

# Improving Gang Scheduling through Job Performance Analysis and Malleability

Julita Corbalan, Xavier Martorell and Jesus Labarta  
Universitat Politècnica de Catalunya (UPC)  
c/ Jordi Girona 1-3, 08034, Barcelona, Spain  
{juli,xavim,jesus}@ac.upc.es

## ABSTRACT

The OpenMP programming model provides parallel applications a very important feature: job malleability. Job malleability is the capacity of an application to dynamically adapt its parallelism to the number of processors allocated to it. We believe that job malleability provides to applications the flexibility that a system needs to achieve its maximum performance. We also defend that a system has to take its decisions not only based on user requirements but also based on run-time performance measurements to ensure the efficient use of resources. Job malleability is the application characteristic that makes possible the run-time performance analysis. Without malleability applications would not be able to adapt their parallelism to the system decisions. To support these ideas, we present two new approaches to attack the two main problems of Gang Scheduling: the excessive number of time slots and the fragmentation. Our first proposal is to apply a scheduling policy inside each time slot of Gang Scheduling to distribute processors among applications considering their efficiency, calculated based on run-time measurements. We call this policy Performance-Driven Gang Scheduling. Our second approach is a new re-packing algorithm, Compress&Join, that exploits the job malleability. This algorithm modifies the processor allocation of running applications to adapt it to the system necessities and minimize the fragmentation and number of time slots. These proposals have been implemented in a SGI Origin 2000 with 64 processors. Results show the validity and convenience of both, to consider the job performance analysis calculated at run-time to decide the processor allocation, and to use a flexible programming model that adapts applications to system decisions.

## 1. INTRODUCTION

Processor scheduling in a multiprocessor machine is a difficult task.

Scheduling algorithms must decide the job scheduling (which and how many jobs should be executed) and the processor allocation (how many and which processors allocate to each job). One of the main problems of the processor scheduling is how to deal with the problem of processor sharing. Traditionally, there have been three approaches to deal with this problem: space sharing, time sharing and space-time sharing (Gang Scheduling).

The space sharing approach [14][1] partitions the machine among a number of applications. Applications run on these partitions as in a dedicated machine. This approach has the advantages that minimizes the context switches and reduces the loss of performance due to synchronizations since all the threads of an application run simultaneously. However, it has the drawback that in execution environments with fixed processor allocation this technique can be unresponsive and introduces fragmentation. Dynamic space-sharing policies such as the equipartition [14] do not have these problems because processor allocation is modified dynamically.

The time sharing approach considers the thread as an individual scheduling basis without knowledge about applications or thread synchronization. This approach introduces a great number of context switches and loss of performance due to the synchronizations usually found in parallel applications. This approach is frequent in SMP machines and has not shown good results when the number of threads exceeds the number of processors in the parallel machine, as Tucker and Gupta showed in [23].

And finally, the combination of the two previous approaches, space-time sharing or Gang Scheduling [4][6], has the following characteristics: Application threads are grouped into gangs, threads in a gang are executed simultaneously, and time sharing is used among gangs. Common reasons to use Gang Scheduling are its responsiveness and efficient use of resources.

This work is based on two main ideas. The first one, and the more important, is that OpenMP [16] is a programming model easy to use and that it provides a very important feature to parallel jobs: job malleability [5]. Job malleability is the capacity of an application to dynamically adapt its parallelism to the number of processors allocated to it. We believe that the flexibility to adapt to the execution environment provided by malleability marks the difference among execution environments and clearly determines the system performance. If applications are malleable, the system can dynamically take the scheduling decisions and applications adapt their parallelism to its decisions. Otherwise, applications are rigid

and the system has to take its decisions as a function of the jobs requirements. Job malleability is the first point that we want to defend in this work, and also provides us the basic mechanism to the second important point in this work. This second point is that the number of processors allocated to an application should be determined by the scheduling policy, based on user request, but also considering run-time measurements to ensure a good processor utilization. Job malleability allows to perform run-time measurements and to dynamically modify the processor allocation of running jobs.

To demonstrate the validity of our ideas we present two new approaches to improve Gang Scheduling attacking the two main problems of Gang Scheduling: the excessive number of time slots [26] and the fragmentation [26][4].

**First approach:** The processor allocation policy used is an orthogonal factor to the fact of using Gang Scheduling. For this reason, we propose to apply a space-sharing policy to each time slot. This policy allocates processors to applications as a function of their performance measured at run-time. We demonstrated in [1][2] that this is a good approach in dynamic space-sharing environments. We will refer to this combination as Performance-Driven Gang Scheduling, PDGS.

**Second approach:** Since OpenMP applications are malleable, there is no need to suffer from the fragmentation. We can modify the processor allocation of running applications to fit an application in an idle processor or by reducing the number of time slots. We will present a re-packing algorithm, Compress&Join, that exploits this feature.

This work has been implemented and evaluated in a SGI Origin 2000 [25] with 64 processors. Since most of the previous works are based on simulations, we have included in this work an initial evaluation comparing Gang Scheduling with several dynamic space-sharing policies and with a batch policy to have a reference about the performance of these policies in our execution environment.

Results show that both approaches, the use of job performance analysis and the job malleability, offer significant gains compared with a traditional Gang Scheduling. However, in the experiments performed in this work, even with our improvements, Gang Scheduling has not outperformed the dynamic space-sharing policies evaluated as a reference in this work.

The remainder of this paper is organized as follows: Section 2 presents some related work, Section 3 describes some space sharing policies and compares them with Gang Scheduling. Section 4 presents our proposals to improve the performance of Gang Scheduling through job performance analysis and job malleability, and finally Section 5 presents the conclusions of this work.

## 2. RELATED WORK

Static space-sharing policies have the drawback that their decisions have a great impact in the response time of applications and in the resource utilization. With these policies, decisions taken by

the scheduler are critical since they can not be modified. Examples of static partitioning are FIFO, Smallest Job First [13] or Worst Fit (these policies can be used with Backfilling [10]). Dynamic space sharing does not have these problems because the processor allocation is dynamically modified. Examples of dynamic space-sharing policies are equipartition [14], equal\_efficiency [15], equip++ [2] or PDPA [1]. However, dynamic space sharing policies have been presented as difficult to implement in real systems.

Gang Scheduling is a technique proposed by Ousterhout in [17] that combines space and time sharing and it was presented as the solution to the problems of static space-sharing policies. Feitelson and Jette argue in [6] that Gang Scheduling solves the problem of taking incorrect decisions while performing job scheduling. Feitelson and Rudolph comment in [8] that academically speaking Gang Scheduling is inferior to dynamic partitioning, but that dynamic partitioning is difficult to implement and that the drawbacks of gang are not so critical. They conclude that “the advantages of Gang Scheduling generally outweigh its drawbacks”.

Several works have analyzed the problem of fragmentation in Gang Scheduling and have proposed several re-packing algorithms as solution to this problem. Feitelson in [4] analyzes several algorithms of job re-packing for Gang Scheduling and concludes that the best option is a buddy system or use migration to re-map the jobs (based on a first-fit algorithm). Feitelson and Rudolph propose and evaluate Distributed Hierarchical Control (DHC) in [7][9]. DHC is a design using a hierarchy of controllers that dynamically re-partitions the system according to changing job requirements. They show through simulations that DHC achieves performance comparable to off-line algorithms. Zhou *et al.* [26] also attack the fragmentation problem and present ideas such as job re-packing, running jobs on multiple slots and minimizing the number of time slots in the system to improve the buddy scheme for Gang Scheduling. Setia [18] shows through simulations that Gang Scheduling policies that support job migration offer significant performance gains over policies that do not use remapping. In this work, we have implemented a migratable Gang Scheduling.

Other authors have analyzed other aspects of the performance of Gang Scheduling under different kind of applications. Silva and Scherson propose Concurrent Gang [20] to improve the performance of I/O intensive applications. Using simulations they show that Concurrent Gang combines the advantages of Gang Scheduling for communication and synchronizations intensive applications with the flexibility of a Unix scheduler for I/O intensive applications. They also classify applications through run-time measurements in [21] to provide better service to I/O bound and interactive jobs under gang. They propose to improve the utilization of idle times (idle slots and blocked tasks) and to control the spinning time of tasks. Gare and Leutenegger [11] analyze the relation between job size and quantum allocation. They allocate a number of quanta inversely proportional to the number of processes per job to reduce the slowdown. Zhang *et al.* [27] combine backfilling and Gang Scheduling in Backfilling Gang Scheduling. As them, we also believe that space sharing scheduling is orthogonal to the Gang Scheduling concept, and this is one of the important points in this work.

### 3. GANG SCHEDULING VS. SPACE SHARING

In this section, we compare the performance of some space sharing policies vs. Gang Scheduling to have a reference and to confirm the problems of Gang Scheduling detected in previous works. The processor allocation policies that we have selected to compare with gang are: batch, equipartition [14], equip++ [2], and PDPA [1]. Equipartition, equip++ and PDPA are dynamic space-sharing policies. The job scheduling policy used in all the cases is an unrestricted Backfilling [12]. We have relaxed the restriction that jobs involved in backfilling do not delay previous jobs in the queue. However, in these experiments, the job scheduling policy only affects the execution of jobs when using the batch policy, because it is a static space-sharing policy.

#### 3.1 Space Sharing Policies

Batch and equipartition are space-sharing policies that take their decisions based on the user request. Equip++ and PDPA also consider the application performance measured at run-time. A brief description of each of these policies follows:

**Batch:** The batch policy is a static space-sharing policy that allocates the number of processors requested to each application, if available. Otherwise, the application waits until there are enough free processors.

**Equipartition:** Equipartition [14] is a dynamic space sharing policy that, to the extent possible, maintains an equal allocation of processors to all jobs. It assigns cyclically processors to applications till the requested of each application is reached or till there are no more processors available to allocate.

**Equip++:** Equip++ [2] is an improvement of the equipartition policy that considers the run-time measured efficiency achieved by applications, and ensures that applications achieve a certain value of processor efficiency.

**Performance-Driven Processor Allocation (PDPA):** The PDPA [1] is a dynamic space sharing policy that allocates a number of processors to each application that achieves an *acceptable* efficiency, calculated at run-time. The concept of *acceptable* is fully described in [1]. It mainly consists of reaching a pre-defined

target efficiency. The PDPA applies a search algorithm to each parallel application running in the system to decide the number of processors that achieves this *acceptable* efficiency.

#### 3.2 Gang Scheduling

Gang scheduling is a term that includes a group of scheduling policies with common features: threads are grouped into gangs, threads in each gang execute simultaneously on different processors, and time sharing is used among gangs. In this section we present the characteristics of Gang Scheduling that we have adopted. In this case, malleability is not involved at all.

Different scheduling policies can fit in the previous definition. We have implemented migratable preemptions [18], that is, threads in a gang can be preempted in a set of processors and resumed in another. Migratable Gang Scheduling has been shown to offer significant gains over policies that do not allow remapping (job migration).

Gang Scheduling manages a two dimensional matrix where one dimension represents processors and the other is time (this structure is known as the Ousterhout matrix [17]). Figure 1 is an example of a matrix in a machine with eight processors, where each column is a time slot, composed by a list of one or several gangs. Time-sharing is performed between slots and the sum of processors allocated to gangs in a slot must be less or equal than the number of processors in the machine.

In our implementation, the number of active time slots has been limited to five and the time-sharing quantum has been set to four seconds.

The packing policy implemented is a first-fit algorithm [4] (combined with job migrations) implemented in the following way:

- A new job is placed in the first slot with a sufficient number of idle processors.
- A job completion does not imply a job re-packing.

At each time-sharing quantum the scheduler performs the following steps:

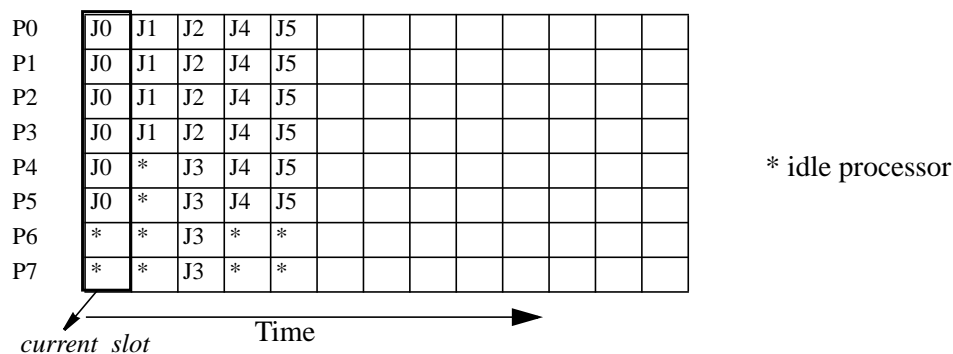


Figure 1: Ousterhout matrix for a system with eight processors

**Table 1: Application characteristics**

Characteristics/ Application(input)	Swim(ref)	Bt(A)	Hydro2d(train)	Apsi(ref)
Exec.Time. in Sequential	212.2 sec.	1066.21 sec.	223.7 sec.	99 sec.
Speedup with 8/16/32/48 proc.	21.6/36.5/ <b>44.2</b> /30.0	6.1/12.4/ <b>20.85</b> /20.59	4.6/5.4/ <b>6.3</b> /3.6	0.93/0.93/0.92

- Stops the currently running gangs (this implies to suspend all threads).
- Advance the *current\_slot* pointer and resume gangs in the new slot (resume all threads).
- If the total amount of processors used by gangs in the *current\_slot* does not fit the total number of processors in the machine, the algorithm searches for gangs that could be moved to this hole (starting from *current\_slot*+1). If it finds a gang that fits in the current slot the algorithm resumes it. Moreover, it tries to move gangs from low loaded slots to heavy loaded slots to reduce the number of time slots and the fragmentation.

We only allow to move gangs from low loaded slots to heavy loaded slots to avoid ping-pong effects that could be produced without this limitation.

### 3.3 Evaluation

In this section, we evaluate the performance of Gang Scheduling compared to the described space-sharing policies.

#### 3.3.1 System, Parallel Applications and Workloads

All the workloads presented in this paper have been executed in an Origin2000 with 64 processors with IRIX 6.5 [19]. Each processor is a MIPS R10000 [25] at 250 MHz, with two separated instruction and data L1 cache (32 Kbytes), and a secondary unified instruction/data cache (4 Mbytes). The memory migrations of IRIX has been activated to reduce the impact of migrating a job.

The execution environment is fully described in [1], and it is mainly composed by the OpenMP parallel applications, a user level scheduler, the CPU Manager, and a queueing system that controls the arrival of applications.

To evaluate our proposal we have selected four different applications: swim, hydro2d, apsi, and BT (class=A). The swim, hydro2d and apsi are applications from the SPECfp95 benchmarks, and the BT is from the NAS Parallel Benchmarks. Each one of them has different behavior considering the speedup. Table 1 presents the characteristics of these applications, from higher to lower speedup. Swim achieves a super-linear speedup, BT has a moderate-high speedup, hydro2d has low speedup and apsi has very bad speedup. In all the applications, except in apsi, the maximum speedup is achieved with 32 processors.

We have defined a mix of applications composed by the four applications with the following requests: swim (32 proc.), BT (32 proc.), hydro (32 proc.) and apsi (2 proc.). These numbers of pro-

cessors are those that achieve the maximum speedup per application.

Using these four applications we have generated several workload logs simulating different frequency of application arrival. These workload logs follow the standard workload format (swf) defined in [22] and they can be found in [24]. The workload logs have been generated assuming that applications arrive following an exponential distribution.

The equation on the left of Figure 2 is the operational analysis equality between demand and resource utilization in an open queueing system. We have taken as the reference of application demand its sequential time  $T_1^i$ . To generate a workload with equal demand for all the applications we use  $P^i=P/4$  (as there are four applications, each application uses 1/4 of the system processors).  $U$  is the system utilization that such an arrival rate would generate. With such arrival rates, we generated workload logs of arrivals during 300 seconds. Then, no more applications are submitted. We have generated workload logs with an utilization of 0.6(normal), 0.8 (high) and 1.0 (very high), that were then used to evaluate all the policies.

With fast arrival rates ( $U=1.0$ ), the experiments will evaluate the system ability to drain the queue of jobs that will build up during bursty job arrival phases.

Note that these workloads are computational intensive, but they do not evaluate the behavior of the policies under other scenarios such as I/O intensive or with interactive applications.

#### 3.3.2 Slowdown

Figure 3 shows the average slowdown achieved by the scheduling policies evaluated. The x axis is the load of the system (60%, 80% and 100%) and the y axis is the average slowdown. The slowdown of an application is calculated as the relationship between the application response time in the experiment and the execution time in a dedicated machine with the number of processors in which achieves the maximum speedup. As we can observe, all the dynamic space-sharing policies clearly outperform Gang Scheduling. The batch policy is one example of those static space-sharing policies that are very affected by job scheduling decisions. Gang Scheduling has not achieved a good performance because of the

$$\lambda^i \times T_1^i = P^i \times U \longrightarrow \lambda^i = \frac{P \times U}{4 \times T_1^i}$$

**Figure 2: Arrival rate calculation for application  $i$**

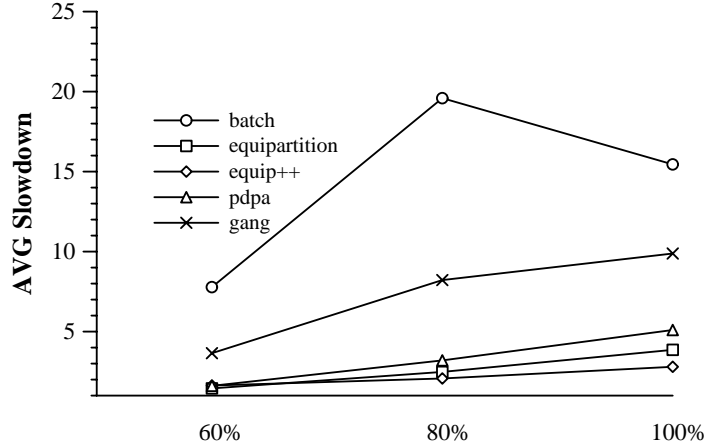


Figure 3: Space-sharing policies vs. Gang Scheduling

fragmentation and the number of time slots. To give an insight, we have measured that Gang Scheduling has an utilization around a 88% compared with an utilization about the 96% of the equipartition (in the experiment with  $U=100\%$ ).

Another important observation is that it results in better performance to queue applications that can not run, rather than to start them and perform time sharing. We have observed that having a large number of time slots is very harmful for the execution time of running applications.

We have separated in Table 2 the average queued time and the average running time in the equipartition, the equip++, the PDPA and Gang Scheduling policies (in the experiment with load  $=100\%$ ). The queued time is the time that applications are waiting in the system, controlled by the queueing system rather than by the short term scheduler. The full response time is the sum of the queued and running times. In space-sharing policies, jobs are queued if there are not available processors to execute. With Gang Scheduling, applications are queued if the maximum number of time slots has been reached. Observe that, despite the fact that jobs under Gang Scheduling run with the number of processors that achieves the maximum speedup, it results in the worst running

times. This is because the excessive number of time slots. Each time we add a new time slot, running jobs receive less proportion of cpu time. Moreover, this cpu time is not totally useful because a proportion is used to resume the applications and restore their status (caches, TLB, etc.). In addition, this slowdown means that applications stay in the system (processors, memory, etc.) much more time that in a space-sharing approach. This is a very important problem because we have executed the same set of experiments with the maximum number of time slots set to eight and we have found that the system crashes due to the unavailability of resources (processes)<sup>1</sup>. If the number of time slots is small, this problem is not so critical, but if the load of the system is very high, this becomes the more important drawback of Gang Scheduling. Observe that Gang Scheduling also have queued time. This is because in these experiments Gang Scheduling reaches the maximum number of time slots (set to five) and applications must be queued.

1. Results of these experiments with eight time slots are not included here due to the lack of space

Table 2: Average queued and running times, in seconds ( $U=100\%$ )

Policy/ Application	Swim		BT		Hydro2d		Apsi	
	queued	running	queued	running	queued	running	queued	running
EQUIPARTITION	0	31	0	268	0	97	0	107
EQUIP++	0	19	0	215	0	73	0	102
PDPA	48	9	59	175	42	61	42	100
GANG	76	22	190	239	96	155	133	227

```

while(1){
    time+=space_sharing_quantum;
    if ((time%time_sharing_quantum)==0){
        Change_slot();
        Restore_state();
    }
    Distribute_processors();
    sleep(space_sharing_quantum);
}

```

**Figure 4: PDGS Algorithm**

On the other hand, the equipartition adjusts the number of processors allocated to applications to adapt it to the load of the system. Then, the execution time is greater than in the PDPA and the equip++ cases because jobs are given a small number of processors but they execute more efficiently and there is no queued time.

The equip++ has a behavior similar to the equipartition but it is able to allocate processors to applications in a more efficient way (based on run-time measurements), resulting in the scheduling policy that achieves the best results.

Finally, in the case of the PDPA, it allocates more processors to jobs than the equipartition and the equip++, then the running time is smaller than the achieved in those policies, but the queued time is greater than the queued time of the equipartition and equip++. In these particular workloads, PDPA does not outperform equipartition and equip++ because all the applications except Hydro2d request a number of processors that initially achieve an acceptable efficiency. In [1], we show that if jobs request processors without knowledge of their performance, PDPA offers great benefits.

## 4. IMPROVING GANG SCHEDULING

The execution of applications in a non-malleable way introduces a lot of fragmentation. Moreover, the time sharing among slots can increase in excess the consumption of resources due to the increment in the number of time slots. The ideas that we will use in the next sections to improve Gang Scheduling are not new. The main contribution is their use along with Gang Scheduling.

### 4.1 Performance-Driven Gang Scheduling

We have observed that users normally ask for the number of processors that achieves the best speedup in their applications, or even they request for the maximum number of processors in the machine without knowledge about their job performance. A lot of jobs achieve their maximum speedup at the expense of processor efficiency, and with much less processors the application would achieve quite the same performance. We believe that the number of processors used by an application should be determined by the scheduling policy. It should be based on the user request but also considering run-time measurements to ensure the processor utilization. And that is orthogonal to the fact of performing time-sharing.

For this reason, we present the Performance-Driven Gang Scheduling, PDGS. The PDGS applies the philosophy used in [1][2] to

Gang Scheduling. The PDGS provides control on the number of processors allocated to each job and ensures that processors are used efficiently. The control of the processor efficiency achieved provides an indirect benefit because it results in a reduction in the number of processors allocated to jobs, generating a reduction in the number of time slots.

The PDGS uses two different quanta: the time-sharing quantum and the space-sharing quantum. The goal of having two different quanta is to decide the processor allocation at a more fine grain than the time-sharing quantum. The scheduler wakes up at each space-sharing quantum to distribute processors among applications active in the current slot (that does not imply necessarily a processor re-allocation). The time-sharing quantum is a multiple of the space-sharing quantum, and implies the activation of applications in the next slot. Figure 4 shows the main loop of the scheduler. At each time-sharing quantum the scheduler changes the slot and selects a new set of applications that will run during the next time-sharing quantum.

## 4.2 Malleable Gang Scheduling

Even adjusting the number of processors that each application uses, Gang Scheduling has the drawback of fragmentation. In our execution environment, fragmentation does not have sense because jobs can adapt their parallelism to the resources available. In this section, we will evaluate the effect of exploiting the malleability of OpenMP applications through a re-packing algorithm, Compress&Join.

### 4.2.1 The Compress&Join Algorithm

The Compress&Join algorithm re-generates the matrix used in Gang Scheduling. The goal of this algorithm is to minimize the overhead introduced by the context switch between slots by reducing the number of time slots. We assume that the speedup provided by reducing the number of slots is greater than the slowdown produced by the reduction in the number of processors allocated to each application.

For instance, consider a simple case when we have one time slot that runs a parallel application with 64 processors. In that case this application does not suffer slowdown because with one time slot there are not context switches. If a new application arrives requesting 64 processors, a normal packing algorithm opens a new time slot and performs time sharing between the two applications. In that case each one receives the 50% of the cpu time and suffers and slowdown of 2 (in average). The Compress&Join algorithm will adjust the processor allocation of each application to 32 processors, reducing the number of time slots from two to one. After applying the Compress&Join algorithm, each application will receive half the number of processors than with a normal algorithm, but it will not suffer any context switches. We assume that the benefit generated by reducing the number of time slots is greater than the penalty by reducing the processor allocation.

Based on this consideration, the *Compress&Join* algorithm works as shown in Figure 5.

```

void Compress_and_Join()
{
    active_slots=1;
    ap=0;
    while(app<MAX_APP){
        slot=0;
        while((slot<MAX_SLOT)&&
            (slot<active_slots)&&
            (compressed==0)){
            compressed=Compress_Slot(app,slot);
            slot++;
        }
        if (slot==active_slots){
            compressed=Compress_Slot(app,slot);
            Check_Error(compressed);
        }
        app++;
    }
}

int Compress_Slot(app,slot)
{
    if (fits(req(app),slot)){
        add(app,slot);
        actualize_active_slots();
        return 1;
    }else{
        calculate_efficiencies(slot);
        cpus_needed=max_cpus-
            (cpus_used(slot) + req(app));
        if (reduce_cpus(cpus_needed,slot)==1){
            add(app,slot);
            return 1;
        }else{
            return 0;
        }
    }
}

```

**Figure 5: Compress&Join algorithm**

The algorithm first tries to add the application in a open slot. If the application does not fit in an open slot, the algorithm adds a new slot. The *fits()* function checks if there are enough free processors in the slot to add the new application. In that case the application is added without modifying its allocation. Otherwise, the algorithm tries to fit the application by *compressing* both the current application and any other applications added previously to the slot. The *reduce\_cpus()* function temporarily modifies the allocation and calculates the resulting efficiencies to check if it is possible to fit the new application in this slot. If the penalty introduced in the set of applications does not exceed the 50% the application is added and the modifications computed by the *reduce\_cpus()* function are fixed in the slot. Otherwise, the slot is not modified and the algorithm tries to add the job to another slot.

Figure 6 shows the resulting matrix after applying the *Compress&Join* algorithm to the matrix presented in Figure 1. We can see how, after applying the *Compress&Join* algorithm, jobs have been proportionally reduced.

P0	J0	J2	J4			
P1	J0	J2	J4			
P2	J0	J2	J4			
P3	J0	J2	J4			
P4	J0	J3	J5			
P5	J1	J3	J5			
P6	J1	J3	J5			
P7	J1	J3	J5			

**Figure 6: Matrix generated by the *Compress&Join* algorithm**

This algorithm uses the job efficiency. If the job efficiency is not available, this algorithm assumes that all the jobs have initially an efficiency of 0.8. Moreover, in that case, we extrapolate the complete efficiency curve using the function proposed by Dowdy in [3].

#### 4.2.2 Evaluation

Figure 7 shows the average slowdown achieved by Gang+Compress&Join, PDGS, PDGS+Compress&Join, and equip++. The equip++ has been added as a space-sharing reference because it achieved the best performance. The execution environment and workloads are the same that in the previous section. We can observe that the Compress&Join algorithm provides an additional benefit to that obtained when considering the job performance analysis through the PDGS. This algorithm is an optimization and can be applied independently of the scheduling algorithm used.

To give an insight about what really happens we have separated in Table 3 the queued time and the running time of Gang Scheduling, Gang+Compress&Join, PDGS, and PDGS+Compress&Join. The more significant improvement is that the queued time has been significantly reduced. This is because with the same number of time slots we are able to run many more applications and they do not have to be queued. As you can see, the queued time has been reduced but not at the expense of the running time. Taking into account that these numbers are averaged, we can say that the execution with *Compress&Join* outperforms the execution without it.

However, despite all our optimizations we can see that Gang Scheduling does not reach the performance of equip++. This is because equip++ also exploits the two points that we defend in this work: the job performance analysis and the job malleability and it does not suffer the slowdown introduced by the time-sharing among applications, improving the resource utilization.

The workload logs evaluated in this work do not introduce a significant fragmentation. Nevertheless, we have performed other experiments, not included in this paper due to the lack of space, that check the behavior of Gang Scheduling with workloads that generate a significant fragmentation. We have found that the use of job malleability provides to Gang Scheduling the additional benefit of the insensitivity to this kind of pathological workloads.

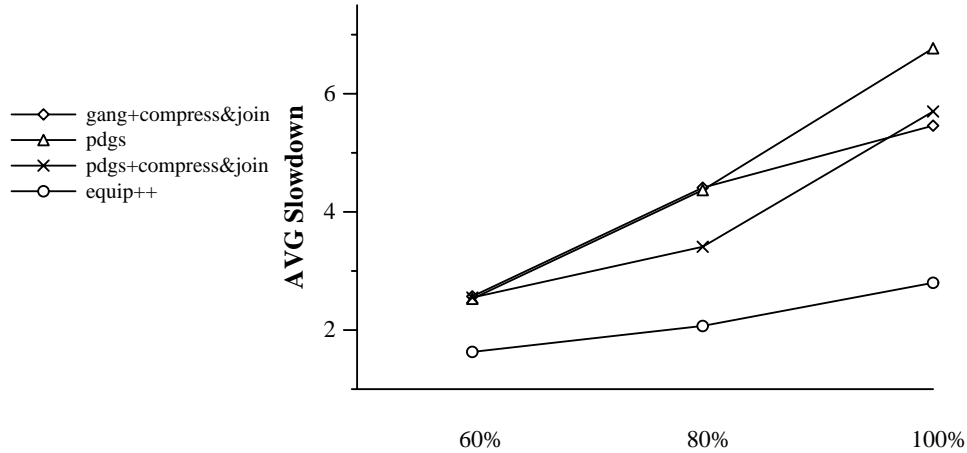


Figure 7: Gang Scheduling compared to PDGS in combination with the *Compress&Join* algorithm

## 5. CONCLUSIONS

Job performance analysis and job malleability have been shown as valid and useful tools to improve the system performance. In this work, we have applied these ideas to the particular case of improving Gang Scheduling. We have presented the Performance-Driven Gang Scheduling policy and the Compress&Join algorithm.

The PDGS policy considers run-time measured performance of jobs to improve the job processor allocation. It is important to track the performance of applications under Gang Scheduling because the job performance highly depends on the number of context switches (generated by the time-sharing among slots).

The Compress&Join algorithm adapts the processor allocation of running jobs to avoid the fragmentation and to minimize the number of time slots. These two approaches to improve Gang Scheduling have shown clear benefits compared to the original Gang Scheduling. The combination of PDGS and Compress&Join algorithm has shown the best improvement.

However, our techniques to improve Gang Scheduling do not reach the same performance that some of the dynamic space-sharing policies evaluated in this work due to the loss of performance that context switches among slots introduce.

As future work, it should be interesting to compare Gang Scheduling vs. dynamic space-sharing policies with workloads with different characteristics or in different multiprocessor architectures such as clusters.

## 6. ACKNOWLEDGEMENTS

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511 and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1999FI 00554 UPC APTIND. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

Table 3: Average queued and running times, in seconds (U=100%)

Policy/Application	Swim		BT		Hydro2d		Apsi	
	queued	running	queued	running	queued	running	queued	running
GANG	76	22	190	239	96	155	133	227
GANG+COMPRESS&JOIN	12	25	24	344	7	137	14	290
PDGS	38	28	68	250	20	153	46	190
PDGS+COMPRESS&JOIN	8	26	8	345	3	241	9	203

## 7. REFERENCES

- [1] J. Corbalán, X. Martorell, J. Labarta, "Performance-Driven Processor Allocation", Proc. of the 4th Symposium on Operating System Design & Implementation (OSDI2000), 2000.
- [2] J. Corbalán, J. Labarta, "Improving Processor Allocation through Run-Time Measured Efficiency", To be published in the International Parallel and Distributed Processing Symposium, IPDPS2001.
- [3] L. Dowdy, "On the Partitioning of Multiprocessor Systems", Tech. Report, Vanderbilt University, June 1988.
- [4] D. G. Feitelson, "Packing Schemes for Gang Scheduling", Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, pp. 89-110, Springer-Verlag, 1996.
- [5] D. G. Feitelson. "Job Scheduling in Multiprogrammed Parallel Systems". IBM Research Report RC 19790 (87657), October 1994, rev. 2, 1997.
- [6] D. G. Feitelson, M. A. Jette, "Improved Utilization and Responsiveness with Gang Scheduling", Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science vol. 1291, pp. 238-261, Springer-Verlag, 1997.
- [7] D. G. Feitelson, L. Rudolph, "Distributed Hierarchical Control for Parallel Processing", Computer 23(5), pp. 65-77, May 1990.
- [8] D. G. Feitelson, L. Rudolph, "Toward Convergence in Job Schedulers for Parallel Supercomputers", Job Scheduling Strategies for Parallel Processing, Lectures Notes in Computer Science vol. 1162, pp. 1-26, Springer-Verlag 1996.
- [9] D. G. Feitelson, L. Rudolph, "Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control", journal of Parallel and Distributed Computing 35(1), pp. 18-34, May 1996.
- [10] D. G. Feitelson, A. M. Weil, "Utilization and predictability in scheduling the IBM SP2 with backfilling", In 12th International Parallel Processing Symposium, pp. 542-546, April 1998.
- [11] G. Ghare, S. Leutenegger, "The effect of Correlating Quantum Allocation and Job Size for Gang Scheduling", Job Scheduling Strategies for Parallel Processing, 1999.
- [12] D. Lifka, "The ANL/IBM SP scheduling system". In Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science vol. 949, pp. 259-263, Springer-Verlag 1995.
- [13] S. Majumdar, D. L. Eager, R. B. Bunt, "Scheduling in multiprogrammed parallel systems", in SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 104-113, May 1988.
- [14] C. McCann, R. Vaswani, J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors", ACM Transactions on Computer Systems, 11(2), pp. 146-178, May 1993.
- [15] T. D. Nguyen, J. Zahorjan, R. Vaswani, "Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling". Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, Springer-Verlag, Univ. of Washington, 1996.
- [16] OpenMP Organization. "OpenMP Fortran Application Interface", v. 2.0 <http://www.openmp.org>, June 2000.
- [17] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems", In Third International Conference on Distributed Computing Systems, pp. 22-30, 1982.
- [18] S. K. Setia, "Trace-driven Analysis of Migration -based Gang Scheduling Policies for Parallel Computers", ICPP97, August 1997.
- [19] Silicon Graphics Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. <http://techpubs.sgi.com>, Document Number 007-3430-002, 1998.
- [20] F. A. B. Silva, I. D. Scherson, "Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang", International Parallel and Distributed Processing Symposium 2000.
- [21] F. A. B. Silva, I. D. Scherson, "Improving Parallel Job Scheduling Using Runtime Measurements", 6th Workshop on Job Scheduling Strategies for Parallel Processing, 2000.
- [22] "The Standard Workload Format", <http://www.cs.huji.ac.il/labs/parallel/workload/swf.html>
- [23] A. Tucker, A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", in 12th Symp. Operating Systems Principles. pp. 159-166, dec. 1989.
- [24] Workload logs, <http://www.ac.upc.es/homes/juli>
- [25] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor". IEEE Micro vol. 16, 2, pp. 28-40, 1996.
- [26] B. B. Zhou, D. Walsh, R. P. Brent, "Resource Allocation Schemes for Gang Scheduling", Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1911, pp. 74-86, Springer-Verlag, 2000.
- [27] Y. Zhang, H. Franke, J. E. Moreira, A. Sivasubramaniam, "Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques", IPDPS2000, Cancun, Mexico, May 2000.