Parallel Processing Letters, Vol. 15, Nos. 1 & 2 (2005) 131–152

World Scientific
www.worldscientific.com

PARALLEL LOOP SCHEDULING APPROACHES FOR DISTRIBUTED AND SHARED MEMORY SYSTEMS

JOSE AGUILAR
CEMISID, Dpto. de Computacion, Facultad de Ingenieria,
Universidad de Los Andes, Merida 5101, Venezuela
aguilar@ing.ula.ve

ERNST LEISS[2]
Department of Computer Science, University of Houston,
Houston, TX 77204-3475, USA
coscel@cs.uh.edu

Received January 2005
Revised March 2005
Accepted by Sélim Akl

Abstract

In this paper, we propose different approaches for the parallel loop scheduling problem on distributed as well as shared memory systems. Specifically, we propose adaptive loop scheduling models in order to achieve load balancing, low runtime scheduling, low synchronization overhead and low communication overhead. Our models are based on an adaptive determination of the chunk size and an exploitation of the processor affinity property, and consider different situations (central or local queues, and dynamic or static loop partition).

Keywords: Loop Scheduling, Parallel Loops, Parallel Systems, Distributed Memory Systems, Shared Memory Systems.

1. Introduction

In this paper, we are concerned with the scheduling of parallel loops on shared-memory and distributed-memory systems [1, 2, 3, 7, 8, 12, 13, 15]. A loop is called a parallel loop if there is no data dependency among all iterations; i.e., iterations can be executed in any order or even simultaneously. They are widely used in scientific application programs. We propose a family of adaptive loop scheduling models based on an adaptive determination of the chunk size and an exploitation of the processor affinity property. Our adaptive chunk size mechanism initially assigns a large number of iterations; then, it increases/decreases the chunk size until all iterations are exhausted, to dynamically allocate loop iterations to processors, based on runtime information to reduce synchronization overhead and balance load more evenly. By exploiting processor affinity,

we reduce the amount of communication required to execute a parallel loop in shared and distributed memory systems, and thereby improve the performance. Our approaches also substantially reduce the number of mutually exclusive accesses avoiding small chunk sizes and predicting the next chunk size. Our approaches make decisions that match the current situation at runtime. In this way, we can handle loops with a wide range of load distributions when no prior knowledge of the execution is available. The organization of this paper is as follows: Section 2 presents the parallel loop-scheduling problem. Section 3 presents the main ideas of our models. Section 4 shows our experiments. Then, we present our conclusions and further work.

2. The Parallel Loop Scheduling Problem

In general, a loop can be defined as parallel when there are not data dependencies between any pair of iterations in it. In this loop all of the calculations are completely independent, so they could be performed in any order and the results would be equivalent. One way to exploit this parallelism is by executing the loop iterations in parallel on different processors. Parallel loops can be *uniform*, in which case the scheduler can assign an equal number of iterations to each processor, or they can be *non-uniform*, in which case iterations have different execution times [5, 6, 9, 10, 11]. In general, an efficient loop scheduling algorithm should optimize its performance by trading off adaptively scheduling overhead (synchronization overhead, loop allocation overhead, and scheduler execution time overhead), load imbalance overhead, and data communication overhead.

Loop scheduling algorithms fall into two distinct classes [1, 2, 13, 15]: i) *Central queue based*: In this approach, iterations of a parallel loop are all stored in a shared central queue and each processor exclusively grabs some iterations from the central queue to execute. ii) *Distributed queue based*: In order to exploit processor affinity inherent in the parallel execution of many iterations and to eliminate the central bottleneck, these approaches distribute the central queue over the processors.

There are two basic loop scheduling methods used to assign iterations of a loop to processors [1, 2, 13, 15]: i) *Static Scheduling*: Static policies depend on the average behavior of the system and not on its current state; they are usually applied to uniformly distributed loops. They assign iterations to processors statically, minimizing run-time synchronization overhead but do not balance the load. ii) *Dynamic scheduling:* Dynamic methods defer the assignment of iterations to processors until run-time. They should not assume any prior knowledge of the execution time of the loop iterations.

Many approaches have been proposed for the parallel loop scheduling problem [1, 2, 3, 7, 8, 12, 13, 14, 15]. All of these loop scheduling algorithms attempt to achieve the minimum completion time by distributing the workload as evenly as possible and/or minimizing the number of synchronization operations required and/or minimizing communication overhead caused by access to non-local data. Each one of the algorithms assumes mainly that we work on a shared-memory system.

The classic optimization criteria are [2, 7, 12, 13, 15]: i) *Minimize loop imbalance*, ii) *Minimize communication cost:* We can minimize this cost by exploiting the processor affinity that favors the allocation of loop iterations close to their data. iii) *Minimize synchronization problems:* In this case, the idea is to reduce the number of mutually exclusive accesses. iv) *Minimize scheduling overhead:* This is the time to allocate the remaining iterations.

There is much recent work on parallel loop-scheduling for shared-memory machines. For example, [8, 15] propose adaptive scheduling algorithms for different control mechanisms. The proposed algorithms apply different degrees of aggressiveness to adjust loop-scheduling granularities, aiming at improving the execution performance of parallel loops by making scheduling decisions that match the real workload distributions at runtime. These approaches are interesting when the initial loop partition is not balanced. In this case, processors should be able to increase or decrease dynamically their allocation granularity based on runtime information. Bull [2] introduces two algorithms for dynamic loop scheduling, which implement a feedback guidance mechanism. The feedback guidance mechanism is designed to utilize knowledge about the workload derived from the measured execution time of previous occurrences of the loop with the aim of reducing the incurred overhead. [12] presents a new methodology for statically scheduling a cyclic data-flow graph whose node computation times can be represented by random variables. Communication cost is also considered as factor: every node in the graph can produce a different amount of data depending on the probability of its computation time. Since such communication costs rely on the amount of transferred data, this overhead becomes uncertain as well. [14] proposes an algorithm to take advantage of the parallelism across a loop iteration while hiding the communication overhead. Results show that the proposed framework performs better that a traditional algorithm running on an input which assumes fixed average timing information. In [3], Cieniak, Javeed, and Li study the problem of scheduling loop at compile time for a heterogeneous network of workstation. They propose a simple model for use in compiling for a network of processors, and develop compiler algorithms for generating optimal and near-optimal schedules of loops for load balancing, communication optimizations, network contention and memory heterogeneity. In [7] Markatos and LeBlanc propose a loop scheduling algorithm for shared-memory multiprocessors in considering communication overhead caused by accesses to non-local data. They show that traditional algorithms for loop scheduling, which ignore the location of data when assigning iterations to processors, incur a significant performance penalty. They propose a scheduling algorithm that attempts to balance the workload, minimize synchronization, and co-locate loop iterations with the necessary data simultaneously. Finally, Tzen and Ni propose a processor self-scheduling scheme, called Trapezoid Self-Scheduling, for arbitrary parallel nested loops in shared-memory multiprocessors [13]. This approach starts the execution of a loop by assigning a large number of iterations and linearly decreases the number of iterations until all iterations are exhausted. In this way, they obtain the best tradeoff between small overhead and load balancing for both uniformly or non-uniformly distributed parallel loops.

3. Our Approach

We consider parallel systems consisting of P processors. Arbitrary nested parallel and serial loops are allowed, and the bound of loops may not be known at compile time. We consider two types of load: uniform and non-uniform, and we propose two versions of our approach: with local queues or with a central queue. The symbols used in our approach are defined as follows:

N: number of chunks.
$LCS_i(t)$: chunk size at time t on processor i.
L: load of the parallel loops (assumed to be larger than k/P).
$LW_i(t)$: workload at time t on processor i.
$LCF_i(t)$: cost function at time t on processor i.
$Cli_j(t)$ load imbalance at time t on processor j.
CS(t): global chunk size at time t.
k: number of iterations.
P: number of processors.
l: final chunk size.
W(t): global workload at time t.
base: is a constant
CF(CS(t), W(t)): chunk function that is based on the chunk size and workload at time t.
Loop-Pattern: is the ratio between the shortest and the largest iteration.
AP: factor that describes the application characteristics (uniform or non-uniform loops, etc.).
$queue_i$: number of iterations allocated to processor i.
Cc(i, j): communication cost which is defined as the distance between processor i and j.
α: factor that describes the importance of the communication cost. It can have the following values: small, large or medium (small may be [0-0.3], medium [0.3-0.7] and large [0.7-1]).
β: factor that describes the importance of the load imbalance cost It can have the following values: small, large or medium.
NRI(t): number of remaining iterations at time t in the case of a central queue. It can have the following values: small, large or medium. The possible values for small is fewer than 30% of the total number of iterations, medium is between 30% and 70% of the total number of iterations, and large is more than 70% of the total number of iterations.
$state_i(t)$: state of the workload at time t on processor i. It can have the following values: heavy ($LW_i(t) > (W(t)/P)*1.5$), light ($LW_i(t) < (W(t)/P)*0.5$) or normal ($(W(t)/P)*0.5 \leq LW_i(t) \leq (W(t)/P)*1.5$).

3.1. *General Algorithm*

Instead of relying on prior knowledge about a loop's execution, our model exploits the potential of using runtime information in order to adjust iteration chunk size adaptively to reduce synchronization scheduling and communication overhead, and dynamically balance the workload. We define a chunk as a unit of work to be assigned to a processor. In our parallel loop scheduling approach, a chunk is a group of m iterations where m is defined dynamically. The general procedure of our approach is the following:

- Initial *Partition phase*. In this phase, a loop with k iterations is partitioned into chunks over P processors.
    - In the case of *distributed queues*, we will partition a parallel loop *statically* into the local queue of each processor. A *deterministic initial assignment policy* is used to divide iterations of a parallel loop into local queues of processors, which ensures that an iteration is always assigned to the same processor at the start (improve processor affinity). With this assignment scheme, if a parallel loop executes repeatedly and each parallel iteration accesses the same data set in different executions, the first execution of the parallel loop will bring data locally to the processor so that the subsequent execution of the parallel loop only involves local data access.
    - In the case of *a central queue*, we define the initial chunk size so as to avoid a load imbalance at the beginning (see Section 3.2.1). In general, the partitions are defined *dynamically* to match the current situation at runtime (see Section 3.3).
- *Scheduling phase*: In this phase, according to the chunk size, each processor executes a part of the remaining iterations. In the case of *local queues*, each processor allocates a part of the remaining iterations (according to its local chunk size) from its local queue until the local queue is empty. In the case of a *central queue*, each processor allocates a part of the remaining iterations (according to the global chunk size) from it. Because all processors share a central queue, a critical section is used. This introduces a synchronization overhead.
- *Remote scheduling phase*: This is used only in the case of *distributed queues*. When a processor finishes the execution of all the iterations in its local queue, it remotely allocates a portion of the iterations from one of the most loaded processor in the system according to certain criteria. The remote data access introduces communication overhead as well as synchronization overhead (local queues are shared by different processors during the remote access).

The phase of scheduling parallel loops (which includes the remote scheduling phase for the local queue case) involves finding the appropriate decomposition of a loop into parallel tasks by finding the appropriate granularity of parallelism, so that the overhead of parallelism is kept small, while the workload is evenly balanced among the available processors. The general procedure of this phase is the following:

1. Until loop is exhausted
    1.1 Plan
    1.2 Compute
2. Wait

That is, during the scheduling phase we distinguish three states in handling a parallel loop for each processor: planning state, computing state, and waiting state. In the *planning state* a processor tries to acquire a chunk from the remaining iterations (scheduling

overhead). The *planning state* is decomposed into the following two processes: Chunk Size Determination and Chunk Assignment. The scheduling overhead is caused by waiting for the availability of a critical section (synchronization overhead) and the scheduler execution time. The synchronization overhead is determined by the implementation of mutually exclusive accesses to shared variables (for example, a central queue). Whenever a processor acquires a chunk, it enters the *computing state* and starts execution. After finishing the computation, it goes back to the planning state. The total time each processor stays in the computing state is very likely greater than the total load of the parallel loop due to network and memory contention. An approach which generates much network and memory traffic will affect the time the processor stays in the computing state (synchronization and communication overhead). A processor repeats planning and computing until the loop is exhausted. Then, the processor goes to the *waiting state* and waits for the completion of the remaining executing chunk. Every processor, except the processor with the last chunk to execute, has to stay in the waiting state due to the synchronization barrier (load balance overhead).

In our approach the main phase is this phase. We must optimize the next criteria during the scheduling phase: i) *Load balancing cost*: Minimize the waiting time for the completion of the last chunk due to the synchronization barrier. ii) *Synchronization cost*: Minimize the number of mutual exclusive accesses. iii) *Communication cost*: Minimize accesses to no-local and far iterations.

3.2. *Criteria Optimization*

3.2.1. *Load Balance overhead*

We solve the load balance problem by assigning iterations dynamically. We avoid the *load imbalance* problem due to a large initial chunk size (*CS(0)*), especially when the workload is not uniformly distributed, because we define a good value for *CS(0)*, and then we increase or decrease the load according to the current situation on the system. In order to provide appropriate load balancing, we choose *CS(0)* initially to be equal to k/2P.

Observations: If *CS(0)* is equal to or larger than L/P, then *CS(0)* likely becomes the last chunk of this loop (load imbalance). If *CS(0)* is always less than L/P then the workload of the rest of the iterations is still adequate to keep other processors busy (k/2P < L/P). For example, if we suppose a linearly decreasing chunk size and the number of chunks is $N=2k/(CS(0) + CS(l))$, then the range of the last chunk of this loop is between 1 and k/2P, and the range of the number of chunks is from 4P to 2P.

We use this idea in both approaches, local and central queue, to define the initial value of the chunk sizes. In the case of *local queue* approach, we consider each processor individually to calculate its $LCS_i(0)$ value. That is, $LCS_i(0) = \text{queue}_i/2$.

3.2.2. *Synchronization Overhead*

Too small a value of the lasts chunk size can introduce a large *synchronization and communication* overhead. We avoid suffering from excessive synchronization and communication overhead because we define a minimal chunk size (see Section 3.2.3). Also, *synchronization* overhead depends on the implementation of chunk dispatching. Chunk dispatching will degrade significantly if the critical section takes much time and the number of processors is large. We must define an efficient chunk dispatching mechanism, which replaces a critical section by a single atomic instruction. We suppose a given chunk function CF (see Section 3.3.2), which determines the chunk size at the time *t*:

$$CS(t) = CF(W(t), CS(t-1))$$

Now, we can predict at time *t* the chunk size of the next chunk at time *t+1* (CS(t+1)) using:

$$CF(W(t), CS(t))*AP \qquad (1)$$

This value can be computed by a processor each time it grabs iterations, and only if W(t+1) changes a lot with respect to the load used in (1), we must recalculate the chunk size. If we use the predicted value, then the index of chunks, *t*, would be the only shared variable that processors have to access atomically.

- In the case of a *central queue*, we apply (1) over the global chunk size that is shared between the different processors and which is adaptively adjusted. In this case, the utilization of our predictable mechanism to reduce the synchronization overhead is very important.
- In our *local queue* approach, we introduce a synchronization cost associated with access to a remote work queue. In this case, each chunk size is adaptively and independently adjusted by a chunk-size function. Under the assumption that an idle processor grabs a number of iteration according to the remote chunk-size, if we use the remote predicted chunk-size, then the synchronization operation will be inexpensive.

3.2.3. *Communication Overhead*

The *communication overhead* problem is considered to improve the processor affinity for both shared and distributed memory systems. The motivation for exploiting processor affinity in loop scheduling derives from the observation that, for many parallel applications, the time spent bringing data into the local or cache memory is a significant source of overhead data (local versus non-local data). That is, loop iterations frequently have an affinity for a particular processor (the one whose local memory or cache contains

the required data). In this way, we try to reduce the *communication overhead* according to the next ideas:

- In the case of *local queues* (for both, shared and distributed memory systems) our approach tries to share the iterations between processors when one of them is overloaded (processor *o*) and the other one is not (processor *i*), according to certain constraints. To minimize communication overhead, we reassign a chunk only if it is necessary to balance the load, according to the next procedure:

  i) Let $LCS_o = \max_{j=1,P \text{ and processor } j \text{ overloaded}} \{LCS_j(t)\}$
  ii) Only if (Loop-Pattern $\approx$ 0) or ($LCS_o/queue_o$ << Loop-Pattern and Loop-Pattern $\approx$ 1)
  ii.1) Send a given number of iterations, according to the current value of LCS_o, from processor *o*
  to processor *i*.

  Observation: In the best case, when the load is uniform, Loop-Pattern is about 1. The ($LCS_o/queue_o$ << Loop-Pattern) condition avoids the thrashing problem and maximizes the locality property. This condition establishes that there is a large load that must be shared with other nodes. In general, we don't allow migration of iterations when there is a low number of iterations that must be executed again in the current node (we exploit the locality property). In addition, our system adjusts the chunk size of the destination node after a migration, to avoid a new migration of the iterations (in this way, we avoid that our system spends more time migrating iterations that executing them, that is, a global trashing problem). When the load is non-uniform (the worst case), Loop-Pattern is close to 0. In this case we must load balance all time.

- In the case of a *central queue* (an application based on the master-slave paradigm) we don't have this problem because we allocate iterations all time.

3.2.3.1. Communication Overhead on Large Networks

The existence of memory that is not equidistant from all processors (such as cache or local memory) implies that some processors are closer to the data required by an iteration than others are. Thus, we can't suppose a binary situation (local and non-local data) because we have a local network. In this case, the communication cost is important and must be reduced:

- In the case of *local queues* (for both, shared and distributed memory systems) our approach tries to share the iterations between closer processors when one of them is overloaded and the other one is not (we schedule a loop iteration on the processor closest to the processor that already contains the necessary data). That is, an idle

processor i examines the work queue of all the other processors and removes iterations from the work queue of a processor j according to the cost function 2.

$$LCF_j(t) = \alpha\, Cc(i, j)*LCS_j(t) + \beta\, Cli_j(t) \tag{2}$$

where $Cli_j(t)$ is:

$$Cli_j(t) = \left(LW_j(t) - \left(\sum_{i=1}^{P} LW_i(t) / P \right) \right)^2$$

and if $\sum_{j=1}^{P} LCS_j(t)/P$ is small, then α should be large and β small

/* We try to minimize the communication cost

if $\sum_{j=1}^{P} LCS_j(t)/P$ is large, then α should be small and β large

/* We try to minimize the load imbalance cost

We use the following procedure to define in which processor j we need to search for a processor i that has no more iterations in its local queue:

If $\sum_{j=1}^{P} LCS_j(t)/P$ is small, then select processor o such as

$$LCF_o = \min_{j=1,\,P \text{ and processor } j \text{ overloaded}} \{LCF_j(t)\}$$

If $\sum_{j=1}^{P} LCS_j(t)/P$ is large, then select processor o such as

$$LCF_o = \max_{j=1,\,P \text{ and processor } j \text{ overloaded}} \{LCF_j(t)\}$$

In this way, we try to minimize the load from the most heavily loaded processor or the communication cost, according to the current load on the system. Particularly, when $\sum_{j=1}^{P} LCS_j(t)/P$ is small we suppose that there is a low number of iterations to be executed in each node. In this way, we share the workload only among neighbor nodes and avoid large communication costs due to the small load imbalance.

- In the case of a *central queue* (an application based on the master-slave paradigm) in a *distributed memory system*, we minimize the number of messages and the size of them, particularly between distant processors, to reduce network contention. We don't use all slaves, only at the beginning. At the end when slave processors are idle, the slave processors closer to the master are selected to minimize communication cost (the load is very low to share with distant processors). We use the next mechanism to select one processor to receive iterations:

If (NRI(t) is large) or (NRI(t) is medium and LW_a is large/medium)

assign load to every idle processor

/* We try to minimize the load balancing cost

else

assign load to the processors close to the master processor

/* We try to minimize the communication cost and the load balancing cost

where LW_a is the average load on the processors close to the master processor. Each node knows its neighbors. In this way, we can know a processor is near to another because we can implement a procedure to know the neighbor of a neighbor, and so forth.

$$LW_a = \frac{\sum_{j \in Nn} \lceil LW_j(t) \rceil}{|Nn|}$$

and Nn is the set of processors that are neighbors of the master processor.

In the last case, we don't send iterations to more remote processors because the communication cost is high.

3.3. *Chunk Size Determination*

In this stage, we must determine the chunk size (number of loops to assign to each processor) in order to minimize the load imbalance and communication costs, and synchronization overhead. We propose a chunk size adjustment mechanism to increase and decrease dynamically the chunk size (allocation granularity) according to the current load on the system. In this way, processors acquire a given set of iterations dynamically. The difference between our approaches is in the manner in which the chunk is modified: the *central queue* approach has a global chunk size, and in the *local queue* approach each local queue has its own chunk size.

- Our *distributed queue* approach partitions iterations of a parallel loop *statically* into local queues, so that each processor is involved in remote access (migration of the iteration execution) only when load imbalance occurs. In our approach, a

deterministic initial assignment policy is used to decompose a parallel loop into local queues of processors, which improves the processor affinity property. Initially, the number of iterations in each queue is equal to the bound of the loop divided by the number of processors.

- In our *central queue* approach, the partitions are defined *dynamically* (determination of the chunk size is calculated dynamically) to match the current situation at runtime.

In each case, we can handle loops with a wide range of load distributions when no prior knowledge of the execution can be used, in order to minimize the load balancing.

3.3.1. *Chunk Size Adjustment Mechanism*

The chunk size adjustment mechanism must minimize synchronization cost, communication cost and load imbalance cost. In general, getting large chunks may cause load unbalancing while getting small chunks will induce too much scheduling and communication overhead (synchronization overhead, etc.). In order to achieve small overhead and load balancing, we propose the following method to define the initial chunk size:

- In the *central queue* approach, our procedure assigns a large number of iterations at the beginning (*CS(0)*), and decreases or increases the chunk size according to the workload until all iterations are exhausted (see section 3.2.1).
- In the *local queue* approach, every local initial chunk size ($LCS_i(0)$) is large. Then, we decrease or increase each local chunk size according to its workload. When a processor *i* grabs iterations from a processor *j* according to the chunk size $LCS_j(t)$, the new chunk size $LCS_i(t+1)$ will be $LCS_j(t)/2$.

In addition, the scheduling algorithm must adapt the number of iterations to execute over one processor dynamically, in order to balance the workload. In this way, we propose the following mechanism:

- In the case of *local queues*, if a processor has a heavy load we decrease its chunk size *(Condition 1)*, so that more iterations remaining in the heavily processor can be executed by those lightly loaded processors, thereby balancing workload more efficiently. If a processor has a normal or little load *(Condition 2)*, we increase the chunk size, so that processor can finish all the iterations in its local work queue as soon as possible, and then immediately starts to help heavily loaded processors. In our approach, we must determine the state $state_i(t)$ of each processor.
- In the case of *a central queue*, if the current load on the system is normal or heavy *(Condition 1),* we decrease the chunk size to balance the workload. If the current load on the system is small *(Condition 2),* we increase the chunk size to minimize synchronization and communication overhead.

In this way, our mechanism is the following:

1. If *Condition 1*
   1.1 Decrease LCS_i (or CS in the case of central queue)
2. If *Condition 2*
   2.1 Increase LCS_i (or CS in the case of central queue)

In both cases, we modify LCS_i or CS each time that we need to allocate a given number of iterations for execution. In the local queue case, each time that we grab iterations from processor i we modify LCS_i. In this way, we obtain a good tradeoff between small overhead and load balancing for both uniformly and non-uniformly distributed parallel loops.

3.3.2. *Chunk Size Functions*

We propose different variations to design the chunk-size (decrease or increase) function based on the work by Yan, Jin and Zhang [15]. Our mechanism uses the current workload and the current value of the chunk sizes (CS(t) in the case of a central queue, or the chunk size of processor i ($LCS_i(t)$) in the case of local queues) as its two input parameters, and adjusts the chunk-size variable as follows:

a) *Exponentially:* Each processor increases or decreases the value of its chunk-size by a constant *base*. The user specifies the constant. This chunk size function can be used in both approaches, local and central queues (we replace LCS_i per CS).

$$LCS_i(t) = \begin{cases} LCS_i(t-1)/base & \text{if we must decrease } LCS_i \\ LCS_i(t-1)*base & \text{if we must increase } LCS_i \end{cases}$$

b) *Linearly:* A processor increases or decreases its chunk size by a constant *base*. The linearity can make the chunk function simple enough to induce a small scheduling overhead. It has less risk of imbalancing the workload than the previous one, but tends to incur in a larger synchronization overhead. This chunk size function can be used in both approaches, local and central queues.

$$LCS_i(t) = \begin{cases} LCS_i(t-1) + base & \text{if we must increase } LCS_i \\ \max\{1, LCS_i(t-1) - base\} & \text{if we must decrease } LCS_i \end{cases}$$

c) *Conservatively:* This means by restricting the varying range of the chunk size within P/2 and 2P to avoid assigning too big or too small a chunk size. This chunk size function can be used in both approaches, local and central queues.

$$LCS_i(t) = \begin{cases} \min\{2P, LCS_i(t-1) + base\} & \text{if we must increase } LCS_i \\ \max\{P/2, LCS_i(t-1) - base\} & \text{if we must decrease } LCS_i \end{cases}$$

d) Greedily: This employs a two-phase consensus method to greedily enlarge the chunking size on non-heavily loaded processors. This mechanism records the previous load state of the processor: if a processor is in a non-heavily loaded state on two consecutive allocations, it sets the chunk-size equal to the remaining iterations on queue *i* (i.e., it grabs all the remaining iterations in the local work queue to execute). Otherwise the processor increases or decreases the chunking size using one of the previous chunk-size functions. This chunk size function can be used only in the local queue approach.

3.4. *Chunk Assignment*

In general, each approach work according to the next procedure:

- For *distributed queues*, initially loop iterations are assigned to a processor's work queue so as to balance the load statically. Then, a given number of iterations of the remaining iterations in the local queue are allocated to the local processor for execution (according to its LCSi(t), $\forall i=1, P$). That is, all processors schedule loop iterations from their local queues. In this case, only the remaining iterations allocated on the local queues of the overloaded processors are shared among the processors when load imbalance occurs. In this way, each processor gets iterations from its own local work queue in parallel, and each access is local, and therefore cheap. The remote access phase is aimed at dynamically balancing the workload. For any idle processor, when the remote access finishes, a new chunk size is calculated (see Section 3.3.1). In this way, the idle processor can share its iterations if it becomes overloaded. We define the number of iterations to share according to the chunk size of each local queue (specific to each processor).
- In the case of *a central queue*, a given number of iterations of the remaining iterations (according to CS(t)) are allocated to the idle processor.

3.5. *Our parallel-loops scheduling approaches*

Now, we present our parallel-loops scheduling approaches according to different platforms:

a) *Distributed Queue Approach and Dynamic Partition (for both distributed and shared memory systems)- DQD*

- Local Queue procedure:

Repeat as long as there are remain iterations in the local queue of processor *i*
    Lock local queue
    Remove LCS_i of the remaining iterations (local queue *i*)
    Unlock local queue
    Execute these loops in processor *i*
    Modify LCS_i size according to Section 3.3

Update state

- Remote Queue procedure (when processor *i* completes execution of the loops in its local queue):

Repeat until all local queues are empty
Select processor *j* to grab iterations according to Section 3.2.3
Lock remote queue on processor *j* ($queue_j$)
Remove LCS_j of the remaining iterations from processor *j* and send it to processor *i*.
Unlock remote queue on processor *j*
Assign these iterations to $queue_i$
Modify LCS_j size according to Section 3.3
Recalculate LCS_i according to Section 3.3.1
Update state
Call "Local Queue" procedure

b) *Central Queue and Dynamic Partition (for both distributed and shared memory systems)-CQD*

Repeat until central queue is empty
Select idle processor *i* to receive iterations according to Section 3.2.3
Lock central queue
Remove CS of the remaining iterations and send to processor *i*.
Unlock central queue
Execute these loops
Modify CS according to Section 3.3
Update state

c) *Distributed Queues and Static Partition (for distributed memory systems)-DQS*

Initial partition of the loop
Allocation of each partition on each local queue
For each processor *i*, repeat until its local queue is empty
Execute these iterations in processor *i*

Our static parallel loop scheduling approach fits in actual parallel compiler, because with our system the compiler can make the partition of the loops. That is, using our system the compiler will define the loop partition to be used during the execution of the program. For the case of our dynamic approaches, it is the operating system at runtime who must control the different variables that our approaches need (CS, LCSj). But again, using our system the compiler will define the initial loop partition to be used during the execution of the program.

4. Results Analysis

We will test our model on several parallel platforms: the Origin 2000 of Silicon Graphic with 8 processors (shared memory system) and the SP2 of IBM with 16 processors (distributed memory system). The performance of our model is measured according to the optimization criteria defined in Section 2.3 and the execution times of the parallel loops. More specific, we use the next set of criteria:

- Execution times of the parallel loops (ET): It is the execution time of the benchmark. It measures how differently the scheduling algorithms work. This time is calculated in seconds.
- Degree of communication overhead (Co): It is only calculated for distributed systems. This is calculated as

$$Co = \frac{\sum_{t=0}^{ET}\sum_{i=1}^{P}\sum_{j=1}^{P} Y_{ij}^{t} Cc(i,j) LCS_j(t)}{ET * P}$$

  Where Y_{ij}^{t} is a state variable equal to 1 if the processor *i* removes LCSj(t) iterations from the work queue of the processor *j* at time *t*, otherwise is equal to 0.
- Degree of Load Imbalance (LI): This is calculated as

$$LI = \frac{\sum_{t=0}^{ET}\sum_{j=1}^{P} Cli_j(t)}{ET * P}$$

- Degree of synchronization overhead (Sy): Calculated as the number of times a processor removes iterations from a work queue (number of synchronization operations required by each algorithm). When we use a central work queue, each access to the work queue is a global synchronization operation. When we use a distributed work queue, each operation performed on remote work queues requires a synchronization operation.

More specific, we use the above criteria because they describe the main characteristic to be solved during the execution of programs with parallel-loop. ET describes the execution time of the program, and the other criteria the optimal utilization of the resources of the system. Co at level of the communication, LI at level of the processors (to avoid idle processors when we have a large workload) and Sy due to synchronization operations. This last can imply large wait times for the processes on the system.

We compare our model with the adaptive affinity algorithm (AA) [15], in the case of shared memory machines, and with the Trapezoid self-scheduling algorithm (TS) for distributed-memory machines [13]. The kernel application programs used for the

performance evaluation were carefully selected for different classes of parallel loops (kinds of affinities and load distributions). We use the next classes of parallel loops in our experiments:

Type 1: Loops with potential affinity and balanced workload. We have chosen the Successive Over-Relaxation (SOR) algorithm. In this case, the entire iterations take about the same time to execute and each iteration always accesses the same data.

```
do i= 1, L
        do parallel j=1, n
                do k=1, n
                        a[j,k]=update(a,j,k)
```

Tables 1 and 2 show the different results. In the case of shared memory systems and DQD, we have used the linear function to modify the chunk size. For distributed memory machines, we have used the greedy function. In the case of CQD and AA, we have used the exponential function for both systems. They are the functions that give the best results in each case (see [1] for more details). We have test the different chunk functions for each algorithm, but for the comparison we have used the best results obtained in each algorithm.

P	Param.	DQD			CQD			AA		
		LI	Sy	ET	LI	Sy	ET	LI	Sy	ET
2	n=1024 L=500	0.4	2	47	0.2	10	50	0.6	3	52
	n=1024 L=250	0.3	2	43	0.1	8	40	0.4	2	40
	n=512 L=500	0.3	3	33	0.1	6	40	0.4	2	42
4	n=1024 L=500	0.6	3	20	0.3	15	20	0.8	8	24
	n=1024 L=250	0.6	5	14	0.2	10	16	0.8	6	18
	n=512 L=500	0.5	5	14	0.2	8	16	0.7	6	18
8	n=1024 L=500	0.7	4	12	0.4	21	15	0.9	12.	16
	n=1024 L=250	0.6	8	9	0.5	16	12	0.8	10	12
	n=512 L=500	0.6	8	10	0.4	16	10	0.8	10	12

Table 1. Results for the Origin 2000.

P	Param.	DQD				CQD				DQS				TS			
		LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET
4	n=1024 L=500	0.5	4	8.1	19	0.2	15	15.1	21	0.1	0	0	22	1.3	12	13.2	20
	n=1024 L=250	0.3	3	4.2	13	0.1	8	10.2	18	0.1	0	0	18	1	6	9.2	14
	n=512 L=500	0.3	3	5.1	10	0.1	8	9.1	13	0.1	0	0	14	1	7	9.4	10
8	n=1024 L=500	0.7	14	4.8	11	0.3	38	9.3	15	0.2	0	0	18	1.6	16	8.1	11
	n=1024 L=250	0.5	10	2.1	9	0.1	29	4.4	12	0.1	0	0	13	1.2	12	5.2	10
	n=512 L=500	0.5	8	1.4	6	0.1	30	4.1	9	0.1	0	0	10	4.1	10	1.2	6
16	n=1024 L=500	0.6	21	2.1	10	0.6	68	4.2	15	0.3	0	0	18	3.8	40	2	12
	n=1024 L=250	0.4	16	1.2	6	0.4	59	2.0	11	0.2	0	0	12	1.8	27	1.7	7
	n=512 L=500	0.4	15	1.2	4	0.4	46	1.8	7	0.2	0	0	8	2.1	24	1.7	4

Table 2. Results for the SP2.

In general, the synchronization overhead is an important factor in this experiment. As can be seen in the tables 1 and 2, CQD performs the worst of all, due to its high synchronization overhead. DQD starts with small chunks of iterations because it uses a distributed work queue, which results in a smaller synchronization overhead. In the case of AA, lightly loaded processors need a large time to finish their jobs, and turn to help heavily loaded processors. In the meantime, the heavily loaded processors have already taken a large number of iterations to execute and did not leave enough iterations for the idle processors.

Type 2: Loops with non-affinity and balanced workload. In this case, we have used a matrix multiplication algorithm. This algorithm doesn't have affinity to exploit (in this case, iterations don't always access the same data).

```
for parallel i=1, n
        for parallel j=1,n
                for k=1,n
                        c[i,j]=c1[i,j]+a[i,k]*b[k,j]
```

Tables 3 and 4 show the different results. In the case of DQD, we have used the greedy function to modify the chunk size for distributed and shared memory systems. In the case of CQD and AA, we have used the exponential function for both systems. They are the functions that give the best results in each case (see [1] for more details).

P	n	DQD			CQD			AA		
		LI	Sy	ET	LI	Sy	ET	LI	Sy	ET
2	1024	0.5	1	11	0.1	3	13	0.8	2	12
	512	0.2	0	8	0.1	2	10	0.6	1	8
	256	0.2	0	6	0.1	1	7	0.6	1	6
4	1024	0.5	3	10	0.3	3	11	1	4	10
	512	0.3	1	5	0.2	3	6	0.7	2	4
	256	0.3	1	4	0.2	2	6	0.7	2	4
8	1024	0.6	4	6	0.5	4	8	1.3	6	6
	512	0.4	2	4	0.3	2	5	1.0	2	3
	256	0.4	2	2	0.3	3	3	1.0	2	2

Table 3. Results for the Origin 2000.

P	n.	DQD				CQD				DQS				TS			
		LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET
4	1024	0.6	2	4.1	9	0.4	4	42.1	14	0.1	0	0	11	0.8	3	40	10
	512	0.4	1	2.1	3	0.2	3	24	10	0	0	0	6	0.5	2	21.7	4
	256	0.3	1	2.4	2	0.2	2	23.2	8	0.1	0	0	5	0.4	2	22.3	3
8	1024	0.8	3	4	7	0.3	5	25	13	0.1	0	0	7	1	4	26.1	7
	512	0.6	2	2.2	3	0.2	4	18.6	9	0.1	0	0	4	0.8	3	20.5	3
	256	0.4	1	2	2	0.2	3	19	7	0.1	0	0	4	0.6	3	19.5	3
16	1024	0.7	3	3.6	6	0.6	6	18.7	10	0.2	0	0	7	1.2	5	15	6
	512	0.6	2	1.8	2	0.4	5	10	6	0.1	0	0	4	0.8	4	7	3
	256	0.4	2	1.4	2	0.4	5	16.1	3	0.1	0	0	3	0.6	3	4.2	2

Table 4. Results for the SP2.

This application is a good example of the fact that the dominant source of overhead in many application is communication, not synchronization. According to our results, for load balanced applications, aggressively adjusting scheduling granularity is an efficient method to reduce synchronization overhead and execution time. As can be seen in the tables 3 and 4, DQD performs the best of all, due to its low synchronization overhead. DQS improve its performance due to its load balancing characteristics.

Type 3: Loops with potential affinity and load imbalance: In this case, we have used the Transitive Closure algorithm. The distinguishing characteristic of this application is that each iteration of the parallel loop may take time O(1) or O(N) depending of the input data. This application will also benefit of an affinity scheduling, since i^{th} iteration of the parallel loop always accesses the i^{th} row of the matrix.

```
For k=1, n
        For Parallel j=1, n
                If a[j,k]=True then
                        For i=1, n
                                If a[k,i]=True
                                        A[j,i]=True
```

Tables 5 and 6 show the different results. We have used the same chunk size functions for each case that the first example (see [1] for more details).

P	n.	DQD			CQD			AA		
		LI	Sy	ET	LI	Sy	ET	LI	Sy	ET
2	1024	3	2	42	2.1	10	40	6	2	45
	512	2.8	1	30	1.6	6	20	5.2	1	20
	256	2.6	1	13	1.8	4	10	5	1	12
4	1024	4.1	3	19	2.8	13	18	8.2	6	22
	512	3.6	2	13	2	10	12	8	4	14
	256	3.5	2	7	2	8	6	7.8	3	8
8	1024	5.2	8	11	3.2	18	10	10	10	12
	512	4.8	5	8	3	10	7	7	7	8
	256	4.4	5	6	2.3	9	6	7.2	7	6

Table 5. Results for the Origin 2000.

P	n	DQD				CQD				DQS				TS			
		LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET
4	1024	3.2	8	1.6	18	2.7	11	23.1	17	7.1	0	0	28	3.8	9	25.1	19
	512	2.6	6	0.9	11	2	9	20.4	13	7	0	0	18	3.2	9	16.7	12
	256	2.4	5	0.7	6	1.6	8	13.2	8	7.1	0	0	11	2.8	7	14.2	7
8	1024	4.2	14	1.7	14	2.9	17	19	14	9.6	0	0	18	4.9	13	16.1	15
	512	4	7	0.9	9	2.5	12	14.1	9	8.8	0	0	11	4.6	9	10.3	9
	256	3.8	5	0.5	5	2.2	7	9.2	5	8.6	0	0	7	4.1	7	9.1	6
16	1024	6	16	1.4	8	4.8	21	12	9	9.8	0	0	15	6.8	21	9.6	9
	512	5.8	10	0.8	6	4.2	18	6.2	6	9.2	0	0	10	6.1	22	7.1	5
	256	4.9	8	0.5	3	2.8	15	5.4	4	8.8	0	0	5	5.6	17	5.2	4

Table 6. Results for the SP2.

This is the first example where there is significant imbalance in the computation across iterations, which explains why DQS performs poorly. The surprising result in tables 5 and 6 are that AA and TS perform worse than CQD. Although AA assigns only a few numbers of iterations at the beginning, the remaining iterations do not contain enough work to balance the load. These experiments show that adjusting scheduling granularity

is an efficient way to handle the load imbalance in unpredictable loop applications. Our models dynamically detect the workload distribution conditions and rapidly modify the chunk size.

Type 4: Loops with non-affinity and load imbalance: We have chosen the Adjoint Convolution algorithm. This application exhibits significant load imbalance and there is no affinity to exploit.

```
For parallel i=1, n*n
        For k=i, n*n
                a[i]=a[i]+b[k]*c[i-k]
```

Tables 7 and 8 show the different results. In the case of DQD, we have used the greedy function to modify the chunk size for both systems. In the case of CQD and AA, we have used the conservative function for both systems. They are the functions that give the best results in each case (see [1] for more details).

P	n	DQD			CQD			AA		
		LI	Sy	ET	LI	Sy	ET	LI	Sy	ET
2	1024	2.6	12	47	4.8	20	51	4.8	14	52
	512	2.4	7	33	4.4	14	30	4.2	10	30
	256	2.2	3	17	4.2	9	18	4	7	68
4	1024	3.8	23	40	7.9	31	40	7.2	26	42
	512	3.6	14	26	7.2	14	26	6.8	16	26
	256	3.4	10	18	7.3	11	14	6.4	10	14
8	1024	4.9	24	28	8.9	38	28	8.4	40	29
	512	4.4	15	14	8.6	21	15	7.9	31	16
	256	4.2	10	7	8.5	16	7	7.3	24	8

Table 7. Results for the Origin 2000.

P	n	DQD				CQD				DQS				TS			
		LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET	LI	Sy	Co	ET
4	1024	3.1	12	0.2	38	3.4	25	1	47	8.6	0	0	59	2.9	26	12	40
	512	2.4	8	0.1	25	2.9	19	0.7	29	8.4	0	0	38	2.7	18	8	26
	256	2.2	6	0.9	15	2.6	8	0.4	20	8.5	0	0	24	2.6	10	6.2	17
8	1024	4	28	0.1	24	4.6	38	0.6	30	9.1	0	0	38	3.2	41	12.1	25
	512	3.6	19	.05	15	4.1	19	0.2	19	9.1	0	0	22	3	22	6.6	15
	256	3.1	12	.05	9	3.7	10	0.4	13	9	0	0	14	2.8	12	3.2	10
16	1024	5.1	42	.07	18	5.8	68	0.7	25	9.8	0	0	30	4.9	70	2.6	20
	512	4.6	26	.08	10	5.3	59	0.2	14	9.6	0	0	19	4.6	61	1.4	13
	256	4.2	18	.05	6	5.3	46	0.3	10	9.6	0	0	11	4.5	52	1.2	9

Table 8. Results for the SP2.

There is significant load imbalance across iterations, the first iteration takes time proportional to $O(N^2)$ while the last iteration take times proportional to O(1). As expected, loop scheduling algorithms that emphasize load balancing, such as DQD and AA, perform the best. CQD assigns too much work to the first few processors and suffer load imbalance as result. In this case, a simple decrease in the chunk size is probably enough to balance the load nearly all programs. The existence of significant load imbalance forces our models to override the initial assignment of iterations to processors instead execute iterations on any available idle processor.

These results show that our approaches can be used like a parallel loop scheduling algorithm. The main advantages of our approaches are that can take into account the current workload to distribute the iterations. In the case of DQD, it minimizes the run-time synchronization overhead. In addition, if all iterations do not take the same amount of time (especially when loops are non-uniformly distributed), then DQD and CQD can achieve good load balancing.

In general, for the DQD, a shared memory system, and an affinity loop is better to use a linear function as chunk size adjustment function to exploit the locality property of the iterations. For the rest of the case is better a greedy function. In the case of the CQD, when we have loops with non-affinity and load imbalance is better to use a conservative function because we need to reduce the synchronization and load imbalance problems. For the rest of the case is better an exponential function.

5. Conclusions

Adaptively changing loop scheduling granularity to minimize load imbalance, synchronization overhead, and communication overhead, for both shared and distributed memory systems, are the major characteristics which distinguishes our models from previous ones. Our approaches attempt to minimize the three sources of overhead for different situations (uniform and non-uniform loops, central or local queues, etc.). We have proposed a predictive approach, an adaptive communication approach, and a chunk-size mechanism to address this problem. Our adaptive approaches can be affected by the system size (the cost to collect runtime information can be important), but it may be insignificant in comparison with the performance improvement and flexibility of our approaches. Our approaches are suitable for a wide range of application programs. For each case (parallel loop on shared memory and distributed memory systems), we have compared our approaches with the best algorithms known (AA and TS). Our experiments demonstrate that our models has load balancing properties comparable to those of the best known loop scheduling algorithms, while maintaining processor affinity and thereby significantly reducing communication overhead. The average number of synchronization operations required by our algorithms are not much larger than the AA or TS algorithms. As result, on most cases our algorithms perform better than any other known algorithms. In our models, we have used a fix chunk size function for each experiment. At the further, we will propose an adaptive chunk-size mechanism.

References

[1] J. Aguilar, E. Leiss, "*General Adaptive Parallel Loop Scheduling for Distributed and Shared Memory Systems*", Technical Report, Department of Computer Science, University of Houston, September 2000.

[2] A. Bull, "*Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments*", Lecture Notes on Computer Sciences, Vol. 1470, pp. 377-382, 1998.

[3] M. Cieniak, M. Javeed, W. Li, "*Compile-Time Scheduling Algorithm for a Heterogeneous Network of Workstation*", The Computational Journal, Vol. 40, N. 6, pp. 356-372, 1997.

[4] H. El_Rewini, T. Lewis, H. Ali, "*Task Scheduling in Parallel and Distributed Systems*", Prentice Hall, New Jersey, 1994.

[5] H. Le, J. Fortes, "*Communication-Minimal Partitioning and Data Alignment for Affine Nested Loops*", The Computational Journal, Vol. 40, N. 6, pp. 303-321, 1997.

[6] C. Lengauer, S. Gorlatch, C. Herrmann "*The Static Parallelization of Loops and Recursions*", Journal of Supercomputing, Vol. 11, N. 4, pp. 333-353, 1997.

[7] E. Markatos T. LeBlanc, "*Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors*", IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 4, pp. 379-400, 1994.

[8] G. Nanliken, G. Belloch, "*Space-Efficient Scheduling of Nested Parallelism*", ACM Transactions on Programming Language and Systems, Vol 21, N. 1, pp. 138-169, January 1999.

[9] F. Rastello , Y. Robert, "*Loop Partitioning versus Tiling for Cache-Based Multiprocessors*", Technical Report RR1998-13, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure. February 1998.

[10] L. Rauhwerger, D. Padua, "*The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization*", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, N. 2, pp. 160-180, February 1999.

[11] Z. Szczerbinski, "*Optimal Distribution of Loops Containing no Dependence Cycles*", Lecture Notes on Computer Sciences, Vol. 1595, pp. 1254-1257, 1999.

[12] S. Togsima, C. Chantrapornchai, E. Sha, "*Probabilistic Loop Scheduling Considering Communication Overhead*", Lecture Notes on Computer Sciences, Vol. 1459, pp-158-179, 1998.

[13] T. Tzen, L. Ni, "*Trapezoidal Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers*", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 1, pp. 87-98, 1993.

[14] J. Xue, "*Communication-Minimal Tiling of Uniform Dependence Loops*", Journal of Parallel and Distributed Computing, Vol. 42, pp. 42-59, 1997.

[15] Y. Yan, C. Jin and X. Zhang, "*Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems*", IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 1, pp. 70-81, 1997.