Transactions on Information and Communications Technologies vol 9, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Scheduling distributed algorithms on heterogeneous computer networks

M. Alfano[a], A. Genco[a,b], G. Lo Re[b]
[a]*Dipartimento di Ingegneria Elettrica, Università di Palermo, Viale delle Scienze, 90128 Palermo, Italy*
[b]*Centro studi sulle Reti di Elaboratori, Consiglio Nazionale delle Ricerche, Viale delle Scienze, 90128 Palermo, Italy*

Abstract

This study deals with a distributed scheduling organization devoted to manage stand-alone and co-operating applications on a workstation network. To this end, it considers different aspects and requirements of the applications, the operating systems, and the distributed environment. In particular, it considers the scheduling activities devoted to the dynamic allocation of processes and proposes a model for differentiating local, global, long-term, medium term, and short-term scheduling. The authors provide a theoretical scheme where the above components are conveniently organized and propose a practical implementation for a network of Unix workstations using the PVM (Parallel Virtual Machine) facilities. In particular, the paper examines the problem of the performance evaluation of co-operating processes and proposes a solution based on delay detection and accusation. The scheduling solution mainly consists of a migration policy and a mechanism to be performed by the distributed components of the medium-term scheduler. This paper reports the results of some optimization problems carried out by parallel implementations as a case study.

1 Introduction

An important aspect to be considered when designing a scheduler for co-operating processes is the performance evaluation criterion. Parallel computation efficiency may be determined by many different parameters. They can be grouped in two main classes: those related to hardware performance and those related to software implementation design. However, if we analyse the parallel execution of co-operating processes, we can observe that, whatever be the causes that affect the application performance, they have a direct relation to the delays accumulated by the processes at the synchronization points.

The highest performance of a parallel application is reached when all the processes running towards the same synchronization barrier spend the same turnaround time to get there, with no interest if they spend less or more cpu time or have the same computational complexity. This consideration can also conveniently apply to parallel applications that should run unbalanced in homogeneous systems. In some cases the different working load of processors can balance the imperfect partitioning of the application.

The user requirements can be accomplished only by means of a suitable scheduling activity. For instance, we could consider a long-term scheduler devoted to perform a suitable initial allocation of the modules, a short-term scheduler for performance evaluation, and a medium term scheduler that performs dynamic re-allocation of modules when necessary. Such a scheduling structure would probably entail an operating system far too complicate. Conversely, a mixed solution that divides the scheduling code between the operating system and the applications can be conveniently considered. To this end, each machine running the local scheduler of its operating system (e.g., the Unix scheduler), will also host the local components of the distributed global scheduler and some application schedulers.

As for long term activity, the global scheduler accepts all the submitted jobs and decides where to allocate the related processes. It can select the local or a remote site according to the user requirements. The application schedulers evaluate the delays provoked by low performing processes and require scheduler interventions for tuning performance when the delay exceeds a given threshold.

Medium-term scheduling decides whether to migrate processes in order to improve the load balancing of the distributed system or to speed up a parallel application. An advantage of such an approach is that the user is also given the possibility of introducing the logic for detecting delays in the application code. This option can be very useful when modules work asynchronously and therefore no synchronization point is available for detecting delays. In this case, only the application logic can determine the current execution stage of the co-operating processes.

2 Distributed scheduling design

The aim of distributed systems is in general the optimal use of shared resources [4]. This optimal use can be intended as the minimization of resource idle time or in terms of the typical performance indexes of a scheduler such as throughput, wait time, response time, etc. These indexes, which are not the only ones to be pursued, are related to a global aim and are mostly measured in terms of average values. However, in many cases the user requirements may not agree with the global criterion of resource optimization or may contrast with the requirements of other users. If we consider that the global aim is something abstract and is determined by a trade-off of user goals, therefore, in general, the scheduler should perform different policies giving higher priority to user requirements than global performance.

To do this, two main aspects must be considered. One is the way of communicating user requirements to the scheduler. The other one is the accounting mechanism to apply different charging rates, for instance, higher rates to the user requirements that more contrast with the global objectives.

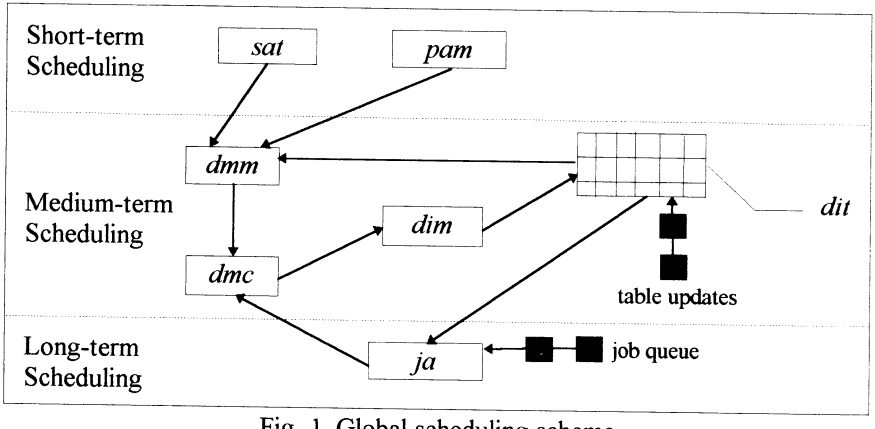

Fig. 1. Global scheduling scheme

Dealing with a heterogeneous network of workstations that run different operating systems, we can consider the scheduling structure of Fig. 1. All the local-scheduler components are parts of the local operating system and, as said above, no intervention can be made on the local scheduler in particular if the local operating system is Unix. For the global scheduler activities we can consider three different components related to long, medium, and short term scheduling.

2.1 Long-term scheduling

The *job acceptor* (*ja*) receives the *job-description file* with the user requirements and verifies that the resources needed to start the application are available in the distributed system. It decides an initial process configuration where each required replica is considered and assigned to a workstation. Next, it transmits this configuration to the migration agent in order to perform the initial process allocation. A template of the job-description file is reported in Table I.
The file begins with a JOB statement that contains the name of the application and a USER statement with the user identifier and its accounting information. The HOSTS statement specifies which workstations can be used for executing the application. The ANYWHERE option indicates that all the machines can be used. Next we have a set of entries for each module of the application. The MODULE statement contains the filename of one executable module. PLACE indicates if the module has to be allocated in a particular host. The AUTOMATIC option allows the scheduler to put the module in any host of the list specified by HOSTS. REPLICAS indicates the desired number of replicas for that module. OBJECTIVE specifies if replicas are requested for speedup, reliabilty, or application-design purposes. The last option (APPLICATION) is for applications where the number of co-operating processes is mandatory for

obtaining a determined result (e.g., scalable problems). COST indicates how much the user is willing to pay for the execution of this module. This cost can be expressed in terms of three values that correspond to the priority assigned to the module. The cost of the whole application is obtained considering the cost of all the component modules. A further option (REAL-TIME) is available if the local scheduler can be directed to manage a module as a real-time one. In this case the time constraints are to be considered by the *job acceptor* when checking the resources for the job. Of course, this requirement will entail the highest priority and the highest charging rate. MOVABLE indicates if the module can be transferred. RESTARTABLE indicates wheter the module can be restarted, for instance, when its host crashes and there is no possibility of correctly perform a migration. In this case, it can only be restarted on another host. The CONNECTIONS statement is followed by as many FROM-TO entries as the pairs of communicating modules.

Table. I : *Job-description file*

JOB:	Application name		
USER:	Userid - Account #		
HOSTS:	Hostlist (Host 1, Host 2, ..., Host n) / ANYWHERE		
MODULE:	Module name		
PLACE:	Hostname / AUTOMATIC		
REPLICAS:	n		
OBJECTIVE:	SPEEDUP / RELIABILITY / APPLICATION		
COST:	MINIMUM / MEDIUM / MAXIMUM / REAL-TIME		
MOVABLE:	YES / NOT		
RESTARTABLE:	YES / NOT		
MODULE:	...		
MODULE:	...		
...			
CONNECTIONS:			
FROM	Module name1	TO	Module name2
FROM	...	TO	...
...			

2.2 Medium-term scheduling

The medium-term scheduler is made up of a *distributed migration manager* (*dmm*), a *distributed migration carrier (dmc)*, and a *distributed information manager (dim)*. All of them are replicated on each workstation.

The *distributed migration manager* accepts the requests of performance adjustment that come from the *parallel-application monitors* (*pam*) and the *stand-alone applications tutor* (*sat*) running on the same machine. In the first

case, an application could require a process to run faster or slower in order to achieve a better synchronization between its modules. In the second case, a stand-alone application is running with performance that disagree with the user requirements. When the *dmm* receives a request of performance tuning, it looks for the possibillity of performing a suitable migration. In migrating one process the *dmm* considers a machine suitable for receiving a new process if its resident processes do not delay. Such a migration can directly apply to the process indicated by the *pam* or the *sat* in order to make it run on a faster or slower machine. Otherwise, the *dmm* can decide to migrate some other processes thus modifying the load distribution of the whole system. This action will indirectly modify the application performance. Other events that activate the *dmm* can be a change in the number of the connected hosts, or the termination of an application. The latter event is signaled by the the *sat* or the *pam*.

The *distributed migration carrier* has the task to carry out the migrations requested by *dmm* and *ja*. In practice, each *dmc* component receives the requests from the *dmm* and *ja* components running on the same machines. If the process to be migrated is running on the same machine, the *dmc* contacts its peer on the destination site in order to carry out the migration. Otherwise, it starts a communication for delegating its peer on the site where the migrating process is hosted. The *dmc* that actually performs the migration, has also the task of updating the status of the distributed system. It does not manage directly the scheduling tables but sends a request to the *distributed information manager* (*dim*). The *dim* will queue this request and perform the updates only when enabled by the access-control mechanism. Each upgrade to the local tables will be propagated by the *dim* to its peers running on the other machines.

2.3 Short-term scheduling

Short-term global scheduling is performed by two modules that are: the *parallel application monitor* and the *stand-alone applications tutor*. Both *pam* and *sat* are regarded as short-term schedulers because they continuously check the process performance.

There is a *pam* for each parallel application and it is devoted to collect information about the delays between processes. The *pam* receives delay warnings and requires a *dmm* intervention when the delays exceed a given threshold. On the contrary there is just one *sat* for each host and it is devoted to check the performance of stand-alone processes in order to verify their adherence to the user requirements. The adoption of a *pam* for each application has two justifications at least: complexity and reliability. The complexity is obviously reduced if the scheduler has to manage only a low number of processes and related synchronizations. Reliability is preserved because only the processes managed by a scheduler are stopped if the scheduler goes down.

3 Performance Evaluation

When some processes co-operate, they usually have to communicate. The communications can be performed either synchronously or asynchronously.

In the first case, a process running slowly may determine a wait condition on the receiving process. In the second case, no execution delay takes place, but some negative consequences may arise about the goodness of the results. In any case co-operating process should run in parallel at similar rates so as to arrive simultaneously at communication rendezvous. Therefore an important goal is to recognize the delays. When a process is forced to wait for a communication incoming from a slow process, it can send an accusation to the *pam*. The *pam* classifies slow and fast processes according to their performance. To implement the above policies we developed some algorithms: an algorithm to recognize delays and decide accusations (in the receive routines), another one to evaluate the performance of the process (in the *pam*), a further one to determine migration paths and finally an algorithm to prevent excessive migration activity (in the *dmm*). Our performance evaluation criteria take into account the process delays at communication rendezvous. The rendezvous concept is based on the consideration that, whatever be the number of the co-operating processes, each pair of them goes along a sequence of common dates and autonomous activities. Such activities can include computations and communications with other processes. Letting the two processes calculate the time elapsed between two successive common rendezvous represents a better solution. The sending process attaches this number to the message; the receiving process compares it with the one calculated by itself and accuses a delay to the scheduler if the difference exceeds a given threshold.

In the second case a process can start a *receive* from a list of processes. This is the situation where no importance is given to the waiting period of the receiver, but the sending process is expected to get to the rendezvous with the receiver at the same time. Therefore, on average, the receiving process should complete the list of the expected communications in whatever order, but without repetitions (two communications from the same process) before the completion of the list. If a repetition takes places, we can be sure that at least one process runs faster than the other ones. In this case, we can also assume that the processes not yet arrived at the rendezvous are the slowest ones. The receiving process accuses this circumstance to the *pam* that updates its performance tables. Each time the *pam* receives an accusation, it awards a prize of one point (+1) to the fastest process and a penalty of one point (-1) to each of the slowest processes. After a certain number of accusations this distribution of points becomes more evident and the *pam* can decide to request a *dmm* intervention. To this end, a threshold of points must be provided to determine when a process is too slow or too fast.

With regards to stand-alone applications, the *sat* of the machine where an application is running will check that the application runs satisfying the user requirements. To do this, for example, the user can supply the *job-descriptor file* with information related to the desired turnaround time for the apllication and the

CPU time required for it. In this case, at given time intervals the *sat* will check how much CPU time has already been spent by the application and what is the elapsed time so far. The *sat* will thus have an approximate idea whether the application is delaying or not. If this is the case, the *sat* will require the *dmm* to make the suitable decisions.

3.1 Enhancements to the operating system

We built our implementation on the basis of the PVM (Parallel Virtual Machine) [1] system. This is a free software allowing UNIX systems to be joined in a distributed environment. It supplies a wide collection of primitives for creating process at any connected host and primitives for communication, synchronization and monitoring. It allocates processess at their creation time and does not supply any dynamic migration facility.

We implemented our extended primitives as wrappers of the PVM corresponding ones, in addition to some new functions. They mainly concern:

- process naming;
- creation of process;
- message packing;
- communication;
- synchronization;
- migration.

Furthermore we implemented a pre-processor in order to manage the entry points of a migrated process. This pre-processor inserts some additional codes to the source of each module. In particular, it adds a label to each statement following a migration primitive. These are the only points where a process can restart from, after its migration. To this end, the pre-processor also inserts a prologue code able to perform a jump to the correct entry point.

4 Case Study

We are currently testing our migration system on a simulated annealing application [2][3]. The computing environment used for the practical experiments is made up of the following workstations:

ALPHA	DEC 3000 AXP 600 with OSF/1 v. 1.3.
POWER	IBM RISC/6000 7011 25T with AIX v. 3.2.5.
SUNIPA	SUN SPARCstation 10/512 biproc. with SUNOS v. 5.3
CUCAIX	IBM RISC/6000 520 with AIX v. 3.2.3.
SUNCUBE	SUN SPARCstation IPC with SUNOS v. 4.1.

Some machines have a fast processor but slow I/O channels or vice-versa. However, limitedly to periods with a roughly stable working load of the host processors, the migration strategy gives good results achieving a virtual load balancing of the parallel processes; they get the rendezvous practically at the same time.

Table. II : *Process distribution of the simulated annealing solution to the job-shop scheduling problem*

	ALPHA	POWER	SUNIPA	CUCAIX	SUNCUBE
(40, 20)	40	29	18	9	4
(40, 20)	48	23	17	9	3
(40, 20)	41	29	18	9	3
(40, 20)	43	27	18	8	4
(40, 20)	41	28	17	10	4
(40, 20)	40	29	19	9	3
(50, 20)	42	30	16	9	3
(50, 20)	37	30	18	11	4
(50, 20)	38	29	19	10	4
(50, 20)	40	30	17	10	3
(50, 20)	37	32	18	10	3
(50, 20)	37	30	19	10	4
(50, 20)	41	31	16	9	3
(80, 20)	39	29	19	9	4
(80, 20)	41	27	20	9	3
(80, 40)	44	26	16	10	4

Table II shows the equilibrium distribution of 100 processes among the different workstations for the simulated annealing solution to job-shop scheduling problems. The first column reports the size of the problems in terms of number of jobs and number of machines. The migration activity was high in the first period of the execution, but after a certain number of migrations, the distribution reached a sufficient stability requiring only few further re-allocations.

Acknowledgements

This study has been partially supported by MURST, the Italian Ministry of University and Researches for Science and Techology and partially by CNR, the Italian National Council of Researches.

REFERENCES

1. Geist, V.S. Sunderam, Network-Based Concurrent Computing on the PVM System, Concurrency: Practice and Experience, VOL 4, N. 4, pp 293-311, 1992
2. Genco, A. Parallel Simulated Annealing: Getting Super Linear Speedups, IEEE Proc. of 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga (Spain) 1994
3. Laarhoven, J.M., Aarts, E.H.L. Simulated Annealing: Theory and Applications, Kluver Academic Publishers
4. Shatz, S.M., Wang, J. & Goto, M. Task Allocation for Maximizing Reliability of Distributed Computer Systems, IEEE Transactions on Computers, **41**, 9, pp 1156-1168, 199