Dynamic Load Balancing for Declarative Distributed Programming Environments*

Małgorzata Steinder
e-mail: gosia@ics.agh.edu.pl
Institute of Computer Science
University of Mining & Metallurgy
Al. Mickiewicza 30, Cracow, Poland

November 16, 1994

Abstract

Execution speed of parallel programs implemented in declarative languages does depend on load conditions. By proper distribution of tasks we can dramatically improve their performance. In this paper we present how such a distribution may be found by means of load balancing introduced into PCN environment. We show that peculiarities of PCN make it profitable to apply statistics to load balancing decision making policies, and how these policies impact on PCN programs performance.

1 Introduction

The load balancing is a very important factor in achieving a good performance in distributed systems where tasks are created in dynamic fashion. However, very little has been done so far to introduce such mechanisms into declarative programming environments.

The execution of a declarative concurrent program consists in creating and processing many light processes in parallel or sequential way, which is done by communicating runtime systems running either on the same or on different processors. These environments are usually equipped with tools that enable computing the location of tasks during execution of the program. This makes the dynamic load balancing feasible.

There are several factors that should be taken into account when designing load balancing mechanisms for the declarative distributed program. The most important of them are: the overhead imposed by the process of task distribution, granularity required to make an impact of tasks distribution costs negligible, speed of execution dependence of load conditions.

In this paper we describe how LB mechanisms may be introduced into PCN programs. Apart from implementing traditional techniques, which proved useless for many PCN applications, we have put forward a semi-stochastic function based on the information about the state of the system.

The rest of this paper is structured as follows: in the next Section we mention computational gain resulting from PCN parallelism and the overhead it imposes; the third Section defines problems which are struggled with when designing LB for this system; Section four presents decision making policies, we have chosen as the most appropriate for PCN; results of their empirical comparison are presented in Section five; this is followed by conclusions.

*This work was sponsored by Polish Research Council (KBN), under grant no.: 8 S503 015 06.

2 Performance of PCN distributed applications

PCN [1, 2, 3] is equipped with several mechanisms making it possible to write parallel applications. Who uses them must be aware of the overhead they impose. The following factors contribute to this overhead:

- communication between runtime systems,
- relocation of a task to another node,
- switching among tasks,
- marshaling parameters, which are characteristic of PCN (tuples, lists).

Impact they have on the performance depends on the granularity of tasks which are distributed [7]. The figure below (figure 1) shows this dependence. One may observe that the high performance may

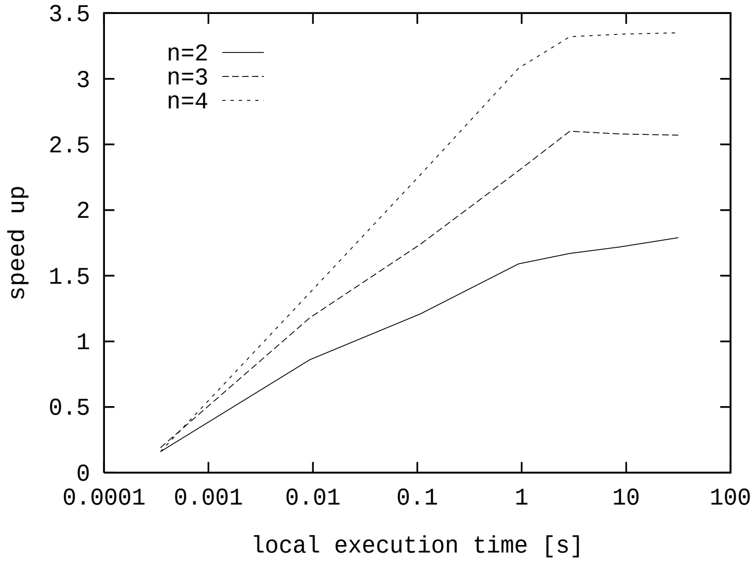

Figure 1: Speed-up versus tasks granularity

only be achieved, if relatively large tasks are being moved around. The cost of relocating small ones may increase their execution time.

3 Background of load balancing for PCN

Load balancing is composed of two phases: information gathering and decision making. When introduced to PCN, they must suit its computational model idiosyncrasies. Each time one runs a PCN program, it is PCN runtime system that is really executed. This runtime system is responsible for processing transformed code of the application. Regardless of how different codes are implemented, the same program is run, and the only thing that changes is its data. The increases of load and CPU time caused by diverse PCN programs are similar (differences result, for example, from using IO operations). They do not make it possible to observe effects of tasks relocations. The performance of PCN program does, however, depend on load conditions. By proper reordering of tasks it is possible to considerably increase execution speed. This is why we bother with load balancing for PCN. We strive to make all nodes be steadily busy, and cease their work at the same time. However, rather than with a set of identical and uniformly l loaded processors, we cope with differing in speed and load ones. This has to be taken into account, when concerning decision making policy. We should not take pattern of decision making strategies designed for environments deprived of the notion of runtime system, because they usually rely on the fact that migration of program, procedure or object does change the state of the processor to which it is moved [8, 6]. It is not the case in the PCN system; in the absence of external circumstances that change the load, we are in danger of relocating many tasks to the same processor, which may result in worse performance than, if they were executed sequentially.

PCN programs, in terms of complexity, vary from ones consisting of a few large, sequentially executed processes to ones involving simultaneous creation of thousands of small tasks. The load balancing must cover all this cases, this is why different policies should be developed.

Dynamic load balancing has to adapt itself to the altering system state, which necessitates possession of as actual and easy to obtain as possible information about it. Each time the call `proc()@node(...)` is invoked, there is a need to access data about the load conditions. We have decided to describe it by means of load1 Unix system parameter which may be taken from `rstat()` rpc service [4] or from ANSA load balancing monitor [5]. We propose four techniques of gathering information:

- on demand; it is obtained each time the map function (`node(...)`) is executed,
- periodically; it is updated from time to time,
- on groups of functions demand; it is updated, when is to be used by more than a specified number of functions,
- on demand when not up-to-date; it is obtained on demand, when the accessible information is no longer actual - the specified period of time has passed.

The first proposal is suitable, when a program is sequentially composed of large procedures; during their execution the state of the system may change, so we would rather the next procedure to use the fresh information. This approach would be however of no avail, if we had to cope with frequent map function calls. The overhead introduced by so often access to information about system's state would be at variance with the sense of load balancing. However, the second above mentioned technique seems appropriate in this case. Unfortunately, it is possible for PCN program to insist on relocating hundreds of newly created tasks almost simultaneously, and then execute them for a long time. Keeping the process of updating the information active during that period, might be expensive, and even, if the state of the system noticeably changed, the tasks ordering would stay the same. This might lead to the degradation of performance. The third technique seems to solve this problem; tasks are relocated in groups, only if t he new information is present, the next group may be served. The last proposal is reasonable, when we know nothing about how the tasks will be created. It is then better to obtain information on demand regarding it, however, as actual for a specified period of time.

4 Decision making strategies

In this section we present three methods, we have chosen as the decision making strategies for load balancing in PCN. Having been given the actual information about the state of the system the decision making strategy must answer the question: to which node should the specified task be relocated. This has to be done each time the map function issues such a request. The most load balancing policies designed for other environments choose the best node (the least loaded one). This approach seams reasonable in PCN case, when the program consists of several large procedures which, in addition, execute in sequential way. Then, if each of them is processed on the least loaded processor, we are guaranteed the highest performance possible. Nevertheless, if the same procedures were executed in parallel block, mapping them to the best would mean mapping them to the same processor. This is not what we expect. Therefore we have developed the 'good nodes' strategy. It relies on choosing the processors whose load does no t exceed the specified value. This way overloaded processors are eliminated. Then, we cyclically scan the ordered list of nodes beginning from the best one, which insures almost equal distribution of tasks among good processors.

There is possible a situation, when a great deal of tasks are to be mapped simultaneously. Using the 'good nodes' strategy is a kind of solution, we think, however, that a number of tasks being moved to each processor should be proportional to its computing capacity, by which we mean how fast processor can execute a program with regard to its load and innate speed. We have based our study on the knowledge how execution speed of PCN programs depends on load conditions. This relation was investigated for different applications on different computers. The figure 2 presents results obtained. One should not be

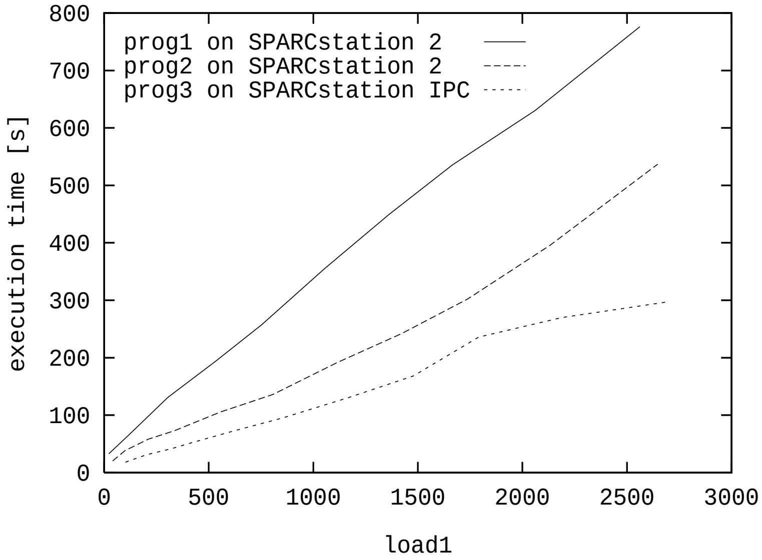

Figure 2: PCN programs execution time dependence of load conditions

surprised with the shape of the curves, since they show execution times of the same program (runtime system). Regardless what PCN programs are being taken into account, their execution times versus load conditions can be approximated with lines having one point in common. We have estimated its coordinates at $(-70, 0)$. We would like each of these lines to delineate the $y = ax$ equation. This can be done by geometrical transformation. By this we have achieved proportionality between load and time taken to execute a task. We suggest, that the number of tasks moved to particular nodes should be reversely proportional to their load. For example, if $l_0, l_1, \ldots, l_n$ denote the loads of processors, then $m_0, m_1 = \frac{l_0+l_s}{l_1+l_s} \cdot m_0, \ldots, m_n = \frac{l_0+l_s}{l_n+l_s} \cdot m_0$ ($l_s = 70$) stand for amounts of tasks being assigned to them respectively. By normalization they can be converted into probabilities:

$$p_0 = k_0 \frac{1}{(l_0 + l_s) \sum_{i=0}^{n} \frac{1}{l_i+l_s}}, p_1 = k_1 \frac{1}{(l_1 + l_s) \sum_{i=0}^{n} \frac{1}{l_i+l_s}}, \ldots, p_n = k_n \frac{1}{(l_n + l_s) \sum_{i=0}^{n} \frac{1}{l_i+l_s}}$$

where k_i denotes a constant, which should be introduced when processors differ in speed. Having been given n processors we have divided subset of real numbers $< 0, 1 >$ into n partitions: $< 0, p_0 >$, $(p_0, p_0 + p_1 >$, ..., $(p_0 + p_1 + \ldots + p_{n-1}, 1 >$. An index of partition, the random value from $< 0, 1 >$ belongs to, is the number of processor that will be chosen.

5 Experimental results

In the previous section we have described three decision making strategies for load balancing in PCN. Below, we present results of experiments which show their effect on PCN performance.

The 'best node' strategy was tested by means of a simple program consisting of a hundred consecutive procedure calls. First, this program was executed on one node and contained no mappings. Then, it was run on three similar processors, and each procedure was mapped to the best node. You can see the results in the table below. With the aim of testing 'good nodes' strategy we placed eight identical

Mapping strategy	Time [s]	Mean loads		
		Processor 0	Processor 1	Processor 2
No mapping	1361	177	-	-
'Best node'	986	173	81	30

procedures in a parallel block. They were mapped firstly to the random node, then the decision making policy was involved. The execution time of the whole block was measured. The experiment was repeated several times. Results are presented below. Finally, when a hundred subsequent procedure calls were

Mapping strategy	Time [s]	Mean loads		
		Processor 0	Processor 1	Processor 2
Random	47	105	75	227
'Good nodes'	31	90	106	207

being invoked, we applied 'random node', 'good nodes' and semi-stochastic strategy. This was repeated for three different procedures (the objective was to check the behaviour of the balanced programs when diverse procedure skeletons, with regard to their complexity and communication scheme, were being used).

- *proc* - summing 20000 natural values,
- *circ* - creating 20 processes exchanging their information a specified number of times,
- *circ1* - creating 20 processes exchanging their information a random number of times.

Execution times are presented in the table below.

6 Summary and suggestions for further study

In the previous sections we proposed several solutions to the problem of equipping PCN with load balancing mechanisms. They embraced traditional techniques suitable for other distributed environments as well as the semi-stochastic function based on the actual information about the state of the system

Procedure name	Time [s]	Mean loads			Mapping strategy
		Processor 0	Processor 1	Processor 2	
proc	301	200	18	320	Random
	280	201	37	294	'Good nodes'
	148	199	66	295	Semi-stochastic
circ	78	178	33	317	Random
	76	138	24	201	'Good nodes'
	52	190	52	307	Semi-stochastic
circ1	78	190	43	315	Random
	70	187	50	289	'Good nodes'
	50	237	45	296	Semi-stochastic

as an individually designed strategy for declarative distributed languages. Experiments described in section 5 proved that dynamic load balancing may have a dramatic effect on the performance of PCN programs.

Nevertheless, we think they should be improved.

During our research we aimed at adjusting the way tasks are being assigned to the processors to their speed and load conditions. Knowing that it partly simulates the processor we would like the runtime system to be applied the same approach. As it was presented earlier, its speed depends on the speed and load of the processor on which it is executed. We showed that it was a linear dependence. It is not, however, certain how load of the runtime system should be expressed. Perhaps, the number of processes in the process pool would be the best solution. Unfortunately, PCN does not give access to such information. We have to seek for another clue.

If we solve this problem, it will be possible to develop load balancing mechanisms which adapt themselves not only to the state of the processors, but also to the internal state of the PCN system. Since the runtime system is able to observe the effects of tasks relocations, we think the decision making policies should be capable of learning from their experience, which involves use of heuristic. That should result in the further increase of performance.

References

[1] I.Foster, S.Tuecke. *Parallel Programming with PCN*, Technical Report, Argonne National Laboratories, 1992

[2] H.C.Cunningham. *Notes on Concurrent Programming with PCN*, 1992

[3] I.Foster, R.Olson, S.Tuecke. *Productive Parallel Programmimg:The PCN Approach*, Scientific Programming, Vol.1, 51-66, 1992

[4] SunOS Reference Manual

[5] A.Uszok, K.Zieliński, A.Król. *Cluster Management Software for Open Object Oriented Systems* , Proceedings of HPCN'94, Lecture Notes in Computer Science, Springer-Verlag, April, 1994

[6] Sayed Atef Banawan *An Evaluation of Load Sharing in Locally Distributed Systems*, University of Washington, tech. report, August 1987

[7] M.Steinder, K.Zieliński. *Badanie efektywności systemu PCN w obliczeniach rozproszonych*, UMM, tech. report, November 1994

[8] O.Kremien, J.Kramer. *Methodical Analisis of Adaptive Load Sharing Algorithms*, IEEE Transactions on Parallel and Distributed Systems, Vol 3, No 6, Noveber 1992