8

Arcadia: A platform for the study of dynamic scheduling of communicating processes

Bernon C., Bétourné C. and Sayah A.
Institut de Recherche en Informatique de Toulouse (IRIT)
Université Paul Sabatier
118 Route de Narbonne - 31062 Toulouse Cedex - France
Tel. (33) 61.55.83.43 Fax (33) 61.55.68.47
E-mail: bernon, betourne, sayah @irit.fr

Abstract

We present a strategy for dynamically schedule communicating processes of a parallel application onto a loosely coupled distributed system. As we must manage two criteria to schedule processes (the workload of the differents sites and the cost of the communication between processes), the algorithm proposed uses two types of agents (system agents and application agents). A system agent manages the workload of its site, an application agent reduces the IPC costs within its application. Our two types of agents cooperate and negociate to make a trade-off between the two criteria. Therefore, we describe the different cooperations needed via a two-layer model. An implementation of this strategy is also described and experimental results analysed. The results obtained justify certain of our design choices and show that the improvement provided by our scheduling algorithm is satisfactory.

Keywords

Dynamic scheduling algorithm, communicating processes, agents, co-operation.

1 INTRODUCTION

The main goal of this paper is to study how an application can run transparently over a distributed network of workstations using dynamic process scheduling. The study is based on several premises:

- the concept of a loosely coupled distributed system, i.e. a system of interconnected workstations communicating over a network, is becoming more and more widespread;
- in such environments, certain machines are overloaded while others remain idle (Krueger,91);
- increasing numbers of system designers are producing parallel applications, but they rarely take full advantage of the resources provided by the underlying distributed system, opting instead for pseudo-parallel execution.

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

Extensive research has been done on the strategies that can be employed for application process scheduling (Bryant, 81; Ferrari,87; Bernard,93; Ju,95). Our work on the behaviour of a multi-agent system developed at IRIT has led us to identify the following application features:

- an application is made up of a large number of entities executing in parallel;
- these entities have long execution times;
- they communicate by message passing, in an unpredictable fashion;
- the application behaves in a nondeterministic manner.

The second goal of our study was to develop a platform to apply the scheduling strategy adopted, with a view to evaluating the strategy and thus providing input for the design of scheduling algorithms.

We shall now look at the broad outline of this algorithm, the model chosen and the policies adopted to implement it. How the test platform was implemented and the experimental results obtained will be discussed later.

2 ALGORITHM SPECIFICATIONS

Two types of scheduling algorithm exist: those that schedule processes statically before execution (Billionnet,89) and those that schedule them dynamically while an application is running (Bernard,93).

We opted for a dynamic scheduling algorithm for the following reasons:

- static scheduling is less flexible than dynamic scheduling;
- the way processes communicate and how much interaction will take place cannot be predicted;
- it is hard to gauge how an application will behave, and therefore to apply predetermined static schemes.

Further, as the processes created by the applications studied have relatively long execution times, we could also look at ways of moving processes during execution, so a suitable process migration mechanism is supposed to already exist (Nutall,94).

The aim of a dynamic scheduling algorithm is to answer the question 'when do we move a process, which process do we move, and where do we move it ?'. Three policies are usually implemented in response to these issues (Zhou,88):

- the information policy, which determines the type of information required to enable scheduling decisions to be made, and how that information is gathered;
- the transfer policy, which determines whether processes need to be moved, and if so which process should be moved;
- the location policy, which determines which processor should be allocated to the process selected above.

We shall now consider the main features of these three policies with respect to the scheduling algorithm adopted.

2.1 Information policy specification

Information policies usually rely on processor characteristics (system configuration, workload, available memory, etc.) as a basis for scheduling decisions. The processor workload index most often used is the number of processes ready to execute on a CPU at a given moment (Ferrari,87; Kunz,91). We shall adopt a similar index.

Where processes are communicating, we need to consider their characteristics (resources requested, estimated execution time, etc.) and how they interact (relationships, frequency and volume of communications, etc.). Neither of these factors is known in advance. First, by

looking at the intensity and volume of intercommunication between processes, we can determine how they interact.

The information policy must then determine to which system processors information applies (to all or just some processors). This information may be gathered on request (Zhou,88), periodically (Litzkow,88), or in response to process state transitions. We opted for a scheme based on state transitions, in which information may be gathered in a centralized or decentralized manner (Douglis,91). We chose the latter solution, given the weak points of centralized management (bottlenecks, lack of fault tolerance, etc.). Information can therefore be communicated between pairs of processors (Bryant,81), distributed to all other system processors (Disted in (Zhou,88)) or to a subset thereof (Ni,85).

To reduce the overhead generated by the algorithm, we also introduced the concept of logical neighbourhood, i.e. each site is interconnected with a certain number of other sites, which we call its logical neighbours. Each site obtains information from its logical neighbours, and can also request to transfer execution of a process to them. Site subsets thus defined may be disjoint, and define a logical topology that exists on top of the physical network topology. Varying this logical topology varies the information on which the algorithm bases its decisions (full information if all sites are interconnected, partial information otherwise). Defining sets of interconnected sites (via 'gateway' sites) also enables processes to be propagated through neighbouring sites along the network.

2.2 Transfer policy specification

The transfer policy starts by examining how processes are scheduled at various instants according to the current load distribution. Two situations may arise:

- Load balancing: work load is distributed equally among all processors. Processes may therefore be rescheduled every time the load on a site varies.
- Load sharing: work load is smoothed out progressively on each individual processor. This avoids any site becoming temporarily overloaded, and a process need only be rescheduled if the load on a site becomes too high.

Although the second technique involves less overhead, we decided to balance the load on each system processor. Logically, if the scheduling algorithm is efficient, load sharing should not have an adverse effect on system performance.

Generally speaking, transfer policies are based on thresholds (Stankovic,84; Shivaratri,92). In other words, a processor transfers a process if its workload exceeds a predetermined threshold, otherwise it can receive processes itself. A relative transfer policy may also be employed (Douglis,91), whereby the level at which a processor is considered overloaded is defined with respect to the load on other processors.

The algorithm proposed employs a combination of the above techniques. A site S interconnected with a neighbouring site S_N with a lower load becomes a candidate to transfer its processes. S_N will receive if its load is, at least, a certain percentage lower than that of S.

Choosing which process to move means deciding which process is the best candidate for remote execution. Some research has been done on manual filters using a special command to indicate which process should be transferred (Folliot,92). Other research has looked at the use of transparent filtering (Svensson,90; Ju,95) based on estimated CPU time used up by a process, or the resources it requires to execute.

Our study places no restriction on remote process execution. We therefore assume that any process can be transferred to and executed at another site. The process that is normally sent to execute at a remote site is the one which causes processor load to increase in the first place, thus switching it to a 'transfer' state. However, to ensure that the algorithm remains stable (i.e. to avoid a process migrating through the network in vain without ever terminating (Stankovic,85)) a process returned to a site where it already been executed must finish executing on that site.

2.3 Location policy specification

The final step in a dynamic scheduling algorithm involves selecting the processor to be allocated to the process identified by the transfer policy in the preceding step.

Again, a centralized policy is considered unsuitable for the reasons already discussed (§2.1). By employing a decentralized policy, a receiving processor can be chosen randomly (Random in (Zhou,88)) or cyclically (Wang,85), i.e. without knowledge of processor status. Where processor selection is predicated on a certain degree of prior information (full or partial), the entities applying the location policy must co-operate. As we shall see in section 3.3.3, we plan to use a scheme whereby sending and receiving processors negotiate before a process is effectively scheduled. This method is also used in (Bryant,81; Stankovic,84).

With the broad outline of the dynamic scheduling algorithm proposed in this paper now established, we shall look at the model adopted to implement it.

3 A TWO-LAYER MODEL

We have seen that the algorithm must be applied in a decentralized manner. This is achieved through a set of entities distributed throughout the system. We shall call one of these entities the 'agent', in the sense given to this term in distributed artificial intelligence (Ferber,88).

Deciding to execute two communicating processes at two remote sites involves a trade-off between the performance gains achieved by executing them in parallel and the cost generated by their inter-communication. While many static scheduling algorithms attempt to reduce inter-process communication costs (Billionnet,89), very little consideration has been given to this problem when designing dynamic algorithms (Stankovic,84; Folliot,92). This is one of the main aims of this study.

Rather than use an objective function based on run times and communication costs, we chose to express these two parameters separately using a two-layer model:

- a 'system agent' (SA) is associated with each site and manages that site;
- an 'application agent' (AA) is associated with each application on each site and handles communications within that application.

3.1 System agents

The purpose of a SA is to reduce the workload on its site by transferring execution of certain processes to its logical neighbours. It does this by evaluating its own site load and the load of its logical neighbours. Co-operation between SAs in the network balances workload over the various network sites.

3.2 Application agents

The purpose of an AA is to reduce IPC costs within the application, by co-operating with other AAs associated with the application. It does this by tracing calls to primitives for inter-process communication and compiling process communication statistics.

3.3 Interaction between agents

SAs exchange information with each other on the status of their associated site. In attempting to reduce workload on their site, SAs tend to disperse processes belonging to the application over the network.

An application's AAs also exchange information on inter-process communication. In attempting to reduce IPC costs, they tend to keep processes as close to one another as possible.

As system and application agents often reach conflicting scheduling decisions, they have to co-operate, sometimes negotiate, to resolve contention. This requires a third level of communication between site SAs and local AAs.

We shall now look at how such co-operation is achieved within each of the three algorithm policies.

Co-operation in information policy

The SA alone decides where processes at a site S shall execute. To ensure this decision takes account of inter-process communication constraints, the local AAs must have a bearing on the SA's choice. But a process P waiting for a CPU time slice to be allocated to it is unable to send or receive messages. Further, execution of other processes waiting for messages from P is also slowed down. To speed up execution of P it must therefore be transferred to a site with a small workload. An SA must therefore eliminate any processes executing locally or prevent remote processes from being executed locally.

The solution adopted therefore consists in increasing site load virtually by including inter-process communication in load computations. An AA does this by giving a 'weight' to each of the processes it controls, according to how communication-intensive it is. This weighting is then taken into account by the local SA when computing its own site load.

Co-operation in transfer policy

An SA detects when its site is overloaded. It can then decide alone which process to move; but it may first consult AAs at its site if required. An AA responds by indicating which process would most reduce communication overhead within the application if it were transferred. The SA then makes its choice on the basis of all responses received.

Co-operation in location policy

To improve the stability of the algorithm, a negotiation phase is introduced between the SAs of the sending and receiving sites. The receiving site commits itself to execute a transferred process on arrival. This means that:

- the process will not be rejected on arrival at the receiving site;
- the receiving site will not become overloaded if several transferring sites select it simultaneously.

A receiving site thus accepts a process before it is transferred and is able to make room for it before it arrives.

Where co-operation between SAs and AAs is necessary, we need to make sure that process scheduling is efficient enough to compensate for the higher cost of the algorithm, and that the latter still reduces application response times.

However, if a certain criterion proves too costly to handle, the corresponding component can be taken out to reduce the overhead generated by the algorithm. Breaking down the algorithm in this way improves its flexibility and reduces its cost. The scheduling policy can thus be based on:

- load sharing alone if SAs alone make scheduling decisions;
- a reduction in communication overhead if AAs alone make these decisions.

4 EXPERIMENTAL PLATFORM IMPLEMENTATION

In this section we shall briefly describe how the test platform is implemented. To evaluate the scheduling strategy described below, we developed a distributed experimentation prototype written in C++ to run in a Sun/Solaris 2.4 environment. This prototype is able to simulate a network of N sites and its architecture is shown in figure 1.

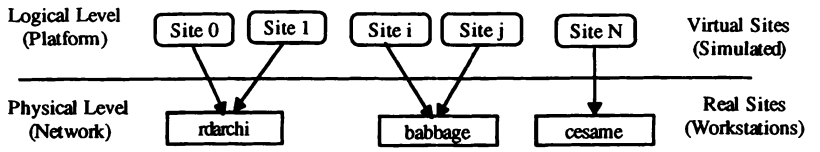


Figure 1 Prototype architecture.

A virtual site, represented by a UNIX process, is activated by the Arcadia command. The user is able to configure simulations by setting this command's parameters (number of virtual sites, scheduling strategy, etc.) and via additional configuration files. The components of a virtual site are represented by lightweight processes:

- the CPU handling processes at the site;
- the site SA;
- the AA associated with each application likely to send a process to execute at the site;
- applications initially started at the virtual site.

These various components are arranged according to the logical scheme of a virtual site shown in figure 2. We shall now look briefly at the overall behaviour of each of these components.

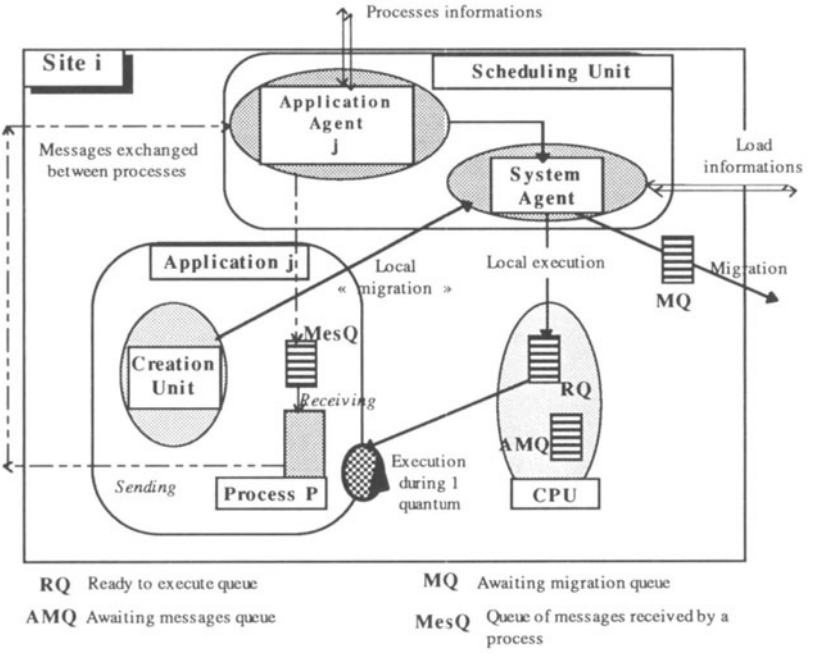


Figure 2 Logical scheme of a virtual site.

4.1 Behaviour of the components of a virtual site

Applications
An application is 'monitored' and managed by a certain number of AAs. It first has to activate its agents, and as an application cannot migrate (only processes can be moved) it has to go through an intermediate SA at each remote site to activate them. The application then creates a 'parent' process from which all application processes will be created, which waits for its 'child' processes to terminate before terminating itself. We thus have a hierarchy in which each child process can also create its own child processes, with the application at the top of the pyramid. The application therefore waits for all its processes to terminate before terminating itself.

Processes
Unlike all the other entities in the prototype, a (virtual) process is not represented by a thread. Indeed, processes must be able to migrate and it would be inexpedient to build a tool for real process migration, even for lightweight processes. Process behaviour is therefore described via a subroutine that simulates process execution. The code run by a process is replicated at each network site. Thus, process migration between two sites amounts to transferring the data needed for a process to resume execution at the remote site and the cost of this migration is the cost of the transformation of data into messages and conversely.

CPUs
Local processes are executed on each CPU in a time-sharing environment. The CPU does this by placing processes in three queues: ready to execute, awaiting message, and finished. It then selects the process at the head of the ready queue at the start of each CPU cycle and allocates it a time quantum.

Agents
We can think of an agent as an entity that reacts to events (the load on a neighbouring entity falls) and takes scheduling decisions (transferring a process to that entity). It detects local or remote events directly by analyzing its own data, or indirectly on the basis of messages it receives and processes according to their priority (urgent, rapid, or normal) and then on a first-come first-served basis.

SAs and AAs do not detect the same events and react differently. For this reason, we shall distinguish between them in our analysis of how the algorithm is implemented in our prototype.

4.2 Implementation of Arcadia scheduling strategy

Information policy
Statistics on a process P indicate the number of messages it has passed or received during an interval of S seconds, the amount of data it has passed or received in that interval, and the identity of its companion processes.

Site workload is represented by the sum of the weight values attached to the processes at a site at a given instant. The weight of a process P depends on the number of receiving processes under its control over the last S seconds. Priority is thus given to processes that are passing messages. Currently, the weight of P is equal to the number of receiving processes P possesses but its calculation should evolve to take into account other criteria.

An SA manages a vector storing information on the load of neighbouring sites. This vector is updated each time a message is received conveying the new load of one of these sites. An SA communicates with its logical neighbour sites whenever its site load increases (when a process is created, execution of a remote process is accepted or the weight of a process increases) or decreases (when a process finishes executing locally, it can execute at another site or its weight decrease).

Transfer policy
An SA considers that its site is overloaded when its load exceeds that of its logical neighbours, and decides that a process needs to be moved. Generally speaking, it is the process at the root of the overload that is moved to execute remotely.

The SA and AA negotiate if the number of processes ready to execute at a site S exceeds a certain threshold, and if the load of S is above the average load of its logical neighbour sites.

An AA calculates the reduction in communication costs achieved for each process P it controls that is sent to execute at the site selected by the SA. This calculation is based on the amount of data exchanged by P during the most recent time interval. Performance gains could also be estimated by attempting to forecast subsequent process communications. An AA then signals to the SA which process should be moved to obtain the highest gain.

Location policy
The location policy, working in tandem with the transfer policy, locates the neighbouring site with the lowest load below a threshold proportional to the load of the transferring site. Our policies are thus relative.

The SA at the receiving site accepts the process to be sent providing its execution will not push its own load above that of the sending site, and above the predetermined threshold.

We chose to apply a threshold to control site selection in order to allow for a certain degree of uncertainty with regard to network status, and thus to render the algorithm more stable.

5 ANALYSIS OF SOME EXPERIMENTAL RESULTS

We shall now evaluate the results obtained with our prototype to ascertain the efficiency of our scheduling strategy. This will lead us in turn to consider to what type of applications the algorithm is best suited.

The results discussed here were obtained by running simulations of a network of 8 virtual sites on a four-processor Sun Sparc-20 workstation. An application comprising a certain number of interacting processes (300, 2, 30, 8, 350, 10, 30 and 260, respectively) executed at each site.

5.1 Performance measurements

Each site simulated by the prototype returns results to an associated file. These results are used to obtain performance measurements enabling the impact of the scheduling algorithm on the applications tested to be assessed. Performance criteria measured were:

- overall performance gains provided by the algorithm, i.e., the difference between the response time obtained, for all the applications, with and without the scheduling strategy;
- distribution of workload over the sites during the simulation. This may be measured in terms of the number of processes ready to execute at a site over time, or the time spent by each site executing processes;
- the amount of negotiations between sending and receiving SAs that end with a process being effectively transferred, in order to evaluate the location policy.

5.2 Algorithm performance

To measure the influence of some parameters of the algorithm, we have set the value of the others and we varied the value of the considered parameters. This method has been applied for all the parameters of the algorithm. The results reported below are averages obtained from several simulations using the same parameters.

Varying the logical topology

In the first applications we tested, processes communicate in pairs, therefore, the weight has no influence on the performance (all processes have the same weight) and we varied the logical topology of the 8-site network. The threshold applied by the location policy was 30% of the load of the sending site. The different logical topologies used are shown in figure 3.

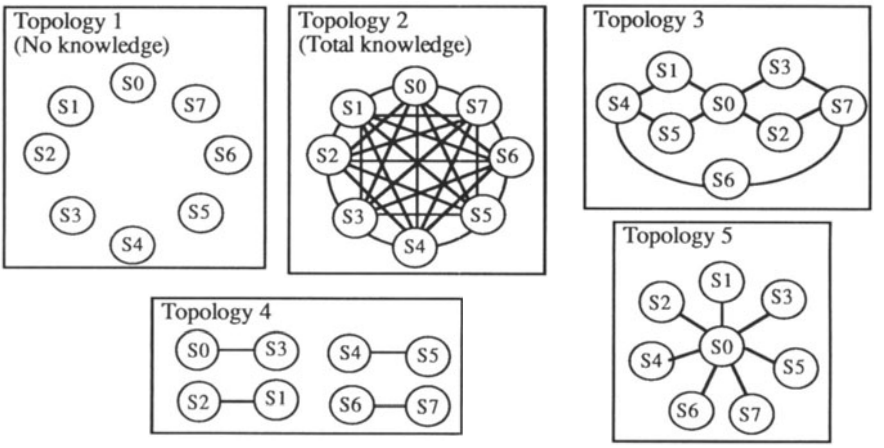
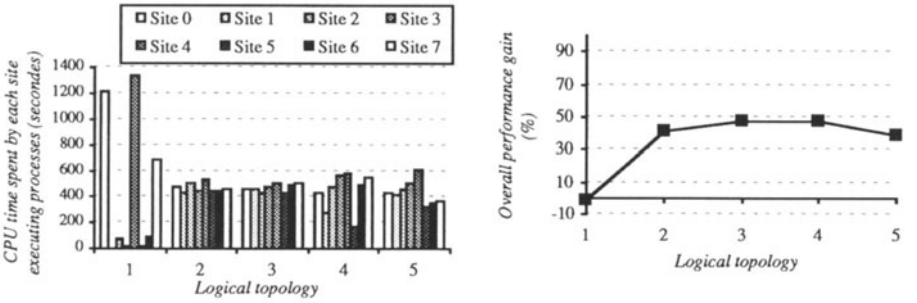


Figure 3 Logical topologies used.

Two of these topologies were diametrically opposite:

- Topology 1: each site had no logical neighbour sites. No remote scheduling was therefore possible, but the fact that sites attempted nevertheless to move processes resulted in an overall performance drop of roughly 1% with respect to when the algorithm is not applied (figure 4a).
- Topology 2: all sites were able to interact. Although this topology enabled load balancing between sites (figure 4b), it did not achieve the highest overall performance gain. This is because the information policy created a lot of overhead, since the algorithm was slowed down by the large number of information or control flow messages.



(a) Load distribution over sites.

(b) Overall performance gain provided.

Figure 4 Varying the logical topology.

To reduce this overhead, we reduced the number of logical neighbours connected to each site and adopted topology 3. Load balancing was still efficient, but this time the overall gains achieved with the algorithm increased by roughly 6%. Practically the same result was obtained with topology 4, in which pairs of sites were able to interact, but load balancing was clearly not as good. This was due to the deliberate link we introduced between a site running a 'small' application and a site supporting a 'large' application. This allows the bigger applications to load off onto the smaller applications in expedient fashion. They do not transfer sufficient load to balance out overall load completely, but enough to achieve a good overall level of performance made easier by using a less costly information policy. It is likely that the impact of these last two topologies would be greater on a larger network, as total knowledge would generate a prohibitive cost for the algorithm.

By adopting topology 5 and allowing a single site to interact with the entire network, we found that this 'gateway' site becomes overloaded with information messages and, although it can distribute load over the network, is therefore less efficient than the other topologies already described.

Varying the logical network topology and, therefore, the knowledge on which our agents rely, shows that SAs and AAs can achieve the overall goal of improved response time with only a partial view of their environment, i.e. a restricted number of logical neighbours.

Linking underloaded and overloaded sites would not appear feasible, however, as their status cannot be predetermined. In choosing a logical topology, we are therefore forced to make a trade-off between the following requirements:

- sites must have a limited number of logical neighbours to limit overhead due to the information policy;
- sites must have a large enough number of logical neighbours to be able to load off excess processes;
- certain sites must serve as gateways between subsets of sites, so that processes can be disseminated through the network.

Our research suggests that a good solution would be to adopt an initial logical topology capable of evolving dynamically in response to site loads and application behaviour. This problem has also been discussed in (Kremien,93).

Bearing of process weight and process migration

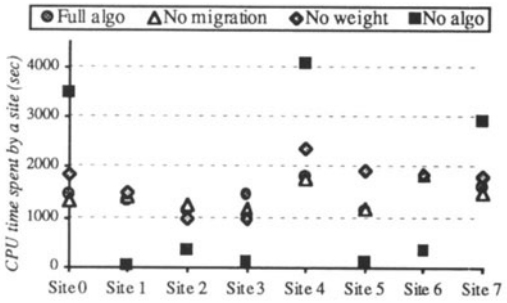
One seeks now to measure the influence of the weight. To do this, some processes of an application passe messages to several other processes thaht only receive. Working on the basis of the same load imbalance, we shall now compare cases where:

- the full algorithm is applied;
- process migration is forbidden, meaning that a process can only be moved when it is created;
- process weight is not taken into account in site load calculations.

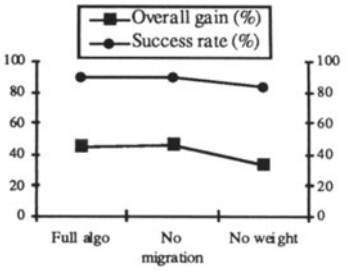
These cases are indicated by the legends 'full algo', 'no migration' and 'no weight' in figure 5.

Process weighting has an affect on scheduling efficiency, and no weighting at all reduces the performance gains achieved on all applications and the success rate of inter-site negotiations by around 10% (figure 5b). A load imbalance tends to persist when weighting is not used (figure 5a). It would therefore seem that weighting a process according to the communication overhead it generates is a reasonable assumption to work on.

We can also see from figure 5 that, in our simulation at least, no benefit is gained from moving a process during its execution. Processes transferred during execution are those whose weight has just changed, meaning they are processes that have already used up a certain amount of CPU time. The execution time remaining is probably too short for remote execution to be of any benefit to them, as the overhead generated by transferring them outweighs any performance gains obtained. Failing to put a limit on remote process execution can therefore lead to perfectly needless process migration.



(a) Load distribution over sites.



(b) Overall performance gain provided and success rate of negociations.

Figure 5 Effects of the variation of the algorithm.

6 CONCLUSION

The algorithm described here is a completely decentralized, dynamic scheduling algorithm whose unique feature is its ability to take into account inter-process communication. The algorithm is split into two parts, thus generating two types of entities to apply its policies: 'system agents' and "application agents'. These agents co-operate and negotiate with each other, i.e.:

- co-operation between SAs allows load sharing between sites;
- co-operation between AAs reduces process communication costs within an application;
- co-operation between SAs and AAs allows load sharing and reduces IPC costs in all applications.

Process execution time and communication overhead are expressed by 'weighting' each process. The weight of a process depends on the number of potential receivers of its messages, and can only be calculated by an AA. SAs use these weight values to calculate their site load.

We developed a prototype capable of running applications over a network of sites to test our scheduling strategy. A lot of improvements still need to be made before we can consider that the scheduling strategy we have adopted is really effective, and before identifying the types of application to which it would be best suited. However, results obtained have justified certain conceptual hypotheses and opened up numerous research prospects:

- Allocating a weight to each process to factor in communication overhead is a viable working hypothesis. However, weight computation needs to be improved and further tests will be required.
- It ought to be possible to make the scheduling algorithm more adaptable by distributing it over the network through the use of agents. An agent could be allowed to request or load off work depending on the perceived status of its environment.
- Process migration does not appear essential for the applications we tested, unless processes executing remotely are filtered. However, migration would certainly prove useful where the risk of site overloading is high, or to provide fault tolerance.
- Overall performance gains vary from application to application, but the 45% to 60% improvement obtained is satisfactory. However, our results were enhanced by deliberately creating large load imbalances between sites.

7 REFERENCES

Bernard, G. Steve, D. and Simatic, M. (1993) A survey of load sharing in networks of workstations. *Distributed Systems Engineering*, **1-2**, 75-86.

Billionnet, A. Costa, M.-C. and Sutter, A. (1989) Les problèmes de placement dans les systèmes distribués. *T.S.I.*, **8-4**, 307-337.

Bryant, R. and Finkel, R. (1981) A stable distributed scheduling algorithm. *Proc. of the 2nd ICDCS* 314-323.

Douglis, F. and Ousterhout, J. (1991) Transparent process migration: design alternatives and the Sprite implementation. *Software-Practice and Experience*, **21-8**, 757-785.

Ferber, J. and Ghallab, G. (1988) Problématique des univers multi-agents intelligents. *Journées Nationales du PRC I.A. Toulouse* , 295-320.

Ferrari, D. and Zhou, S. An empirical investigation of load indices for load balancing applications. *Proc. of PERFORMANCE'87* , 515-528.

Folliot, B. (1992) *Méthodes et outils de partage de charge pour la conception et la mise en œuvre d'applications dans les systèmes répartis hétérogènes.* PhD Thesis - Université Pierre et Marie Curie - Paris VI.

Ju J. Xu, G. and Yang, K. (1995) An intelligent load balancer for workstation clusters. *Operating Systems Review* , **29-1**, 7-16.

Kremien, O. Kramer, J. and Magee, J. (1993) Scalable load-sharing for distributed systems. HICSS-26.

Krueger, P. and Chawla, R. (1991) The Stealth distributed scheduler. *Proc. of the 11th ICDCS*, 336-343.

Kunz, T. (1991) The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, **17-7**, 725-730.

Litzkow M.J. Livny, M. and Mutka, M.W. (1988) « Condor: A hunter of idle workstations. *Proc. of the 8th ICDCS*, 104-111.

Ni, L. M. Xu, C-W. and Gendreau T. B. (1985) A distributed drafting algorithm for load balancing. IEEE Transactions on Software Engineering, **11-10**, 1153-1161.

Nutall, M. (1994) A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, **28-4**, 64-80.

Shivaratri, N. Krueger, P. and Singhal, M. (1992) Load distributing for locally distributed systems. *IEEE Computer*, **25-12**, 33-44.

Stankovic, J. and Sidhu, I. S. (1984) An adaptative bidding algorithm for processes, clusters and distributed groups. *Proc of the 4th ICDCS*, 49-59.

Stankovic, J. (1985) Stability and distributed scheduling algorithms. IEEE Transactions on Software Engineering, **11-10**, 1141-52.

Svensson, A. (1990) History, an intelligent load sharing filter. *Proc. of the 10th ICDCS*, 546-553.

Wang, Y-T. and Morris, R.J.T. (1985) Load sharing in distributed systems. *IEEE Transactions on Computer*, **34-3**, 204-217.

Zhou, S. (1988) A trace-driven simulation study of dynamic load balancing. IEEE Transactions on Software Engineering, **14-9**, 1327-41.