Global Scheduling on Clusters of Workstations: Development and Performance Study*

M. Hobbs, A. Goscinski, D. Rhodes†, D. Temple†

{mick, ang, rhodesd, temple}@deakin.edu.au

School of Computing and Mathematics
Deakin University, Geelong
Victoria 3217, Australia.

Abstract

The ability to fully utilise the computational resources of a cluster of workstations (COW) through the initial placement and movement of workload is desirable not only by the user of the system, but also from an efficiency point of view since the modern COWs are an expensive resource. The ability to have an even spread of load over an entire COW can be achieved through the employment of Global Scheduling. This paper introduces the concept of global scheduling exploiting static allocation and dynamic load balancing along with the implementation of such a facility and support services, remote process creation and process migration, respectively, on the RHODOS distributed operating system. The suitability of the RHODOS implementation is demonstrated by results obtained from initial test runs.

* This work was partly supported by the Deakin University Research Grant 0505009151.
† Supported by a School of Computing and Mathematics, Deakin University, 1996/1997 Summer Vacation Scholarship.

INDEX

1 Introduction

The overall performance of a Cluster of Workstations (COWs) and of individual users, as well as the efficient use of computational resources can be improved and even increased if the load of the COW is made even. This is becoming critical when a COW is used as a platform for parallel processing and many users execute their programs in parallel. To achieve these goals there must be some way for processes to be created on remote workstations or moved to another workstation when the workstation they are running on becomes overloaded. The mechanisms which support these requirements include remote process creation and process migration. For these mechanisms to be effective, there must be knowledge of the current COW's load and the loads of individual workstations, i.e., resource discovery must be provided. It is no good moving a process if the workstation it is migrated to is already heavily loaded; this only leads to a degradation in performance. Thus there is the need for a coordinator, a Global Scheduler, which makes decisions on where to create a new process, selection of a process to be migrated and where to migrate the selected process, thus supporting static and dynamic load balancing, based on information it has stored regarding workstation resources and loads. These decisions can be made centrally or in a decentralized manner. The former approach suffers from reliability and scalability. The latter from heavy communication and lack of up-to-date state information on the COW.

The three key aspects to global scheduling and process migration can be summarized by *when*, *which*, and *where*; when to migrate the process, which process to migrate and where to migrate it. With global scheduling supported by remote execution we are only concerned with *where* to create the process, as the *when* and the *which* questions are dictated by the user process. There are several algorithms which can be used to dictate the policies of global scheduling [Goscinski 91]. In this paper we are not dealing with the selection of such policies. Instead we address the interaction between a global scheduling server and remote process creation and process migration servers, and the parameters on which decisions to remotely create and migrate processes are made.

This paper examines the static and dynamic load balancing services of the RHODOS distributed operating system [De Paoli et al. 95]. The objective of this paper is to present the facilities that form the RHODOS static and dynamic allocation facility, their cooperation, and their ability to improve performance of individual applications. Other work has been carried out on distributed operating systems in the area of load distribution — these systems include MOSIX [Barak et al. 93], Amoeba [Zhu 95] and MACH [Milojicic 94]. The main difference between these systems and RHODOS is that RHODOS employs both static and dynamic load balancing methods [De Paoli et al. 95].

This paper is organised as follows. The RHODOS Global Scheduling facility, its requirements and functions in supporting load balancing, as well as the operation of the support mechanisms such as remote process creation and process migration which are provided by the Remote Execution Server (Manager) and Process Migration Server (Manager), respectively, are presented

in Section 2. A description of the event based Global Scheduling Server which was implemented for RHODOS, including its coordination of the remote process creation and process migration mechanisms is given in Section 3. The results achieved by implementing an example parallel application supported by the RHODOS Global Scheduling facility are presented in Section 4. This section in particular demonstrates that the approach we have taken in combining static allocation and dynamic load balancing is feasible. A conclusion gained from this work and future actions to be taken are presented in Section 5.

2 Global Scheduling Server and Support Mechanisms

Global scheduling is concerned with making the load of all workstations within a COW equal, with the objective of improving the overall and individual process performance, and to increase resource utilisation. By examining the load on individual workstations within the COW decisions can be made on how to remove load imbalances, by deciding which workstations need a decrease in load and which require an increase in load. Thus once balanced, the load on all workstations should be even allowing the overall system to operate at an optimal level. Moreover, enabling the overall load of the COW to be balanced the amount of work each workstation has to perform is equal; this in turn leads to an overall better performance.

2.1 Global Scheduling in RHODOS

The RHODOS distributed operating system was designed and built following the client/server model using a microkernel architecture [De Paoli et al. 95]. These foundations have provided an ideal environment to implement new and advanced services including process migration, remote process creation and execution, data collection and global scheduling. Another benefit of the enforced modularity of RHODOS (primarily due to the microkernel architecture) is that policies, mechanisms and services can be cleanly separated, simplifying their implementation and extension.

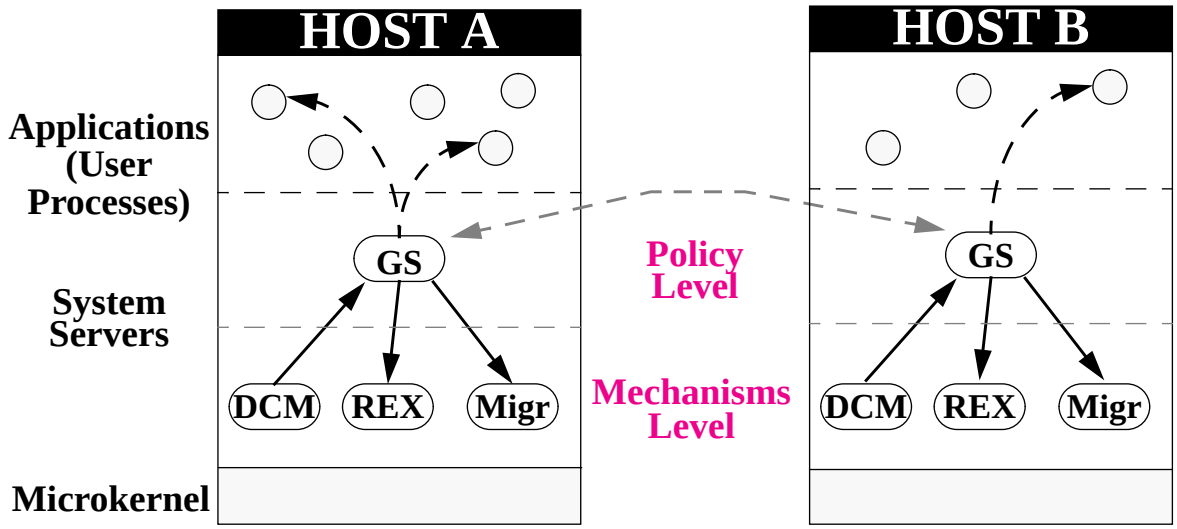

Figure 1: RHODOS Global Scheduler and Support Mechanisms

Figure 1 shows the position of the Global Scheduler, Process Migration Manager (Migr), Remote Execution Manager (REX) and Data Collection Manager (DCM) within the framework of the RHODOS service hierarchy. These server processes cooperate to provide the foundations of the RHODOS load balancing facility, each providing a valuable and advanced service.

2.2 Requirements of a Global Scheduler

To balance the load of a COW we need to employ global scheduling which combines the load balancing methods of static allocation and dynamic load balancing. These methods allow the load balancing system to react to large fluctuations in system load (using dynamic load balancing), and also to avert the case when system load remains steady (static allocation) or when a new process (processes) are created. In the RHODOS system, global scheduling is supported by process migration and remote process creation.

Separation of the policy decisions of the Global Scheduler (i.e., the decisions on when to employ static allocation and dynamic load balancing) from the support mechanisms (i.e., remote process creation and process migration respectively) needed to enact such policies was chosen to simplify the design and implementation. Another requirement of the RHODOS Global Scheduling facility was to provide a testbed for the analysis of existing and new static allocation and dynamic load balancing algorithms. The modular framework and clean separation of the policies from the mechanisms allows varying algorithms to be tested and compared effectively.

We also require the RHODOS Global Scheduling facility to be flexible enough to implement various control and management structures including: centralised, hierarchical and distributed. This will allow us to investigate the impact the various decision structures have on the effectiveness of the static allocation and dynamic load balancing facilities. The overheads from each structure can be measured and the overall impact gauged.

A centralised Global Scheduling structure is based on a single Global Scheduling Server for an entire COW. This central server collects all load information and based on this makes load balancing decisions. This structure is simple to design and implement but suffers from the scalability and fault tolerance problems faced by all centralised system.

The second structure proposed is that of a hierarchical or master/slave global scheduling arrangement. In this structure, every workstation within the COW has a Global Scheduling Server, with one acting as a master or controlling Global Scheduler and all others act as slaves or agents. The master Global Scheduling Server collects all load information from slave Global Scheduling Servers and makes decisions upon this information, which is then dispersed out to all slave Global Scheduling Servers. The slaves in this architecture do not make any decisions on their own, they follow the decisions made by the central/master Global Scheduler. This structure has similar benefits and disadvantages as experienced by the centralised structure, but the master/slave relationship reduces the communication between the central workstation and other workstations.

The final structure proposed is that of a fully distributed system. In this structure, every workstation within the COW has a Global Scheduling Server which interacts with all other Global Schedulers within the cluster. Load information is passed between the Global Schedulers and each server makes decisions independent to others within the COW. This structure has good fault tolerance and good scalability can be achieved if the set of global scheduling servers is divided into subsets, with all global schedulers within the subset exchange load information and not with any global schedulers outside their subset. This reduces communication overheads since only subsets exchange load information, not every global scheduler.

The Global Scheduling Server structure chosen to implement and test was that of the centralised version. As previously stated, in the centralised structure a single RHODOS Global Scheduling server exists that collects load conditions from all workstations, upon which it bases its static allocation and dynamic load balancing decisions.

2.3 Remote Process Creation

The first mechanism that RHODOS provides to support load balancing, more precisely static allocation, is that of transparent remote process creation. The mechanism is employed when the load of a given workstation is continually high and a request to create an instance of a process is issued. Remote process creation is the activity of transparently creating a child process on a workstation other than that which is executing the parent process; ideally the workstation with the lowest load at the time the process is created. The RHODOS Remote Execution (REX) Manager [Hobbs and Goscinski 96] interacts with the Global Scheduler to carry out this task.

The REX Manager acts as a proxy agent that interacts with the variety of resource kernel server processes, on behalf of the user, to complete requests for process duplications, creations, exits and terminations. Therefore the REX Manager operates as a kernel server process which follows a ‘policy’ on how processes are created and uses the ‘mechanisms’ of the other kernel servers to achieve its objectives. The kernel server processes that the REX Manager interacts with include: the Space Manager, Process Manager, File Manager, Device Manager and the IPC Manager.

The decision of which workstation to create a process on is not made by the REX Manager but is the responsibility of the Global Scheduler. When a user process requests the creation of some other process by issuing the *process_create()* call, the REX Manager picks up this request and contacts the Global Scheduler requesting the address of the workstation on which to create the process. The decision whether to have the process created locally or remotely (and on which remote host) is decided by the Global Scheduler. This decision is passed back to the REX Manager. This may of course be the local workstation.

The REX Manager either creates the process locally or contacts the REX Manager on the remote workstation to create the process on its behalf. Figure 2 illustrates the message passing based architecture of this implementation. The interface that the REX Manager has with the other kernel servers is via message passing and RPCs. The REX Manager is required to coordinate the

allocation and control of resources on behalf of the calling user process. The interaction the REX Manager has with the user process, peer REX Managers (for remote process creations), and other kernel servers can be found in [Hobbs and Goscinski 96].

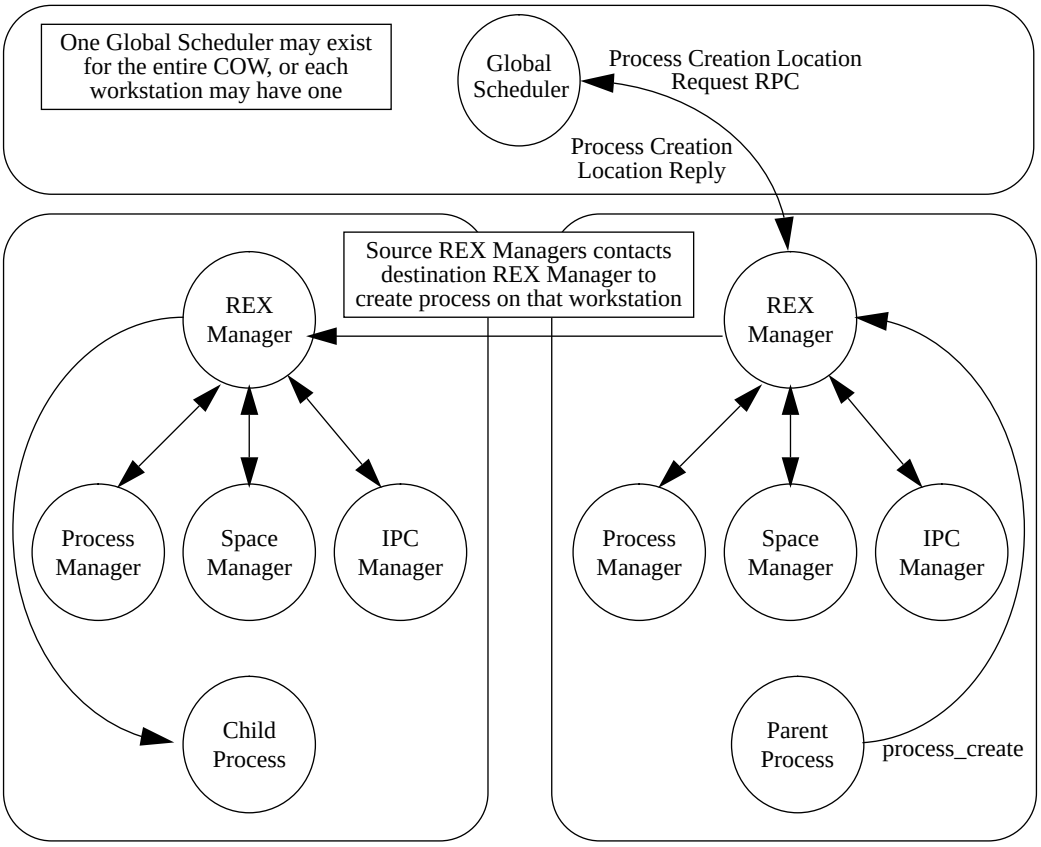

Figure 2: Global Scheduling and Process Creation

The order of steps involved in creating a remote process are as follows. The REX Manager sends a request to the Process Manager to create a new Process Control Block (PCB) for the process it is about to load in from disk. Once the Process Manager has completed this task, the REX Manager then reads the Text and Initialized Data regions in from disk. The REX Manager then requests the Space Manager to create the Text, Data and Stack spaces for the new process. These spaces are then attached to the REX Managers address space and populated with the information read in from the disk. Once populated, the new process' Text, Data and Stack spaces are detached from the REX Managers address space. The REX Manager then informs the Process Manager to place the new process on the ready queue, and concurrently the parent process is notified of the successful creation of the child process. If this process is created remotely, the Process Manager on the parent process workstation is notified to update its' process table indicating that the child process of the parent is located on a remote workstation.

2.4 Process Migration

The second mechanism that RHODOS provides to support load balancing, in this instance

dynamic load balancing, is that of transparent process migration. This mechanism is employed by the Global Scheduler when the load within the COW (or given workstations) varies significantly and load (in the form of processes) is required to be moved from overloaded workstations to lightly loaded or idle workstations.

Process migration is the method of moving running processes from one workstation to another, transparent to the process being migrated, its parent process or peer processes with which the migrated process is communication with. The RHODOS migration facility [De Paoli and Goscinski 97] is able to undertake migration of processes, moving them from one workstation to another, by interacting with the key kernel and system servers (Process Manager, Space Manager, IPC Manager, File Manager, etc.) through the message passing (IPC) mechanism.

The Migration Manager does not make any decisions on *when* a migration of *which* process to *where*, rather the Global Scheduler instructs it on the destination workstation and a source process should be involved in the migration. Furthermore, to prevent the process chosen to be migrated from executing further it is moved from the ready queue, onto the frozen queue. At the same time, the processes spaces and ports are also frozen.

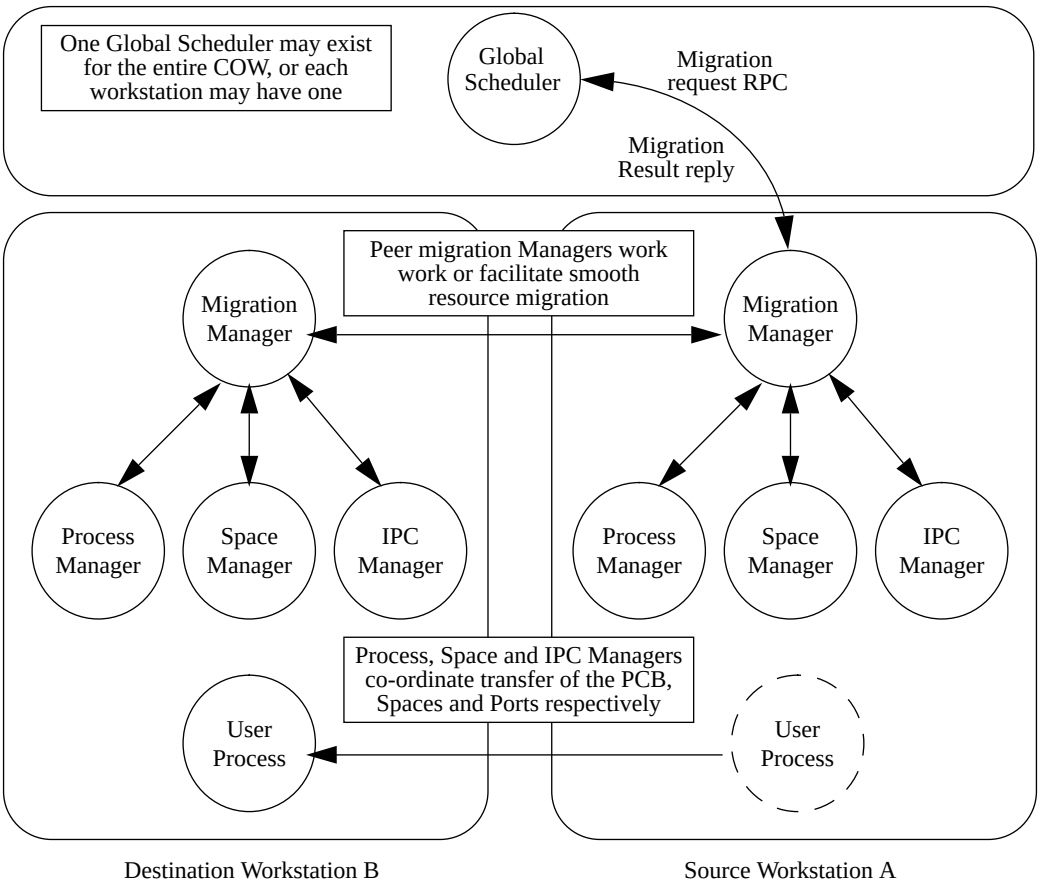

Figure 3: Global Scheduling and Process Migration

The order of steps involved in a process migration is shown in Figure 3. The Migration Manager receives a migration request from the Global Scheduler. This request identifies the selected

process to be migrated and the destination workstation to migrate to. The Migration Manager on the source workstation then sends a message to the Migration Manager on the destination workstation to indicate that it wishes to migrate an object to it. This allows the Migration Manager on the destination workstation to prepare a location in which to store information about the state of the process and the migration. The source Migration Manager also sends a message to the Process Manager on the source workstation indicating it's desire to migrate the required process. When the Process Manager receives the request to migrate the control block of a process, it checks the current state of the process. This is to prevent the migration of processes that are not migratable, for example, processes that are frozen because they have caused a fault and are in the middle of being handled or processes that are already being migrated. Another cause for a process to be non migratable are when processes are executing system calls (processes blocked on waiting for a message are migratable with the RHODOS migration facility). If the process can't be migrated, a message is passed back to the Migration Manager, cancelling the migration, otherwise the local Migration Manager is advised the migration may continue, and the local Process Manager transfers the processes PCB to the destination Process Manager. Once the destination Process Manager has populated a new PCB with the information from the recently transferred PCB, it then sends a message to it's own Migration Manager indicating the successful transfer of the PCB.

Meanwhile the source Migration Manager sends a request to the source Memory Manager to migrate the process' memory space to the destination. Similarly it sends a message to the source IPC Manager to transfer the ports and messages queued on them. Both managers then freeze their respective resources and send them to their destination counterparts. The respective managers on the destination side allocate spaces for the incoming resources and fill them appropriately. They then send acknowledgment to their Migration Manager to indicate a successful movement of the resources. Other resource managers such as the File Manager, Terminal Manager and Name Server are notified of the migration if the process has resources allocated to it on the source workstation.

Once all the resources have been accounted for and checked off by the destination Migration Manager, it sends a message back to the source Migration Manager indicating a successful migration. This message must be received by the source Migration Manager before it time-out's on the receive. Otherwise the migration is cancelled.

If the migration is successful, the local Migration Manager depending on the migration technique used [De Paoli and Goscinski 97] sends a message to the other local managers to free up any resources they hold associated with the migrated process. On the destination workstation, the Migration Manager transmits a message to activate the resources allocated to the migrated process.

When dealing with the migration of spaces the RHODOS Migration Manager is able to undertake either direct copy or copy on reference [De Paoli and Goscinski 97] migration strategies. Each method has it's advantages and disadvantages. For example, by copying on reference, we have a higher run-time cost when compared to direct copy. This is due to when a copy on ref-

erence migration of a process occurs the pages associated with the process remain on the source workstation and are only copied to the destination when the migrated process tries to access them. The advantage is that this method reduces the time required to migrate a spaces, as portions of the space may never be used. For example, if a process is duplicated using the *process_twin()* call, the child may never access the parent's code. To copy the entire text region (including both the code for the parent and the code for the child) during the process migration from source to destination would be wasteful. Alternatively the use of direct copy migration maybe more favourable when the load on the source is high, as we are then freeing memory for the remaining processes to utilize. Direct copy migration migrates the entire memory space from source to destination, leaving no residual memory on the source. This produces a higher migration time, but lower run-times for the migrated process as it does not require to page from the source workstation.

3 Event Based Global Scheduling

The RHODOS Global Scheduler is an event based entity, which acts as a coordinator and collects all load events from the entire COW. These load events include: process creations, process twins and process exits. As all events are logged with the central Global Scheduler, an exact picture of the COW wide load is obtained. Therefore load imbalances can be rectified by employing static allocation or dynamic load balancing methods.

The centralised Global Scheduler was chosen to be implemented initially. In this implementation of global scheduling, there is no work to be performed by slave Global Schedulers. This implies that, the architecture of this centralized global scheduling facility could be reduced to just the master Global Scheduler. Thus, this architecture has a single Global Scheduler which maintains load data of all workstations in the domain. To achieve this, all REX Managers within the COW must know the address of the central Global Scheduler. This becomes a simple resource discovery problem. The broadcast discovery method is used in this implementation to ensure all REX Managers obtain the address of the Global Scheduler. When the Global Scheduler receives this message it adds the workstation to its list of workstations in the domain and sends back a reply with its own location. This leads us to the stage where the Global Scheduler has knowledge of all REX Managers and all REX Managers have knowledge of the Global Scheduler.

3.1 Collecting and Measuring Load

The term “load” in the context of this paper is defined as the amount of work being performed by individual workstations or by the entire COW. Load is produced on workstations or COWs by the execution of processes. A measure of the load can therefore be simplified to the number of processes executing (process queue lengths) on a workstation, or averaged out over a COW. This form of load index is satisfactory within homogeneous COWs, where the capacity of each individual workstation to perform work is equal. In heterogeneous COWs, weighting factors must be applied to normalize the load indices such that they can be compared without biasing more powerful workstations. Another problem with the process queue length is that it can change

dramatically over a short period of time. These changes can be avoided by smoothing or averaging the process queue lengths over a period of time to remove transient load fluctuations, allowing load trends to be determined. Other methods of measuring load can include processor utilisation, context switch rates, system call rates and memory/communication usage.

The load index chosen for our implementation is that of the process queue length or the sum of runnable processes located on a given workstation. A process is defined to be runnable if it exists in a state that allows it to execute; this may include the states: ready to run and blocked. The initial implementation uses instantaneous load values, not load averages, which provide the Global Scheduler with current load conditions for it to base its decisions upon. The effects of various smoothing values upon the global scheduling will be an area of further research.

3.2 Event Based Global Scheduling and Remote Execution

When a user process makes a *process_create()* call this is issued to the local REX Manager (Figure 4). The local REX Manager then contacts the Global Scheduler and requests a decision on which workstation to create the child process on.

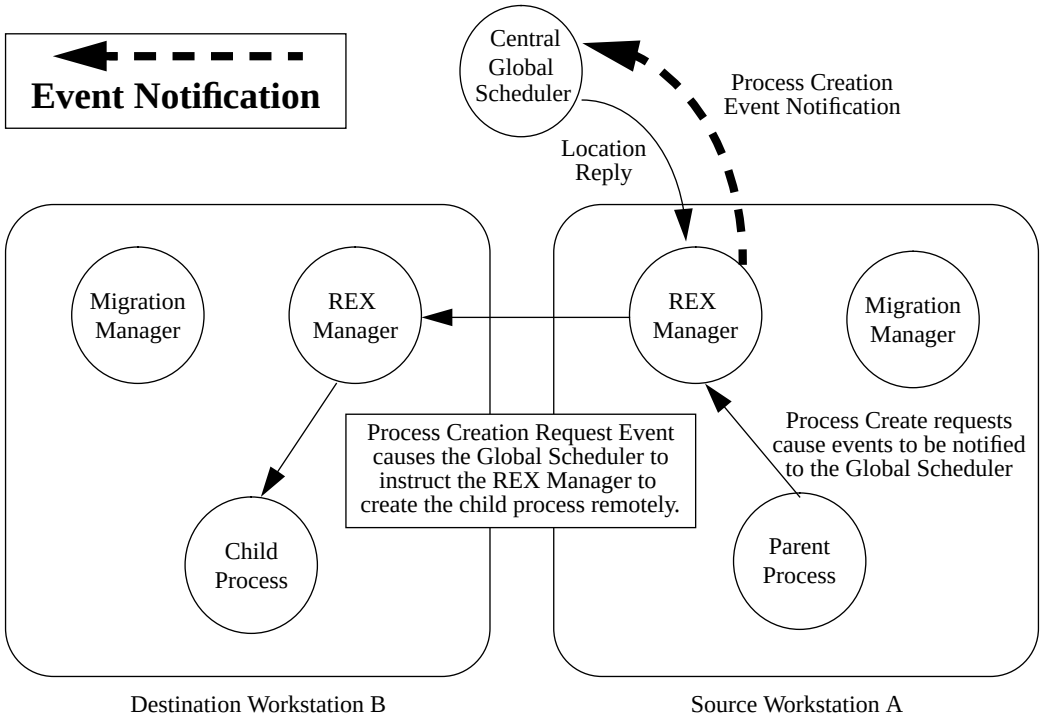

Figure 4: The occurrence of an event causing a remote process creation

When notified of a process creation event the Global Scheduling facility makes a decision on *where* the process should be created, either remotely or locally. The Global Scheduler makes this decision by inspecting the load on the source workstation and compares it with the load of all other workstations within the COW. If the load of the source workstation is found to be greater than the other workstations within the COW the Global Scheduler then instructs the REX Man-

ager to create the process remotely. The workstation with the lowest load is chosen as the destination on which the new process is to be created, which will improve the overall COW wide load. If the load of the source workstation is found not to be overloaded when compared with the other workstations within the COW, then the source workstation is chosen as the destination on which the new process is to be created. The Global Scheduler updates the load value for the destination workstation.

3.3 Event Based Global Scheduling and Process Migration

When a user process issues a *process_twin()* or *process_exit()* request to the REX Manager, these events are notified to the Global Scheduler. These events relate to load changes within the COW and therefore a decision on whether or not to invoke dynamic load balancing needs to be made.

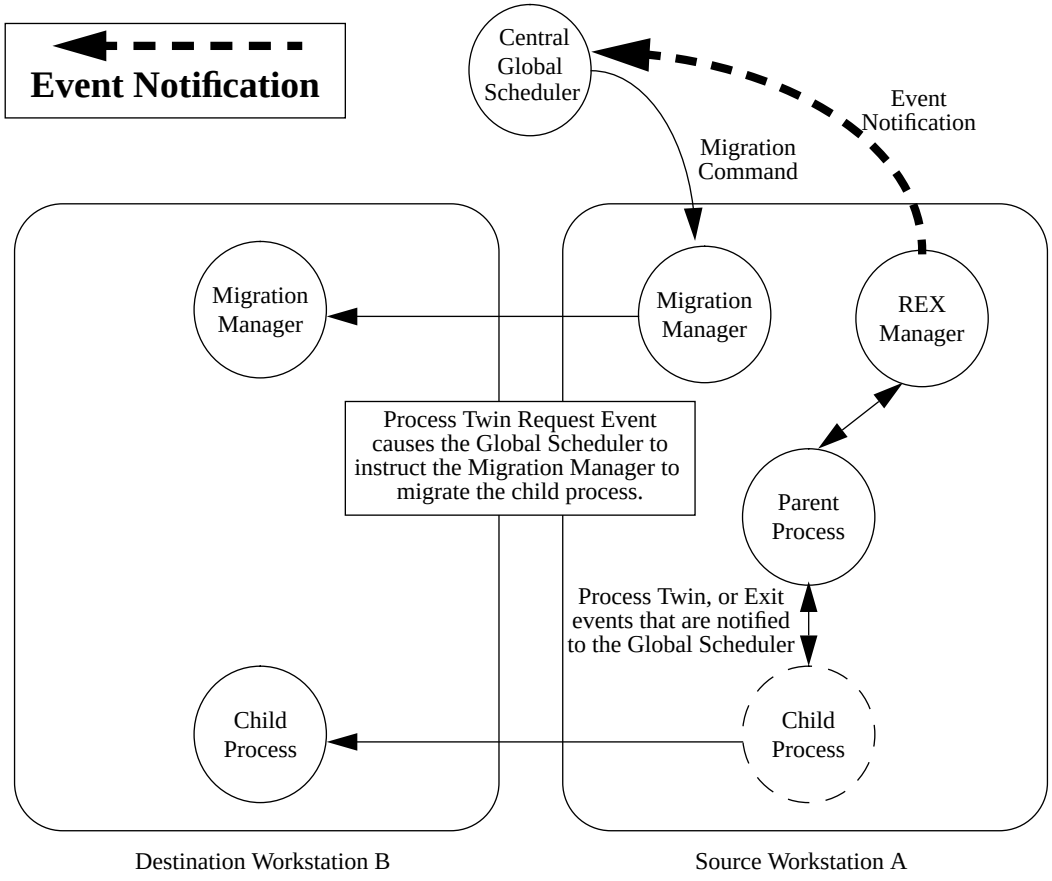

Figure 5: The occurrence of an event causing a migration to occur

A user process issues the request for a *process_twin()* to the REX Manager, which then proceeds to duplicate the process. Once the duplication of the process has been accomplished successfully, with communication ports and memory spaces assigned, the REX Manager then sends an event notification to the Global Scheduler (Figure 5). This is to indicate that an increase in load has occurred. With this event notification the Global Scheduler increments the current load value

for that workstation to represent the additional process. Since this additional process may then cause a load imbalance within the COW, the Global Scheduler must make a decision whether to engage dynamic load balancing. This is achieved by searching through load table to find the workstation with the smallest load, along with the workstation with the largest load. If the source workstation is found to be overloaded then a process migration is employed. A process on the workstation with the highest load is chosen to be migrated to the workstation with the lowest load. The algorithm used in selecting a process to migrate is currently based on random selection but a more detailed algorithm can be installed. Once the migration has completed successfully the Global Scheduler adjusts the load table, decrementing the source workstations load and incrementing the destination workstations load. No acknowledgement to the event is sent back to the REX Manager.

An almost identical mechanism exists for the case when a *process_exit()* request is issued, but with the loads decremented when a *process_exit()* event message is received. Once again a check is done of the current load to assess whether a process migration would even the network load.

4 Performance of the Global Scheduling Facility

Parallel processing on COWs is an ideal choice to gauge the effectiveness of the RHODOS Global Scheduling facility due to the variety of load conditions that this style of application produces. Various parallel computational models can be implemented on COWs including both Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) [Goscinski 97]. For this study we decided to use the SPMD model which maps quite easily to the cluster of workstations environment and can be implemented simply following the master/slave hierarchy.

The basic concept of the Single Program Multiple Data parallel application is that identical sets of program code are executed in parallel (on a number of workstations) with each individual program working on a subset of the total problem data space. The data may be divided equally between each of these programs or different sizes allocated to each program.

The SPMD application we chose to implement allocates an equal amount of data for each program to complete. The computation performed by each parallel program has been simplified to a number of iterations of a loop. The decision to use equal partitioning of the data was made so that performance variations due to data imbalances could be removed. We named this application ‘loop parallel’ as it provides a skeleton framework for more involved parallel applications to built upon.

4.1 SPMD Based Parallel Application

We implemented the loop parallel application following the traditional master/slave approach. The master process acts as a coordinator of the parallel processing. Its role is to initialise the data to be worked on and to start the set of slave processes which will perform this work.

After starting the slave processes the master process then sends each slave the data on which it must perform work upon. The slave processes then proceed to work on the data they have been given and after completion send the results back to the master process. The master process collects all results from the slave processes and calculates the final result. The bulk of the computation for the loop parallel application is performed in user mode (no system calls are issued).

```
/* Create instances of slave processes */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     process_create();
     /* GS may decide remote creation */

/* Send initial data to slaves */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     send( slave[cnt, initial data ] );

/* Wait for every slaves response */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     recv( slave, result );
```

a) Process Create Method

```
/* Create instances of slave processes */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     process_twin();
     /* GS may use process migration */

/* Send initial data to slaves */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     send( slave[cnt], initial data );

/* Wait for every slaves response */
for( cnt = 0 ; cnt < MAXPROC ; cnt++ )
     recv( slave, result );
```

b) Process Twin Method

Block 1: Sample Master Process Code

The pseudo code for the master process is presented in Block 1. Two versions are shown in this code block, the first is where the master process initiates a number of slave processes by issuing *process_create()* requests, as shown in Block 1a). The second is where the master initiates slave processes by issuing *process_twin()* requests, as shown in Block 1b). These methods allow the static allocation and dynamic load balancing components of the Global Scheduler to be tested and evaluated.

```
/* Recv data to work upon */
recv(master, initial work);

/* Do work on data */
for(cnt = 0; cnt < maxdata; cnt++)
     our_work(cnt);

/* Send final result back to master */
send(master, final result);

/* This exit event may cause the GS to
 * invoke a process migration */
process_exit();
```

Block 2: Sample Slave Process Code

The pseudo code for the slave process is presented in Block 2. The slave process is divided

into four phases. The first stage is initialisation and receipt of the initial data stage, through the receipt of a message from the master process. The second stage involves the execution of the work, which is performed on the allocated data. Thirdly, once the work has been completed the final result is sent back to the master process. The final stage of the slave process is to exit and to free the resources owned by this slave, the exit event is notified to the Global Scheduler which may invoke a process migration to balance the COW wide load.

The pseudo code presented in Block 1 and Block 2 were implemented and tested on a cluster of Sun 3/50M workstations managed by the RHODOS distributed operating system. In this section we will demonstrate that the RHODOS Global Scheduling Facility and underlying mechanisms are able to improve the performance of the loop parallel application if more workstations are added to the COW.

The number of workstations forming the COW was varied from a single workstation through to a maximum of ten workstations. For each of these COW sizes the number of slave processes was varied from a single slave (equivalent to sequential processing) through to a maximum of 20 process. This allowed us to gauge the impact of the case when we have more slave processes executing than we have workstations.

4.2 Event Based Global Scheduling employing Remote Process Creation

The pseudo code presented in Block 1a) was implemented and tested with the support of the static allocation component of the RHODOS Event Based Global Scheduling Facility. We demonstrate in this subsection that we can improve the processing time for a SPMD parallel application with equally divided data space, which uses the *process_create()* mechanism to instantiate and execute slave processes; by transparently employing local and remote process creation and execution load balancing methods.

In these experiments new processes were produced using the process creation method which allowed the static allocation component of the Global Scheduling facility to distribute the load over the COW. This distribution led to a round robin pattern as the COW was dedicated to the execution of this application and no background or contending processes were executing.

The loop parallel application using process creation as the method of initiating slave processes (as in Block 1a)) were implemented and the performance results obtained. We tested the loop parallel application varying both the number of slave processes and the number of hosts used. For each test we ran 10 successive iterations to obtain an average result. These results are presented in Figure 6.

In this example, only static allocation was employed, which meant that any load imbalance after the creation of a process that occurred could not be rectified with dynamic load balancing and process migration. This situation would arise when a process exits, for example. Since this experiment produce an ‘initial placement’ environment, the results show a periodic wave effect seen in the experimental results, due to the imbalance of process distribution over the COW (e.g.,

5 processes on 4 workstations has a distribution of 1, 1, 1 and 2 processes).

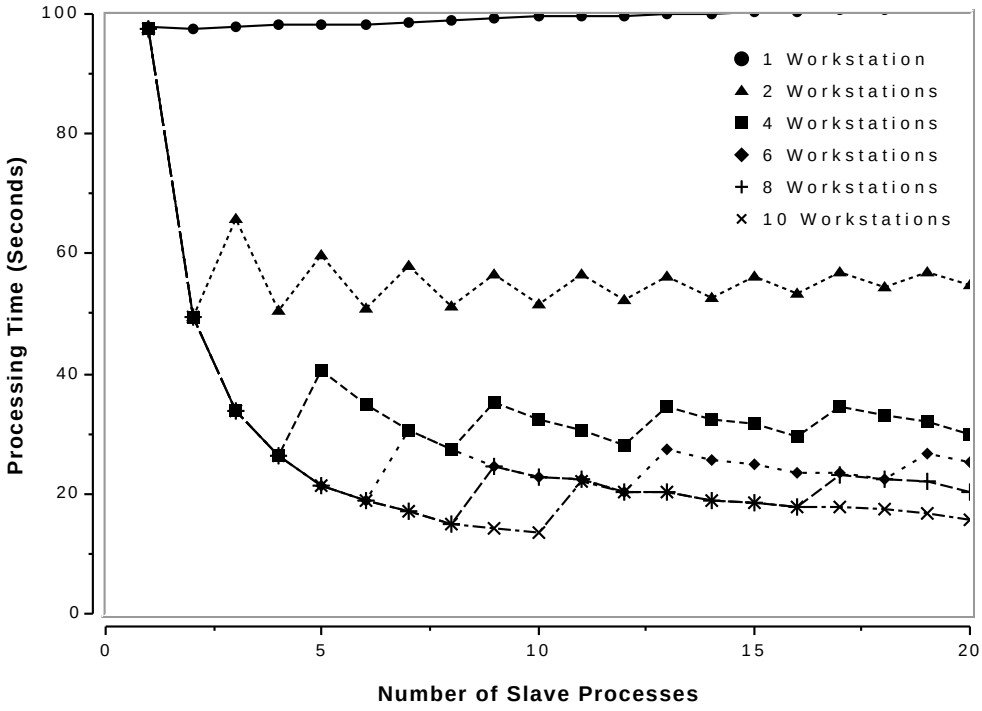

Figure 6: Results of Event Based Global Scheduling and Remote Process Creation

The implementation of the RHODOS Event Based Global Scheduling facility receives notification of all events (process creation, twin and exit events). On receipt of each of these event notifications the central Global Scheduler updates its load tables. In this example, only process creation events are acted upon by the Global Scheduler. Therefore, during testing of remote process creation under event based global scheduling, the Global Scheduler made a correct static allocation decision (whether to create the process locally or remotely, and on which remote host) every time.

A slight linear increase can be seen in the experimental results shown in Figure 6. This slight increase in processing time as the number of slave processes increases can be attributed to the overheads for process creation, communication and context switching between multiple slave processes on the same workstation.

We observe from these results that a parallel application that uses process creation as the primitive for slave instantiation and execution are improved when static allocation methods are employed. The number of workstations used in the parallel application has the greatest impact on the overall processing time achieved but the number of slave processes employed also has an effect. Although, the ratio of slave processes to workstations is more important when only static allocation is employed as computation resources can not be regained when they become available. In summary, the RHODOS Event Based Global Scheduler employing static allocation improves the performance of the loop parallel application by transparently utilising the available computational resources of the COW.

4.3 Event Based Global Scheduling employing Process Migration

The pseudo code presented in Block 1b) was implemented and tested with the support of the dynamic load balancing component of the RHODOS Event Based Global Scheduling Facility. As in Section 4.2, we demonstrate in this subsection that we can improve the processing time for a SPMD parallel application with equally divided data space, which in this case uses the *process_twin()* mechanism to instantiate and execute slave processes and by transparently employing process migration managed by the dynamic load balancing.

In this example all slave processes are started (twinned) on the same workstation as the master process. Dynamic load balancing using process migration, migrates processes away from this overloaded workstation to idle or lightly loaded workstations. As in the previous example, using static allocation and remote process creation, all events (process create, twin and exit events) are notified to the central Global Scheduler. Two situations can be tested with this framework. If only process twin events are considered while process exit events are ignored by the Global Scheduler, then we revert to an initial placement' environment presented in the previous section. If both process twin and exit events are considered then we have a full load balancing facility.

We present in this subsection, firstly, the results obtained from a static allocation environment using the *process_twin()* mechanism and dynamic load balancing (achieved by disabling the notification of process exit events to the Global Scheduler). Secondly, we present the results obtained from a full dynamic load balancing facility achieved when both process twin and process exit events are notified to the Global Scheduler.

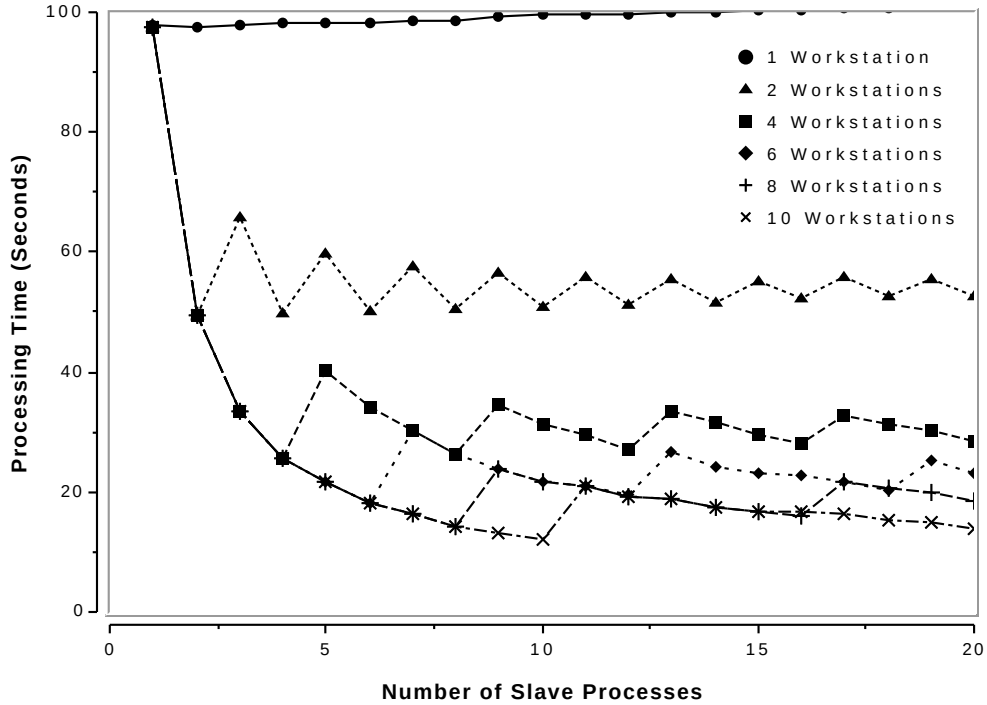

Figure 7: Results of Event Based Global Scheduling, Process Twin and Process Migration (without exit event)

The first test performed only allowed process migrations to occur on the creation of a process i.e., *process_twin()* not on *process_exit()* events. The new slave processes are then migrated away from the workstation that they were created on to workstations that are less loaded (equivalent to static allocation since once located on the destination workstation, they remain until they terminate). The experimental results of the loop parallel application utilising the *process_twin()* mechanism and supported by static allocation using process migration are presented in Figure 7. We notice that these results that the 'initial placement' environment produced are nearly identical to the results shown in Section 4.2.

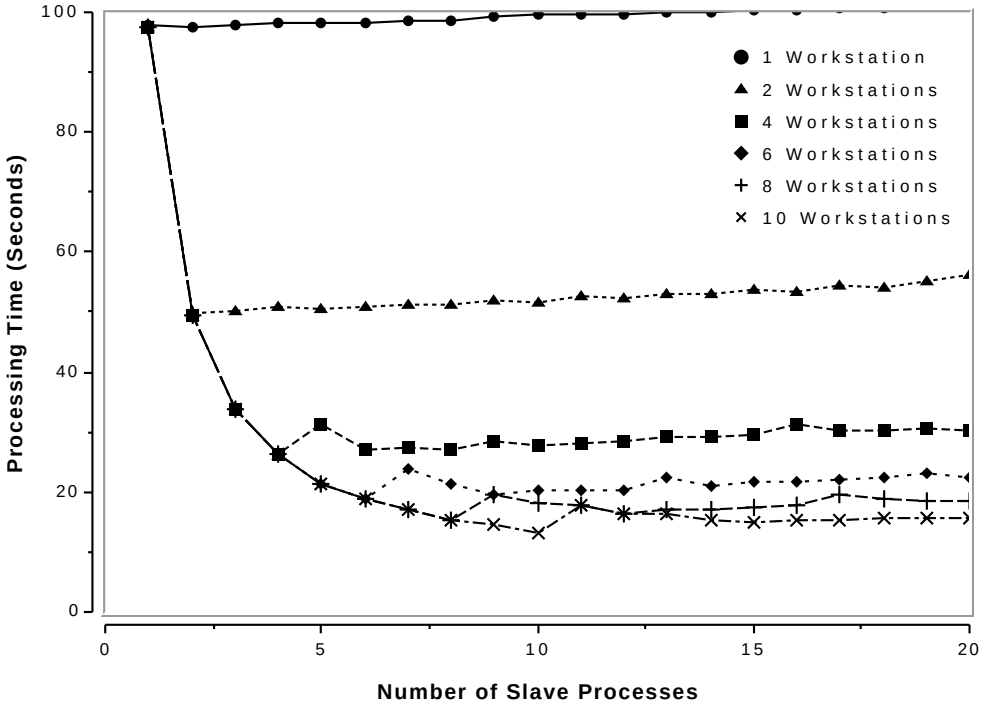

Figure 8: Results of Event Based Global Scheduling, Process Twin and Process Migration (allowing process exits)

The second test performed was to allow process migration to occur on both process twin and process exit events. The experimental results obtained from implementing the loop parallel application using *process_twin()* mechanism and supported by the full dynamic load balancing component of the RHODOS Global Scheduling facility are shown in Figure 8.

4.4 Event Based Global Scheduling employing Remote Process Creation and Process Migration

The pseudo code presented in Block 1a) was implemented and tested with the support of both static allocation and dynamic load balancing components of the Scheduling Facility. In these experiments new processes were produced using the process creation method; the static allocation component of the Global Scheduler distributed the load over the COW. Furthermore, *process_exit()* events were enabled to be considered by the Global Scheduler. This then provides

a full load balancing services to a loop parallel application. The results obtained from this implementation are shown in Figure 9.

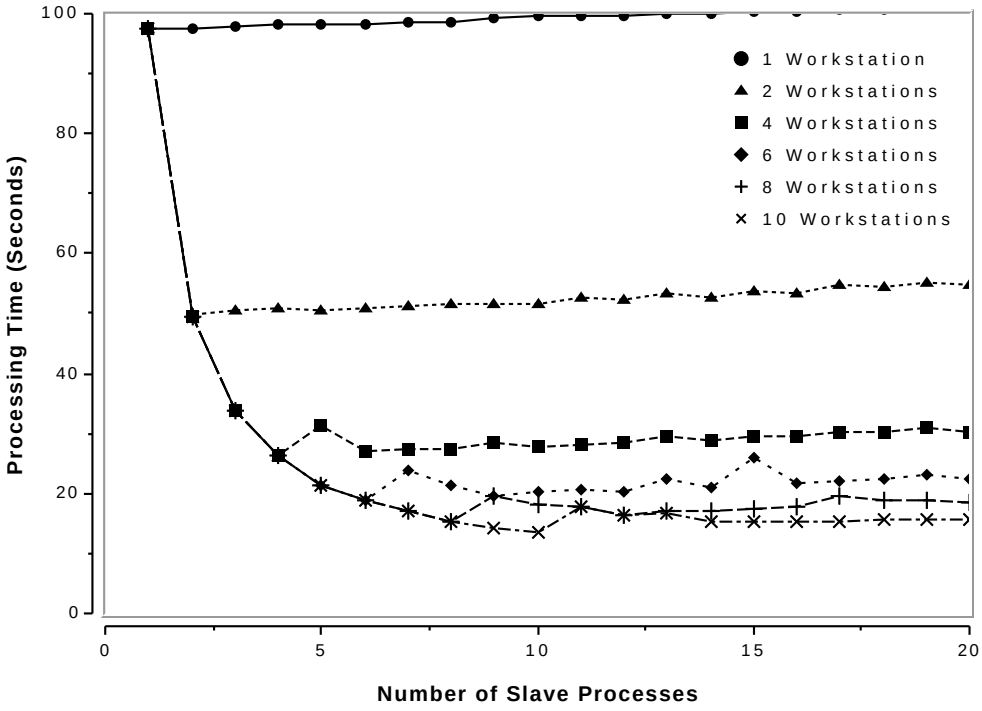

Figure 9: Results of Event Based Global Scheduling, Process Creation and Process Migration (allowing process exits)

4.5 Observations

In these experiments new processes were created (duplicated) using the *process_twin()* mechanism. This allowed the investigate of both the static allocation and dynamic load balancing facilities of the Global Scheduler by disabling and enabling notification of process exit events, respectively.

The experimental results obtained for the static allocation version using *process_twin()*, shown in Figure 7, are similar to the results obtained for the static allocation version using *process_create()* shown in Figure 6. Thus, by disabling process exit event notification to the Global Scheduler, we can support static allocation when processes are created using the *process_twin()* mechanism. Both methods improve the performance of the loop parallel application and both have the wave-like pattern evident as the number of slave processes is increased. We notice (Table 1) a slight performance improvement of the *process_twin()* and process migration version over the *process_create()* and remote execution version as the migration is slightly faster than the remote process creation. Table 1 shows the best case processing times of the loop parallel application executed in an environment supported by the Global Scheduling, Remote Process Creation and Process Migration for 1 to 10 slave processes on 1 to 10 workstations.

We observe a slight linear increase in processing times for both the static allocation and

dynamic load balancing methods utilising process migration as the number of slave processes is increased. This increase can also be attributed to the overheads for process migration, communication and context switching between multiple slave processes on the same workstation.

The RHODOS Global Scheduling facility can provide full dynamic load balancing by enabling the notification of *process_exit()* events. We can support dynamic load balancing when both *process_create() and process_twin()* mechanisms are used to initiate slave processes. The experimental results of both process initiation cases are presented in Figures 8 and 9, respectively. The most noteworthy observation we can present from these results are the loss of the periodic waveform as the number of the slave processes increases. This can be attributed to the dynamic characteristic of allowing all computation resources to be fully utilised by the loop parallel application. When a slave exits the slave processes can be redistributed using the process migration and dynamic load balancing methods, such that computational resources are left idle. The results show a marked performance improvement this modification to the Global Scheduler provides.

Table 1: Minimum Loop Parallel Processing Times (Best Times)

Number of Workstations	Static Allocation & Process Create (seconds)	Static Allocation & Process Twin (seconds)	Global Scheduling & Process Create (seconds)	Global Scheduling & Process Twin (seconds)
1	97.71	97.70	97.62	97.70
2	49.60	49.46	49.86	49.93
4	26.30	25.69	26.27	26.30
6	18.83	18.20	18.85	18.84
8	15.07	14.32	15.36	15.29
10	13.59	12.24	13.46	13.12

Table 2: Minimum Loop Parallel Processing Times (20 Slave Processes)

Number of Workstations	Static Allocation & Process Create (seconds)	Static Allocation & Process Twin (seconds)	Global Scheduling & Process Create (seconds)	Global Scheduling & Process Twin (seconds)
1	101.29	101.51	101.43	101.51
2	54.67	52.69	54.69	56.36
4	29.89	28.33	30.16	30.11
6	25.22	23.29	22.39	22.40
8	20.40	18.50	18.34	18.37
10	15.57	13.86	15.62	15.67

5 Conclusion

The RHODOS Global Scheduling Facility provides a unique service that is not found within any other distributed operating systems supporting the COW model. RHODOS transparently supports a combination of the static allocation and dynamic load balancing services. The RHODOS Global Scheduling Facility can balance the load over a COW under a variety of load conditions.

We have presented in this paper the need for a distributed operating system and advanced mechanisms required for the implementation of an Event Based Centralised Global Scheduling Facility which supports concurrently both static allocation and dynamic load balancing. The importance of advanced mechanisms such as remote process creation and process migration were highlighted to be critical in providing COW-wide load balancing.

The feasibility of such a Global Scheduling Facility was shown to be true through the testing and analysis of a computationally intensive parallel application run on a COW managed by the RHODOS system. A SPMD parallel application using both *process_create()* and *process_twin()* slave instantiation mechanisms was run over the RHODOS COW. These tests were supported separately by both static allocation and dynamic load balancing components of the Event Based Global Scheduler. We observed from these results that a performance improvement can be achieved from increasing the number of workstations in the COW. We also showed the benefits of using process migration and dynamic load balancing in removing the periodic performance, seen when static allocation is used, as the number of slave processes increases.

The initial experiments with the RHODOS Event Based Global Scheduling Facility are promising. Further work is required to investigate the impact which other parallel processing models have on the performance when executed using this environment. Other areas that also need further work include the unbalanced distribution of work to the slave processes; the impact of higher communication from the master to slave processes; the suitability of group communication in parallel processing on COWs; and the impact of background workloads on the overall performance.

6 References

[De Paoli and Goscinski 97] D. De Paoli, A. Goscinski. *"The RHODOS Migration Facility"*. The Journal of Systems and Software.

[Barak et al. 93] A. Barak, S. Guday, R. Wheeler. *"The MOSIX Distributed Operating System: Load Balancing for UNIX"*. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg.

[De Paoli et al. 95] D. De Paoli, A. Goscinski, M. Hobbs and G. Wickham. "*The RHODOS Microkernel, Kernel Servers and their Cooperation*". IEEE First International Conference on Algorithms and Architectures for Parallel Processing (ICA^{3}PP). Brisbane Australia. April.

[Goscinski 91] A. Goscinski. *"Distributed Operating Systems: The Logical Design"*. Addison-Wesley Publishing. 1991.

[Goscinski 97] A. Goscinski. *"Parallel Processing on Clusters of Workstations"*. Invited Paper, SICON-97, Singapore, April.

[Hobbs and Goscinski 96] M. Hobbs and A. Goscinski. "*The RHODOS Remote Process Creation Facility Supporting Parallel Execution on Distributed Systems*". Journal of High Performance Computing. Vol. 3, No. 1. December.

[Milojicic 94] D. S. Milojicic. *"Load Distribution. Implementation fro the Mach Microkernel"*. Vieweg Advanced Studies in Computer Science.

[Zhu 95] W. Zhu. *"Dynamic Load Balancing on Amoeba"*. Proceedings of IEEE First International Conference on Algorithms and Architectures for Parallel Processing. Brisbane, Australia. April.