J. Functional Programming 2 (4): 387–405, October 1992 © 1992 Cambridge University Press

A run-time algorithm for managing the granularity of parallel functional programs[1]

GAD AHARONI, DROR G. FEITELSON[2] AND AMNON BARAK

Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel

(e-mail: gadi@cs.huji.ac.il)

Abstract

We present an on-line (run-time) algorithm that manages the granularity of parallel functional programs. The algorithm exploits useful parallelism when it exists, and ignores ineffective parallelism in programs that produce many small tasks. The idea is to balance the amount of local work with the cost of distributing the work. This is achieved by ensuring that for every parallel task spawned, an amount of work that equals the cost of the spawn is performed locally. We analyse several cases and compare the algorithm to the optimal execution. In most cases the algorithm competes well with the optimal algorithm, even though the optimal algorithm has information about the future evolution of the computation that is not available to the on-line algorithm. This is quite remarkable considering we have chosen extreme cases that have contradicting optimal executions. Moreover, we show that no other on-line algorithm can be consistently better than it. We also present experimental results that demonstrate the effectiveness of the algorithm.

Capsule review

In a parallel implementation of a functional programming language, several issues arise that do not arise in a sequential implementation. One of these is controlling the granularity of tasks. Functional programs tend to have a potential for a large number of small tasks, which often results in more parallelism than a system can support. At the same time, there is an overhead associated with creating a task, and these overhead can easily dominate the cost of computation. It is better to combine a number of potential tasks into a single real task to reduce overhead. The question under consideration is when should a new task be spawned to keep parallelism high and overhead low.

This paper proposes that for each task spawned an amount of work equal to the cost of spawning be performed by the spawning process before the spawn. One immediate result is that in the worst case a parallel program will take at most twice as long as a sequential program, since at most half the time can be spent on spawning tasks. On the other hand, in several typical situations with lots of parallelism it is shown that the proposed policy will yield a performance that is within a constant factor of optimal. Finally, it is noted that the proposed spawning policy can lead to very poor performance in some cases. However, the authors show that any algorithm that determines granularity at run time will have equally poor performance (to

[1] This research was supported in part by the Basic Research Foundation administered by the Israel Academy of Science and Humanities.

[2] Current address: IBM TJ Watson Research Center, Yorktown Heights, NY, 10598, USA.

within a constant factor) in some cases. Some experimental results support the claims made for the proposal.

1 Introduction

During the execution of parallel functional programs many independent tasks may be created, some of which may be sent for parallel execution to other Processing Elements (PEs). However, the size of each task is generally not known before its execution, which makes it difficult to decide whether to execute it locally, or to perform the relatively expensive operation of sending it for remote execution. On the one hand, sending (spawning) a small task for remote execution may result in a slowdown due to the communication overhead; on the other hand, delaying the sending of a large task may result in loss of parallelism. This problem is often referred to as the *granularity problem.*

Controlling the degree of actual concurrency to achieve effective granularity is especially important in fine-grained computation models such as functional programming, logic programming (Clark, 1990), and the dataflow paradigm (Arvind and Nikhil, 1990; Kirkham, 1990). In such models there is often too much parallelism, which burdens the run-time system with the handling of many small tasks.

We investigate the granularity problem using a simple model that addresses the main difficulties of determining when to allow the spawning of parallel tasks. In this model, the computation graph of the program is represented by a tree in which the nodes represent the tasks and the arcs represent their dependencies. The tree, whose precise shape and evolution is not known *a priori*, is traversed in parallel on a loosely-coupled multicomputer system. The shape of the computation graph is usually not known in advance because it is generally difficult to predict the exact execution patterns of recursive functions and loops. The evaluation of a task is represented by a visit to a node; this visit also reveals the successor nodes (children), which are pointed at by the out-going arcs of the visited node. The objective is to visit all the nodes of the tree in parallel in the minimum amount of time.

In this paper we present a controlled granularity (CG) algorithm for the run-time management of parallelism. The execution platform for this algorithm is a multicomputer system, consisting of several loosely-coupled PEs that communicate via messages. In such systems, sending (spawning) a task from one PE to another usually involves non-negligible overhead. The CG algorithm balances between local computation and the cost of spawning parallel tasks. This is achieved by ensuring that for every spawn of a task, an amount of work that equals the cost of the spawn is performed locally. We show that this algorithm exploits useful parallelism but curtails superfluous parallelism when appropriate. The overhead of this algorithm is rather low, merely requiring some means of keeping track of the amount of work that has been performed locally.

Our model of computation captures the main issue in deciding when to spawn tasks, i.e. the communication overhead. But it makes some simplifying assumptions and does not include all the details found in real-life situations. For example, we assume that the overhead costs of spawning a task are known, whereas in real programs it is sometimes difficult to estimate the costs of handling the result of a task, since that could be a structure whose size is hard to determine in advance. Real programs also have complex sharing between tasks, which is not considered here. Nevertheless, the simplicity of the model enables us to provide formal proofs about the presented algorithm, while taking all the details into account would make the analysis intractable.

The CG algorithm is an *on-line* algorithm, i.e. an algorithm that is required to traverse the tree without having information of future development of the tree. To analyse the performance of the CG algorithm we use as a reference point the optimal *off-line* algorithm, i.e. an algorithm that has complete knowledge about the shape of the tree and its future evolution. The CG algorithm applies *competitive* considerations (Sleator and Tarjan, 1985) to minimize the effect of inadvertently sending small tasks. An on-line algorithm x is defined to be c-competitive with respect to another algorithm y, if the worst possible ratio between the performance of x to that of y, taken over all inputs, is bounded by some small constant c (Sleator and Tarjan, 1985). The constant c is referred to here as the *competitive ratio*. We analyse several typical cases, and show that the CG algorithm competes well with the optimal off-line algorithm for these cases. This is quite remarkable considering we have chosen extreme cases that have diverse optimal executions.

Previous work on granularity control included both on-line and compile-time algorithms. Compile-time algorithms (Debray *et al.*, 1990; Hudak and Goldberg, 1985) try to decompose the source program into sufficiently large tasks by using only the static information available at compile time. These algorithms differ from on-line algorithms in that they do not have knowledge of the program's run-time behaviour, and therefore suffer from inefficient handling of recursive functions and loops. This is due to the unknown depth of recursion of some recursive functions, and the unknown number of iterations of some loops. Previous on-line algorithms (Mohr *et al.*, 1991; Peyton Jones *et al.*, 1990; Rao and Kumar, 1987) were mainly suited for *full-tree-like* computation graphs, in contrast to the CG algorithm, which can handle most computation trees efficiently.

Many theoretical models either assume zero communication time between PEs, or assume that only the receiver incurs the cost of spawning (latency). In either case, since the sender does not incur any costs for spawning a task, it appears to be worthwhile to spawn almost *every* parallel task, which is obviously not true in reality. Therefore, such models avoid the question of granularity, and address only the problem of achieving a good load balancing between the PEs (Wu and Kung, 1991). In contrast, in our model both the sender and the receiver of a task incur some overhead cost, which introduces the dilemma of when it is worthwhile to spawn a task (i.e. the granularity problem).

This paper is organized as follows: the computational model is defined in section 2. The CG algorithm is presented in section 3. Section 4 compares the performance of the CG algorithm with that of the optimal *off-line* algorithm for several typical cases. Section 5 analyses the worst-case performance of the CG algorithm.

Experimental results of the CG algorithm are given in section 6. Several previous algorithms are examined and compared to the CG algorithm in section 7, and section 8 concludes the paper.

2 The computational model

The computational process is expressed by the traversal of a tree, whose shape and run-time evolution are not known *a priori*. Each node in the tree is a *task*. Visiting the node represents the execution of the task, and is defined to take one unit of time (tasks that take more time may be represented by strings of unit-time tasks). The objective of the algorithm developed here is to visit all the nodes of the tree, using a multicomputer system, in the minimum amount of time. We assume a loosely-coupled distributed-memory system with a bounded number of PEs, which communicate via messages. On such systems the overhead costs of spawning a task are relatively high, which accentuates the granularity problem. The execution of a program on such a platform begins with one PE visiting the root node of the tree. This visit reveals the successor nodes, which are then placed in a local task pool. Traversal is performed by selecting a node from the task pool, visiting that node, and then adding its successor nodes to the pool. Nodes (tasks) in the pool may also be sent for traversal (execution) to other PEs. This process continues until all the nodes in the tree have been visited, i.e. all the local pools are empty.

Sending a node to be traversed in parallel on a different PE is called *spawning a task*. Let M (units of time) be the cost of spawning. M is a system-dependent parameter, which includes the costs of packing the task into a message, finding a suitable PE, sending and receiving the task, unpacking the task at the receiving end, and the sending and receiving of the result. We assume that the cost of spawning M is incurred by *both* the sender and the receiver of the task. In other words, when PE $\mathscr{A}$ sends a task to PE $\mathscr{B}$, $\mathscr{A}$ will resume computation after a delay of M and $\mathscr{B}$ will start to execute the task also after a delay of M. In spite of the simplifying assumption that both PEs incur the same cost, this model is more realistic than the usual model, in which only the receiver incurs the latency suffered by the message passing. First, the act of sending requires some work (packing the task into a message, unpacking the result etc.). Second, if synchronous messages are used, the sender has to wait for an acknowledgement. Finally, if we assume that sending is free, it seems as if it is beneficiary to send almost *all* parallel tasks for remote execution. However, practical experience shows that a considerable degradation of performance occurs when too many tasks are sent.

One important issue is the minimal effective 'granularity' (amount of work) of the tasks that are spawned. The essence of the granularity problem is the relation between the amount of processing represented by the task, which is the size of the subtree rooted at the task, and the cost of spawning it. A simple scheme that guarantees a performance gain is one that ensures that the amount of processing performed by the spawned task is greater than M, and the local computation that remains is also greater than M. However, since there is no prior knowledge about the execution times of the tasks, it is impossible to design a scheme in which only sufficiently large tasks are spawned. The alternative is to bound the number of spawnings of tasks that may be too small. This is the basic idea in the CG algorithm, described in section 3.

Since the number of PEs in this model is bounded, some load-balancing scheme has to be assumed to treat cases in which there are more tasks than PEs (Eager *et al.*, 1986). We assume a simple load-balancing scheme that spawns a task only if there is an idle PE. Some information-dissemination scheme has to be applied to locate idle PEs. The model is not limited to a specific information-dissemination scheme since its cost is included in M, the cost of spawning. The actual scheme used in our implementation is described in section 6.2. To simplify the analysis we assume depth first search (DFS) traversal of the tree and the selection of the oldest task in the task pool for spawning. But we will show that identical results are obtained for any other deterministic traversal scheme and task-selection policy.

To clarify the details of the computation model, consider as an example the traversal of the tree in Fig. 1. The dark nodes in this figure represent nodes that have

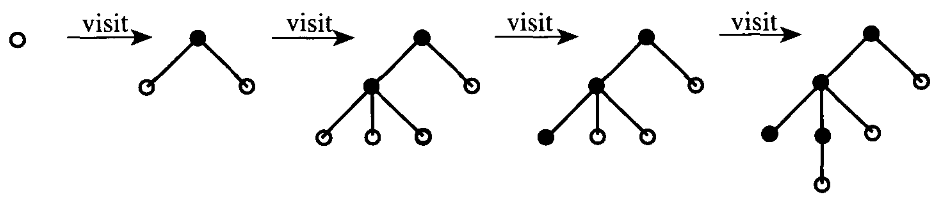

Fig. 1. DFS traversal of an unknown tree.

been visited. The light nodes are those that are in the task pool at each stage. The visit of the root node of this example reveals two successor nodes. The left successor node is visited next, revealing three additional nodes. The next visit does not reveal any new nodes, because the visited node is a leaf. DFS traversal continues, and the next visit reveals a single node. In this example, there is at least one node in the task pool after the root node has been visited, and one of these nodes may be sent to another PE to be traversed in parallel.

3 The CG algorithm

The CG algorithm attempts to resolve the granularity problem by minimizing the effect of inadvertently sending tasks that are smaller than M (the cost of spawning). This is achieved by balancing the amount of local computation performed with the cost of a spawn. More specifically, this balance is attained by ensuring that for every parallel task spawned, M nodes are processed (visited) locally.

The CG algorithm, outlined in Fig. 2, performs each spawn in two phases. In the first phase, called the traversal phase, the successor nodes of each visited node are added to the task pool. When at least $M+1$ nodes (tasks) have been added to the task pool, the task allocation phase is performed. In this phase, one task is spawned from the task pool, provided there is an idle PE to which to send it. These two phases are repeated iteratively as the tree is traversed.

To clarify the details of the CG algorithm, consider first a simpler scheme that ensures that at least M nodes are processed locally for each task spawned. This

```
$M$ : **const**; /* cost of spawning a task, $M > 0$ */
$n$:= root node; /* points at the current node */
$t$:= 0; /* counts the excess of local work over spawn overhead */
**repeat**
  **while** $t \leqslant M$
    /* visit node $n$ */
    add the children of $n$ to task pool;
    $t$:= $t$ + number of children of $n$;
    $n$:= get a node from the task pool;
  **if** there is an idle PE
    spawn *one* task from the task pool to an idle PE;
    t:=t−1; /* one less local task */
  $t$:= $t-M$; /* see text */
**until** no more nodes to visit;
```

Fig. 2. Outline of the CG algorithm.

scheme first visits M nodes and only then spawns a (single) task. However, this scheme may cause an unnecessary delay. To see why, consider a case where after k nodes have been visited there are already more than $M-k$ nodes in the task pool. In this case, a task can be spawned immediately, provided the $M-k$ nodes in the task pool are guaranteed to be executed locally. This is exactly what the CG algorithm does. The variable t is used to count the excess of guaranteed local work (nodes added to the pool) over the spawning overhead. When $t > M$, a spawn is allowed. Then t is decremented by M, effectively guaranteeing that M nodes are traversed locally to pay for the spawn. Some of these nodes have already been traversed, and the others will be traversed in the future. If t is still larger than M, an additional spawn may take place.

Note that when there is no idle PE, t is decremented by M even if no task was spawned. This is necessary to prevent the algorithm from entering an infinite loop. The algorithm may be modified slightly to keep a record of the number of tasks that could have been sent, but were not sent because there were no idle PEs available. When a PE subsequently becomes free, it can be sent a task without having to wait until an additional $M+1$ tasks are added to the task pool.

The algorithm is not limited to a specific traversal order, or to a specific selection of tasks to be spawned, but the analysis in the following sections assumes DFS traversal order, and the selection of the oldest task in the task pool for spawning. Note the adaptive nature of the algorithm, whereby the rate at which tasks are spawned is proportional to the out-degree of the tree (number of children of a node). In other words, tasks are spawned quickly when the tree is 'dense', whereas the spawning is delayed when the tree is 'sparse'.

The CG algorithm can also be extended to handle weighted trees, in which the nodes include information obtained from the compiler about the size of some sequential tasks. In such trees, the sum of the weights (rather than the number of the nodes) should be compared with M. If information on the amount of data that has to be sent is also available, the value of M may be modified. An extra amount of local computation should be performed to compensate for extra cost of sending more data, thus maintaining the balance between the cost of spawning and the amount of local computation performed.

4 Performance evaluation

This section analyses the performance of the CG algorithm by comparing its performance with that of the optimal off-line algorithm for several typical and diverse cases. We show that the CG algorithm competes well with the off-line algorithm in these cases.

It is important to emphasize the difference between this and the following section (section 5). Here we try to show that the CG algorithm is generally an effective algorithm. We present drastically different cases, for which there are contradicting optimal executions (the optimal algorithm in the sequential case does not spawn any tasks, while the optimal algorithm in the full tree spawns every available task), and yet the CG algorithm manages to compete favourably with all of them. We have examined many cases, of which we present the results of four here. We believe that for any practical purposes the CG algorithm would serve as a most effective algorithm. However, from a theoretical point of view, section 5 shows that there are cases (albeit contrived) in which the CG algorithm is not competitive with the optimal off-line algorithm.

4.1 The sequential case

We begin the comparison of the CG algorithm and the optimal off-line algorithm with a class of trees whose optimal execution is sequential. For such trees, all potential parallelism should be ignored since the size of every parallel task is smaller than the cost M of spawning a task. We show that in this case the CG algorithm is within a factor of two of the optimal algorithm. This claim is proven by showing that for *any* tree the CG algorithm is within a factor of two of any sequential algorithm (Corollary 4.1 below).

Lemma 4.1

The cost of traversing a tree with the CG algorithm is at most $2n$*, where* n *is the number of nodes in the tree.*

Proof

Let M be the cost of spawning a task. The CG algorithm may spawn at most one task for every M nodes visited. The overall execution time of the CG algorithm has an upper bound of $(n/M)M+n = 2n$, which accounts for the maximum cost of spawning, in addition to the n operations required to traverse all the nodes of the tree. □

Corollary 4.1

The CG algorithm is 2*-competitive with any sequential algorithm.*

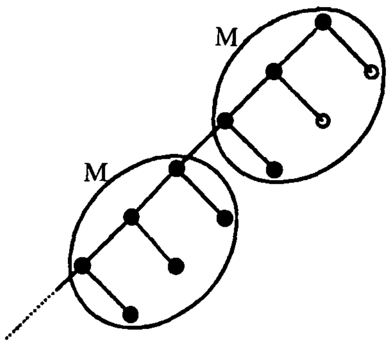

Fig. 3. The CG algorithm applied to a comb tree.

Proof
The cost of sequentially traversing a tree of n nodes is n units of time. By Lemma 4.1, the cost of the CG algorithm is at most $2n$. Hence, the cost of the CG algorithm is within a factor of two of the cost of any sequential algorithm. □

The importance of this corollary is that it guarantees that no program will run slower (by more than a factor of two) on a parallel system than on a single processor machine, a phenomenon that often occurs in parallel systems when executing fine-grained programs. In cases where sequential execution gives optimal performance, CG is 2-competitive with the optimal.

4.2 The comb-tree case

Consider the comb tree depicted in Fig. 3. In this example the tree produces mostly small tasks. The optimal off-line algorithm for such a tree is not strictly sequential; nevertheless, its execution cost is bounded from below by the height of the tree, which is $n/2$. The cost of the CG algorithm when applied to such a tree is at most $2n$ (Lemma 4.1); therefore, the CG algorithm is within a factor of four of the optimal off-line algorithm.

4.3 The full-tree case

We continue the comparison of the CG algorithm with the off-line algorithm by considering another typical case – a full (binary) tree. This case deals with a computation in which *every* potential parallelism should be exploited, namely when all the tasks are large and should be sent to idle PEs. Let P be the number of PEs, and let n be the number of nodes in the tree. Assume n is large, say $n > PM$.

Lemma 4.2
In the full binary tree case, the cost of the optimal parallel algorithm is $(M+1)\log P+(n-P+1)/P$.

Proof
An optimal off-line algorithm executes such a tree in two stages: distribution of tasks and local computation. In the distribution stage, every task created is sent to an idle PE, until all the PEs are busy. In the local-computation stage, each PE executes an

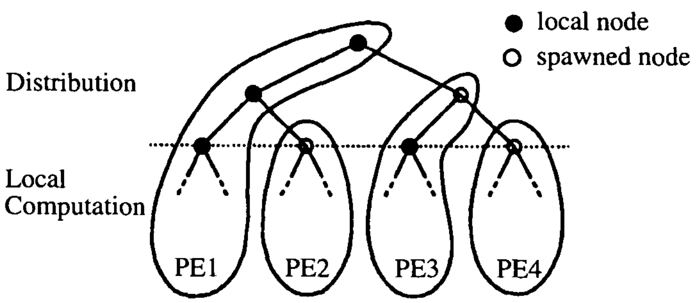

Fig. 4. The optimal algorithm applied to a full binary tree.

equal part of the tree in parallel. Since the distribution of tasks is done in parallel, it takes $\log P$ steps before all the PEs are busy, where each step includes the traversal of one node and the sending of one task. The cost of the distribution stage is therefore $(M+1)\log P$. Once all the PEs are busy, each executes an equal part $(1/P)$ of the rest of the tree in parallel. Since the distribution stage forms a binary tree with P leaves, the total number of nodes traversed in the distribution stage is $P-1$, and the size of the rest of the tree is $n-P+1$. The cost of the local-computation stage is therefore $(n-P+1)/P$. Hence, the overall cost of the optimal algorithm is $(M+1)\log P+(n-P+1)/P$. □

Fig. 4 illustrates the proof of Lemma 4.2 by giving the optimal execution of a binary tree for an example of four PEs. The dotted line in the figure indicates the end of the distribution stage and the beginning of the local-computation stage. Visiting the root of the tree creates two tasks (two subtrees to traverse), one of which (the rightmost child) is sent to one of the other PEs. The next node visited creates two more tasks, and again the right subtree is sent to an idle PE. At the same time that this subtree is spawned another subtree is spawned by the PE that received the first task sent. Hence, at this point all four PEs are busily working, each on a quarter of the rest of the tree. The cost of the distribution stage in this example is therefore two spawns, and two nodes traversed. All other spawns and traversed nodes are not counted since they occurred in parallel. The cost of the local-computation stage is calculated by subtracting the nodes traversed during the distribution stage from n, the total number of nodes in the tree, and dividing by four. The overall optimal cost of this example is therefore $2M+(n+5)/4$.

Lemma 4.3

In the full binary tree case, the cost of the CG algorithm is $(3M/2)\log P+(n-P+1)/P$.

Proof

The cost of distributing the tasks for the CG algorithm is also $M\log P$, but there are $(M/2)\log P$ local operations performed before the last task is sent (the factor is $M/2$ rather than M because a task is spawned after M tasks are exposed, but not necessarily visited). The cost of the local-computation stage remains

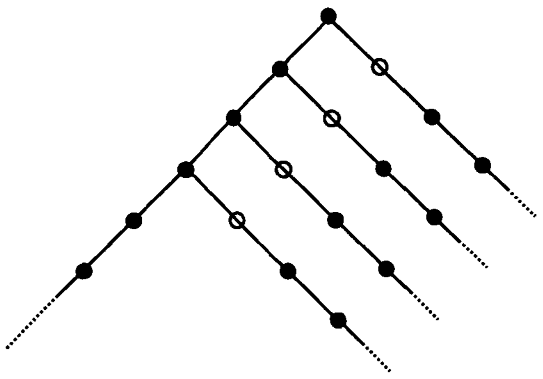
Fig. 5. A client-server computation tree.

$1/P$ of the rest of the tree. Therefore, the overall cost of the CG algorithm is $(3M/2)\log P+(n-P+1)/P$. $\square$

Theorem 4.1

In the case of a full binary tree, the CG algorithm is within a factor of 3/2 *of the optimal algorithm.*

Proof

The ratio between the performance of the CG algorithm, given in Lemma 4.3, to that of the optimal algorithm, given in Lemma 4.2, is always less than 3/2, and tends to one for a large n. $\square$

4.4 The client-server case

We now analyse the performance of the CG algorithm when applied to a client-server tree, which is portrayed in Fig. 5. This case has one 'server' PE handing out sequential work to the rest of the PEs.

The analysis of this case is similar to the analysis of the binary tree, only the work distribution is linear, rather than logarithmic, in P. The optimal algorithm spawns the first $P-1$ right subtrees. The time taken for the last task to start executing on its PE in the optimal algorithm is $(P-1)M+(P-1)$. In the CG algorithm there is a longer delay in the distribution of the tasks, and the overall time taken for the last task to begin execution on its PE is $(P-1)M+(P-1)M$. Again, the optimal algorithm beats the CG algorithm by at most a factor of 2.

5 Worst-case analysis

In this section we prove that there does not exist a competitive deterministic on-line algorithm for solving the problem discussed in this paper. This theorem implies that the CG algorithm is not a competitive algorithm; that is, there exists a case in which the CG algorithm executes a tree sequentially, whereas the optimal off-line algorithm manages to execute this tree almost completely in parallel. This means that the CG algorithm does not always succeed in fully exploiting all the useful parallelism that exists. Nevertheless, no other on-line algorithm can ever achieve this either.

We first give an upper bound on the worst-case performance of the CG algorithm in comparison to the optimal off-line algorithm.

Lemma 5.1

The competitive ratio between the CG algorithm and the optimal off-line algorithm has an upper bound of $2\min\{P, n/H\}$, *where P is the number of PEs, and H is the height of the tree.*

Proof

The worst-case performance of the CG algorithm when applied to a tree of n nodes is $2n$ (Lemma 4.1). The optimal off-line algorithm can at best fully use the available PEs, so it has a lower bound of n/P. Moreover, it is also bounded from below by the height of the tree. Hence, the optimal algorithm actually has a lower bound of max $\{n/P, H\}$. Therefore, the ratio between the cost of the CG algorithm to that of the optimal off-line algorithm is bounded by $\min\{2P, 2n/H\} = 2\min\{P, n/H\}$ for any input. □

Assume the existence of a competitive deterministic on-line algorithm for the parallel traversal of a tree whose shape is not known in advance. The performance of such an algorithm should be within a small constant factor of the performance of the optimal algorithm for *any* tree. The following counter example proves that such an on-line algorithm does not exist, at least if we require that the constant be small enough.

Consider a T-tree which consists of T full binary subtrees that are placed one on top of the other, as shown in Fig. 6. The T-tree is constructed such that the root of each

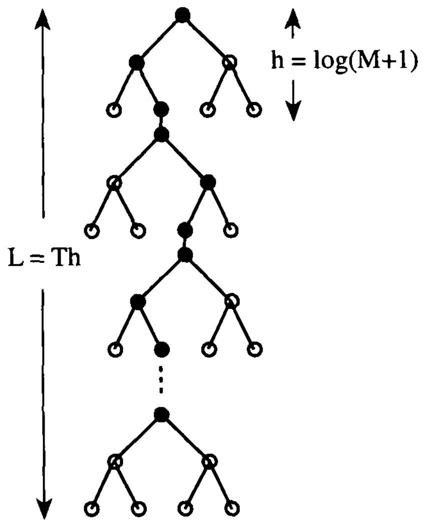

Fig. 6. T-tree.

full binary subtree (apart from the top one) is a successor node of one of the leaves of the full binary subtree above it. Let there be M nodes in each such subtree, where M is the cost of spawning a task, and let $h = \log(M+1)$ be the height of the subtree. The height of the T-tree is therefore $L = Th$, and its total number of nodes is $n = TM$. The path that leads from the root of the T-tree to the subtree at the lowest level is called the *spine*.

Lemma 5.2

For every deterministic on-line algorithm, there exists a T-tree such that the cost incurred by the algorithm in traversing this tree is at least n time units.

Proof

Given a deterministic on-line algorithm, an adversary can construct a T-tree in a way that will cause it to incur a cost of at least M for each full binary subtree. If the algorithm spawns a task while traversing a certain subtree, then this already costs M operations. If it does not, the deterministic traversal order allows the adversary to identify which leaf will be visited last. The adversary then places the next subtree as a successor node of the last leaf node visited in the subtree above it. This forces the algorithm to visit all M nodes of the subtree before reaching the root of the next subtree. Therefore, the algorithm incurs a cost of at least $TM = n$ time units for the whole T-tree. □

Lemma 5.3

There exists an off-line algorithm that traverses any T-tree in less than $n/P + L + MP$ time units.

Proof

The off-line algorithm has the advantage of knowing the structure of the tree in advance. Consider an off-line algorithm where the first PE begins traversal of the T-tree at the root, and continues down the spine until it visits some r spine nodes (the way r is determined is shown below). All the other tasks exposed while going down the spine are collected in the local pool. The first PE then sends the whole subtree under the $r+1$ spine node to be traversed in parallel on an idle PE, while it works on the tasks left in the local pool. The PE that receives the spawned task applies the same algorithm to decide when to spawn again.

Let A be the number of nodes left for local traversal after the task has been sent. Let B be the number of nodes in the task (the subtree whose root is the $r+1$ spine node). The number of nodes in the T-tree is therefore $n = A+B+r$. Let $P > 1$ be the number of PEs, and let $\mathcal{T}_P(m)$ be the time it takes to process a T-tree with m nodes on P PEs using this algorithm. The algorithm traverses down the spine in hops of h nodes and tries to select an r such that $\mathcal{T}_p(B) = A$, where p is the number of PEs that are still idle. As shown below, setting r to $1/p$ of the height of the tree is a good choice.

The following case analysis uses induction on the number of PEs to evaluate the cost of the above off-line algorithm. To keep the equations simple, we assume that all the necessary values divide each other without a remainder:

- Base case: the number of PEs is $P = 2$. The algorithm selects $r = L/2$, so that $B = n/2$. From $n = A+B+r$ it follows that $A < B$, so that the time to perform the parallel task is greater than the cost of the local remaining tasks. The total parallel cost is therefore

$$\mathcal{T}_2(n) = r + M + B = L/2 + M + n/2$$

which is made up of sequential traversal of r spine nodes, then spawning one task, followed by sequential traversal of B nodes.

- In the case of $P = 3$, the off-line algorithm selects $r = L/3$, so that $B = 2n/3$. The time to compute the parallel task on the two available PEs is (using the result for the case $P = 2$)

$$\mathcal{T}_2(B) = (2L/3)/2 + M + (2n/3)/2 = L/3 + M + n/3.$$

The time to compute the parallel task, $\mathcal{T}_2(B)$, is greater than the time it takes to traverse the remaining local A nodes, since $A = n/3 - L/3$. The total parallel cost of executing the tree in this case is therefore

$$\mathcal{T}_3(n) = r + M + \mathcal{T}_2(B) = 2L/3 + 2M + n/3.$$

- For the general case, $P > 2$, the induction hypothesis is that

$$\mathcal{T}_{P-1}(B) = (P-2)\,L'/(P-1) + (P-2)\,M + B/(P-1),$$

where L' is the length of the spine of the subtree with B nodes. The off-line algorithm chooses $r = L/P$, so that $B = (P-1)\,n/P$ and $L' = (P-1)\,L/P$. The total cost on P PEs is therefore

$$\begin{aligned}\mathcal{T}_P(n) &= r + M + \mathcal{T}_{P-1}(B)\\ &= L/P + M + (P-2)(P-1)\,L/(P(P-1)) + (P-2)\,M + (P-1)\,n/(P(P-1))\\ &= (P-1)\,L/P + (P-1)\,M + n/P.\end{aligned}$$

The cost of the above off-line algorithm is therefore bounded by

$$n/P + L + MP. \quad \square$$

Theorem 5.1

The competitive ratio of any deterministic on-line algorithm to the optimal parallel algorithm has a lower bound of $\frac{1}{3}\min\{P, n/L\}$, *provided the tree size satisfies* $n > MP^2$.

Proof

The cost of any on-line algorithm is at least n time units for some T-tree (Lemma 5.2), and the cost of the optimal off-line algorithm is at most $n/P + L + MP$ (Lemma 5.3). Note that we do not know which of the three terms is the largest; nevertheless, we can infer that the actual cost is bounded from above by $3\max\{n/P, L, MP\}$. Therefore, the ratio of the performance of the on-line to the off-line algorithms is bounded from below by $\frac{1}{3}\min\{P, n/L, n/MP\}$. The third term can be as large as we want, because n can be as large as we want. The second term does not necessarily increase with n, since L is also a function of n. If we limit the discussion to inputs (trees) that satisfy $P \leqslant n/MP$, which implies $n \geqslant MP^2$, then the bound is actually determined by the first two terms, that is, $\frac{1}{3}\min\{P, n/L\}$. $\square$

The implication of Theorem 5.1 is that there does not exist a competitive deterministic on-line algorithm for solving the problem presented in this paper (deciding when to spawn tasks). In simple terms this means that for any on-line algorithm there exists at least one tree that the on-line algorithm executes in sequential time, whereas the optimal off-line algorithm can run this tree almost completely in parallel. Since this proposition applies to any on-line algorithm it also applies to the CG algorithm. In other words, the CG algorithm may sometimes be unsuccessful in fully exploiting all the useful parallelism that exists. However, it is important to emphasize that the cautiousness feature of the CG algorithm (which sometimes causes the algorithm to overlook useful parallelism) guarantees 'safety', in the sense that the algorithm will never run significantly slower than sequential time (Corollary 4.1).

We note that because there does not exist a competitive on-line algorithm it is not easy to compare the CG algorithm with other on-line algorithms. The problem with comparing two on-line algorithms is that each algorithm may perform better than the other on different trees. This makes it difficult to define the meaning of one algorithm being better than the other.

6 Experimental measurements

This section presents performance results of the CG algorithm. The execution platform is the MOSIX system (Barak and Wheeler, 1989), a distributed operating system with a built-in dynamic process migration mechanism. The MOSIX system integrates a cluster of loosely-coupled, independent processors to a virtual, single machine UNIX environment. The specific configuration used includes eight NS32532 computers, each with its local own memory and communication devices. These computers are arranged in two identical enclosures, each with four processors that communicate via a shared VME bus. The two enclosures are connected by a ProNET-80, an 80 Mbits/second token-ring LAN.

We present results of two different implementations: a distributed tree traversal, which implements the abstract model described in section 2, and a distributed λ-calculus evaluator.

6.1 Performance of the Lambda-calculus evaluator

Our implementation of the λ-calculus evaluator is based on compiled graph reduction techniques (Peyton Jones, 1987), but without many of the optimizations. The evaluator accepts a functional program written in the usual λ-calculus notation, and produces target code in C. The tasks are realized as Unix processes, and task-spawning as Unix forks. In these experiments we have used $M = 2000$, which means that we have estimated the cost of a spawn (forking a Unix process and its migration) to be 2000 reduction steps. This implementation takes advantage of the automatic load-balancing of the MOSIX system. To comply with the CG algorithm's requirement of stopping task spawning when there are no more idle PEs, this implementation stops forking new processes when the load on the local machine is

Table 1. *Performance of the CG algorithm on the λ-calculus evaluator*

	power(22)		*fib*(32)		*comb'*(500, 8)	
Configuration	Time (sec)	Speedup	Time (sec)	Speedup	Time (sec)	Speedup
Serial	3660·3	1	3062·0	1	126·5	1
1 PE	3935·0	0·93	3290·3	0·93	135·9	0·93
2 PEs	1980·3	1·9	1672·3	1·8	139·0	0·91
4 PEs	1024·5	3·6	877·2	3·5	143·6	0·88
8 PEs	541·1	6·8	478·7	6·4	142·4	0·89

above some fixed threshold (which is equal to the load of about three processes). This approach is effective since the automatic load-balancing work well (Barak and Shiloh, 1985), and the processes are evenly distributed across the system. Hence, if the local load is high, this indicates that all the PEs are not idle. The results show that the CG algorithm manages to bridge the gap between the fine granularity of λ-calculus and the coarse granularity of Unix.

Let the functions *fib*, *power* and *comb'* be defined by:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$
$$power_2(0) = 1$$
$$power_2(n) = power_2(n-1) + power_2(n-1)$$
$$load(0) = 1$$
$$load(n) = load(n-1) \text{ and } load(n-1)$$
$$comb'(0, n) = 1$$
$$comb'(h, n) = comb'(h-1) + load(n)$$

Table 1 presents the performance results of the CG algorithm when applied to these functions. The table gives the execution time (in seconds) and the speedup for different numbers of PEs. The speedup is calculated by the ratio T_s/T_p, where T_s is the *serial* execution time obtained from a purely sequential evaluator, and T_p is the parallel execution time obtained from the distributed evaluator running on p PEs. The ratio T_s/T_1 indicates the overhead of distributing the serial evaluator, including the overhead of the CG algorithm. This overhead turns out to be about 7%.

The power function represents the case with much useful parallelism, the *fib* function is the standard Fibonacci function, and the *comb'* function represents the case which generates many small tasks. The performance results show that the CG algorithm manages to exploit the useful parallelism in the case of the *power* and *fib* functions, while there is no significant degradation of performance in the case of the *comb'* function in spite of the large number of small tasks. We note that the execution

Table 2. *Performance of the CG algorithm on the abstract model*

	power(17)		*comb*(32000)		*fib*(23)		*serv*(24, 5000)	
PEs	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	8·33	1	4·46	1	5·89	1	7·87	1
2	4·20	1·98	4·84	0·92	3·02	1·95	4·47	1·76
4	2·26	3·69	4·81	0·93	1·69	3·49	2·80	2·81
6	1·81	4·60	4·89	0.91	1·31	4·50	2·26	3·48
8	1·50	5·53	4·84	0·92	1·20	4·91	2·03	3·88

of the function *load* is sequential due to the non-strict *and* operation. The execution time of *load*(8) is approximately 0·2 seconds, which results in flooding the system with many small tasks when executing the *comb′* function.

6.2 *Performance of the abstract model*

The abstract model is defined to be the traversal of a tree on a multicomputer system, where the objective is to visit all the nodes in the tree in the minimum amount of time. Our distributed implementation of such a model first builds a tree of a desired shape, and then traverses the tree according to the principles of the CG algorithm. One advantage of such an approach is the ability to isolate the granularity problem from other parameters, and thus examine it closely. Another important advantage is the possibility of measuring the performance of the execution of random trees.

Each PE is realzied as a Unix process, and tasks are realized as messages containing references to subtrees. To simplify the task distribution policy, we assume that the PEs are connected along a (logical) directed ring, although the physical connection allows full connectivity among all the PEs. In our policy, whenever a PE becomes idle, it sends a request for work to the next PE along the ring. A PE that receives a request for a task when its task pool is empty passes on the request to the next PE along the ring. A PE that receives a request when its task pool is not empty responds by sending a task to the requesting PE. These tasks are sent directly, and not through the ring.

Table 2 depicts the execution time (in seconds) and the speedup of the four types of trees: *power* tree (full binary), *fib* tree, *comb* tree and *serv* tree. The traversal of these trees represents the execution of the functions *power*, *fib*, *comb* and *serv*. The latter two functions are defined by:

$$\begin{aligned} &comb(0) = 1\\ &comb(h) = comb(h-1) * (h+1)\\ &chain(0) = 0\\ &chain(n) = 1 + chain(n-1)\\ &serv(0, m) = 0\\ &serv(n, m) = serv(n-1, m) + chain(m). \end{aligned}$$

Table 3. *Average speedup of the CG algorithm when applied to random trees*

	Expected value of out-degree			
PEs	1·1	1·5	2·0	2·7
1	1	1	1	1
2	1·37	1·85	1·96	1·97
4	1·55	2·63	3·15	3·32
6	1·63	2·87	3·65	4·07
8	1·54	2·93	3·79	4·35

The *power* function represents the case with much useful parallelism, the *comb* function represents the case with much ineffective parallelism, and the *serv* function represents the client-server case. The experiment was run with $M = 800$. Note that the CG algorithm manages to exploit the useful parallelism in the *power*, *fib* and *serv* trees, whereas it avoids falling into the trap of trying to utilize the superfluous parallelism of the *comb* tree.

Table 3 presents the performance figures of the CG algorithm when applied to random trees. The table lists the expected values of the out-degree used for constructing the random trees, and the corresponding average speedups obtained. The random trees are constructed with a bounded height, and the number of children (out-degree) at each node is determined randomly, using a binomial distribution. Thus, the trees become denser as the expected value of the out-degree increases, which also increases the potential useful parallelism in the trees. In the experiment, we ran 200 different random trees for each entry in the table, and calculated the speedup by comparing the sequential time with the parallel execution time. Consider for example the fourth column in the table, which presents the results of running random trees with an expected out-degree value of 2·0, and compare it to the execution of a full binary tree depicted in the third column of Table 2. It is important to emphasize that being random trees they are of different sizes and shapes, and therefore do not all contain only useful parallelism, as is the case in completely full trees. The results show a clear trend of a higher average speedup for a denser tree, i.e. a tree with larger amount of useful parallelism on average. Hence, again, this experiment demonstrates the effectiveness of the CG algorithm, and its ability to exploit useful parallelism while ignoring superfluous parallelism.

7 Related work

Previous on-line algorithms were mainly oriented towards programs with a full-tree-like computation graph. Two such algorithms are described and examined here. One algorithm, called 'task stealing' (Mohr *et al.*, 1991), has idle PEs 'steal' tasks from non-idle PEs. The second algorithm, which is used in the GRIP project (Peyton Jones *et al.*, 1990), assumes that every PE knows the global load of the system. Then, based on this knowledge, each PE 'sparks' (spawns) new tasks if that load is below some threshold. Both of the above algorithms attempt to reduce the communication overhead when all the PEs are busy, by processing all the tasks that are created locally.

Let us now examine the application of the above two algorithms to a program that produces many small tasks. Consider a comb-like tree, as shown in Fig. 3. The task-stealing algorithm has idle PEs repeatedly stealing tasks from the PE that executes the main spine of the comb tree, only to find that these tasks are small. Similarly, in the GRIP algorithm, the PE that executes the main spine continuously sparks new tasks, since it relies on the global system's load, which remains low because the other PEs are executing only small tasks.

The above two algorithms may therefore spawn a separate task for each node along the spine of the comb tree. This results in a performance loss of $M-1$ for each task spawned. The overall cost is therefore $(M-1)n/2+n/2 = nM/2$, which accounts for $n/2$ spawns and $n/2$ nodes traversed locally. Hence, in the comb-tree case, the performance of the CG algorithm is an order of M better than the above algorithms, where M may be rather large.

Other on-line schemes (Hudak and Goldberg, 1984; Lin and Keller, 1987) concentrate more on the load balancing aspect of the computation, and aim to keep the PEs busy nearly all the time. These algorithms do not consider the grain size of the task, and therefore would also exhibit poor performance when applied to a program that produces many small tasks, such as the comb tree shown in Fig. 3. Our implementation shows that the load balancing is actually a secondary issue: good performance was obtained when the CG algorithm was used to decide when to spawn in order to control the granularity. Once a task was spawned, MOSIX automatically moved it to an idle PE. Granularity control is tightly linked to functional programming and other fine-grained models of computation. Load balancing is of general importance, but should not take precedence over the granularity considerations.

8 Conclusions

We have described a run-time algorithm that controls the degree of concurrency of parallel computation to achieve effective granularity. The CG algorithm that was presented is significantly better than existing strategies for solving this problem. Moreover, no other on-line algorithm can be consistently better than it. This algorithm increases granularity by exploiting useful parallelism when it exists, and ignores ineffective parallelism in programs that contain many small tasks. The overhead of this algorithm is rather small, consisting mainly of an additional program counter. Furthermore, the CG algorithm has some adaptiveness quality which distributes graphs with a high out-degree faster.

In the future we shall be looking to enhance the CG algorithm to handle more general computation graphs, such as DAGs. In addition, we shall look into the possibility of making a more sophisticated choice of which task to spawn, based on the depth of recursion observed for different functions at run time.

Acknowledgements

The authors are indebted to Sandy Irani, Michal Parnas, and Ronitt Rubinfeld for their help in the proofs of section 5. We are also grateful to Moshe Ben Ezra for implementing the λ-calculus evaluator. Special thanks are due to the referees' careful reading of the manuscript, which helped eliminate a number of errors.

References

Arvind and Nikhil, R. S. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* **39** (3): 300–318.

Barak, A. and Shiloh, A. 1985. A distributed load-balancing policy for a multicomputer. *IEEE Trans. Softw. – Practice and Experience* **15** (9): 901–913.

Barak, A. and Wheeler, R. 1989. *MOSIX: an integrated multiprocessor. Proc. Winter USENIX Conf.*, San Diego, CA, pp. 101–112.

Clark, K. L. 1990. Parallel logic programming. *The Computer Journal* **33** (6): 482–500.

Debray, S. K., Lin, N.-W. and Hermenegildo, M. 1990. Task granularity analysis in logic programs. *Programming Languages Design and Implementation*, ACM SIGPLAN, White Plains, New York, pp. 174–188.

Eager, D. L., Lazowska, E. D. and Zahorjan, J. 1986. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.* **12** (5): 662–675.

Hudak, P. and Goldberg, B. 1984. Experiments in diffused combinator reduction. *ACM Symp. on Lisp and Functional Programming*, Austin, TX, pp. 167–176.

Hudak, P. and Goldberg, B. 1985. Serial combinators: 'optimal' grains of parallelism. *Functional Programming Languages and Computer Architecture.* Volume 201 of *Lecture Notes in Computer Science.* Springer-Verlag, pp. 382–399.

Kirkham, C. 1990. The Manchester dataflow project. In Fountain, T. J. and Shute, M. J., editors, *Multiprocessor Computer Architectures.* North-Holland, pp. 141–153.

Lin, F. C. H. and Keller, R. M. 1987. The gradient model load balancing method. *IEEE Trans. Softw. Eng.* **13** (1): 32–38.

Mohr, E., Kranz, D. A. and Halstead, R. H. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel & Distributed Syst.* **2** (3): 264–280.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall.

Peyton Jones, S. L., Clack, C., Salkild, J. and Hardie, M. 1990. GRIP – a high-performance architecture for parallel graph reduction. In Fountain, T. J. and Shute, M. J., editors, *Multiprocessor Computer Architectures.* North-Holland, pp. 101–119.

Rao, V. N. and Kumar, V. 1987. Parallel depth first search. *Int. J. Parallel Programming,* **16** (6): 479–519.

Sleator, D. and Tarjan, R. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM,* **28** (2): 202–208.

Wu, I.-C. and Kung, H. T. 1991. Communication complexity for parallel divide-and-conquer. *32nd Symp. Foundations of Computer Science,* San Juan, Puerto Rico, pp. 151–162.