than in a single long message. Let n denote the number of chunks. If the sequence of processors receiving the load chunks is known then our problem can be reduced to a linear program. Let α_i denote size of chunk i. Let $d_i \in \{1, \ldots, m\}$ be the number of the processor receiving chunk i. We will denote by $H_i \subseteq \{i, \ldots, n\}$ the set of chunks sent to processor d_i, starting from chunk i. C_{max} denotes schedule length. Fig. 1 depicts an example schedule with multiple installments. The linear program can be formulated as follows:

minimize C_{max}
on condition that:

$$\sum_{j=1}^{i}(S_{d_j} + \alpha_j C_{d_j}) + A_{d_i} \sum_{j \in H_i} \alpha_j \leq C_{max} \quad i = 1, \ldots, n \tag{1}$$

$$\sum_{i=1}^{n} \alpha_i = V \tag{2}$$

In constraint (1) sum $\sum_{j=1}^{i}(S_{d_j} + \alpha_j C_{d_j})$ expresses communication time for chunks $1, \ldots, i$. $A_{d_i} \sum_{j \in H_i} \alpha_j$ is computation time on processor d_i starting from chunk i. Thus, (1) guarantees that all processors stop computations before the end of the schedule. All work is done by equation (2). Thus, it is possible to find optimum distribution of the load using formulation (1)-(2) if we know the sequence of the processor activation (i.e. values d_i for $i = 1, \ldots, n$).

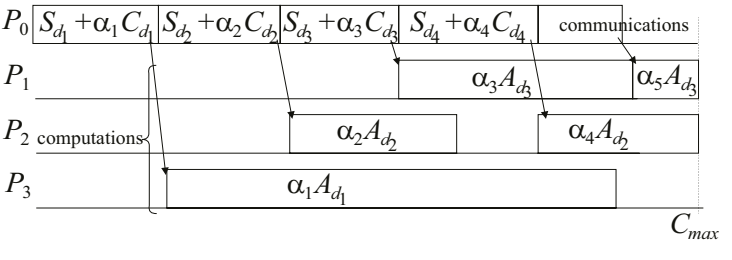

Fig. 1. Example of load distribution pattern.

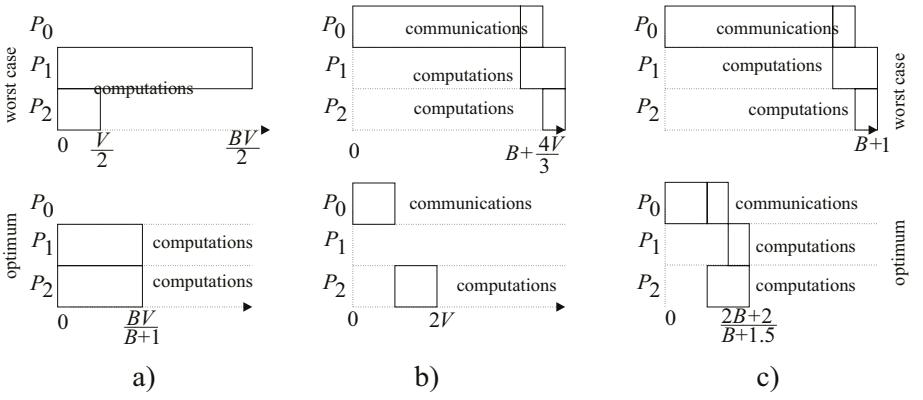

Fig. 2. The worst case examples. a) ignoring heterogeneity, b) ignoring processor set selection, c) ignoring sequencing of the processor activation.

Before proceeding to the further details let us consider *worst cases* that may appear if scheduling decisions ignore certain information. Suppose that we ignore the heterogeneity of the system, and send load parts of equal size to the processors. For instance (Fig. 2a), consider two processors P_1 with parameters $S_1 = 0, C_1 = 0, A_1 = B$, and P_2 with parameters $S_2 = 0, C_2 = 0, A_2 = 1$. We divide the load into two equal chunks of size $\frac{V}{2}$. Resulting schedule has length $\frac{BV}{2}$ but processor P_2 is idle in interval $[\frac{V}{2}, \frac{BV}{2}]$. If we use sizes $\alpha_1 = \frac{V}{B+1}, \alpha_2 = \frac{BV}{B+1}$, then both processors stop computing simultaneously, and schedule length is $\frac{BV}{B+1}$. The ratio of the two schedule lengths is $\frac{B+1}{2}$ which can be arbitrarily big. Hence, in the worst case solutions based on load equipartitioning can be arbitrarily bad in heterogeneous systems.

Suppose that we adjust chunk sizes to the parameters A_i, C_i, but all processors are always used. Let us present another example (Fig. 2b). There are two processors with parameters: $S_1 = B, A_1 = 1, C_1 = 1, S_2 = 0, A_2 = 1, C_2 = 1$. If $V < \frac{B}{2}$ then there is no point in using processor P_1 because load of this size may be processed in a shorter time than the communication activating P_1. If we use P_1 then the schedule has length at least B. If we don't, then schedule has length $V(A_2 + C_2) = 2V$. The ratio of the two lengths is at least $\frac{B}{2V}$ which can be arbitrarily big. Thus, if the set of processors is always the same, the resulting schedule can be arbitrarily bad.

Suppose that we adjust chunk sizes, and select the processors wisely, but we always use the same sequence $(P_1, \ldots, P_m)$ of processor activation. Let us analyze one more instance (Fig. 2c), $m = 2, V = 2, S_1 = 0, C_1 = B, A_1 = 1, S_2 = 0, C_2 = A_2 = 0.5$. If we use sequence (P_1, P_2) of processor activation, then the optimum load distribution is $\alpha_1 = \alpha_2 = 1$, and schedule length is $B + 1$. For sequence (P_2, P_1) the optimum distribution is $\alpha_1 = \frac{1}{B+1.5}, \alpha_2 = \frac{2B+2}{B+1.5}$, and schedule length is $\frac{2B+2}{B+1.5}$. The ratio of the two lengths is $\frac{B+1}{2-\frac{1}{B+1.5}}$ which can be arbitrarily big.

Thus, the subset of processors $P_1, \ldots, P_m$ exploited in the computations and the targets of the communications are unknown, and must be determined. This task has combinatorial nature. In Section 3 we propose algorithms that determine destinations for the load chunks. If one ignores proper selection of the chunk destinations, the problem becomes easier to solve because only linear program (1)-(2) has to be solved for some assumed chunk destinations $d_1, d_2, \ldots, d_n$. Then, the resulting schedules can be arbitrarily bad in the worst case, as demonstrated in the preceding paragraph. How bad the solutions can be on average, if we skip the combinatorial part of the problem, is unknown. We attempt answering this question in Section 4.

3 Optimization Algorithms

3.1 Branch and Bound Algorithm

Two elements constitute a branch-and-bound algorithm. The first is *branching* procedure which divides the solution space into disjoint subsets. These subsets are either eliminated if they do not include the optimum solution, or are further divided until selecting a unique solution. Partition of the solution space can be represented as a tree. Each node is a representative of a set of solutions. Dividing such a set is equivalent to generating successors of a node. In our problem we have to select the sequence of the targets for n load chunks. In the root of the tree the sequence is empty. The first chunk may be sent to one of processors P_i, for $i = 1, \ldots, m$. Therefore, the root has m successors each representing sequences starting with a message sent to processor P_i. The second level of the tree includes two-processor sequences (P_i, P_j). Branching a node representing a leading sequence of l chunk targets consists in appending one more processor to which chunk $l + 1$ will be sent. The branching procedure is continued until constructing a sequence of the assumed length n.

The maximum number of the search tree leaves is m^n. As this number grows exponentially with n, it is necessary to prune the search tree by eliminating nodes representing solutions certainly not better than some already known solution. This procedure is the *bound* element of the algorithm. To determine if a node should be eliminated its *lower bound* of the schedule length is calculated. Suppose the node represents a sequence of l chunks. Thus values $d_1, \ldots, d_l$ are already determined. The remaining $n - l$ chunks still need to be selected. We assume that these $n - l$ chunks are sent to $n - l$ ideal target processors. The ideal target processor has parameters $A^{id} = \min_{i=1}^{m}\{A_i\}, C^{id} = \min_{i=1}^{m}\{C_i\}, S^{id} = \min_{i=1}^{m}\{S_i\}$, and processes only one load chunk. For such a sequence of l real processors, and $n-l$ ideal ones, a linear program (1)-(2) is solved for C_{max} which is the lower bound.

The best known solution used in comparisons with the lower bound is found by the algorithm itself. It is the best solution found in any leaf of the search tree. The tree is searched in the depth-first least lower bound order.

3.2 Genetic Algorithm

Genetic algorithms imitate evolution of genome. Solutions are encoded as strings of symbols analogously to the encoding of the chromosomes. Some initial population of solutions is generated randomly. Genetic operators transform populations in a direction improving quality of the solutions. Selection, crossover, and mutation are typical genetic operators. *Selection* elects better solutions for the next population. *Crossover* operation generates offspring solutions by randomly combining pieces of the parent strings. Though the offspring is constructed in a random manner, the fragments of a string encoding an optimum solution are indirectly discovered and combined due to the selection and crossover. *Mutation* changes randomly some solutions to diversify the search, and to escape local optima. Genetic search is a classic technique for solving combinatorial optimization problems, including scheduling problems. We direct interested readers to monographs [9, 10] for detailed presentation of the genetic search method.

In our implementation a chromosome is a string $(d_1, \ldots, d_n)$ of chunk destinations. The measure of a chromosome fitness is the value of schedule length C_{max} obtained from the linear program (1)-(2) formulated for the sequence of chunk targets given in the chromosome. In the crossover operation two chromosomes are randomly selected, and combined using one point crossover. For example, let $(a_1, a_2, \ldots, a_n)$, $(b_1, b_2, \ldots, b_n)$ be two parent solutions, and let k denote a randomly selected crossover point. The two offspring solutions are $(b_1, \ldots, b_{k-1}, a_k, \ldots, a_n)$ and $(a_1, \ldots, a_{k-1}, b_k, \ldots, b_n)$. The total number of new chromosomes constructed in crossover is Gp_C, where G is the size of the population, and p_C is a tunable algorithm parameter which will be called crossover probability. Mutation changes Gnp_M random genes (i.e. d_is) to different values. Gn is the total number of genes, p_M is a tunable algorithm parameter which we will call mutation probability. The selection of the chromosomes for the new population is done by a combination of elitist and roulette wheel method. The best half of the old population is always preserved. A string is passed to the second half of the new population with probability $\frac{1}{C^j_{max}} / \sum_{j=1}^{G} \frac{1}{C^j_{max}}$, where C^j_{max} is the schedule length for chromosome j. The algorithm stops after a fixed number of iterations without an improvement in the quality of the best solution ever found. There is also a limit on the total number of iterations.

4 Computational Experiments

4.1 Experiment Setting

All the experiments were performed on a PC computer with Pentium IV 1.8GHz, 512MB RAM memory, and Microsoft Windows XP. The executable code was generated by Borland C++ Builder 6.0. All LP formulations were solved by a code derived from `lp_solve` [1]. Unless stated otherwise, the test instances of the scheduling problem were generated in the following way: Processor parameters A, C, S, were generated with uniform distribution from the range [0,1]. Problem size was $V = 1\text{E}6$. The processor number was $m = 4$, and the number of chunks

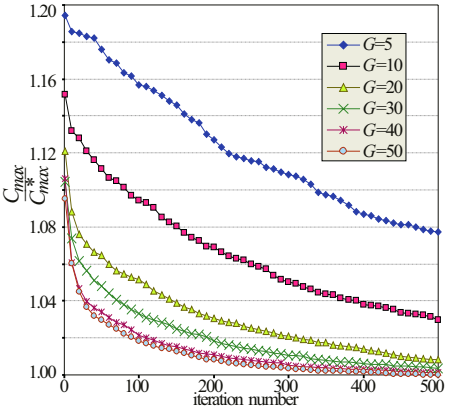

Fig. 3. Average distance from optimum vs. iteration (population) number and G.

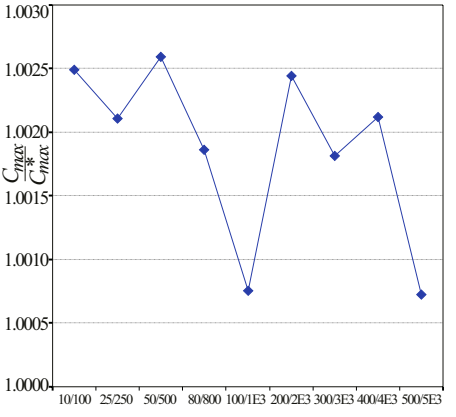

Fig. 4. Average distance from optimum vs. iteration limits.

was $n = 8$. Each point on the following charts is an average of at least 10 instances.

In the genetic algorithm genes of the initial population were generated with uniform distribution from set $\{1, \ldots, m\}$. The following procedure has been applied to tune the genetic algorithm. A set of 100 random instances were generated as a reference benchmark. An indicator of algorithm performance was the average quality of the best solutions obtained for these benchmark instances. Population size $G = 50$ has been selected as the convergence improvement stops at this size (cf. Fig. 3). For the fixed G crossover probability $p_C = 80\%$, and then mutation probability $p_M = 3\%$ were selected. We used a limit of 10 iterations without solution improvement, and an upper limit of 100 iterations in total, which give acceptable solution quality on average (cf. Fig. 4), but still result in a shorter running time than other iteration limits combinations.

4.2 Performance of the Algorithms

Running Times. The execution times of the algorithms are collected in Fig. 5, and 6. The running time of the branch and bound is denoted by B&B, and of the genetic algorithm by GA. It can be seen that the branch and bound algorithm has exponential running time in n for fixed m (cf.Fig. 5). The execution time grows slower as a function of m for fixed n (cf.Fig. 6) because the maximum number of the search tree leaves is m^n. Nevertheless, execution time of the branch and bound algorithm allows only for solving instances with small m, and n. Execution time of the genetic algorithm grows with n (Fig. 5) because the length of the string encoding solution is n. For $m = 3, \ldots, 20$ execution time grows less than twice (Fig. 6). We also tested dependence of the execution times on size V of the problem. For small V execution time of the branch and bound was shorter than for big sizes because less processors had to be activated,

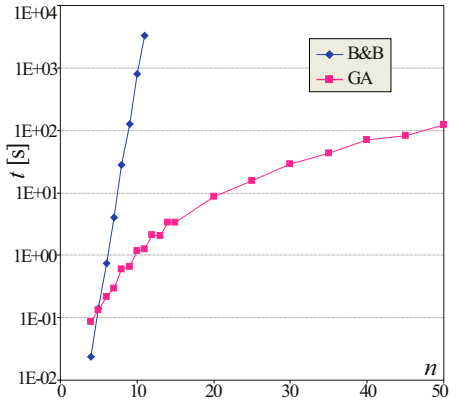

Fig. 5. Running time vs. n.

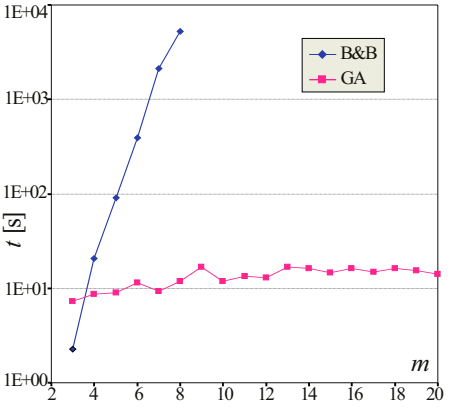

Fig. 6. Running time vs. m.

and therefore the search trees were smaller. The execution time of the genetic algorithm was independent of V.

Quality of the Solutions. The results of our study on the quality of solutions are collected in Fig. 7-8. The instances in Fig. 7 had A parameter equal to a given value on all processors. The remaining C, S parameters were generated as described previously. Analogously, for Fig. 8 parameter C was fixed on all processors, and A, S were randomly generated. Each figure represents quality of the solutions, i.e. the relative distance from the optimum, in three cases: the average solution of a genetic algorithm (denoted GA), the average random solution (denoted RND), and the worst selection of the chunk targets ever observed (denoted Worst). Note that the worst case (Worst) has its own 'y' axis different than RND, and GA cases. The random solutions (RND) have random chunk destinations. In all cases load chunk sizes were calculated by linear program (1)-(2).

These three cases demonstrate weaknesses and strengths of the two parts in the solution of our problem: the combinatorial part which finds targets for the chunks (d_is), and the linear programming part which calculates optimum chunk sizes (α_is) for the given destinations. It can be seen that genetic algorithm constructs solutions that are very close to the optimum. On average its solutions were not further 0.2% from the optimum. The worst solution obtained by the genetic algorithm was 1.1% away from the optimum. Thus, the genetic algorithm is a practical replacement for the optimization branch and bound algorithm which has exponential running time. The random solutions (RND) are also good on average because their distance from the optimum is not greater than approximately 30%. This is good news because solving a complex combinatorial problem of determining chunk targets (be it by a branch and bound or by a genetic algorithm) may be too time consuming and unprofitable on average. A random, or reasonable selection of processors and their activation sequence, supplemented by a linear program (1)-(2) gives solutions of acceptable quality on average. This

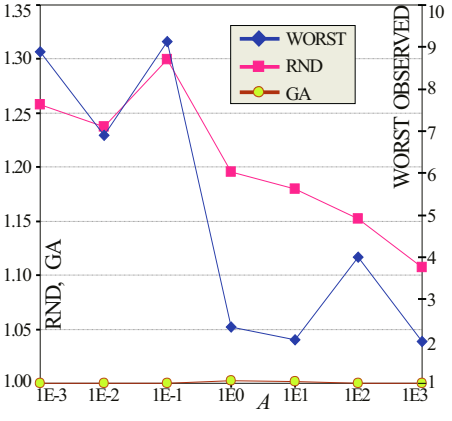

Fig. 7. Relative distance from the optimum vs. A.

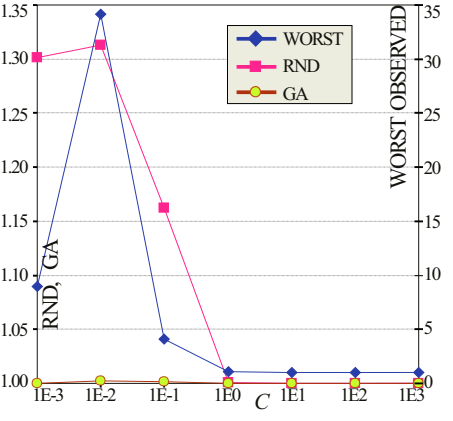

Fig. 8. Relative distance from the optimum vs. C.

tells us also about the nature of the problem we are solving. Since relatively good results can be obtained only by adjusting chunk sizes (even for random chunk destinations), the chunk size selection is an important element in the solution of our problem. In other words, linear programming can compensate for some bad decisions in combinatorial part of the algorithms. It can be said that on average the combinatorial part of our problem (i.e. target selection) improves a random solution by approximately 30%. Finally, the worst case really exists. In the worst observed case of the chunk target selection a schedule 35 times worse than optimum was constructed (cf. Fig. 8).

It is possible to infer from Fig. 7-8 on the features of the solutions and performance of the algorithms. With growing A, C the quality of the random and the worst case is improving. When A is very big, the schedule length becomes dominated by the computation time. The selection of the chunk destinations is nearly meaningless because the schedule length is determined by the computation time which is approximately $\frac{AV}{m}$. Similar conclusions can be drawn for parameter C. When C is very big, chunk target selection tends to be immaterial because the schedule length is determined by the communication time which is approximately VC. We also tested dependence on S in range [1E-3,1E3]. It turned out that S constitutes at most $\approx 2\%$ of the communication time, and hence this dependence was not strong.

5 Conclusions

In this paper we studied multi-installment divisible load processing in heterogeneous distributed system. The problem we analyzed consists in determining optimum destinations for the load chunks and adjusting their sizes to the speeds of processors and communication links. Hence, we divided solution methods into two parts: combinatorial one which finds destinations for the load chunks, and

linear programming part which finds optimum chunk sizes for the given targets. We have shown that in the worst case solutions can be arbitrarily bad if any of the two parts is ignored. In a set of computational experiments we demonstrated that on average the combinatorial part improves the solution quality by approximately 30 %. The linear part is a very important element in the construction of the schedule, and to some extent it is able to compensate bad decisions in the combinatorial part.

References

1. Berkelaar, M.: `lp_solve` - Mixed Integer Linear Program solver. `ftp://ftp.es.ele.tue.nl/pub/lp_solve` (1995)
2. Bharadwaj, V., Barlas, G.: Access time minimization for distributed multimedia applications. Multimedia Tools and Applications **12** (2000) 235-256
3. Bharadwaj, V., Ghose, D., Mani, V.: Multi-installment Load Distribution in Tree Networks With Delays. IEEE Transactions on Aerospace and Electronic Systems **31** (1995) 555-567
4. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.: Scheduling divisible loads in parallel and distributed systems. IEEE Computer Society Press, Los Alamitos CA (1996)
5. Błażewicz, J., Drozdowski, M., Markiewicz, M.: Divisible task scheduling - concept and verification. Parallel Computing **25** (1999) 87–98
6. Drozdowski, M.: Selected problems of scheduling tasks in multiprocessor computer systems. Series: Monographs, No 321, Poznań University of Technology Press, Poznań (1997). Downloadable from `http://www.cs.put.poznan.pl/mdrozdowski/txt/h.ps`
7. Drozdowski, M., Wolniewicz, P.: Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In: A.Bode, T.Ludwig, W.Karl, R.Wismüller (eds.), Euro-Par 2000. Lecture Notes in Computer Science, Vol. 1900. Springer-Verlag, Berlin Heidelberg New York (2000) 311-319
8. Drozdowski, M., Wolniewicz, P.: Out-of-Core Divisible Load Processing, IEEE Trans. on Parallel and Distributed Systems **14** (2003) 1048-1056.
9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, Massachusetts (1989)
10. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, Berlin Heidelberg New York (1996)
11. Robertazzi, T.: Ten reasons to use divisible load theory. IEEE Computer **36** (2003) 63-68
12. Wolniewicz, P.: Divisible Job Scheduling in Systems with Limited Memory. PhD Thesis, Poznań Univ. of Technology (2003). Downloadable from `http://www.man.poznan.pl/~pawelw/phd.pdf`
13. Yang, Y., Casanova, H.: Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation. Univ. of California, San Diego, Dept. of Computer Science and Engineering, Tech. Rep. CS2002-0721 (2002)