Experiments with scheduling divisible tasks in clusters of workstations

Maciej Drozdowski[1]* and Paweł Wolniewicz[2]

[1] Institute of Computing Science, Poznań University of Technology, ul.Piotrowo 3a, 60-965 Poznań, Poland.
[2] Poznań Supercomputing and Networking Center, ul.Noskowskiego 10, 61-794 Poznań, Poland.

Abstract. We present results of a series of experiments with parallel processing divisible tasks on various cluster of workstations platforms. Divisible task is a new model of scheduling distributed computations. It is assumed that the parallel application can be divided into parts of arbitrary sizes and the parts can be processed independently on distributed computers. The experiments were performed on various applications and distributed computing platforms. Though practical verification of the divisible task scheduling model was the primary goal of the experiments also an insight into the behavior and performance of cluster computing platforms has been gained.

Keywords: Parallel processing, scheduling, divisible tasks, clusters of workstations.

1 Introduction

The first work analyzing divisible tasks seems to be [4]. It was motivated by the need of finding optimal balance between parallelism of computations and necessary communication in a linear network of intelligent sensors. Later it turned out that divisible task model is versatile vehicle to represent distributed computations in: linear arrays of processors, stars, buses and trees of processors, hypercubes, meshes, multistage interconnections (cf. monographs [1, 2, 5]). It was demonstrated that divisible task theory can be a valuable tool in performance evaluation of distributed computations. The results of experiments performed in a dedicated Transputer system [3] confirm the correctness of the theory predictions. This work is dedicated to a challenging goal of verifying divisible task model in contemporary parallel processing environments available to the masses.

The remaining parts of this paper are organized as follows. In the next section we formulate the problem of scheduling divisible task in star networks. In Section 3 we describe test applications. In Section 4 the way of experimenting and the results obtained are presented. In Section 5 the results are discussed and conclusions are proposed.

* The research has been partially supported by a KBN grant and project CRIT2.
Corresponding author.

2 Processing divisible tasks on star and bus topologies

In the divisible task model it is assumed that computations (or work, load, processing) can be divided into several parts of arbitrary sizes which can be processed on parallel processors. In other words, granularity of parallelism is fine because the work can be divided into chunks of arbitrary sizes. There are no precedence constraints (or data dependencies) because the parts can be processed independently on parallel processors. Applications conforming with divisible task model are e.g. distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; many problems of simulation, linear algebra and combinatorial optimization.

We assume that initially whole volume V of work that must be performed (or e.g. data to be processed) resides on one processor called *originator*. In the star (equivalently bus) interconnection, which is considered in this work, the originator activates other processors one after another by sending them some amount of load for processing. It is assumed that the load is sent to the processors only once. α_i denotes the amount sent to processor P_i, for $i = 1, \ldots, m$. The transmission time is equal to $S_i + \alpha_i C_i$, where S_i is communication startup time, and C_i is transfer rate. The time of processing α_i units of work on P_i is A_i. The units of V, S_i, C_i, and A_i can be e.g. bytes, seconds, and seconds per byte (twice), respectively. Communication rates and startup times can represent not only the network hardware but also all the layers of communication software between the user application modules. Having received its share of work, each of the computers immediately starts processing it and finally returns to the originator the results in the amount of $\beta(\alpha_i)$. $\beta(x)$ is an application-dependent function of the amount of results produced per x units of input data. In Fig.1 a Gantt chart with communications and computations in the star network is presented. In Fig.1a results are returned in the inverted order of activating the processors (which we will call LIFO), or (Fig.1b) in the same order in which processors obtained their work (FIFO).

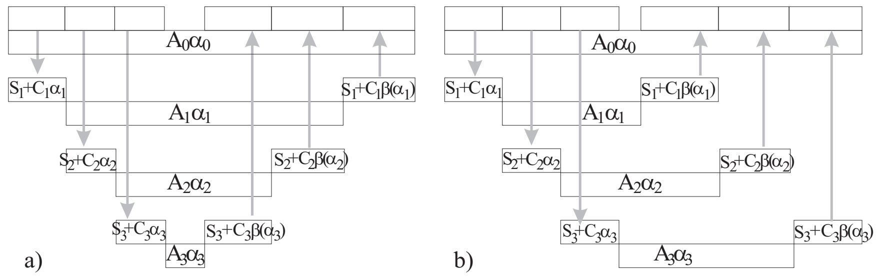

Fig. 1. Communications and computations in star. a) LIFO case, b) FIFO case.

Our goal is to distribute computations, i.e. find α_i, such that the duration of all communications and computations is minimal. Let us observe (cf. Fig.1a) that in the LIFO case processing on the processor activated earlier lasts as long as sending to the next processor, computing on it and returning the results. Using this observation we can formulate a set of linear equations from which distribution of the load can be found:

$$\alpha_i A_i = 2S_{i+1} + C_{i+1}(\alpha_{i+1} + \beta(\alpha_{i+1})) + A_{i+1}\alpha_{i+1} \quad i=0,\ldots,m-1 \tag{1}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{2}$$

P_0 denotes the originator. In the FIFO case (cf. Fig.1b) the time of processing on P_i and returning results from processor P_i is equal to the time of sending to P_{i+1} and processing on P_{i+1}. Hence, distribution of the work can be calculated from equations:

$$\alpha_i A_i + S_i + \beta(\alpha_i)C_i = S_{i+1} + \alpha_{i+1}(C_{i+1} + A_{i+1}), \quad i=1,\ldots,m-1 \tag{3}$$

$$\alpha_0 A_0 = \sum_{i=1}^{m}(S_i + \alpha_i C_i) + \alpha_m A_m + S_m + C_m\beta(\alpha_m) \tag{4}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{5}$$

Due to specific structure the above two equation systems can be solved in $O(m)$ time. However, they may have no feasible solution (because some $\alpha_i < 0$) when volume V is too small and not all m processors are able to take part in the computation. In this case less processors should be used.

3 Test applications

3.1 Search for a pattern

The problem consists in verifying whether some given sequence S of characters contains substring x. If it is the case the position of the first character in S matching x must be returned as a result. Having calculated quantity α_i of data the originator sends to processor P_i amount of $\alpha_i + \texttt{strlen}(x) - 1$ bytes from the sequence S starting at position $\sum_{j=1}^{i-1}\alpha_j + 1$. The chunks overlap in order to avoid cutting substring x placed across the border of two different chunks. As the files for the tests were known the amount of returned results was also known. $\beta(x) \approx 0.005x$ which is typical of search in databases holding e.g. surnames and personal data.

3.2 Compression

In this application originator sends to the processors parts of a file. The part sent to processor P_i has size α_i. Each of the processors compresses the obtained data using LZW [10, 11] compression algorithm. The resulting compressed strings are returned to the originator and appended to one output file. The original file can be obtained by decompressing each part in turn. The achieved compression ratio determines the amount of returned results. It was measured that $\beta(x) = 0.55x$. The compression ratio and speed depend on the contents and size of the input. In order to eliminate (or at least minimize) this dependence only parts of at least 10kB were sent to the processors for remote compression.

3.3 Join

Join is a fundamental operation in relational databases. Suppose there are two databases: A e.g. with a list of supplier identifiers, names, addresses etc., and B with a list of products with names, prices, etc. and supplier identifier. The result of join operation on A and B should be one file with a list of suppliers (names, addresses, etc.) and the products the respective supplier provides. The join algorithm can be understood as calculation of cartesian product $A \times B$ of the two initial databases. $A \times B$ can be viewed as a 2-dimensional array in which one row corresponds to one record a_j from file A and one column corresponds to one record b_k from database B. On the intersection of row a_j and column b_k pair (a_j, b_k) is created. (a_j, b_k) is transferred to the output file only if the fields of the supplier identifier match. In our implementation of distributed join, one of the databases (say A) was transmitted to all processors first. Then, the second database (B) was cut into parts B_i according to the calculated voulmes α_i, and sent to processors P_i $(i = 1, \dots, m)$. Each of the processors calculated join on A and B_i and the results were returned to the originator. Databases A and B were artificially and randomly generated, therefore the amount of results was known. β expressed the ratio of the amount of results and database B size.

3.4 Graph coloring and genetic search

Consider graph $G(V, E)$, where V is a set of vertices, and $E = \{\{v_i, v_j\} : v_i, v_j \in V\}$ is a set of edges. Node coloring problem consists in assigning colors to the nodes such that no two adjacent nodes v_i, v_j have the same color. More precisely, node coloring is a mapping $f : V \to \{1, \dots, k\}$, where $\{v_i, v_j\} \in E \Rightarrow f(v_i) \neq f(v_j)$. Find minimum k, i.e. chromatic number χ_G.

Determining graph chromatic number is a hard combinatorial problem, therefore heuristic algorithms are suitable for solving it approximately. Genetic search [8] is one of the metaheuristics applicable with this respect. In our implementation of the genetic search each solution is a gene represented by a string of colors assigned to the consecutive nodes. Good solutions from the initial population are combined using genetic operators to obtain a new population. The measure of solution quality is called fitness function which in our case was the number of the colors used plus the number of infeasibly colored nodes. Two genetic operators were used to obtain new 'individuals': crossover and mutation. Crossover is a binary operator exchanging tails of the strings in two genes starting at a randomly selected place. The effect of the crossover and selecting good solutions to produce offsprings is that certain properties of the initial solutions can be implicitly identified and combined. Mutation operator makes random changes in the individuals and diversifies the population. Solutions were selected to produce offsprings with probability increasing proportionally to decreasing of the fitness function (note that we have minimization). Originator generated the initial population of 1000 random solutions (genes). This population was distributed among the processors according to the calculated values of α_i's. Each processor created a fixed number of new generations and returned final population to the originator. Thus, $\beta(x) = x$. The gene with feasible coloring and the smallest number of colors used was selected as a final solution.

4 The results

In this section we outline results obtained in the experiments (cf. also [6, 7, 9]). We experimented over several years on several different hardware and software platforms. Due to time and workforce limitations not all applications were performed on every considered platform. Some platform did not exist (Java) at the time we started. In Table 1 we summarize which application was tested on which platform. All experiments were made on Ethernet network. Abbreviation ded. stands for dedicated single-segment interconnection, and pub for public, multisegment network.

Table 1. Platforms vs applications

application→ (year)platform	search for a pattern	com- pression	join	coloring
A: (1995) 7 heterogeneous Sun workstations: SLC, IPX, SparcClassic, PVM, ded.10Mb	yes			
B: (1997) 6 heterogeneous PCs: 486DX66, RAM 8M - P166, RAM 64M, Linux, PVM, pub.10Mb		yes		
C: (1997) 7 nodes of IBM SP2, PVM, ded.45Mb		yes		
D: (1999) 6 homogeneous PCs: P-133, RAM 64M, WinNT, MPI, ded.100Mb	yes	yes	yes	
E: (1999) 4 heterogeneous PCs:P-100, RAM 24M - Celeron-330, RAM 64M, Win98, Java, pub.10Mb				yes
F: (1999) 6 homogeneous PCs: P-200MMX, RAM 32M, Linux, Java, ded.100Mb				yes

The main goal of the experiments was to apply divisible task model in practice and to verify correctness of its predictions. The verification was done by comparing the real and the predicted execution times of some application when data is distributed in chunks of sizes (α_i's) calculated from equations (1)-(5). To formulate any of the above equations we needed data, i.e. parameters A_j, C_j, S_j for $j = 1, \ldots, m$. Therefore, we had to measure these parameters first. The communication parameters were measured by a ping-pong test. Originator sent to a processor some amount of data. The processor immediately returned these data. A symmetry of the communication links was assumed and half of the total bidirectional communication time was taken as the unidirectional communication time. The time of the communication and the amount of data were stored. After collecting a number of such pairs (for various sizes of the message), parameters S_j, C_j were calculated using linear regression. Processing rate A_j was measured as an average of the ratios of the computation time and the quantity of data processed. The method of obtaining $\beta(x)$ has been explained in the previous section. The measured communication parameters are presented in Table 2. Standard deviations are reported after the $\pm$ sign. The last two rows apply to the same hardware suite as for platform F. These data were obtained in some other set of experiments. Table 2 requires some comment and explanation.

Table 2. Typical values of communication parameters

(year)platform	$C_j[\mu s/B]$	$S_j[\mu s]$
A: (1995) various Sun workstations: SLC, IPX, SparcClassic, PVM, ded.10Mb	70.7±0.3	636000±86000
B: (1997) various PCs:486DX66,RAM 8M - P166, RAM 64M, Linux, PVM, pub.10Mb	7031±13	2861±9312
C: (1997) IBM SP2, PVM, ded.45Mb	68.6±0.1	205±144
D: (1999) homogeneous PCs: P-133 RAM 64M, WinNT, MPI, ded.100Mb	1.04±0.13	6200±7200
(1999) homogeneous PCs: P-200MMX, RAM 32M, Linux, PVM, ded.100Mb	18.3±0.2	11000±25000
(1999) homogeneous PCs: P-200MMX, RAM 32M, WinNT, PVM, ded.100Mb	15.9±0.3	248000±35000

Firstly, these numbers may differ from system to system and from implementation to implementation. Thus, they should be understood rather as indicators than the ultimate truth about communication performance. The measurements were taken on unloaded computers (no other user applications were running). The values represent one pair of communicating computers. We do not report results for the Java platform because there is no permission of the software producer.

In Table 3 examples of typical processing rates (A_j) are given. Note, that these values not only depend on the raw speed of the hardware or the operating system, but also on the application, its implementation, and run-time environment. All results refer to a single computer. Though values A_j are often greater than C_j it does not mean that parallel processing is less efficient than sequential execution because in most of such cases a single computer is not able to hold equivalent quantity of data as distributed computers and process it with the same speed as for small amounts of data.

Due to space limitations we present only a selection of results. The contents of the following diagrams is organized as follows. Difference between the expected execution time and the measurement divided by the expected execution time

Table 3. Examples of processing rates (A_j)

(year)platform	application	$A_j[\mu s/B]$
A: (1995) Sun SLC, PVM	search for a pattern	6.99±0.03
B: (1997) PC 486DX66,RAM 8M, Linux, PVM	compression	1500±20
C: (1997) IBM SP2, PVM	compression	650±60
D: (1999) homogeneous PCs: WinNT, P-133, RAM 64M, MPI	search for a pattern	0.838±0.007
D: (1999) homogeneous PCs: WinNT, P-133, RAM 64M, MPI	join	1176±6

(i.e. relative error) is presented on the vertical axis. The horizontal axis shows the size of the problem. In Fig.2 results of the "search for a pattern" application on platform D are shown. In all cases real running time was longer than the expectation. For platform A the results were similar. The difference is stable and around 35% in LIFO case. In the FIFO case the difference has bigger variation, and grows slightly with V from approx. 25% to 30%. In Fig.3 results of the "compression" application on platform C are shown. Real running time was longer than the expectation. The LIFO/FIFO orders of returning results have similar behavior as in the previous application. LIFO case is more stable and real execution times oscillates around 10.5%. In FIFO case difference is growing with the size of the problem from approx. 6% to approx. 13%. For the same application on platform D relative error was decreasing from approx. 55% to approx. 7% with V increasing.

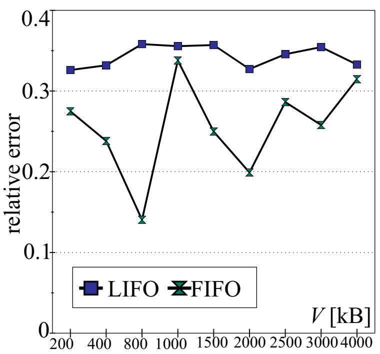

Fig. 2. Difference between model and measurement on platform D in "search for a pattern" application.

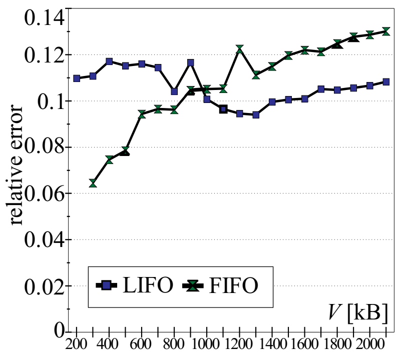

Fig. 3. Difference between model and measurement on platform C in "compression" application.

In Fig.4 relative error for "join" application on platform D are displayed. In both LIFO and FIFO cases the difference decreases from approx. 40% to less than 0.5%. Intuitively, it seems reasonable that there should be a good coincidence between the expectation and the measurement for big values of V, because processing and communication times are long and transient (or "noisy") effects are compensated for. In Fig.5 relative difference between the model and experiment for "coloring" application on platform F is shown. As it can be seen with growing volume V the relative error decreases from approx. 30% to less than 5% and then increases to approx. 30%. Real execution time was longer up to 30kB, and from 40kB on it was shorter than the expectation. As these results differ from the previous ones further critical verification is needed.

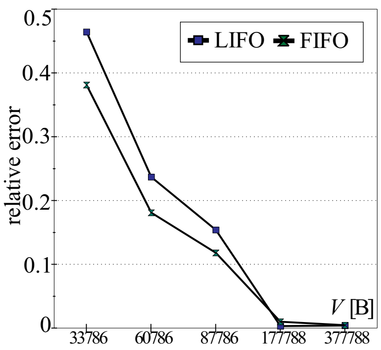

Fig. 4. Difference between model and measurement on platform D in "join" application.

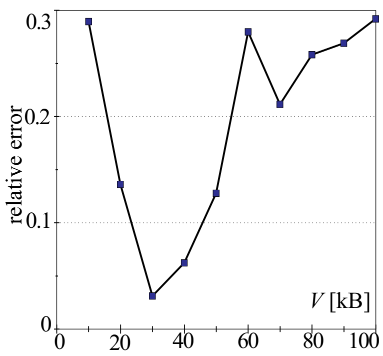

Fig. 5. Difference between model and measurement on platform F in "coloring" application LIFO case.

5 Discussion and conclusions

Let us observe that in most of the cases relative difference between the model and the measurement is $\approx$30% and less. We believe that the coincidence of the model and experimental results can be improved if more effort is devoted to better understanding the computing environment, and more carefully setting up the experiments (e.g. if we have more control on the computer software suite). On the other hand differences below 10% (cf. Fig.3 and Fig.4) indicate that there are applications and platforms where the divisible task model is accurate.

It can be observed that the more uniform system we used and the more the system was dedicated to our experiments, the better the coincidence with the model was. Basing on the experience we gained it can be observed that calling operating system and runtime environment services is one of error sources in our results. For example, references to disk files or memory allocation procedures introduces great amount of uncertainty and dependence on the behavior of other software using the computer. This was also the case for long messages for which the efficiency of communication decreased as soon as the message size exceeded free core memory size. Virtual memory was used by the operating system to hold big data volumes to be communicated. In such situations assumption about linear dependence of the communication time on the volume of data was not fulfilled, and communication speed decreased with data size. This observation applies also to the dependence of processing time on the volume of data: in wide ranges of data sizes the assumption on linearity of this function may be not satisfied. Distribution of the results can be another reason for disagreement of the real running time and the expectation. This applies e.g. to "search for a pattern" and "join" applications. In the model distribution of the results is uniform and any fraction of the total volume of data induces some results. This may not be the case in reality because records or text patterns may be abundant in data for one processor, and may be absent from the data for another processor. Our experiments were performed on Ethernet network. The access time to this kind of network is not deterministic. Also the software running in parallel with our programs (e.g. interrupt drivers, operating system in general) causes that processing speed is not stable. As a result both communication and computing parameters include some amount of uncertainty, which can be estimated by the value of these parameters standard deviation. The standard deviation of C_j and A_j parameters on most of the platforms was less than approx. 0.01. Deviation of startup time parameters (S_j) is much bigger, even as much as 3.3 in case of platform B.

In this work we presented results of experiments on applying divisible task theory in practice. It has been demonstrated that the model is capable of accurately describing the reality. There are also cases when the predictions of the model are not satisfactory yet and these cases should be a subject of further analysis. The collected experience gives rise to a better understanding of distributed processing environments and may help improve the theoretical models. A static and single-chunk distribution of work was assumed. In everyday practice a dynamic on-line algorithm would be more welcome. Proposing and analysing such algorithms can be a subject of the future research.

References

1. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T., *Scheduling divisible loads in parallel and distributed systems*, IEEE Computer Society Press, Los Alamitos CA (1996).
2. Błażewicz, J., Drozdowski, M., Ecker, K., Management of Resources in Parallel Systems, in: Błażewicz, J., Ecker, K., Plateau, B., Trystram, D. (eds.), *Handbook on Parallel and Distributed Processing*, Springer, Berlin-Heidelberg (2000) 263–341.
3. Błażewicz, J., Drozdowski, M., Markiewicz, M., Divisible task scheduling - concept and verification, *Parallel Computing* **25** (1999) 87–98.
4. Cheng Y.-C., Robertazzi, T.G., Distributed computation with communication delay, *IEEE Transactions on Aerospace and Electronic Systems* **24** (1988) 700–712.
5. Drozdowski, M., *Selected problems of scheduling tasks in multiprocessor computer systems*, Poznań University of Technology Press, Series: Rozprawy, No.321, Poznań, (1997). Available also from: `http://www.cs.put.poznan.pl/~maciejd/h.ps`
6. Drzewiecki, D., Drozdowski, M., Rozdział obciążeń w rozproszonym systemie komputerowym metodą zadania jednorodnego, Technical Report RB - 008/97 of the Institute of Computing Science, Poznań University of Technology, Poznań (1997).
7. Kemp, M., Przetwarzanie w sieci Internet metodą zadania jednorodnego, M.Sc. Thesis, Institute of Computing Science, Poznań University of technology, Poznań (1999).

8. Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin (1992).
9. Piotrowski, A., Zastosowanie metody zadnia jednorodnego w rozproszonym systemie komputerowym, M.Sc. Thesis, Institute of Computing Science, Poznań University of Technology, Poznań (1999).
10. Welch, T.A., A technique for high-performace data compression, *IEEE Computer* **17** (1984) 8–19.
11. Ziv, J., Lempel, A., Compression of individual sequences via variable-rate coding, *IEEE Transactions on Information Theory* **24** (1978) 530–536.