Chapter 5

Experimental Results (Implementation)

In this chapter, the coscheduling mechanisms are analyzed and compared by means of results obtained experimentally. The experimentation was performed in a non-dedicated Linux Cluster made up of eight PVM-Linux PCs with the following characteristics: 350Mhz Pentium II processor, 128 MB of RAM and 512 KB of cache. All of them are connected through a 100Mbps bandwidth Ethernet network and a minimal latency in the order of 0.1 ms.

The choice of the above experimental environment was not arbitrary. We are interested in commodity Clusters, made up of commercial hardware, that are widely used in actual LANs. The choice of Linux and PVM has been already explained in chapter 1. Furthermore, the open source feature of Linux and PVM simplified the implementation of the different coscheduling techniques that are analyzed in this chapter.

The evaluation of the different implemented coscheduling models was performed by using three kinds of distributed application:

- The EP, IS, MG, FT, CG, BT and SP kernel benchmarks from the NAS parallel benchmarks suite ([56],[57]). These are explained in section 5.1.1.

- The COMMS1, COMMS2 and SYNCH1 Low_Level benchmarks ([57]). Explained in depth in section 5.1.2.

- Synthetic benchmarks. Some coscheduling features (or differences) cannot be measured by using the above mentioned benchmarks. For example, it is very difficult (or impossible) to modify the message delivering frequency or the message pattern of the NAS benchmarks. With this in mind, some sort of synthetic benchmarks have been developed. So, in some situations, it was necessary to design new synthetic benchmarks. All of them are presented in the sections where they are used.

All these benchmarks are designed to be executed in the PVM environment. There are versions for MPI but these were discarded because for comparing the coscheduling models it was necessary to always use the same applications and DCE. The communication between remote tasks uses the *RouteDirect* (i.e. TCP/IP) mode because this avoids the intermediate participation of the PVM daemon (*pvmd*), and thus more accurate results can be obtained.

As mentioned above, the experiments were performed in a non-dedicated Cluster (the kind of system we are interested in). This non-dedicated property means that distributed applications execute jointly with the owner, local or user workload (or simply workload). Accordingly, the way the workload was characterized in this experimentation is another important point to be analyzed separately.

All the NAS and Synthetic benchmarks were executed in 4 and 8 Cluster nodes. The Low_Level benchmarks used 2 nodes. Almost all the benchmark mappings was performed by assigning one forming task per processor. Every result shown in this chapter is the arithmetic mean of five executions.

5.1 Benchmarks

A broad range of benchmarks are available and explained in the literature. The choice of a representative set of distributed benchmarks for measuring coscheduling performance in a Cluster system was the prior experimental objective of this chapter.

Due to historical causes, most of the available benchmarks are implemented for shared-memory multiprocessors (a MPP property), which cannot be imple-

mented in distributed-memory environments such as Cluster systems. This kind of benchmark was thus discarded.

As already introduced in chapter 1, the distributed applications can be coarse-grained and fine-grained, which have respectively a low and high degree of communication/synchronization. The coscheduling methods can be applied in both kinds of applications, but as coscheduling methods attempt to minimize the waiting time spent in synchronizing distributed tasks which communicate remotely, one should expect coscheduling gains to be more significant in the fine-grained category. Thus, a representative set of benchmarks should contain fined-grained applications representing as large a message-pattern range as possible.

In general, the NAS Kernel Benchmarks meet the above explained necessities and for this reason were chosen.

Another key question to be analyzed is how well the coscheduling methods behave in combination with a group of synchronization or message sending and receiving primitives, such as: simple send and receive, barriers, etc... The Low_Level benchmarks were chosen in doing so. They are principally intended to measure the performance in the execution of this kind of communication primitives.

5.1.1 Kernel Benchmarks

The kernel benchmarks ([56, 57]) are a set of parallel programs developed by the NAS (Numerical Aerodynamic Simulation) program, based at NASA Ames Research Center. Together they simulate the computation and data movement characteristics of large scale Computational Fluid Dynamics (CFD) applications. The motivation was to develop a set of benchmarks for the performance evaluation of highly parallel computers.

Those benchmarks have also been widely used in distributed environments, such as Cluster systems. They involve substantially larger computations than previous kernel benchmarks, such as the Livermore Loops [58] or Linpack [59], and are therefore more appropriate for the evaluation of parallel machines. The use of these benchmarks in the performance evaluation of the different coscheduling models is certainly a validation guarantee.

Bench.	Class T		Class A	
	Problem size	**Memory (MBytes)**	**Problem size**	**Memory (MBytes)**
EP	2^{24}	0.1	2^{28}	0.58
IS	2^{16}-$\lceil 0..2^{11} \rceil$	0.3	2^{23}-$\lceil 0..2^{19} \rceil$	39
MG	32^3	15	256^3	112
CG	1400	3.2	14000	55
FT	64^3	4.7	256^3	OM
BT	12^3	1.2	64^3	11.8
SP	12^3	1.3	64^3	42

Table 5.1: NAS Parallel Benchmarks. Memory: max. resident set size of each node when the benchmark is executed in 4 nodes. OM: Out of Memory.

For each benchmark, four different classes are defined (classes T, A, B and C). They differ in the problem size (mainly, in the size of the principal arrays). Table 5.1 shows the problem sizes (also are the input arguments) of each benchmark as well as the resident memory used in one Cluster node. This table only shows values for class T and A, because they are the only classes used in the experimentation. Normally, the resource necessities (principally Main Memory) of the remaining classes exceed the Cluster capabilities, and so they are not appropriate for executing in such systems. Consequently, they were discarded.

Next, the general properties of the kernel benchmarks used are described. As we are interested in increasing performance of fine-grained applications, the communication pattern of the different benchmarks were analyzed in depth. In doing so, PGPVM2 [75] and Paragraph [46] were used. PGPVM2 is an enhancement package for PVM that produces trace files for posterior use with standard ParaGraph. ParaGraph is a graphic display tool for visualizing the behavior and performance of parallel programs.

The results obtained in the execution of the kernel benchmarks in a PVM-Linux environment, which were traced with PGPVM2, are shown in different manners. The communication pattern of the benchmarks can easily be obtained by observing three different Graphics figures (generated with ParaGraph) and the TIME tables. Times in the Graphics figures are much greater than when the same applications (with also the same class and number of nodes) were executed with-

out the PGPVM2 tracing facility (shown in the TIME tables).

- Graphics. To obtain them 4 nodes were used.

    1. One of these graphs is the *Utilization Gantt Chart*. It shows the CPU utilization over time (in seconds) of the different Cluster nodes where tasks making up the applications were executed.
    2. The *SpaceTime Diagram* illustrates the communication between the different processes. It shows the message length (with its histogram graph, MSG LTH, in bytes) and the elapsed time in their transmission (in seconds).
    3. Finally, the Phase Portrait graph depicts the relationship over time between communication and processor utilization. At any time, the current percentage utilization and the percentage of the maximum volume of communication currently in transit together define a single point in a two-dimensional plane. This graph is particularly useful for revealing repetitive or periodic behavior in a parallel program, which tends to show up in the Phase Portrait as an orbit pattern.

- TIME tables: they show the total execution time and the communication time of each benchmark in this order. Experiments were done in 4 and 8 nodes. The number of local tasks (workload) was varied between 0 and 5. The characterization of the local workload is explained later in section 5.2.

The different NAS kernel benchmarks are introduced below. Special attention is given to their communication features.

EP: An "embarrassingly parallel" kernel. It provides an estimate of the upper achievable limits for floating point performance, i.e., the performance without interprocessor communication. Fig. 5.1 shows the Utilization Gantt Chart and the SpaceTime Diagram of this benchmark. It can be seen that the benchmark is entirely CPU bound. The slave tasks only communicate with the master task in the final part to send it their respective results.

Figure 5.1: EP class A.

		IS times (in sec.)					
		number of local tasks					
nodes	*Class*	0	1	2	3	4	5
4	*T*	0.63/0.52	1.1/0.9	1.7/1.6	2.2/2.1	2.6/2.5	2.9/2.8
	A	101/86	234/206	302/259	379/325	457/390	520/440
8	*T*	1.6/1.36	3.5/3.4	3.4/3.3	3.2/3.1	3.8/3.7	3.9/3.7
	A	84/75	194/179	235/213	267/240	302/267	332/292

Table 5.2: IS Execution/Communication times.

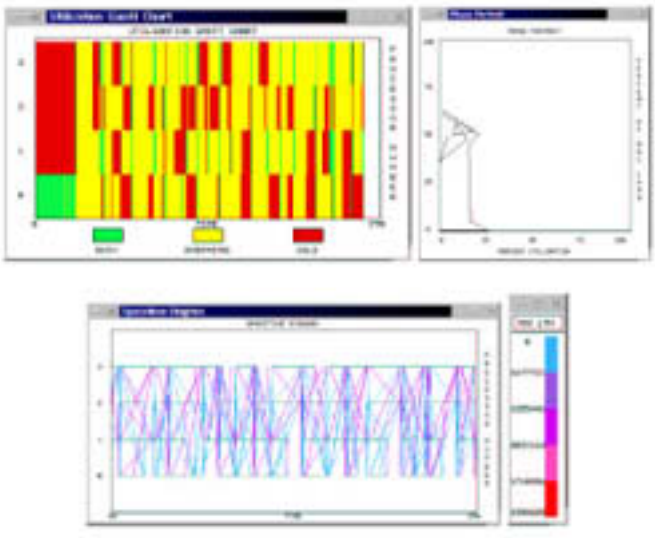


Figure 5.2: IS class A.

		MG times (in sec.)					
		number of local tasks					
nodes	*Class*	0	1	2	3	4	5
4	*T*	13.5/5.9	27/19	40.5/29	53/46	66/60	79/68
	A	102/44	208/90	301/120	399/180	498/200	595/240
8	*T*	9.9/4.9	20.4/15	25/17.7	31/20.8	39/29	46/36
	A	63/35	124/79	173/96	223/140	275/178	327/201

Table 5.3: MG Execution/Communication times.

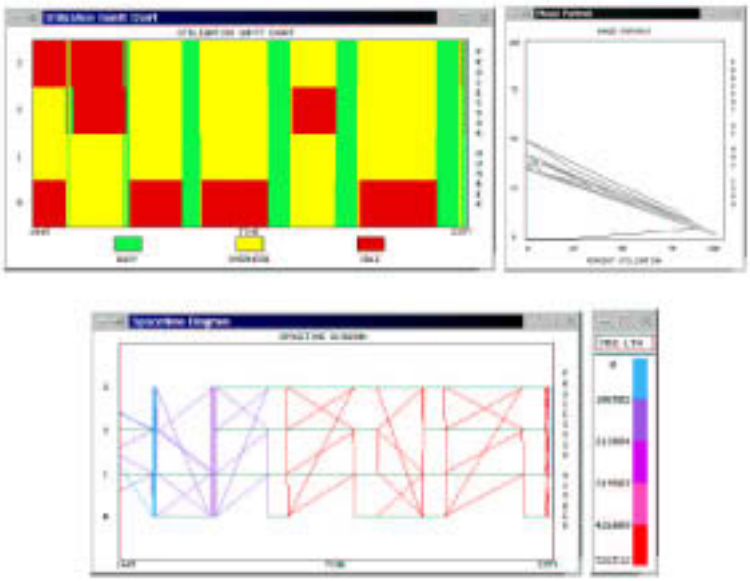


Figure 5.3: MG class A.

		FT times (in sec.)					
		number of local tasks					
nodes	*Class*	0	1	2	3	4	5
4	*T*	13.5/9	22/15.5	26.8/19.5	34.3/26	40.6/30.5	48.4/38.4

Table 5.4: FT Execution/Communication times.

IS: A large integer sort. This kernel performs a sorting operation (it sorts N keys in parallel). It tests both integer computation speed and communication performance. The keys are generated by a sequential key generation algorithm which initially must be uniformly distributed in memory.

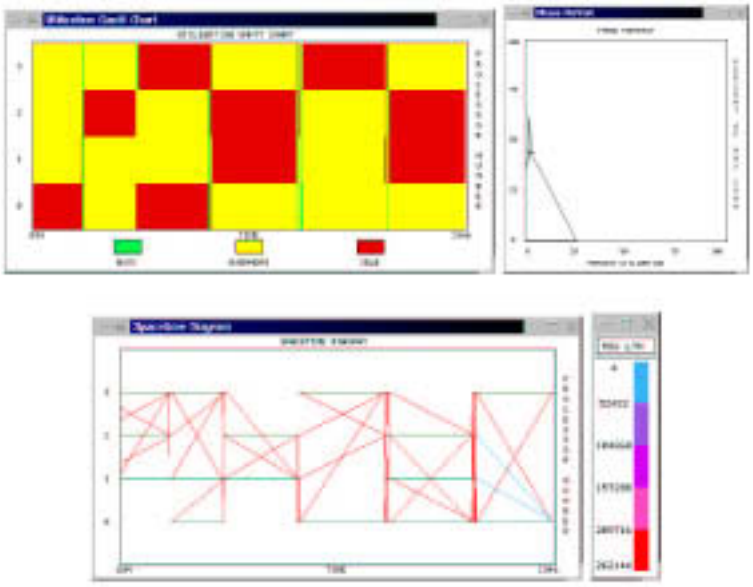


Figure 5.4: FT class T.

		CG times (in sec.)					
		number of local tasks					
nodes	*Class*	0	1	2	3	4	5
4	*T*	3.6/2.2	9/7.3	16/14.5	17.5/15.9	23/21.2	31/29.4
	A	58/36	131/91	205/165	279/238	356/314	431/390
8	*A*	64.3/47	102/85	130/113	166/149	203/186	242/224

Table 5.5: CG Execution/Communication times.

		BT times (in sec.)					
		number of local tasks					
n	*C*	0	1	2	3	4	5
4	*T*	40.4/14.4	99/73	149/123	197/171	249/223	305/279
	A	571/199	1159/562	1702/983	2263/1439	2834/1885	3405/2370
8	*A*	353/173	619/420	892/654	1168/813	1456/1074	1731/1233

Table 5.6: BT Execution/Communication times. *n*: nodes, *C*: Class.

MG: A simplified 3D multi-grid kernel. It requires highly structured long distance communication and tests computation speed and both short and long distance data communication.

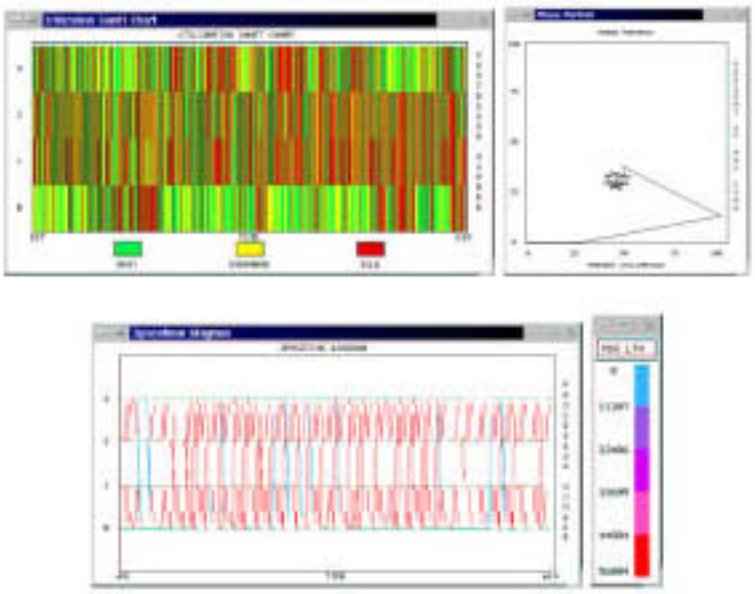


Figure 5.5: CG class T.

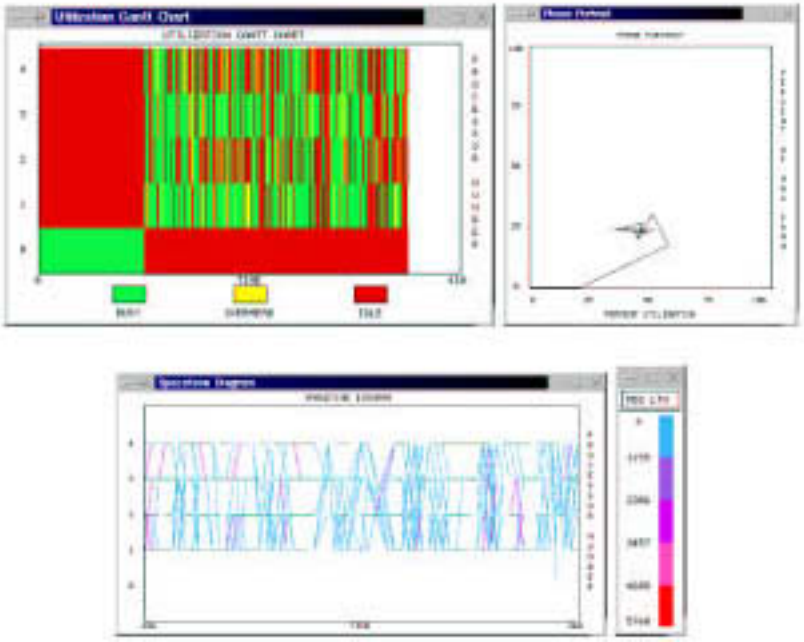


Figure 5.6: BT class T.

		SP times (in sec.)					
		number of local tasks					
nodes	*Class*	0	1	2	3	4	5
4	*T*	22.2/15.3	52/45	79/72	102/96	128/121	154/147
	A	397/275	546/381	782/550	1018/722	1260/893	1489/1065
8	*A*	210/168	353/284	458/369	567/457	683/551	794/644

Table 5.7: SP Execution/Communication times.

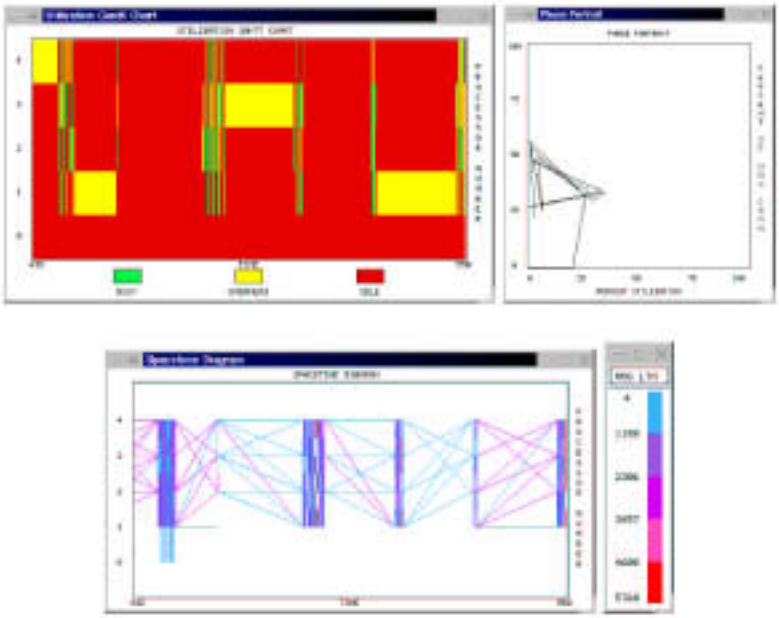


Figure 5.7: SP class T.

FT: A 3-D partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long-distance communication performance.

CG: A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication.

BT and SP: The SP and BT algorithms have a similar structure: each solves three sets of decoupled systems of equations, first in the x, then in the y, and finally in the z direction. SP and BT differences are in the kind of computation used in solving these systems of equations. In both the SP and BT, the granularity of communications is kept large and fewer messages are sent.

5.1.2 Multiprocessor Low_Level Benchmarks

The PARKBENCH suite of benchmark programs ([57]) provides low-level benchmarks to characterize the basic communication properties of an MPP (or message passing MIMD computer or Cluster system). From among all of these, we have chosen the COMMS1, COMMS2 and SYNCH1 benchmarks.

The benchmarks COMMS1 and COMMS2 were used to measure the communication rate (r).

For long messages, r is approximated by r_∞, defined as the asymptotic bandwidth of communication which is approached as the message length tends to infinity. This is $r \approx r_\infty$.

On the other hand, for short messages, the communication rate (r) is approximated by $\left(r_\infty / n_{1/2}\right) * n$, where n is the message length and $n_{1/2}$ is the message length required to achieve half the asymptotic rate r_∞. This is $r \approx \left(r_\infty / n_{1/2}\right) * n$.

The SYNCH1 benchmark measures the number of barrier statements that can be executed per second as a function of the number of Cluster processors taking part in the barrier.

5.1.2.1 Communication Benchmarks

COMMS1, or ping-pong benchmark, measures the basic communication properties of a Cluster system. A message of variable length, n (in Bytes), is sent from the master to a slave node. The slave node receives the message in a Fortran data array, and immediately returns it to the master.

In the COMMS2 benchmark there is a message exchange in which two nodes simultaneously send messages to each other and return them. In this case, advantage can be taken of bidirectional links, and a greater bandwidth can be obtained than is possible with COMMS1.

5.1.2.2 Synchronization Benchmark

SYNCH1 measures the time to execute a barrier synchronization statement as a function of the number of processes taking part in the barrier. The practicability of massively parallel (or distributed) computation depends on the execution performance of barrier statements. The results are quoted both as a barrier time, and as the number of barrier statements executed per second (barr/s).

5.2 Local Workload Characterization

The workload characterization was carried out by means of running an application (named *calcula*) which performs floating point operations indefinitely. There were no system calls inside it, and thus its execution only occupies user time (and no system time). This way, it was possible to fix its mean CPU utilization at 99%.

We want to simulate a mixed workload formed by processes that repeatedly require CPU execution time slices of variable length. The wide range of possibilities in the choice of this workload influenced the search for an intermediate solution. We provided *calcula* with the ability to yield the CPU (but without leaving the RQ) after a predetermined computing phase ($\simeq$ 10 ms) had elapsed.

This way we do not collapse the CPU by executing local tasks. The execution slices were selected this way to simulate the interactivity that generally characterizes local tasks.

One variant of *calcula* (*calcula2*) was also implemented. As we will see, it served to obtain and compare the execution penalties into the local tasks produced by the coscheduling models. Unlike *calcula* (which executes forever), the total execution time in a dedicated node with the plain Linux is fixed to $\simeq$ 80s.

To quantify the local workload overhead introduced in the execution of one distributed application in a determined coscheduling environment, the *Local Overhead* (*LO*) metric is used. The local benchmark *calcula2* will be used in doing so. The execution time of *calcula2* is used as a reference for comparing the added overhead of the different coscheduling models. *Local Overhead* (*LO*) is defined as follows:

$$LO = \frac{ET_{dedicated}}{ET_{nondedicated}} \tag{5.1}$$

where $ET_{dedicated}$ is the execution time spent by *calcula2* in a dedicated workstation with the plain Linux, and $ET_{nondedicated}$ is the execution time of *calcula2* when it is executed jointly with one (or various) parallel application(s) in the same coscheduling environment.

Unlike CPU bound applications, the variable behavior of I/O bounded applications should produce excessive variation in obtaining the *Local Overhead* (*LO*). Thus, and in addition to the studies performed by Ferrari [54] and Kunz [55] (already commented on in section 1.1.1.4), CPU bound applications were chosen as the kind of application to characterize the local workload.

Moreover, benefits in distributed applications will be more accurately obtained if the local workload is CPU bound. No means to fix the load of a system can be performed if local applications perform mainly I/O.

Bearing in mind those considerations, the workload characterization in the rest of the experimentation is performed by using the CPU bound applications *calcula* and *calcula2*.

5.3 Explicit Coscheduling

In this section, the DTS system and the different execution modes are analyzed. The Gain metric was used for this. This metric gives the performance of the different models with respect to PVM and is defined as follows:

$$Gain = \frac{T_{pvm}}{T_{cosched}} \tag{5.2}$$

where T_{pvm} is the execution time of one application in the original PVM-Linux environment and $T_{cosched}$ is the execution time of the same application in the *cosched* (STATIC, BALANCED or DISTRIBUTED) environment.

5.3.1 IP Interval and Local Overhead

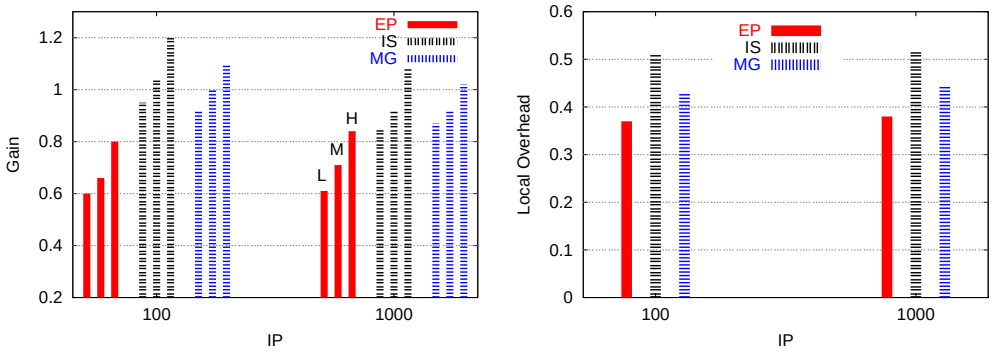
As was shown in the synchronization algorithm, the method used in synchronizing *PS* and *LS* in the STATIC and BALANCED modes is by broadcasting a short message to all the nodes in the Cluster. So, first of all, the cost of the synchronization phase must be found. This cost is determined by the coscheduling skew.

The *coscheduling skew* (δ) is defined as the maximum out of phase between two arbitrary nodes, formally:

$$\delta = max(broadcast) - min(broadcast) \tag{5.3}$$

where *max(broadcast)* and *min(broadcast)* are the maximum and minimum time in sending a short broadcast message. With the aid of the *lmbench* [60] benchmark, we have measured the coscheduling skew. An average value for $\delta = 0.1$ ms was obtained.

The next step is to determine the *IP* value, one of the most important DTS parameters. If a large enough *IP* interval is chosen, for example, more than 100 ms, δ should be insignificant in relation to *IP*. Thus initially, the lower bound for *IP* was fixed at 100 ms.



(a) (b)

Figure 5.8: (a) STATIC mode results. (b) local workload overhead.

Fig. 5.8(a) shows the gain obtained in the execution in four nodes of three NAS

benchmarks, EP, IS and MG class A, when they were executed in the DTS environment (mode STATIC) with different workload (instances of *calcula*, 1 Light, 2 Medium and 3 Heavy) and with two different *IP* periods, 100 ms and 1000 ms. Furthermore, $PS = LS = IP/2$.

DTS performance increases with the local workload. This means that when the workload increases, DTS tends to give more execution opportunities to the distributed tasks than PVM. This was also corroborated in the simulation chapter (see chapter 4, table 4.1, cell *MRQL*-EXP), where only a slightly increment of the explicit model was observed. Here, the results are significantly better.

Performance of distributed benchmarks also increases (decreases) with the communication (computing) requirements. This is, the obtained gain is ordered as follows IS > MG > EP. This means that better coscheduling gains are produced in communication intensive applications. The synchronization time spent in the communication phases is reduced significantly in the DTS environment.

In message-passing intensive benchmarks (IS and MG), better results were obtained for $IP = 100$ ms. Also, any other experiment carried out (where *IP* was varied between 100 and 1000 ms) did not improve the results obtained for $IP = 100$ ms. This means that synchronization of distributed tasks must not be delayed excessively, but doing this frequently, the added overhead overrides the synchronization gain.

When applications are CPU bound (EP) slightly better results were obtained for $IP = 1000$ ms, but they were not very significant. In this case, the gain in performance was due to the reduction in context switching.

Obviously, the DTS gain is at the expense of introducing local workload overhead. Fig. 5.8(b) shows the local workload overhead in the execution of *calcula2* when it was executed together with the different benchmarks. As can be seen in the figure, an *IP* of 100 ms introduces slightly more overhead in the local tasks than an *IP* of 1000, due to the added overhead of increasing the context switching. Really, *IP* intervals above 1000ms drop response time of the local tasks excessively. On the other hand, a value of *IP* below 100 ms would reduce the efficiency of the DTS model by the addition of excessive context switching.

The local overhead introduced into local tasks also depends on the kind of distributed application. This is, more local overhead is introduced by increasing

the computing requirements of distributed applications. This result is very coherent because, message-passing intensive applications can give more execution opportunities to the local ones in the *PS* interval than compute-bound distributed applications.

In general, for message-passing intensive distributed applications, better results were obtained when $IP = 100$ms. Also, an $IP = 100$ms does not damaged the response time of local applications as does an $IP = 1000$ms. In the case of CPU bound applications (EP), an $IP = 1000$ms behaves slightly better. In further experiments, as we are interested in reducing the waiting time of communicating benchmarks, an *IP* value of 100 ms will be used.

5.3.2 DTS Modes

Fig. 5.9 summarizes the gain obtained by running the EP, IS and MG class A benchmarks on the three different DTS modes (STATIC, BALANCED and DISTRIBUTED) with respect to PVM. As before, four Cluster nodes where used. Based on the previous experimentation, we chose an *IP* of 100ms. In the STATIC mode, the *PS* and *LS* values were $PS = 60$ms and $LS = 40$ms respectively.

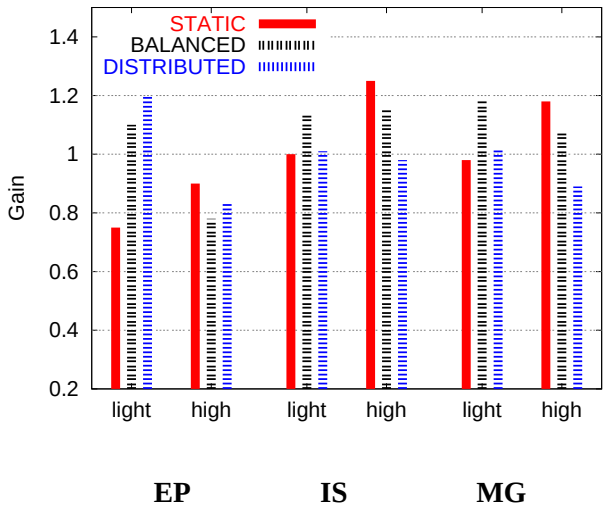


Figure 5.9: Comparison between the three DTS modes.

From Fig. 5.9, we observe that in EP, a computing intensive benchmark, the results obtained in the STATIC mode for a light load were worse than in PVM

due to the additional overhead introduced by DTS (which is basically produced by context switching), and *PS*, which should be increased. On the other hand, in the heavy case, since the DTS gives more scheduling chances to distributed applications, better results were obtained but also performance did not reach the level of PVM.

The BALANCED mode behaved significantly better than STATIC for a light workload. Instead, the results decreased when the workload increased. In this case, despite the added overhead in the synchronization phase, the *PS* and *LS* intervals were adjusted to the local workload.

In general, the DISTRIBUTED mode, works better than the other modes, and especially in light workloads. Heavy workload also damaged distributed performance by adjusting *PS* and *LS* (like BALANCED), but here gains are improved by avoiding synchronization. When the system was heavily loaded, STATIC was the most favored mode (*PS* was the highest one in STATIC). This fact shows that only the adjustment of *PS* and *LS* in each node is necessary without any other kind of synchronization when distributed applications are CPU bound.

For message-passing intensive applications (IS and MG), the STATIC mode gave equal or better performance values than PVM. This gain was due to the synchronization between periods this mode introduce. Also, for heavy workload, it was the better mode because the PS and IS intervals remain constant.

The BALANCED mode gave the best results when the system was lightly loaded. Both the synchronization of *PS* periods and the adjustment of its length to the local workload significantly overtook the overhead it introduced. For heavy workload instead, the *PS* (*LS*) period decrease (increase), thus reducing distributed performance.

The DISTRIBUTED mode always gave lower results than BALANCED because in message-passing intensive applications the *PS* period was out of phase (as was *LS*).

In the IS and MG, synchronization is required, and as the DISTRIBUTED mode does not provide this feature, the performance results were not as good as in the BALANCED mode.

Globally, for high message passing applications (IS and MG), better results were obtained when synchronization between communicating periods was applied

(STATIC and BALANCED).

5.4 Explicit versus Implicit

In this section, various coscheduling schemes are compared. Basically, comparisons are made between PVM, the STATIC mode of DTS and combinations of the HPDT and the spin-block (Implicit) techniques.

5.4.1 Implemented Environments

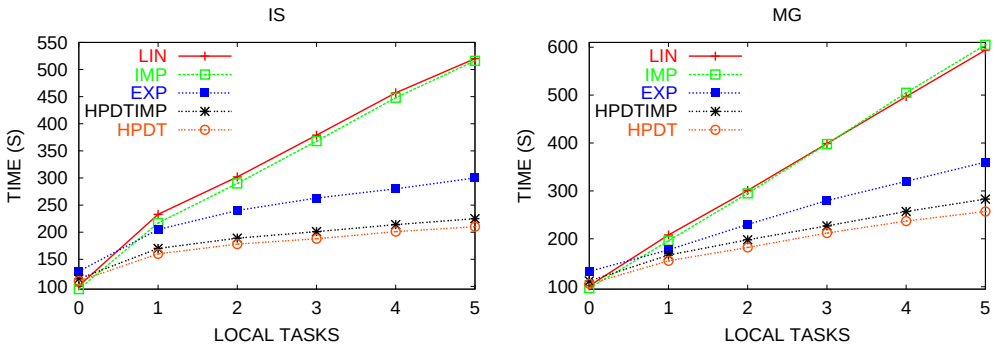
The next five distributed environments were created with the help of the coscheduling models detailed in chapter 3 (and the respective algorithms which implement them):

- LIN: plain Linux (and PVM) environment.
- IMP: Implicit coscheduling (spin-block technique). PVM uses two phases in the reading of a fragment, the PVM transmission unit (see chapter 3, sec. 3.1.1.1). In the first phase, only the header of the fragment is read. In the second, the data of the fragment is obtained. In our case, the spin-block is only performed in the first phase. That is, Algorithm 7 (of section 3.4) is applied only in the reading of the fragment header.
- HPDT: distributed tasks always have higher scheduling priority assigned than the local ones. Algorithm 6 (of section 3.3) is applied.
- HPDTIMP: HPDT model. In addition, distributed tasks also perform spin-block in the reading of the fragment header. Algorithms 6 and 7 (of sections 3.3 and 3.4 respectively) are applied.
- EXP: STATIC DTS mode. Periodically, after 80 ms the *DTS Scheduler* in each node delivers a STOP signal to all the local distributed processes and then, after 20 ms, the *DTS Scheduler* delivers a CONTINUE signal to reawaken them. That is, $IP = 100$ ms, $PS = 80$ms and $LS = 20$ms.

With the help of *lmbench* [60], we measured $spin_{max} = 2 * C$, where C is the context switch cost (see sec. 3.4.1, formula 3.5). As we will see, the benchmarks used were IS and MG, which gave similar C values ($C \simeq 100$ μs) with the same workload (instances of *calcula*). Also, due to the low memory requirements of *calcula*, no significant differences were obtained when the workload was varied. So, in the spin-like models a *spin* interval of 200 μs was chosen, independently of the benchmark to be used.

5.4.2 Distributed Performance

Fig. 5.10 shows the results obtained by executing IS and MG class A in four nodes. They were tested in the five models. The local workload was also varied between 0 and 5.



(a) (b)

Figure 5.10: Execution of the NAS parallel benchmarks. (a) IS (b) MG.

Note that the behavior of the two parallel benchmarks in the overall models is very similar.

IMP gains were only obtained for low local workload. Furthermore, IMP scales fine and its performance tends to reach the LIN one when the local workload is increased. In some cases (i.e. MG), the IMP model even behaved worse than LIN when the system was heavily loaded. This model is the only one that behaves slightly better than LIN (which can hardly be appreciated in the figures) when the

workload is 0. The IMP benefits are in general moderate. This means that the saving of context switching does not increase the distributed performance.

As was expected, optimal execution of the HPDT model can be observed. EXP without local tasks is the worst model, but by increasing the workload, its performance increases and it becomes one of the most efficient model for heavy workloads.

The HPDTIMP case gave worse results than HPDT. The unnecessary spin-block phase added in the HPDTIMP model only adds overhead in the reading of PVM fragments.

5.4.3 Local Performance

The influence of the models on the local tasks was also based on the *Local Overhead* (*LO*) metric (as in DTS), which measures the overhead in the execution of *calcula2* in the different environments (see Fig. 5.11).

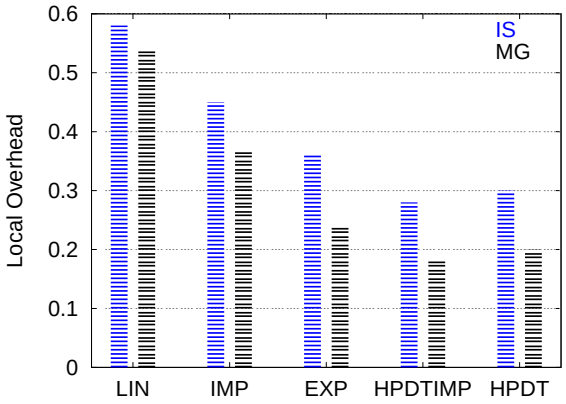


Figure 5.11: *Local Overhead* (*LO*) of local tasks.

When message-passing intensive applications are executed (IS), less negative effects on the local tasks are produced. On the other hand, the overhead increases with the CPU requirements of the distributed applications (i.e. MG).

In general, a great negative impact on the local task performance was produced by the EXP model. A *PS* interval of 80 ms (and IS=20ms) for the execution of distributed applications caused excessive overhead in the local task. It behaved

worse than the IMP model. In this case, the active waiting for messages does not overtake the overhead added by the DTS into local tasks.

HPDT introduced a great slowdown in the local tasks and the HPDTIMP model even more. The addition of spinning to HPDT only introduced additional overhead into the local tasks.

5.5 Predictive and Dynamic

In this section, various Predictive and Dynamic models are analyzed and compared. The analyzed models are:

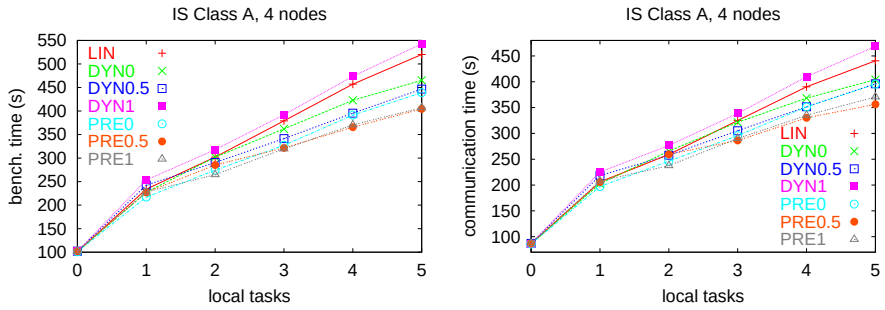
- LIN: plain Linux (and PVM) environment.
- DYN0, DYN0.5 and DYN1: Dynamic with P (Percentage assigned only to the past receiving frequency) equal to 0, 0.5 and 1 respectively. The value $P{=}0$ and $P{=}1$ represent values for P close to 0 ($P{\simeq}0$) and 1 ($P{\simeq}1$) respectively.
- PRE0, PRE0.5 and PRE1: Predictive with P (percentage assigned to the past receiving and sending frequency) equal to 0, 0.5 and 1 respectively. Also in this case $P{=}0$ and $P{=}1$ means $P{\simeq}0$ and $P{\simeq}1$ respectively.

5.5.1 NAS Results

First of all, the behavior of various coscheduling models was analyzed. This was accomplished by comparing execution times of the same benchmark in each model.

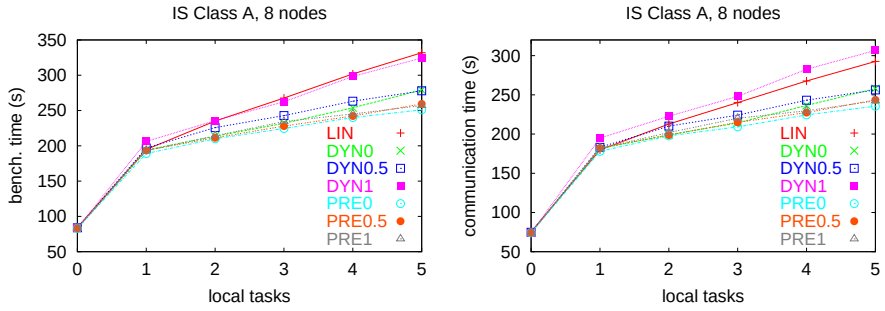
Figures 5.12 and 5.13 show the results obtained for the IS class A benchmark in 4 and 8 nodes respectively when the number of local tasks (instances of *calcula*) was increased (form 0 to 5). Class T results are omitted because times are too low. These figures reflect the total executing time (the figures on the left, labeled (a)) and the corresponding communication times (on the right, labeled (b)).

Note that the shapes of the execution benchmark and the communication times are identical. This confirms that coscheduling techniques influence in the communication phases and do not affect the computing ones. So, gains in coscheduling



(a) (b)

Figure 5.12: IS class A, 4 nodes.



(a) (b)

Figure 5.13: IS class A, 8 nodes.

are obtained by reducing the waiting times in communicating and synchronizing remote tasks.

One interesting comment about those figures is that when the Cluster is scaled, the behavior of the coscheduling models does not vary. This means that the presented models scales well and no malfunction or performance drop is produced when the communication traffic requirements increase. As we will see, the same results were obtained in the rest of the experimentation.

In general, as the multiprogramming level increased, the performance of both coscheduling models (Dynamic and Predictive) was better than the Linux (LIN)

one. Furthermore, the Predictive results are very close to each other and considerably better than the Dynamic ones.

Also, when no local applications were executed, the results between the different models (including Linux) were very close. This means that the coscheduling mechanisms added almost no overhead to the system. The added overhead is not always as low as the execution alone of IS. Really, this overhead depends on the number of distributed applications to be jointly executed in the Cluster. This can be observed in the next section (sec. 5.5.1.1), in Figures 5.16 and 5.17, which show the results obtained when various NAS benchmarks were executed simultaneously.

When only one distributed application was executed, sometimes, the current frequency had slightly more influence than the past one. This fact can be better observed in the results obtained by the MG benchmark (see Fig. 5.14, where the Class T results are also shown). Furthermore, the MG benchmark changed rapidly between computing and communication phases, and thus the current frequency has more influence in determining the need of the MG tasks to be coscheduled. This fact can be observed in Figs. 5.14(b) and 5.14(d), where the coscheduling models increase performance by increasing the weight of the current frequency.

The highest gains were obtained in the Predictive model. Predictive coscheduling takes both the sending and receiving messages into account, whereas Dynamic only considers the receiving ones, so the opportunity for promoting distributed tasks through the ready queue is greater in the Predictive than in the Dynamic case. The necessity for coscheduling, which is closely related to the communicating pattern of each distributed application, is more approximated by the Predictive model.

Other results, obtained with CG, FT, BT and SP are shown in Appendix B. Similar results were obtained for these benchmarks. In CG, BT and SP, which are benchmarks with higher computing requirements and even more regular communication patterns than IS for example, the Predictive models, which assign more coscheduling weight to past frequency (and especially PRE1) gave slightly better results. Their regular behavior favors their effectiveness when these models are applied, because those models act over the average and not the current frequency.

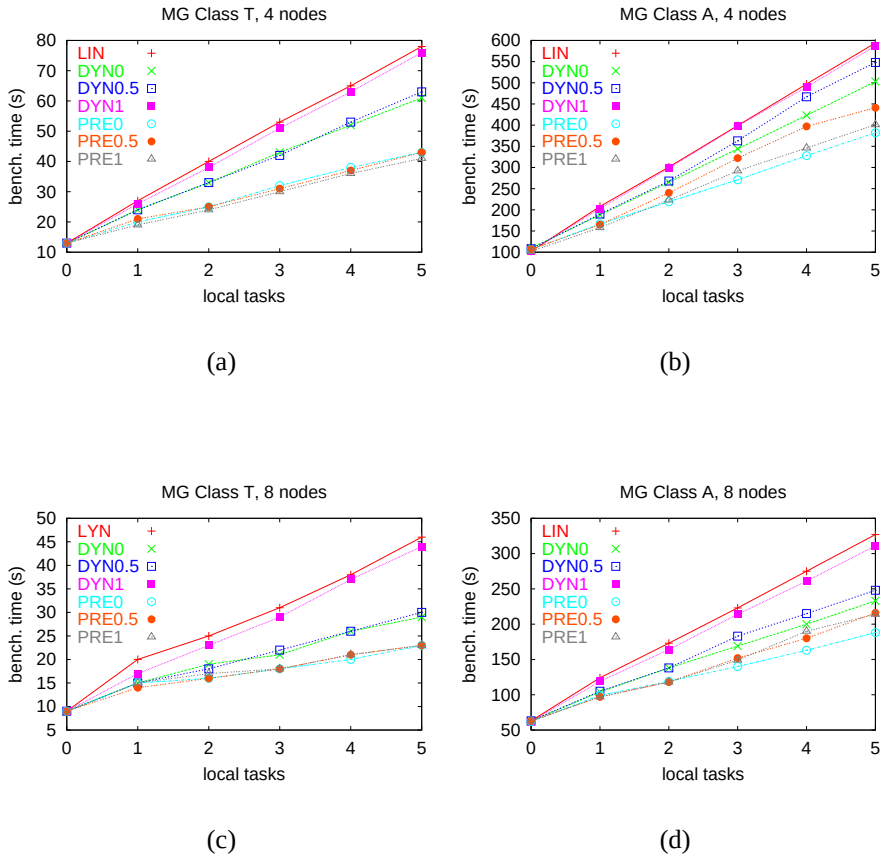
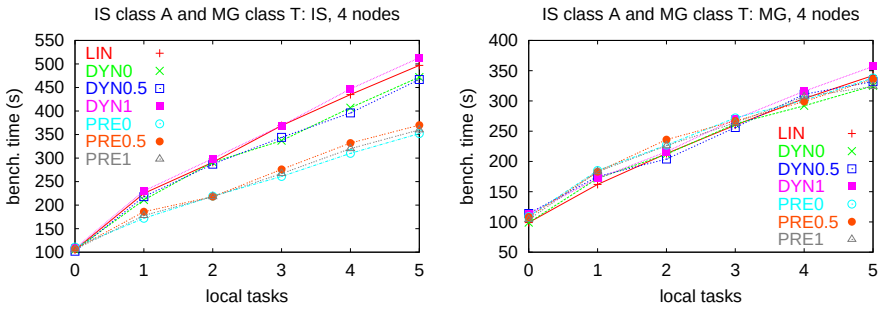


Figure 5.14: MG class T and A, 4 and 8 nodes.

5.5.1.1 Executing Together

Next, the behavior of the coscheduling models is analyzed when various distributed applications are executed in various workloads. It is important to mention that the benchmarks to be executed jointly with the workload must fit in the main memory. If not, the page faulting mechanism (one or two orders of magnitude slower than the network latency) would corrupt the performance results of the coscheduling models.

First, IS class A was executed with MG class T (see Fig. 5.15) in 4 nodes. In this case, due to the memory fitting requirement explained above, IS is class A and MG is class T (with 600 iterations, instead of 4 as in the original benchmark).



(a) (b)

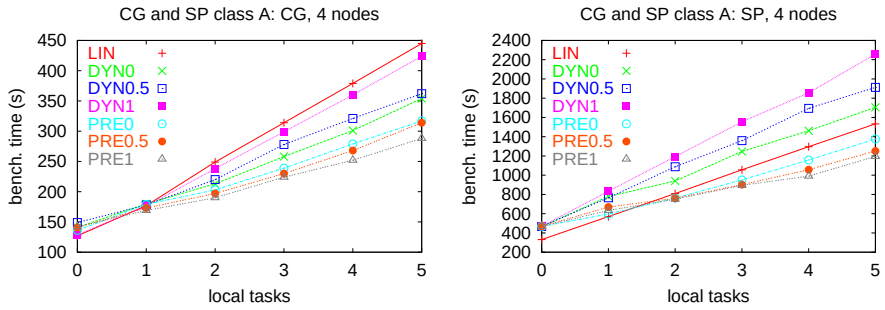
Figure 5.15: IS and MG, 4 nodes. (a) IS (b) MG.

Better IS results were even obtained for the Predictive case (see Fig. 5.15(a)). This confirms the good behavior of the Predictive model with respect to Linux and Dynamic. There were more opportunities (more chances for overtaking tasks) for increasing the IS performance when the MG was also executed, so IS performance increased. The MG results (Fig. 5.15(b)) are very similar because class T is not as message-passing intensive as class A or even as IS class A, and consequently synchronization measures provided by the Dynamic and Predictive models have fewer opportunities to improve its performance.

Also, note that the IS behavior suffers no excessive variations with respect to its single execution (see Figs. 5.12(a) and 5.15(a)). Once again, the low communication of MG does not influence the coscheduling of IS.

Other executions, such as CG with SP (see Fig. 5.16(a)), produced analogous results for CG. In this case, PRE0.5 and PRE1 were the finest models. In the execution of various distributed applications, PRE0.5 and PRE1 identified the correspondents better. Also, these models obtained the best performance in SP (see Fig. 5.16(b)), whereas the Dynamic ones behaved even worse than Linux. In this case, the Dynamic models only favored one task, CG, with higher communication frequency (and a more regular communication pattern) than SP (see Figs 5.5 and 5.7).

More experiments (shown in Appendix B) were performed by executing IS



(a) (b)

Figure 5.16: CG and SP. (b) CG (b) SP.

with CG, BT and SP. Similar results were obtained in all of them. The Predictive model always gave better results than the others. Also, the Dynamic models, as in the execution of SP in Fig. 5.16(b), did not always coschedule all the distributed tasks correctly.

Finally, the CG, IS and SP benchmarks were executed jointly. Fig. 5.17 shows their execution times obtained in four nodes. CG was the most favored benchmark. In general, the behavior of the coscheduling models did not change. It can be seen that the PRE0.5 and specially PRE1, obtained the best results. This fact also confirms that when various distributed applications are executed in parallel, the need for coscheduling is better approximated by considering the past communication frequency.

5.5.2 Low Level Results

Low Level Benchmarks were selected to measure different aspects of the remote communication between tasks forming distributed applications. Instead of measuring different distributed applications with different communication patterns, we measured different communicating primitives. The different communication patterns are normally formed by some sort or combination of these primitives, so by measuring their performance in the different coscheduling models we will approximate the behavior of the different communication patterns. As in the pre-

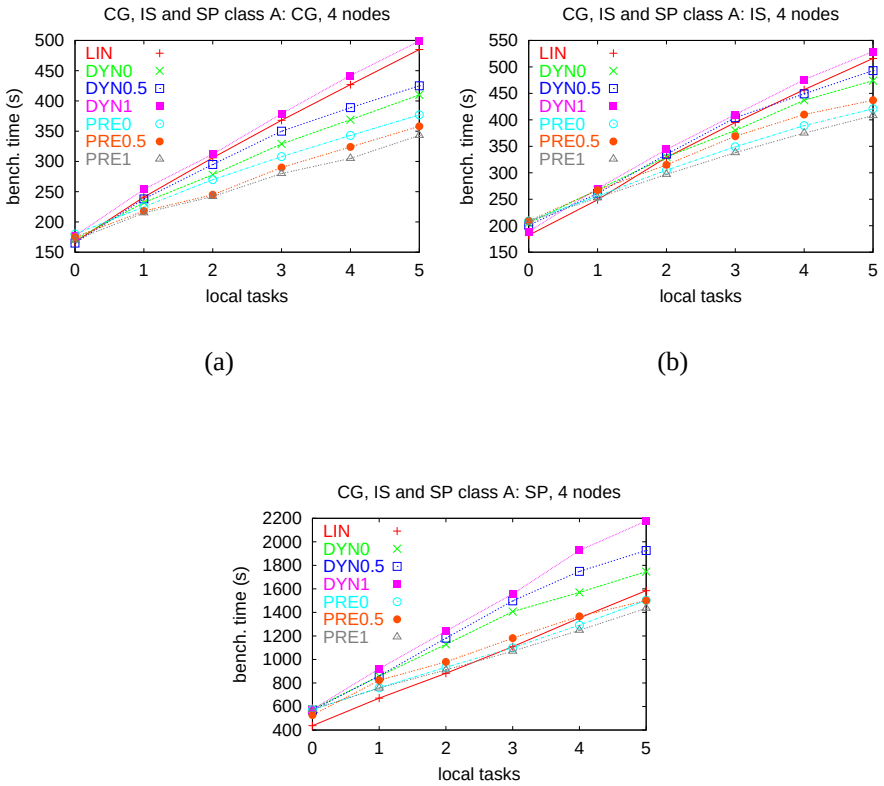
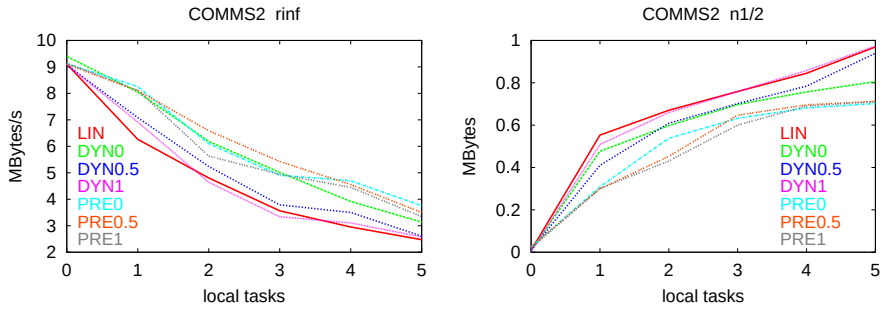


Figure 5.17: CG, IS and SP. (a) CG (b) IS (b) SP.

vious section, the models used are LIN DYN0, DYN0.5, DYN1, PRE0, PRE0.5 and PRE1.

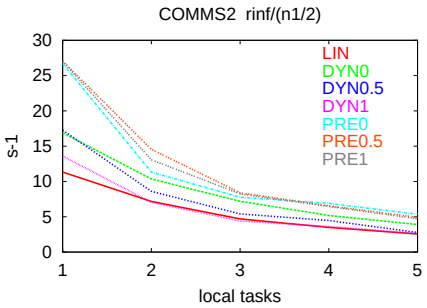
The r_∞, $n_{1/2}$ and $r_\infty/n_{1/2}$ values were collected for the Low Level benchmarks COMMS1 and COMMS2. As was shown in section 5.1.2, r_∞ approximates the communication rate (r) for long messages. The communication rate for short messages, is instead approximated by $\left(r_\infty/n_{1/2}\right) * n$. So, for short messages, the best performance will be reached for high $r_\infty/n_{1/2}$ values. Furthermore, as $n_{1/2}$ is the message length required to achieve half this asymptotic rate it must be as low as possible.

Fig. 5.18 shows the results obtained in the execution of the Low Level bench-



(a)

(b)



(c)

Figure 5.18: Low Level benchmark COMMS2.

mark COMMS2 in two nodes and in the different models: (a) r_∞, (b) $n_{1/2}$ and (c) $r_\infty/n_{1/2}$. The COMMS1 results were very similar to the COMMS2 ones, so only COMMS2 results are shown. This fact demonstrates that the models have the same performance when the communications are performed in only one direction (COMMS1) and when the communication links are bidirectional (COMMS2).

In general, good results were obtained for the Predictive models, which in turn were very close to each other. No significant differences between the Predictive models were produced for normal sending and receiving communications.

Fig. 5.19 shows the results obtained for the SYNCH1 benchmark (also with important improvements in the Predictive models). In the barrier processes there is

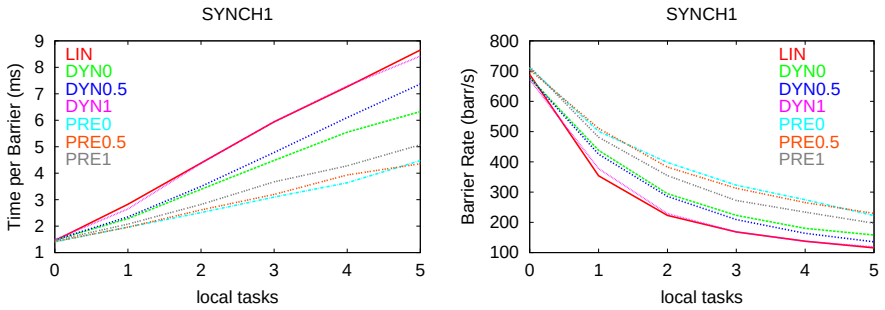


Figure 5.19: Low Level benchmark SYNCH1.

a great quantity of all-to-all communication. In this case, PRE0 increase the priority of this kind of communication more efficiently than PRE0.5 and even more than PRE1. Tasks making barriers send and receive asynchronous short messages, and their efficiency depends on the rapid response from the nodes. So, as in the MG case, where communication was more irregular than other benchmarks, the current frequency has more influence than the past one.

5.5.3 Local Tasks Overhead

Fig. 5.20 shows the overhead introduced by the coscheduling models in the execution of *calcula2* together with the following NAS (class A) parallel benchmarks: IS, MG, CG, BT and SP.

The execution time of *calcula2* (80s) will be the reference for knowing the added overhead when it is executed jointly with each benchmark in three different coscheduling models: Linux, DYN0 and PRE0. Only DYN0 and PRE0 were used because no important differences were produced when the weight assigned to the current and past frequencies was varied.

Figure 5.20 shows how Linux always obtained the best results, and the Dynamic model introduced slightly less overhead than the Predictive one.

Linux obtained the best results because, unlike the other two, it does not add extra overhead in obtaining the communication frequency and does not delay execution of the local tasks. Also, the Dynamic model performs fewer communication inquires (only receiving information is required) than the Predictive one.

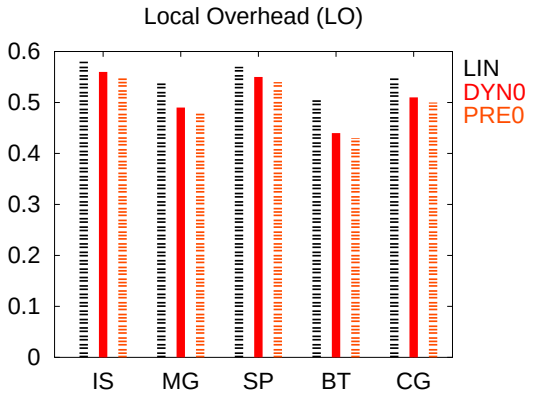


Figure 5.20: Local Overhead (LO) of *calcula2*.

The slight differences are also due to the better coscheduling performed by the Predictive model, that, in turn, produces non-proportional local task penalties.

In any case, the coscheduling models avoid the starvation of the local tasks. The local tasks are executed when all the PCB *counter* fields of the distributed applications reach the value 0 (they have consumed their time slices), thus an opportunity arises for the execution of the local tasks. The coscheduling methods advance the execution of distributed tasks (because they must be coscheduled with their correspondents) without delaying the local ones excessively.

In the coscheduling models, as the benchmark computing requirements grow (which in increasing order is: IS, SP, CG, MG and BT), more overhead is added into the local task. But, as was mentioned above, this overhead is very low because the distributed tasks must wait to be executed again when their counter values reach 0.

Another kind of local experiment was also performed. In this case, *calcula2* was modified to provide an additional feature: to fix the CPU utilization by using system calls which suspend the benchmark for a predetermined time. The *calcula2* benchmark (with three different CPU utilizations: 10%, 50% and 90%) was executed in four nodes (in the LIN and PRE0 models) jointly with two different distributed workloads. The first distributed workload was formed by IS and CG, and the second by EP and MG (see Fig. 5.21). As we have seen, communicating benchmarks does not produce significant overhead into local tasks (case IS and

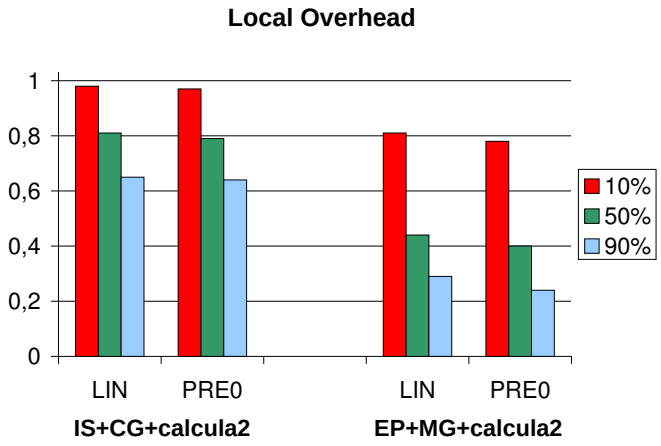


Figure 5.21: Local Overhead (LO) of *calcula*2.

CG). In the second case, where the distributed workload were formed by more CPU-bound applications (EP and MG), the local slowdown was increased.

In general, the most unfavored cases were produced when local tasks had more computing requirements. Furthermore, from these results we can affirm that I/O bound local applications will be scarcely affected by the Predictive model (and also by the Dynamic one). Note that for the 10% cases, the local task does not suffer excessive overhead. Usually, I/O applications have CPU utilizations even under 10%, thus performance penalties should be even narrower. This fact also justifies the choice of CPU bound applications to model the local workload.

5.5.4 Varying the Message Size

The influence of the message size in the behavior of the Predictive model was also obtained.

Two synthetic applications, sinring and sintree, representative of two types of communication patterns were implemented. It was necessary to develop these two new synthetic applications due to the impossibility of changing the message size in standard benchmarks.

Each benchmark performs the same loop repeatedly: *sinring* implements a logical ring (see Figure 5.22(a)), and *sintree* attends to the communication from

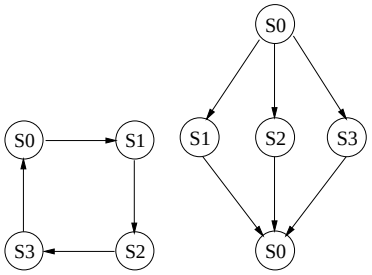


Figure 5.22: benchmarks: (a) *sinring* and (b) *sintree*

one to various, and various to one (see Figure 5.22(b)). In both applications, every forming task, in the reception of a synchronization message basically performs floating point operations during a fixed period of time, this being an input argument (1ms by default). The number of iterations of both benchmarks is also an input argument.

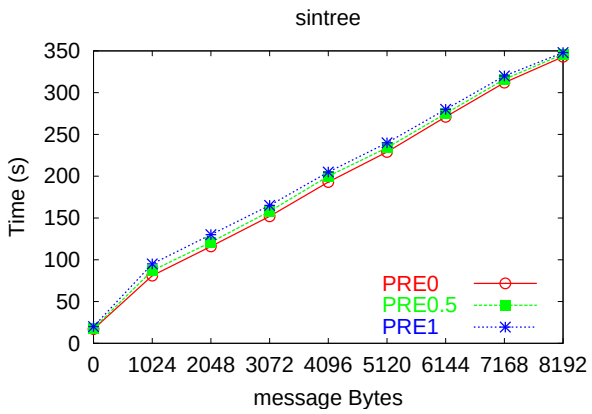


Figure 5.23: Varying the message size.

We only collected results for the Predictive models PRE0, PRE0.5 and PRE1. As the behavior of the two synthetic benchmarks was very similar, only results obtained for *sintree* are shown. Fig. 5.23 shows the results obtained in the execution of *sintree* in 4 nodes when the message size is varied between 0 and 8192

bytes. The number of local tasks was fixed to 5.

Note that the behavior of each model is very similar and these tend toward each other. Basically, the behavior in every model was not modified by increasing the message sizes. That is, the time increases linearly with increasing message size.

5.5.5 Additional Measurements

A new synthetic application, named *master-slave* is used here to demonstrate the differences between the Predictive and the Dynamic model. This application will also be used to illustrate a case where the weight of the past frequency is of more importance than in the previous ones.

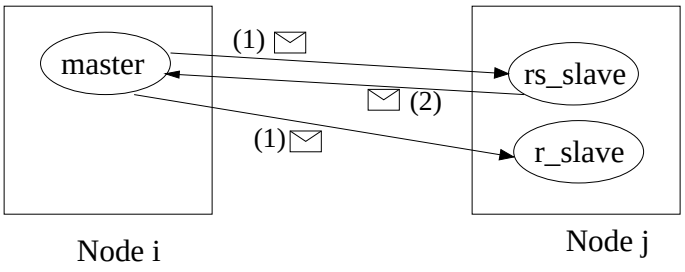


Figure 5.24: *master-slave* benchmark.

The *master-slave* application (see Fig. 5.24) is made up of one master and the slaves. There are two kinds of slaves, one that only receives messages (*r_slave*) and other that performs both receiving and sending messages (*rs_slave*). The mapping in four Cluster nodes is performed as follows: the master is assigned to one node, and two slaves (of different kinds) to each remaining node. The master performs a predetermined number of iterations and then both master and the slaves finish execution and the master reports the return time for the application. In each iteration, the master sends a message to all the slaves. All the slaves, after receiving the message, perform a simple floating-point computation. In addition, the *rs_slave* tasks reply to the master with the computation result. After receiving all the *rs_slave* messages, the *master* repeats the process again.

Each *rs_slave* waits for the reception of one message from the Cluster before

returning another message to it. The *r_slave* tasks only receive messages. Note that the *master-slave* performance does not depend on the performance of the *r_slave* tasks. To acquire maximum performance, the scheduling priority of the *r_slave* tasks must not be promoted as the *rs_slave* one.

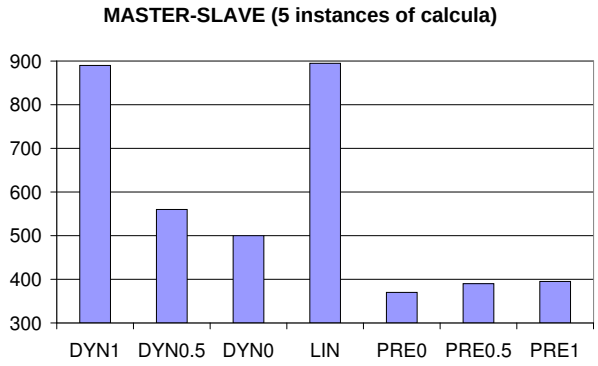


Figure 5.25: *master-slave* execution times (in seconds).

Fig. 5.25 shows the good performance of the Predictive model in the execution of the *master-slave* application with a local workload of five local tasks (instances of *calcula*) in each node.

As was expected, the Predictive models promoted the *rs_slave* tasks earlier than the Dynamic ones and Linux. So, the round-trip time of each iteration was minimized in the Predictive model and consequently major gains were reached. Moreover, $P \simeq 0$ obtained the best results because immediate response is required for achieving performance and this model favors it. Any mean between the current and past receiving frequency caused a drop in performance. Note that P has more influence in the Dynamic cases: performance decreases strongly on increasing P.

5.6 Summary

The distributed and local performance results of the different coscheduling mechanisms analyzed in this chapter are summarized below.

IM (Implicit). It was the only mechanism that slightly (2% as much) improved distributed performance in the absence of local workload.

Distributed: in general, Implicit behaves worse than the other coscheduling mechanisms. Distributed performance was not increased by varying the spin interval. In real applications, even when the communication pattern is very regular, message arrival is very irregular. For this reason, the gain was not increased for *spin* values above twice the context switching cost. The Gain was very moderate and do not exceed 5% with respect to LIN when the workload was low. However, by increasing the workload, the distributed performance tends to reach the one provided by LIN. Slightly better results were obtained in [19], where gains where as much 10%, but using another messaging protocol (Active Messages).

Local: it was one of the worst model (only the EXP with high *PS* values and HPDT gave poorer results). The spinning for arrival of messages damaged local performance excessively. Gains obtained in distributed performance were not proportional to the lost in local one.

HPDT (Hight-Priority Distributed Tasks). The main advantage of this mechanism is its implementation in the user space.

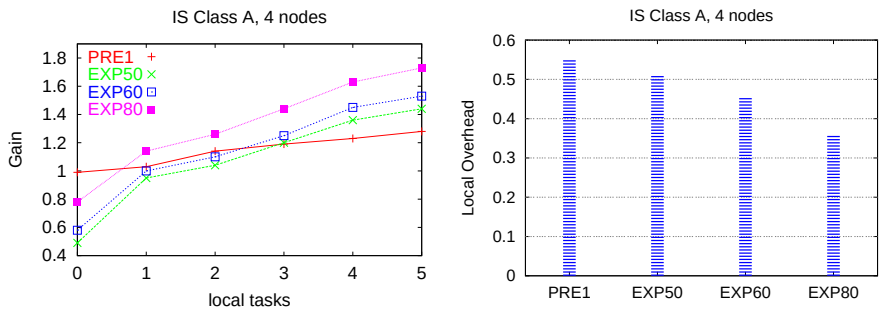
Distributed: this mechanism always gave the best Distributed performance.

Local: the excessive overhead that it introduced into the local tasks is a strong enough argument to discard it.

Through the experiments performed in this chapter, we demonstrated that IMP and HPDT are not as appropriate coscheduling mechanisms for use in Cluster computing as EXP, Predictive and Dynamic. In general, the Predictive model also behaved better than Dynamic. Consequently, more attention to the performance of the EXP and Predictive models (the ones presented in this project) is performed in this summary. Discussion about the EXP and PRE models is also based on Figures 5.26 and 5.27.

The Figures 5.26 and 5.27 show the relation between the Gain in the distributed tasks and the Local Overhead in the execution of IS and MG class A in 4 nodes. The features of both mechanisms are:

- PRE1: Predictive model with $P \simeq 1$. Here, the Gain is obtained as was defined in formula 5.2 for the EXP model. $T_{cosched}$ is in this case the execution time of the benchmark to be evaluated in the PRE1 model.
- EXP: denotes the STATIC operation mode of the Explicit (or DTS) model with $PS = 50$ ms and $LS = 50$ ms (EXP50), $PS = 60$ ms and $LS = 40$ ms (EXP60), and $PS = 80$ ms and $LS = 20$ ms (EXP80).

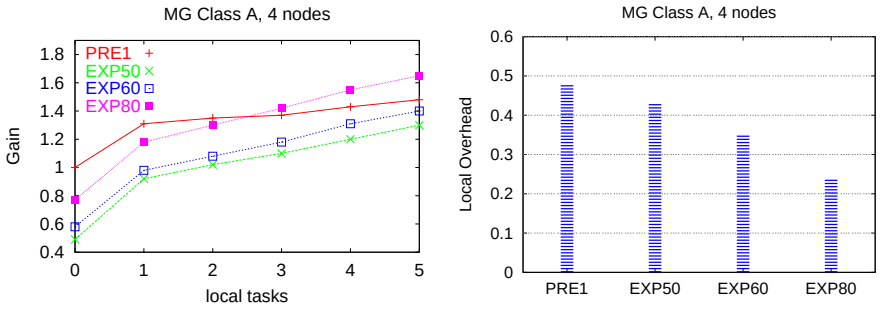


(a) (b)

Figure 5.26: Explicit vs. Predictive (IS). (a) Distributed Gain (b) LO.

EXP (Explicit). The main advantages of this mechanism over the implicit-control ones (Implicit, Dynamic and Predictive) is its implementation in the user space (as HPDT) and the ability to limit gains (slowdown) in performance in advance for distributed (local) tasks.

Distributed: EXP, an explicit-control mechanism, gave an intermediate performance for both distributed and local applications. In this environment, the distributed or local performance can be tuned in function to the execution periods assigned to the distributed (i.e. *PS*) and local



(a) (b)

Figure 5.27: Explicit vs. Predictive (MG). (a) Distributed Gain (b) LO.

(i.e. *LS*) tasks. Consequently, it is possible to achieve Distributed gain very close to HPDT.

Distributed Gain always increases with the local workload and also with increasing the period assigned to the distributed tasks (*PS*). Unlike the Predictive model, the potential gain depends more on the *PS* interval than on the kind of the distributed applications. However, higher intensive communicating applications (i.e. IS) also obtained higher performance improvements. For example, in the EXP80 model and with 5 local tasks, IS and MG obtained improvements of 73% and 65% respectively (see Figs. 5.26(a) and 5.27(a)).

To simplify the summary no results from the BALANCED mode are shown (also, as was commented previously the DISTRIBUTED mode is not appropriate for message-passing intensive applications). However, note that for example, for 1 local task, the *PS* and *LS* periods will be adjusted to 80 ms and 20 ms respectively, thus giving similar results to EXP80.

Local: for PS intervals higher than approximately 40 ms, the Explicit local overhead always exceeded that of the Predictive. They grow excessively by increasing the PS intervals. Also, but with fewer negative implications, increasing the CPU requirements of the distributed

tasks also increases the overhead. See for example Figures 5.26(b) and 5.27(b). In IS, the added overhead with respect to the plain LIN environment ranged from 12% (in EXP50) to 48% (in EXP80). In the MG case, it ranged from 21% (in EXP50) to 66% (in EXP80).

PRE and DYN (Predictive and Dynamic). The avoidance of the expli-cit-control mechanisms, provided by EXP for example, makes the Dynamic and Predictive models more efficient. That is, less overhead is introduced in both distributed and local tasks. Unlike the other models, Predictive and Dynamic were implemented in the system space (inside the Linux o.s.).

Distributed: as in the EXP case, it is possible to reach gains very close to HPDT. Distributed applications obtained better results in the Predictive than in the Dynamic model. In the Dynamic model, only the consideration of the current communication (receiving) frequency was advantageous. On the other hand, in the Predictive model, the past frequency favored distributed tasks with a more regular behavior. Also it was determinant when various distributed applications were executed in parallel.

The distributed Gain increases with the local workload, but the slope of the Gain function is not very pronounced. The potential Gain depends not only on the communication rate, but also on the synchronization requirements. Thus, in the Predictive model for example (see Figs. 5.26(a) and 5.27(a)), the obtained Gain in MG (48% for 5 local tasks) exceeds the IS one (28% for 5 local tasks). This fact also shows that the need for coscheduling is more important in MG than in IS.

Local: local overhead is minimum and is very similar in both the Dynamic and Predictive models. The local task performance decreases proportionally to the CPU requirements of the distributed applications. As more CPU requirements are needed more overhead is added into the local tasks. For example, the slowdown introduced into the local tasks by the DYN0 (PRE0) model with respect to LIN ranged from 3.5% (5%) in IS to 14% (16%) in BT. IS and BT are the NAS benchmark

with less and more computing requirements respectively. Note also as in the PRE1 model (see Figs. 5.26(b) and 5.27(b)), IS (MG) only added 5% (12%) more overhead than LIN.

A final conclusion to be made is that in light or medium loaded systems, Predictive is the best coscheduling model. The Explicit model (and more specifically, the STATIC mode) will serve instead to achieve distributed Gain in heavily loaded systems, but at the cost of introducing an excessive overhead into local tasks.