Out-of-Core Divisible Load Processing

Maciej Drozdowski, *Member*, *IEEE Computer Society*, and Pawel Wolniewicz

Abstract—In this paper, we analyze processing divisible loads in systems with a memory hierarchy. Divisible loads are computations that can be divided into parts of arbitrary sizes and these parts can be independently processed in a distributed system. The problem is to partition the load so that the total processing time, including communications and computations, is the shortest possible. Earlier works in the divisible load theory assumed distributed systems with a flat memory model. The dependence of the processing time on the size of the assigned load was assumed to be linear. A new mathematical model relaxing the above two assumptions is proposed in this article. We study distributed systems which have both the hierarchical memory model and a piecewise linear dependence of the processing time on the size of the assigned load. Performance of such systems is modeled and evaluated. Finally, we compare the efficiency of distributed processing divisible loads in multiinstallment and out-of-core modes. Multiinstallment processing consists in sending multiple small chunks of the load to processors instead of a single chunk which needs external memory. It turns out that multiinstallment is an advantageous strategy for reasonably selected load chunks sizes.

Index Terms—Divisible load theory, scheduling, performance evaluation, memory hierarchy, communication delays.

1 Introduction

The divisible load model applies to a class of distributed applications allowing to divide computations into parts of arbitrary sizes. The parts are processed in parallel by a set of distributed computers. The sizes of the load parts must be adjusted to the speeds of communication and computation, and possibly also other system parameters, to finish processing in the shortest possible time. Examples of divisible computations include search for patterns in text, audio, graphical files, database processing, measurement data processing, data retrieval systems, video on demand, some linear algebra algorithms, simulation, and combinatorial optimization algorithms [3], [8], [11], [15], [18]. For example, a big file searched for patterns can be divided into parts of arbitrary sizes, and these parts can be processed in parallel.

A general scenario of distributed computations assumed in the divisible load theory (DLT) is that, initially, some volume V of the load resides on one processor called *originator*, which sends chunks of the load to other processors. The time of transferring x load units (e.g., bytes) is $S + Cx$ time units (e.g., seconds). Computing x load units takes Ax time units. After receiving the message, a processor intercepts its share of the load and immediately starts computing. The rest of the load is relayed to the processor's inactive neighbors. This procedure is repeated until activating all the used processors. After the completion of the computations, the results can be returned to the originator. Since communication delays are inherent in the divisible load model, the underlying interconnection topology is explicitly analyzed. DLT started with a linear array of processors architecture [10], other interconnection topologies such as buses, trees, hypercubes, and meshes, were included later [6], [7], [13], [14]. In this paper, we consider a star interconnection (also called a single level tree) which is topologically equivalent to a bus. Extensive surveys of DLT can be found in [4], [6], [13].

In the earlier DLT literature, the processing time dependence on the size of the load was linear. This is justified in flat (nonhierarchical) memory systems. Though core memory sizes grew rapidly over the years, the memory size limitations are an important factor in high-performance computing. Under DLT assumptions, memory limitations have been considered first in [19]. A fast heuristic method has been proposed which constructs load distributions not exceeding the given memory limits. Later, a more general method with guaranteed optimality has been proposed in [16]. In both [19] and [16], it was assumed that memory limits are restrictive, i.e., assigning load beyond the memory limit is forbidden, and results in an infeasible solution. Yet, in most contemporary computer systems, memory is hierarchical. The higher the certain level of memory hierarchy is, the faster transmission can be achieved. However, the higher the certain level of memory hierarchy, the smaller the memory size. The lowest memory levels are implemented either as virtual memory storing memory pages on disks or as files directly accessed by the application. Huge sizes of disk storage can be achieved at relatively low costs using off-the-shelf components. Thus, instead of strictly forbidding a load assignment exceeding certain memory level size, it is more practicable to use the next memory level with longer access time and, hence, a smaller computing rate. We will call the applications using external memory (i.e., disks) the *out-of-core* computations. In Fig. 1, we demonstrate that using out-of-core memory makes a big difference in the computation speed. A dependence of the processing time of a simple search for a pattern in a linear array on the array size is shown in Fig. 1. Even for this simple application, with a predictable memory

- *M. Drozdowski is with the Institute of Computing Science, Poznań University of Technology, ul.Piotrowo 3A, 60-965 Poznań, Poland. E-mail: Maciej.Drozdowski@cs.put.poznan.pl.*
- *P. Wolniewicz is with the Poznań Supercomputing and Networking Center, ul.Noskowskiego 10, 61-794 Poznań, Poland. E-mail: pawelw@man.poznan.pl.*

Manuscript received 5 Aug. 2002; revised 17 Jan. 2003; accepted 17 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 117075.

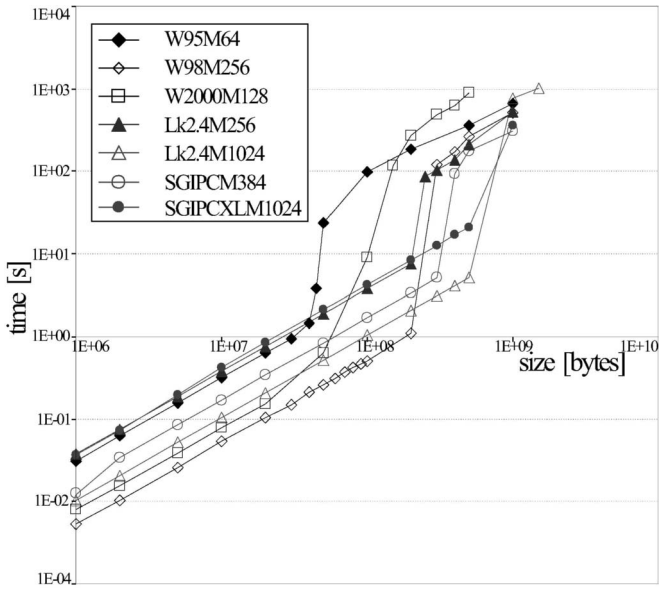

Fig. 1. Processing time of a simple search for a pattern in a linear array versus array size. In the legend, Wx denotes Windows version x, Lk2.4 denotes Linux with kernel version 2.4, SGIPC—SGI Power Challenge, SGIPCXL—SGI Power Challenge XL, My denotes core memory size y MB.

access pattern using more memory than available in the core results in an increase of the execution time by at least an order of magnitude.

There is a broad class of the out-of-core parallel applications. These include data-intensive algorithms [21] for processing information from large scientific experiments, data mining, visualization [1], [12], and simulation, often with the need to solve large linear algebra problems [22]. Gaussian [17] is an example of a commercial package using out-of-core memory. In [9], an environment for out-of-core parallel applications has been proposed. Computational fluid dynamics and large linear algebra problems have been used as benchmarks. The access to the major data arrays was achieved by using indirect addressing that has been known at the runtime only. It turned out that, by using locality in the algorithm and dividing the arrays into small sections that fit in the core memory a fourfold reduction of the execution time has been obtained compared to the use of virtual memory.

Thus, an alternative to the out-of-core processing is to divide the load into many small chunks that fit into the available core memory. The chunks are sent to processors in an iterative manner. Hence, it is possible to perform fast computations at the cost of additional communications. We will call this way of computing a *multiinstallment* divisible load processing. Multiinstallment processing has been considered, e.g., in [5], [23]. In this work, we compare the efficiency of the out-of-core with the efficiency of multiinstallment computations.

The original contribution of this paper can be summarized in the following way: The DLT is extended by the inclusion of hierarchical memory systems in its methodology. To the best of our knowledge, this is the first attempt of modeling nonlinear dependence of the processing time on the load size in the DLT. Simple and tractable mathematical tools are used to solve the proposed model. We believe that this is a good compromise between precision and tractability in modeling parallel applications. The performance of the data-intensive applications and utilization of the computational resources should benefit from the new method of determining the optimum distribution of the computations in the systems with hierarchical memory and communication delays.

The rest of this paper is organized as follows: In Section 2, the problem of constructing optimum load distribution is formulated in a mathematical form. In Section 3, the influence of the system parameters on the performance is studied. In Section 4, the performance of the out-of-core computations is compared with that of the multiinstallments computations. The notation used in this paper is summarized in the Appendix.

2 Mathematical Models

In this section, we formulate mathematical models for divisible load computations in a system with hierarchical memory, and for multiinstallment computations. We will use the word processor to denote a single processing element with CPU, memory, disks, and network interface.

Let us start with the description of the communication subsystem. We assume a star interconnection network. In this topology, originator P_0 is located in the center of the star. Processors $P_1, \ldots, P_m$ can communicate only with the originator. Only one communication can be executed at a time. Thus, star topology is equivalent to a bus interconnection here. The load is sent to the processors in a single communication. P_1 receives the load first, P_2 as the second processor, etc. P_m receives its load as the last one. The originator does not compute, but communicates only. This assumption does not limit the generality of our considerations because computations on the originator can be represented as an additional processor. The time of transferring x units of load to processor P_i is $S_i + xC_i$. Note that the communication time depends linearly on the size of the message, but also includes fixed startup cost S_i. For the simplicity of the presentation, we assume that the time of returning the results can be neglected. We do not exclude applications which return some results; this assumption is done for the sake of simplicity of mathematical models. The process of returning the results can be easily represented in the DLT, which was demonstrated both theoretically in monographs [6], [13], and practically by the applications [3], [8], [11], [15], [18].

The computations are performed by processors connected to the hierarchical memory systems. The highest level is constituted by processor registers. The lowest level is disk storage. The memory sizes increase and transfer rates decrease with the decreasing hierarchy level. Hence, the processing time depends on the amount of allocated memory. Processing time t_i on processor P_i is a piecewise linear function of the assigned load x: $t_i = \max\{A_{ij1} + xA_{ij2}\}$ (cf. Fig. 2), where A_{ij1}, A_{ij2} are the coefficients of the linear function describing the processing time on processor P_i at the jth hierarchy level. Note that A_{i11} is the cost of starting computations on processor P_i. For practical reasons, only two levels of memory hierarchy: core memory and virtual memory (or other form of disk storage) are considered in this paper. The reason for this simplification is that divisible load model is well-suited for data parallel applications processing large volumes of data. Therefore, high levels of

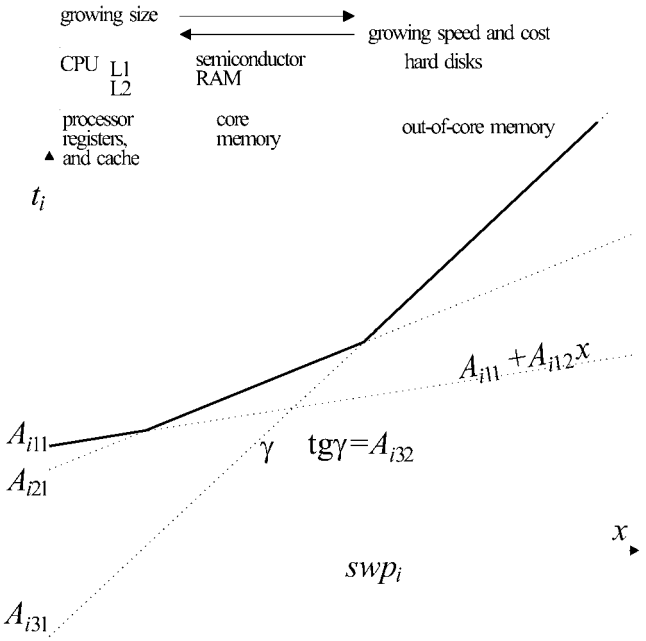

Fig. 2. Memory hierarchy diagram and a piecewise linear dependence of processing time on the size of the load.

memory hierarchy, such as processor registers and caches, are not able to hold a substantial part of the assigned load. Due to the uniform and regular structure of divisible load applications, memory access patterns are very predictable and cache management algorithms make this memory level transparent. The processor cache level of memory hierarchy could be visible for the application if the memory access pattern were random. However, to the best of our knowledge, no divisible load processing problem has been presented with random memory access patterns. The simplification of the model to only two memory levels can be easily relaxed as we explain in the further part of this section. We denote the processing time on processor P_i as $t_i = \max\{A^l_{i1} + xA^l_{i2}, A^h_{i1} + xA^h_{i2}\}$, where A^l_{i1}, A^l_{i2} are the coefficients of the linear function describing the computing time in the core memory, and A^h_{i1}, A^h_{i2} for computing out-of-core using disk storage. The size of the load swp_i beyond which the operating system starts using the disk, and for which the above two functions are equal, i.e., $A^l_{i1} + swp_i A^l_{i2} = A^h_{i1} + swp_i A^h_{i2}$, will be called a *swap point* of processor P_i.

Let us observe that the above piecewise linear dependence of the processing time on the load size may also have a different nature. Not only can the memory hierarchy be modeled in this way, but also, referencing memory on remote hosts or nonlinear dependence of the processing time on the problem size can be dealt with in this way. Hence, after approximating a nonlinear convex function of the processing time by a piecewise linear convex function, our method can be used to represent more complex DLT applications.

We will formulate the problem of constructing optimum distribution of the divisible load computations as a linear program. Linear programming is a special case of mathematical programming. It is used for modeling problems in science and engineering [20]. Let us denote by α_i the amount of load assigned to processor P_i, and by C_{max} the completion time of processing. Our problem can be formulated as a linear program:
minimize C_{max}
subject to:

$$\sum_{j=1}^{i}(S_j + \alpha_j C_j) + t_i \le C_{max} \quad i = 1, \ldots, m \tag{1}$$

$$A^l_{i1} + \alpha_i A^l_{i2} \le t_i \quad i = 1, \ldots, m \tag{2}$$

$$A^h_{i1} + \alpha_i A^h_{i2} \le t_i \quad i = 1, \ldots, m \tag{3}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{4}$$

$$\alpha_i \ge 0 \quad i = 1, \ldots, m.$$

The above formulation has $2m+1$ variables and $4m+1$ constraints. On the left-hand side of inequality (1), communication time $\sum_{j=1}^{i}(S_j + \alpha_j C_j)$ until activating P_i is added to processing time t_i on P_i. Hence, inequality (1) guarantees that all processors stop computing before the end of the schedule. Inequalities (2) and (3) together, model a piecewise linear processing time function of the assigned load. Observe that (2) and (3) restrict processing time t_i from below, but do not bind it from above. Sufficiency of these two constraints is guaranteed by the features of linear programming [20]. As the linear program constraints formulate a $2m+1$-dimensional convex polyhedron, and the objective function is a linear function of the program variables (C_{max}), the optimum solution is a point in $2m+1$-dimensional space located in an extreme corner of the polyhedron. The constraints intersecting in the optimum corner of the polyhedron are limiting the optimum value of the objective function, and are called active. If one of the constraints (2), (3) is active for some i, then it is satisfied with equality and t_i is exactly equal to the piecewise function expressing the processing time. If none of the constraints (2), (3) is active for some i and both are satisfied with inequality, then it means that processor P_i is idle for some time after completing the computation phase. By inequality (4), all the load is processed.

Note that formulations (1)-(4) can be augmented by adding a constraint of the form $\alpha_i \le B_i$ to limit the total memory usage on some processor P_i. Constraints analogous to (2) and (3) can be added to represent additional memory hierarchy levels. For the feasibility of this method, it is necessary that the dependence of processing time on the volume of load be a piecewise linear convex function. The shape of the convex polyhedron and the location of the optimum extreme corner depend on the numerical values of the coefficients in constraints (1)-(4). Therefore, no closed form expression of α_i seems possible. Consequently, general analytical solutions are hard to be expected.

Let us use an example to compare the above model with the earlier DLT approach. Consider a homogeneous system with two processors and computing time function described by parameters: $A^l_{11} = A^l_{21} = 1$, $A^l_{12} = A^l_{22} = 1$, $A^h_{11} = A^h_{21} = -9$, $A^h_{12} = A^h_{22} = 10$. Hence, the swap points are at load size $\frac{10}{9}$, and core memory is approximately 10 times faster because $\frac{A^h_{12}}{A^l_{12}} = \frac{A^h_{22}}{A^l_{22}} = 10$. The communication transfer rate is $C = 1$ and the startup time is $S = 1$. The load volume is $V = 2$. By solving formulations (1)-(4), we obtain a solution $\alpha_1 = 1.25, \alpha_2 = 0.75, C_{max} = 5.75$ and the

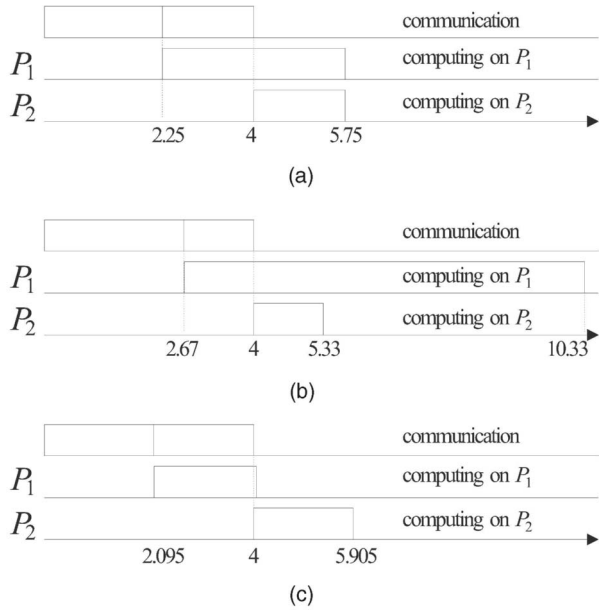

Fig. 3. Schedules for load distributions calculated assuming (a) hierarchical model of the memory, (b) computing in the core memory only, and (c) computing out-of-core only.

schedule shown in Fig. 3a. If standard DLT methodology were used, we would have to assume that the processing time is a strictly linear function of the load. Thus, computing x units of the load would take either xA_{12}^l (if we optimistically assume that only the core memory is used) or xA_{12}^h (if computing takes place out-of-core only). In the first case, the standard DLT theory [4], [6], [13] gives solution $\alpha_1 = \frac{5}{3}, \alpha_2 = \frac{1}{3}, C_{max} \approx 4.333$. However, the real schedule length for this load distribution would be approximately 10.333, due to the hierarchical structure of the memory (see Fig. 3b). In the second case, the standard DLT solution is $\alpha_1 \approx 1.095, \alpha_2 \approx 0.905, C_{max} \approx 13.048$. Yet, the real schedule length for this load distribution is approximately 5.905 (see Fig. 3c). As it can be seen in Fig. 3, neglecting memory hierarchy results in significant load imbalance. The decisions made on the basis of the average processing rates can be even worse in heterogeneous systems. Let us consider a two processor system with processor P_1 core memory size V and processor P_2 core size 0. The computing speed on the second memory level is equal for both processors. A decision made on the basis of the speed at the second memory level splits the load equally between the processors. The optimum, however, is to give the majority of the load to P_1. Depending on the speed of P_1 for the in-core computations the ratio of the optimum schedule length and the length of the schedule based on average speed can be very large.

Now, we shall formulate a simple algorithm for multiinstallment divisible load processing and a method for adjusting its parameters. By the use of installments, we want to exploit fast computing within the limits of the available core memory, while keeping the communication costs low. Let us assume that the multiinstallment algorithm divides the whole volume V into equal chunks of size δ. The processors are assigned load repetitively in rounds, i.e., in the manner $P_1, P_2, \ldots, P_m, P_1, \ldots$. The selection of the optimum chunk size δ is a nontrivial problem. Therefore, we give bounds on reasonable δ sizes and propose a heuristic method indicating a potentially good value.

Chunk size δ cannot exceed the swap point of any of the processors, i.e., $\delta \leq swp_i$ for $i = 1, \ldots, m$. Second, it cannot be too small because too many messages would be used and communication costs would dominate the processing time. Let us calculate the minimum chunk size for which multiinstallment processing is still better than the computations out-of-core. When the second memory level is used, the load must be at least as big as $m \times swp$. Thus, we may assume that the processing time is dominated by the computation time. A rough estimate of out-of-core processing rate for large volumes is

$$\lim_{V\to\infty} \frac{C_{max}}{V} = \frac{1}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}},$$

where $\sum_{i=1}^{m} \frac{1}{A_{i2}^h}$ is the total speed of all the processors. An estimate of the processing time for multiinstallment processing with small load chunks and dominating communication time is $\frac{V}{m\delta}\left(\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i\right) + A_{1i}^l + \delta A_{2i}^l$, where $\frac{V}{m\delta}$ is the number of communication rounds, $\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i$ is the communication time per round, and $A_{1i}^l + \delta A_{2i}^l$ is the computation time for the last chunk. Hence, an estimate of the processing rate is

$$\lim_{V\to\infty} \frac{C_{max}}{V} = \frac{1}{m\delta}\left(\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i\right).$$

The multiinstallment mode is faster when its processing rate is smaller than the one for the out-of-core mode:

$$\frac{1}{m\delta}\left(\sum_{i=1}^{m} S_i + \delta \sum_{i=1}^{m} C_i\right) < \frac{1}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}},$$

from which we get

$$\delta > \frac{\sum_{i=1}^{m} S_i}{\frac{m}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}} - \sum_{i=1}^{m} C_i}.$$

Thus, chunk size δ should be selected from the range

$$\left(\frac{\sum_{i=1}^{m} S_i}{\frac{m}{\sum_{i=1}^{m} \frac{1}{A_{i2}^h}} - \sum_{i=1}^{m} C_i}, \max_{i=1}^{m}\{swp_i\}\right).$$

For uniform computing systems, this expression can be simplified to $(\frac{mS}{A_2^h - mC}, swp)$, where swp is the swap point.

When δ increases, the load imbalance may arise and some processors may have to wait idle for the completion of the computations on other processors. Furthermore, the bigger δ, the longer the processors must wait before starting the computations. On the other hand, if δ decreases, the number of messages grows and communication overhead increases. Hence, it can be expected that, for some instances, of the system parameters an optimum value of δ exists for which the processing time is minimum. We propose a heuristics to select δ. The value of δ should be such that a processor is computing during the whole communication round, while the originator is sending the load to the

processors. This results in requirement $A_{i1}^l + \delta A_{i2}^l \geq \sum_{j=1}^{m}(S_j + \delta C_j)$, for processor P_i. Taking into account all processors:

$$\delta = \max_{i=1}^{m}\left\{\frac{\sum_{j=1}^{m} S_j - A_{i1}^l}{A_{i2}^l - \sum_{j=1}^{m} C_j}\right\}.$$

For a uniform computing system, the above formula expressing δ can be simplified to:

$$\delta = \frac{mS - A_1^l}{A_2^l - mC}, \tag{5}$$

where A_1^l, A_2^l are parameters of the linear function of the processing time in the core memory, and C, S are communication time parameters. Note that δ can be calculated in this way only if the numerator and the denominator are of the same sign. In (5), the numerator is positive when $mS > A_1^l$, which means that a processor is able to start computation within the duration of activating communication to all processors. If the numerator is negative, messages arrive faster than the processors are able to process them, and the load will accumulate in communication buffers. Consequently, the numerator and the denominator must be positive. Denominator $A_2^l - mC$ is positive when $\frac{A_2^l}{m} > C$, which means that the computing rate of all processors together is greater than the communication rate or, in other words, communication speed is greater than the total computing speed of all processors. If the denominator is negative, the total computing speed of processors is greater than the communication speed and some idle times will arise on some processors. The negative denominator, or the fact that (5) expresses a value outside the admissible interval introduced in the preceding paragraph, do not limit the applicability of the multiinstallment strategy. It means that chunk size δ must be selected in a different way. In practice, by selection of δ, applications can be experimentally tuned to obtain good performance.

3 Performance Modeling

In this section, we shall present the results of modeling dependence of a computing system performance on the model parameters. Over 2,400 instances of the linear programs were solved by `lp_solve`, a free linear programming code [2].

Let us analyze the optimum distribution of the load under various swap point values and speeds of the processors. This dependence for two processors ($m = 2$) and load size $V = 2\text{E}8$ is presented in Fig. 4. We assumed that parameters of P_1 are fixed to $A_{11}^l = 0, A_{12}^l = 1\text{E}-3$, $A_{11}^h = -9\text{E}5$, $A_{12}^h = 1\text{E}-2$ (hence, $swp_1 = 1\text{E}8$). The parameters of speed and swap point of P_2 were variable, except for $A_{21}^l = 0$. In Fig. 4, load α_2 assigned to processor P_2 is presented on the vertical axis, on the horizontal axis the ratio $\frac{A_{22}^h}{A_{12}^h} = \frac{A_{22}^l}{A_{12}^l} = \beta$ of the processor speeds is shown, various values of the processor P_2 swap points $\frac{swp_2}{V}$ are represented by different curves. As we move to the right along the horizontal axis, the speed of processor P_2 decreases, and its load also decreases. The curve for $\frac{swp_2}{V} = 0$ represents P_2 using the second level of memory only (disk), while $\frac{swp_2}{V} = 1$ represents P_2 able to hold all load V in the first level of memory (core). As swap point swp_2 increases, load size α_2 also increases. Curves for $\frac{swp_2}{V} < 1$ do not cross the curve $\frac{swp_2}{V} = 1$ because, at the point of such intersection, the load assigned to processor P_2 is small enough to be held in the core, i.e., $\alpha_2 \leq swp_2$, and the real location of the swap point of P_2 is meaningless. Three intervals of processing rate ratio β can be distinguished in Fig. 4. When $\beta < 1\text{E}-1$, then P_2 has the second memory level faster than the first memory level of P_1. In interval $[1\text{E}-1, 1\text{E}0]$, P_2 is faster than P_1, but only when the core memory is used on P_2. In the third interval of $\beta > 1$, P_2 is slower than P_1, independently of the memory level used. In these three intervals, α_2 changes with different speeds under β changes. This can be seen, especially for swap points $swp = 0$ and $swp = 1$, for which the curves are not smooth. When the two processors are identical ($\frac{swp_2}{V} = 0.5, \beta = 1$), the distribution of the load is not exactly equal because processor P_1 receives the load first and computes longer. It can be concluded that, even though the mathematical model is linear, the optimum distribution of the load changes nonlinearly with the growing difference of the processors.

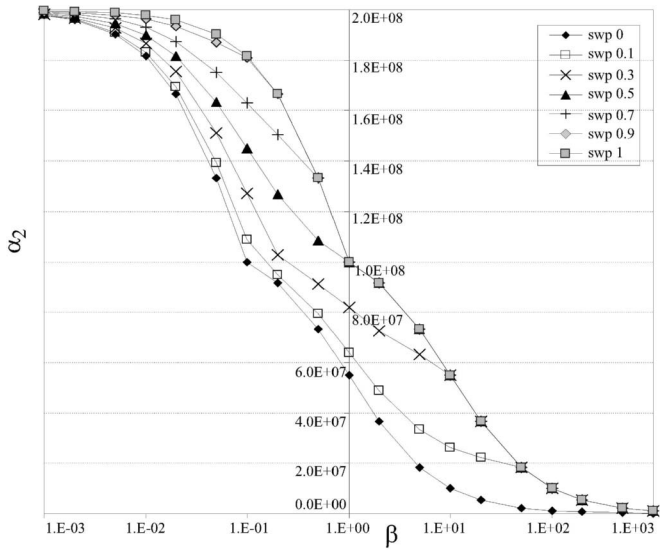

Fig. 4. Changes in the load partition for $m = 2$ and various $\beta = \frac{A_{22}^h}{A_{21}^l}$, and swp_2.

In the following part of this section, we shall consider homogeneous computing systems only. Therefore, we shall use a simplified notation, in which A_1^l, A_2^l are parameters of the linear function of processing time for the core memory, and A_1^h, A_2^h for the out-of-core memory. C, S are communication time parameters. Unless otherwise specified, we considered a system with $m = 10$ processors, $A_1^l = 0, A_2^l = 1\text{E}-3$, $swp = 1\text{E}8$, $\frac{A_2^h}{A_2^l} = 10$, and communication parameters $C = 1\text{E}-6$, $S = 1\text{E}-3$.

Fig. 5 demonstrates the dependence of processing time C_{max} on problem size V for various values of the ratio of processing rates in core and out-of-core $\frac{A_2^h}{A_2^l}$, and swap points $swp = 1\text{E}8$ or $swp = 1\text{E}11$. As it can be predicted, $\frac{A_2^h}{A_2^l}$ has

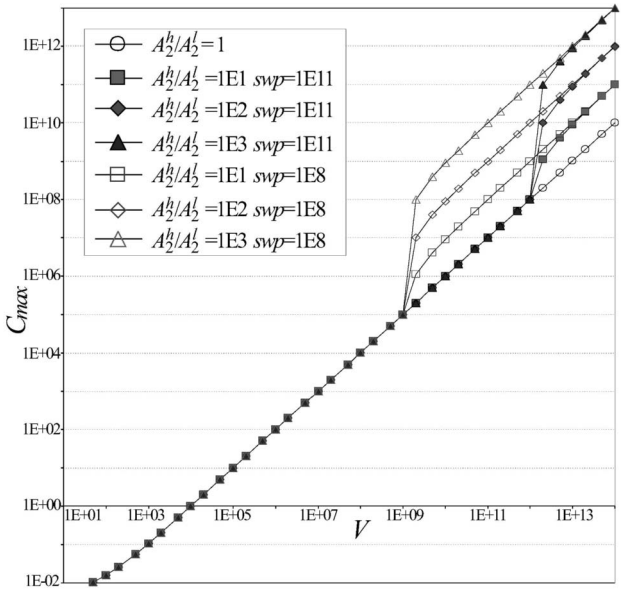

Fig. 5. Processing time versus V for various A_2^h/A_2^l and swp.

some influence on the processing time, when the second memory level is used, which is the case for $V > m \times swp$, i.e., load size exceeding the total core memory size.

In Fig. 6, the dependence of the processing time on the size of problem V for various computing speeds is shown. The curves represent systems with different speeds. A dotted reference line shows the communication time equal to $mS + VC$, which is a lower bound on the processing time. In all cases, ratio $\frac{A_2^h}{A_2^l}$ of the processing rates in core and out-of-core and swap points swp were fixed. It can be observed that increasing speed beyond a certain level is not profitable because communication becomes a bottleneck. Note that for $A_2^l = 1\mathrm{E}-3$, $A_2^l = 1\mathrm{E}-5$, and $A_2^l = 1\mathrm{E}-7$, some points are missing in Fig. 6. It is the case when some of the processors receive no load. This means that not all

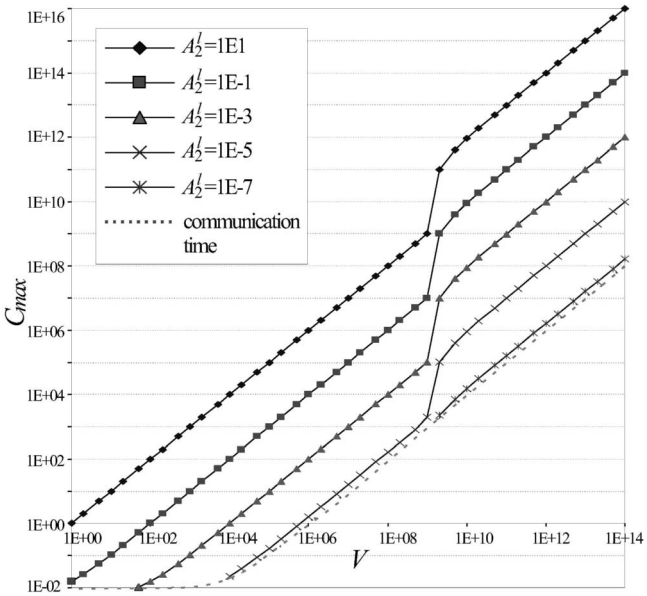

Fig. 6. Processing time versus V for various A_2^l and fixed swp, A_2^h/A_2^l.

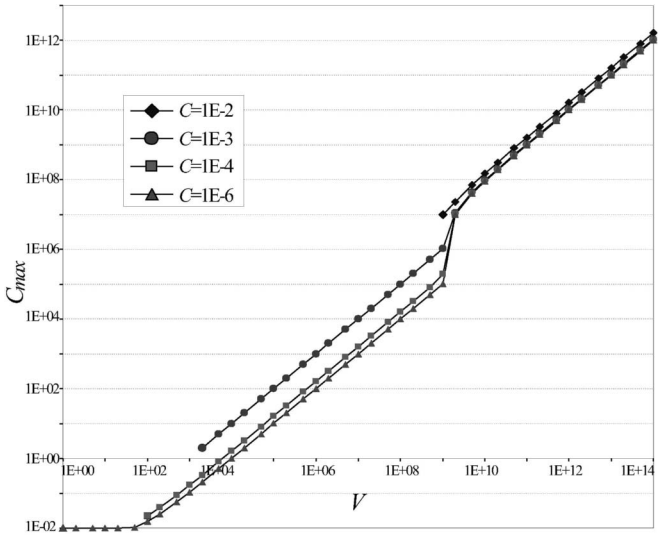

Fig. 7. Processing time versus V for various C and fixed swp, A_2^h/A_2^l.

m processors can be effectively used because computing on less than m processors is shorter than activating all the processors.

Dependence of the processing time on the size of problem V for various communication speeds is shown in Fig. 7. It can be observed that the processing time decreases with C decreasing only up to a certain limit beyond which the computing speed is the limiting factor. Not in all cases, $m = 10$ processors can be used here. When the communication speed is small $C = 1\mathrm{E}-2$, all processors can be used for load sizes $V \geq 1\mathrm{E}9$. As the communication speed increases (i.e., C is decreasing), the size of the problem for which all processors can be used also decreases.

In Fig. 8, speedup for various processor numbers m and problem sizes V is shown. The size of $V = 1$ (e.g., byte), certainly, is not practical, but it shows the behavior of the model. As it can be seen, for problem sizes $V = 1$, speedup decreases all the time. It is so because the load is too small

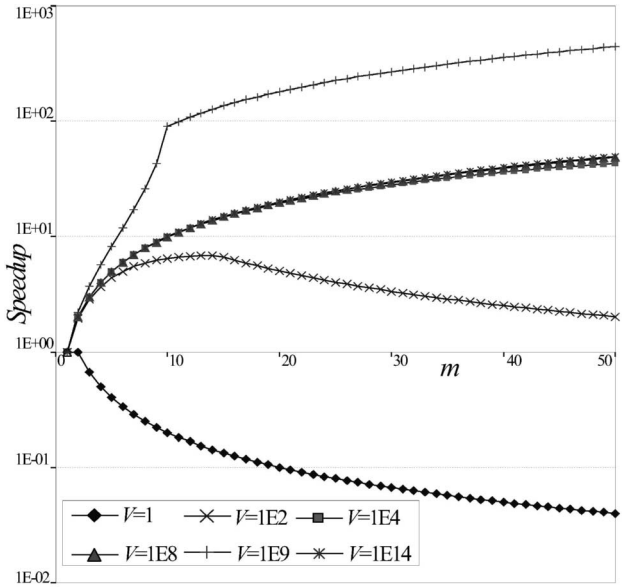

Fig. 8. Speedup versus m for various V.

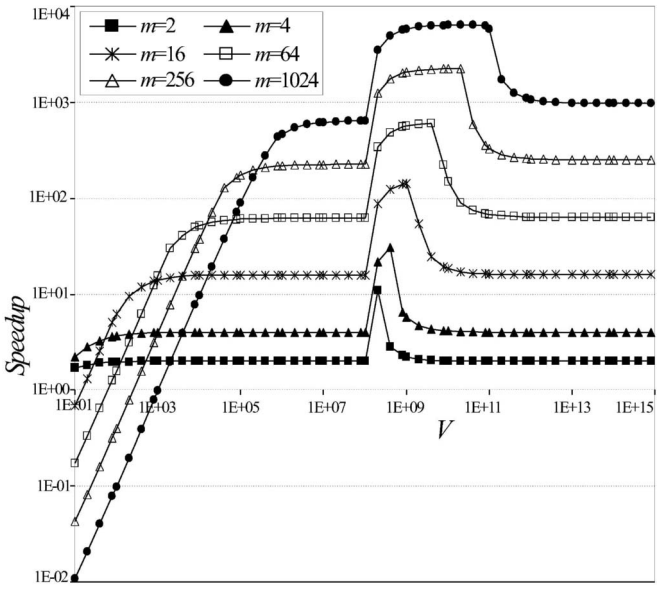

Fig. 9. Speedup versus V for various m.

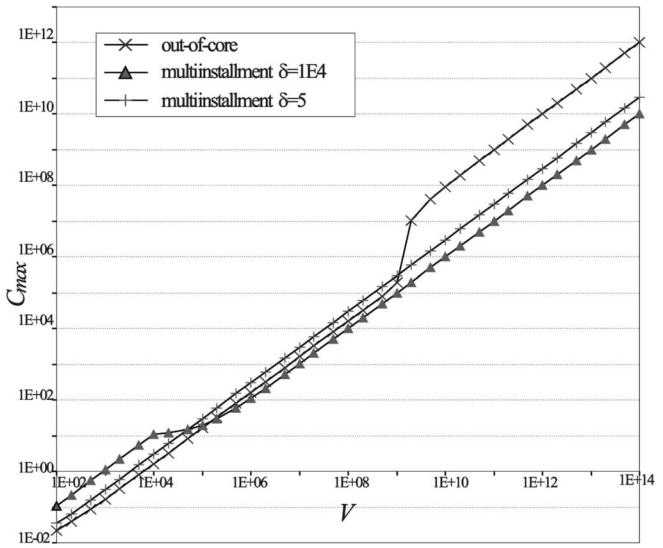

Fig. 10. Processing time versus V for multiinstallment and out-of-core computations.

and one processor is able to perform all the computations within the time of activating additional processors. The additional processors receive no load and only unnecessary communication cost is induced. The case of load size $V = 1\text{E}2$ is similar when the number of processors m exceeds 14. For $m \leq 14$, speedup is growing which indicates some profit from parallelism. For other problem sizes V, speedup is growing. Both when the load size is smaller than the core memory $V \leq swp$ and when the problem size by far exceeds the total core memory size $V \gg m \times swp$, the speedup is similar and close to linear. Therefore, lines for $V = 1\text{E}4$, $V = 1\text{E}8$, $V = 1\text{E}14$ overlap. When $V \approx m \times swp = 1\text{E}9$ superlinear speedup can be observed, because using m processors allows for holding most of the load in the core memory, while computing on one processor requires using slower external memory. Fig. 8 shows speedup obtained on the assumption that exactly m processors are activated by an appropriate message even if some of them receive no load to process. It has been observed that the number of processors for which speedup achieved its maximum corresponds to the maximum number of processors for which all processors receive some load. More insight into the behavior of the speedup is given by Fig. 9, which presents speedup versus V for various m. It can be seen that superlinear speedup is achieved for problem sizes V in range $(swp, m \times swp]$. When the number of processors is too large, speedup decreases with decreasing load V.

4 Out-of-Core and Multiinstallment Load Processing

In this section, we compare two modes of processing divisible loads: out-of-core computations which use external memory with multiinstallment processing of small pieces of the load on the first level of memory hierarchy, but at the cost of additional communications.

We considered a homogeneous system with $m = 10$ processors, communication rate $C = 9.99\text{E} - 5$, communication startup time $S = 1\text{E} - 3$, and computing time function coefficients $A_1^l = 0, A_2^l = 1\text{E} - 3$, $A_1^h = -9.9\text{E}6$, and $A_2^h = 1\text{E} - 1$ (hence, $swp = 1\text{E}8$). We used (5) to calculate the load chunk size $\delta = 1\text{E}4$. The dependence of processing time (C_{max}) on problem size V is shown in Fig. 10. Note that both axes are logarithmic and a small constant difference in this figure can be a big difference in the absolute terms. The three lines in Fig. 10 depict the processing time in the out-of-core mode, multiinstallment mode using $\delta = 1\text{E}4$, and using $\delta = 5$. For $V < 1\text{E}4$, multiinstallment with $\delta = 1\text{E}4$ is the worst because only one chunk of the load is sent and only one processor works, while the other processors remain idle. For $V \in [1\text{E}4, m \times swp]$, the processing time increases slowly, in the case of multiinstallments with $\delta = 1\text{E}4$, because more than one load chunk must be sent and additional processors are activated. For load chunk $\delta = 5$, processing time in multiinstallment mode is shorter than with $\delta = 1\text{E}4$ for loads V smaller than approximately 1E5. It is also better than the out-of-core computation when the second level of memory comes into use. Multiinstallment with $\delta = 5$ is worse than distributing the load according to the linear program (1)-(4) when the core memory is used. It is because the latter distribution has only one communication per processor, and perfect load balance resulting in simultaneous completion of computations on all processors. As it can be seen in Fig. 10, the multiinstallment mode of processing outperforms the out-of-core computations even for the chunk sizes δ smaller than the one selected according to (5).

The predictions of our model are confirmed by the computational experiments conducted on a cluster of $m = 3$ Pentium III computers with 1Gbyte of the core memory. The operating system was Red Hat Linux 6.2. The test application was searching for a pattern in a binary file. Communications were done on the basis of a socket library. Fig. 11 shows processing time vs. V/m, for out-of-core computations using virtual memory, and multi-installment processing with chunk sizes 1E3, 1E4, 1E6, and 1E8. A dotted line representing the linear part CV of the communication time has been added as a reference line. The

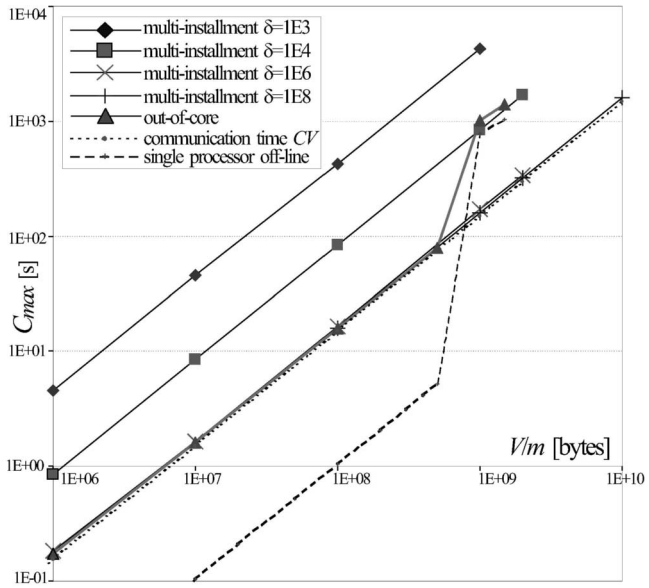

Fig. 11. Processing time versus $\frac{V}{m}$ for multiinstallment and out-of-core computations on a cluster of Linux PC computers.

dashed reference line at the bottom is the computation time on a single processor working offline. In the out-of-core processing, the use of virtual memory is evident when the load assigned to a processor exceeds the core memory size. In multiinstallment mode, the processing time is even worse than the out-of-core processing for $\delta = 1\text{E}3$ because communication overheads dominate. Increasing the chunk size δ reduces the total processing time, but only to the limit of the communication time required to scatter the load. Therefore, the lines for $\delta = 1\text{E}6$ and $\delta = 1\text{E}8$ overlap.

We shall conclude this section with an observation that multiinstallment processing can outperform out-of-core computations if the proper size of the load chunk is chosen.

5 Conclusions

In this work, we have proposed a new mathematical model for distributed processing divisible loads. The model is based on linear programming and is capable of representing piecewise linear convex processing time functions of the assigned load. In particular, systems with memory hierarchy can be represented in this way. The influence of the model parameters on the performance of the computing system has been studied. The efficiency of processing divisible loads in multiinstallment and out-of-core modes were compared. Multiinstallment processing appears to be advantageous for reasonably selected load chunks sizes.

Appendix

Notation Summary

$A_{ij1} + xA_{ij2}$: computing time for load size x, on processor P_i, memory level j in a heterogeneous system.

$A^l_{i1} + xA^l_{i2}$: computing time for load size x, on processor P_i, core memory level in a heterogeneous system.

$A^h_{i1} + xA^h_{i2}$: computing time for load size x, processor P_i, out-of-core memory level in a heterogeneous system.

$A^l_1 + xA^l_2$: computing time in core memory for a homogeneous system.

$A^h_1 + xA^h_2$: out-of-core computing time for a homogeneous system.

C: transfer rate in a homogeneous system.

C_j: transfer rate of the link to processor P_j in heterogeneous system.

C_{max}: schedule length.

δ: size of the load chunk in multiinstallment processing.

m: number of processors.

S: communication startup time in a homogeneous system.

S_j: communication startup time of the link to processor P_j in heterogeneous system.

swp: swap point in a homogeneous system.

swp_j: swap point of processor P_j.

V: total size of the load.

Acknowledgments

The authors would like to thank the referees for their suggestions on improving the quality of this paper. This research was partially supported by a grant of the Polish State Committee for Scientific Research.

References

[1] C. Bajaj, V. Pascucci, D. Thompson, and X.Y. Zhang, "Parallel Accelerated Isocontouring for Out-Of-Core Visualization," *Proc. IEEE Parallel Visualization and Graphics Symp.*, pp. 97-104, http://www.ticam.utexas.edu/CCV/papers/papera1.pdf, 1999.

[2] M. Berkelaar, `lp_solve`—Mixed Integer Linear Program Solver, ftp://ftp.es.ele.tue.nl/pub/lp_solve, 1995.

[3] V. Bharadwaj and G. Barlas, "Access Time Minimization for Distributed Multimedia Applications," *Multimedia Tools and Applications,* vol. 12, no. 2/3, pp. 235-256, 2000.

[4] V. Bharadwaj, D. Ghose, and T. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems," *Cluster Computing,* vol. 6, no. 1, pp. 7-17, 2003.

[5] V. Bharadwaj, D. Ghose, and V. Mani, "Multiinstallment Load Distribution in Tree Networks With Delays," *IEEE Trans. Aerospace and Electronic Systems,* vol. 31, no. 2, pp. 555-567, 1995.

[6] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems.* IEEE Computer Soc. Press, Los Alamitos, Calif., 1996.

[7] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram, "Scheduling a Divisible Task in a 2-Dimensional Mesh," *Discrete Applied Math.,* vol. 94, nos. 1-3, pp. 35-50, 1999.

[8] J. Blazewicz, M. Drozdowski, and M. Markiewicz, "Divisible Task Scheduling—Concept and Verification," *Parallel Computing,* vol. 25, pp. 87-98, 1999.

[9] P. Brezany, M. Bubak, M. Malawski, K. Zajac, "Irregular and Out-of-Core Parallel Computing," *Proc. Conf. Parallel Processing and Applied Math.,* pp. 299-306, 2001.

[10] Y.-C. Cheng and T.G. Robertazzi, "Distributed Computation with Communication Delay," *IEEE Trans. Aerospace and Electronic Systems,* vol. 24, pp. 700-712, 1988.

[11] S.K. Chan, V. Bharadwaj, and D. Ghose, "Experimental Study on Large Size Matrix-Vector Product Computations Using Divisible Load Paradigm on Distributed Bus Networks," *Math. and Computers in Simulation,* vol. 58, no. 1, pp. 71-92, 2001.

[12] W.T. Corrêa, J.T. Klosowski, and C.T. Silva, "Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays," *Proc. Eurographics Workshop Parallel Graphics and Visualization,* http://www.cs.princeton.edu/omnimedia/papers/piwalk.pdf, 2002.

[13] M. Drozdowski, *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems.* Poznan Univ. of Technology Press, Series: Monographs, no. 321, http://www.cs.put.poznan.pl/~maciejd/txt/h.ps, 1997.
[14] M. Drozdowski and W. Glazek, "Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors," *Parallel Computing,* vol. 25, pp. 381-404, 1999.
[15] M. Drozdowski and P. Wolniewicz, "Experiments with Scheduling Divisible Tasks in Clusters of Workstations," *Proc. Euro-Par,* pp. 311-319, 2000.
[16] M. Drozdowski and P. Wolniewicz, "Divisible Load Scheduling in Systems with Limited Memory," *Cluster Computing,* vol. 6, no. 1, pp. 19-29, 2003.
[17] "Storage, Transformation, and Recomputation of Integrals,"Gaussian 98 Technical Info, http://www.gaussian.com/g98.htm, 2000.
[18] K. Ko, "Scheduling Data Intensive Parallel Processing in Distributed and Networked Environments," PhD thesis, Dept. of Electrical and Computer Eng., State Univ. of New York at Stony Brook, 2000.
[19] X. Li, V. Bharadwaj, and C.C. Ko, "Optimal Divisible Task Scheduling on Single-Level Tree Networks with Buffer Constraints," *IEEE Trans. Aerospace and Electronic Systems,* vol. 36, no. 4, pp. 1298-1308, 2000.
[20] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization.* New York: John Wiley and Sons, 1988.
[21] "Parallel Data-Intensive Algorithms and Applications," *Parallel Computing,* D. Talia and P.K. Srimani, eds., vol. 28, no. 5, pp. 669-860, 2002.
[22] S. Toledo, "A Survey of Out-of-Core Algorithms in Numerical Linear Algebra," *External Memory Algorithms,* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Am. Math. Soc., J.M. Abello and J. Scott Vitter, eds., http://www.math.tau.ac il/~stoledo/Pubs/oocsurvey.pdf, pp. 161-179, 1999.
[23] P. Wolniewicz, "Multi-Installment Divisible Job Processing with Communication Startup Cost," *Foundations of Computing and Decision Sciences,* vol. 27, no. 1, pp. 43-57, 2002.

Maciej Drozdowski received the MSc degree in control engineering in 1987, and the PhD degree in computer science in 1992. In 1997, he defended his habilitation in computer science. Currently, he is an associate professor at the Institute of Computing Science, Poznan University of Technology. His research interests include design and analysis of algorithms, complexity analysis, combinatorial optimization, scheduling, and computer performance evaluation. He is a member of the IEEE Computer Society.

Pawel Wolniewicz received the MSc degree in computer science from the Poznan University of Technology in 1997, and the PhD degree in computer science from the same university in 2003. Currently, he works for the Poznan Supercomputing and Networking Center, Poznan, Poland. His research interests include metacomputers, distributed environments, and scheduling.