Distance: a New Metric for Controlling Granularity for Parallel Execution

Kish Shen*
IC-Parc
Imperial College
London SW7 2AZ, U.K.
k.shen@icparc.ic.ac.uk

Vítor Santos Costa
FCUP & LIACC
Universidade do Porto
4150 Porto, Portugal
vsc@ncc.up.pt

Andy King
Computing Laboratory
University of Kent at Canterbury
CT2 7NF, U.K.
A.M.King@ukc.ac.uk

Abstract

Granularity control is a method to improve parallel execution performance by limiting excessive parallelism. The general idea is that if the gain obtained by executing a task in parallel is less than the overheads required to support parallel execution, then the task is better executed sequentially. Traditionally, in logic programming, task size is estimated from the sequential time-complexity of evaluating the task. Tasks are only executed in parallel if task size exceeds a pre-determined threshold.

We argue in this paper that the estimation of complexity on its own is not an ideal metric for improving the performance of parallel programs through granularity control. We present a new metric for measuring granularity, based on a notion of *distance*. We present some initial results with two very simple methods of using this metric for granularity control. We then discuss how more sophisticated granularity control methods can be devised using the new metric.

1 Introduction

Granularity control is a method to improve parallel execution performance by limiting excessive parallelism. The general idea is that if the gain obtained by executing a task in parallel is less than the overheads required to support parallel execution, then the task is better executed sequentially [9]. Granularity control has been recently applied to Prolog/Logic Programming [26, 12, 11] and to Functional Programming [10]. In logic programming, a "task" is considered to be a goal or an alternative to be executed in parallel, and "useful work" the amount of computation performed to solve the goal or try the alternative. Granularity control in these systems boils down to a two-phase process. First, a global analysis system calculates at compile-time the time-complexity of a potentially parallel goal, generally approximated by the number of reductions performed to solve the goal[1]. At run-time, a pre-determined threshold for time-complexity controls whether a task is to run in parallel or not: if the task's time-complexity is less than the threshold, it will not be executed in parallel. Results for small benchmark type programs have shown that these methods can and do increase performance of parallel logic programming systems. The main motivation for controlling grain size is to reduce overheads so that the overall execution time for a program will decrease.

A key assumption of the complexity-based approach is that there is a linear relationship between a task's complexity and the amount of overheads it incurs. In other words, one assumes there is a threshold complexity

*The work described in this paper was carried out while this author was at University of Manchester, Manchester M13 9PL, U.K.

[1] Or more precisely, an upper or lower bound on the goal's complexity, In [12, 11], a relationship between the inputs and the bound is derived for recursive goals, and this is used at run-time to determine the time-complexity.

which would induce a critical amount of overheads, below which it is not profitable to execute tasks in parallel. We shall now examine how valid such an assumption is.

Direct parallel task execution overheads correspond to the cost of creating and maintaining a parallel task. These overheads usually include the cost of scheduling the task for parallel execution, the cost of initialising the new parallel task, and hardware related costs such as interprocess communication and locking. Up to a first approximation, direct overheads are either fixed, such as publishing a task, or proportional to task size, such as the interprocess communication overheads, and are therefore proportional to the goal's time complexity for sizeable tasks.

Quantitatively, we assume that for a sequential execution, the time taken to execute a particular piece of computation (T), is proportional to the amount of work performed (W), say, as measured in number of reductions:

$$T = cW$$

In a parallel computation, if we assume that the same piece of computation was spawned to run in parallel, then the direct parallel overheads, H_D, for the task can be approximated as:

$$H_D \simeq kW + f$$

where f measures the fixed overheads, such as the overheads in creating and completing the task, and kW are the overheads incurred during execution. With the traditional view, the execution time for a task T_p is given by the sequential time plus the direct parallel overheads, i.e.

$$T_p \simeq cW + kW + f \tag{1}$$

Thus, there is an approximately linear relationship between W and H_D. With complexity-based approaches, the assumption is that there is a threshold W_t which corresponds to a H_{D_t} for which it is not profitable to run a task in parallel[2]. Note that in many systems, as long as there is parallel execution, and given the same configuration (same number of processors, same machine, etc.), The H_D probably do not vary (on average) greatly from program to program, i.e. k and f are constants across a large number of programs. Thus, a "universal" threshold can be set for a given parallel Prolog system and a given configuration. Lastly note that it is not necessary to determine what the various constants are individually if a threshold is to be used – all we need to know is that the threshold itself can be regarded as a constant for a given configuration, and experimentally determine this constant. Of course, the constant would need to be derived individually for each configuration.

However, in many parallel systems, including most parallel logic programming systems, tasks can and do create other potentially parallel tasks dynamically. Sub-task creation adds to the available parallelism, but it also adds to total overheads, which are not measured by the direct parallel task execution overheads. Instead, these new overheads should be considered a separate class, the *indirect parallel execution overheads*. Indirect overheads can contribute significantly to the total overhead. Indeed, and as shall be discussed later, experimental results from [11] strongly suggest that, at least for those two configurations and programs studied, the major factor for the increase in performance of the programs with granularity control is avoiding largely superfluous parallel work which the system does not have the resources to exploit.

In contrast to direct overheads, indirect overheads may not be very dependent on the size of the task being executed in parallel and, in addition, may vary greatly from programs to programs. With the consideration of subtask creations, the total overheads, H (assuming no thresholding), is H_D + H_I, where H_I is the indirect parallel overheads, which consists of subtask creations (and perhaps other overheads as well, as shall be considered later). H can now be approximated as:

$$H \simeq kW + f + Nq \tag{2}$$

[2] More precisely, the sizes of other tasks that are running in parallel may have to be considered as well. See [12] for an example.

where q is the (average) cost of subtask creation and spawning, and N is the number of times subtasks were created by this task. N does not have a direct relationship with W, and is highly program dependent. We thus believe that using task size (W) alone in general is not sufficient to capture the full complexities of overheads in parallel execution. We would instead prefer a more robust metric, one that is generally applicable, and could apply to cases where it is hard to estimate time-complexity, and to cases where work creation is unevenly balanced.

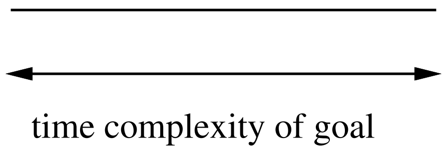

Figure 1: Sequential Execution of a Task

Towards this goal, we propose in this work a different metric for measuring "granularity", the *distance metric*, defined as the amount of work performed between successive points at which major parallel overheads are incurred. Figure 1 illustrates a task executing sequentially, without any overheads.

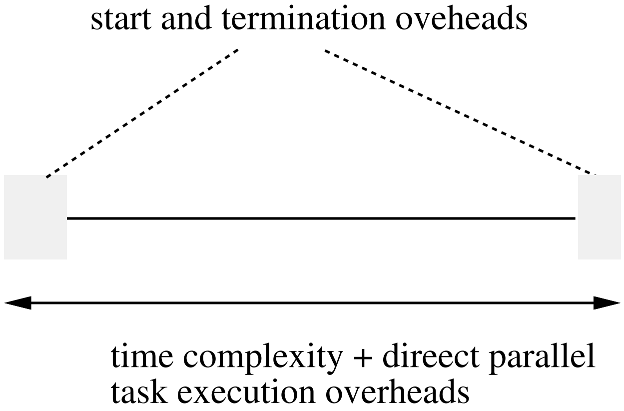

Figure 2: Conventional View of Grain-Size in Task Execution

Figure 2 shows the conventional view of granularity. The same task is executed in parallel, with initialisation and termination overheads explicit. Moreover, the task itself takes somewhat longer to execute due to run-time parallel overheads. Together these form the direct parallel execution overheads.

Figure 3 shows the situation assuming dynamic creation of parallel work. Overheads arise at several points in the execution. These overheads add to the cost of executing the task in parallel, and are unfortunately unaccounted for by the conventional view of granularity control. Note that in the general case the creation points (q in equation 2) occur at irregular intervals. Figure 3 also shows the new distance metric. The metric measures the distance (or amount of work performed) between major fixed parallel overhead points, e.g. the start and end of tasks (where large fixed overheads can occur as modelled in equation 1), and subtask creation events (where a particular q occurs in equation 2). Between such points, the amount of overheads incurred does have a (approximately) linear relationship with W, and in fact, it is approximately proportional to W, because of the kW component in equations 1 and 2. The other components of these equations do not figure in this view because we are measuring W between points where they occur. The key insight is that in order to minimise the overheads we would like to reduce the number of task creation points, and that to do so *we should increase the distance between the points that create work.*

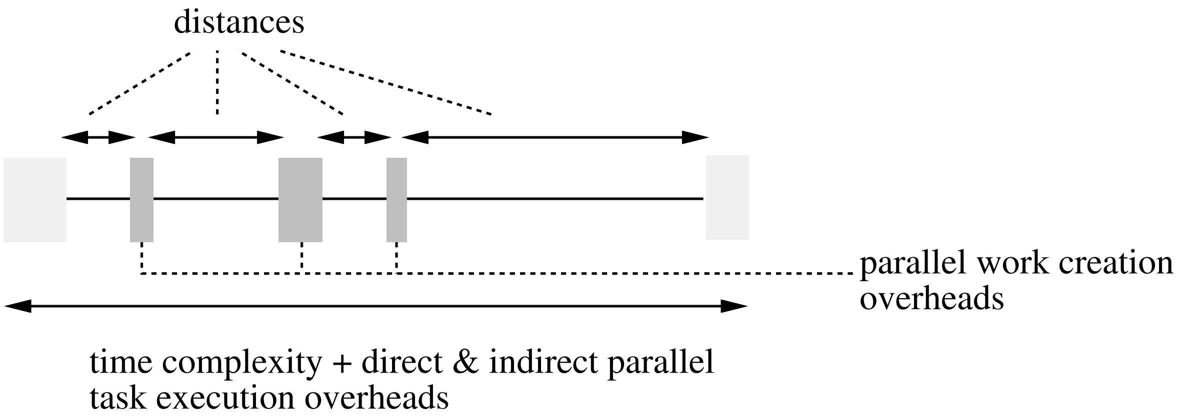

Figure 3: Dynamic Creation Of Parallel Work

The intuition is that once we found the metric for which overheads can be expressed as a simple relationship with the amount of work perform, we can then use this distance metric as a thresholding tactic. To justify this intuition, we present a more formal argument based on a simple cost model. The cost model has an overhead for creating parallelism, called q, and an overhead of scheduling, starting and communicating with a remote process, called f. If n parallel tasks, for example, are created at the same program point, and these tasks have execution times of $t_1, \ldots, t_n$ respectively, then (provided there are sufficient free processors) the overall parallel execution time is

$$q + max(t_1, f + t_2, \ldots, f + t_n)$$

Note that the first task is executed locally and thus does not incur an additional overhead of f. This cost model can be refined because each of the parallel tasks will themselves have (internally) parallel overheads. These overheads stem from q. We shall suppose that these q overheads occur, on average, at a distances of $d_1, \ldots, d_n$ apart in the n tasks. In other words, task i has a cumulative q overhead of qt_i/d_i. Factoring these additional overheads into the overall parallel execution time, the following criteria is obtained. This criteria must be satisfied for worthwhile parallelisation:

$$\sum_{i=1}^{n} t_i \geq max((1 + q/d_1)t_1, r + (1 + q/d_2)t_2, \ldots, r + (1 + q/d_n)t_n)$$

Put simply, the inequality says that the parallel execution should not be slower than sequential execution. Now let $1 \leq i \leq n$. Note that this inequality cannot hold as $d_i \to 0$ and hence there exists $\epsilon > 0$ such that $d_j > \epsilon$ for all j. Put another way, a d_j cannot be arbitrarily small and thus we must ensure that the distance between overheads be above a certain threshold. This justifies thresholding.

There are several important considerations when using distance as a metric for granularity control. Firstly, if the creation of parallel work occurs at irregular intervals directed by the program, then the individual distances between such creation points can vary considerably. We will therefore try to maximise the *average* distance over the whole program (or over parts of the program in which granularity control is applied).

Clearly, one could maximise distance by never creating work. A second principle, that we will call the *load-balancing principle*, is that in order to to achieve the best execution time for a particular configuration, *the amount of parallelism produced should be just sufficient to keep the processors in the system busy, and doing useful work, while average distance is maximised.*

Note that the two goals are conflicting: the less tasks we create, the less parallelism is available, hence the harder it is to keep all processors busy. An optimal configuration will depend on a large number of factors, many of which cannot be known with high precision before runtime (for example, the ideal amount of parallelism can vary for different queries to the same program [11]). In addition, different sources of parallelism (such as different or-parallel nodes in or-parallel Prologs or different and-parallel conjunctions in and-parallel systems) can lead to very different amounts of parallelism. The point is that for a given average distance, the amount of parallelism that is available can differ significantly, depending on which actual sources are removed. In other words, to increase distance we would like to first remove the "worst" sources of parallelism, that is, those that lead to the least amount of parallelism, first.

This ideal is almost certainly impossible to achieve in practice, but a system which seeks to perform close to the ideal would require both run-time and compile-time controls. On the one hand, the "worth" of a particular source of parallelism is only available after the execution of the particular task, not before, when the granularity control system actually needs it. On the other hand, it is impossible to determine at compile-time what the ideal amount of parallelism should be, because the query (and indeed the load and even the configuration of the parallel system) is usually unknown. Thus, a good granularity control system should try to determine at run-time the actual amount of parallelism that should be made available, guided by information generated by compile-time analysis as to the best sources of parallelism to make available.

Such a sophisticated system has not yet been designed. In this work we concentrate on validating the usefulness of distance as a metric for granularity. To do so, we will consider two simple schemes, one compile-time and the other run-time. These strategies simply try to increase the distance between *individual* sources of parallelism by increasing average distance.

The paper is organised as follows. Section 2 discusses the basic concepts of independent and dependent and-parallelism in logic programming systems. Section 3 presents in detail the novel run-time and compile-time granularity control methods, describing their performance for selected benchmarks. Section 4 compares the two methods with each other and with traditional methods. Section 5 analyses some possible improvements and discusses performance of these methods for the DDAS dependent and-parallel system. Lastly, we present some conclusions and suggest future work.

2 And-Parallelism

The previous discussion on granularity applies to any parallel system in which creation of parallel work incurs overheads. For example, in a conventional parallel programming environment, the parallel work creation overheads might correspond to the creation of a (lightweight) process or to sending an array to another processor. With parallel Prolog, for traditional or-parallel systems such as Aurora [13], Muse [1] and PEPSys [2], a task corresponds to the execution of a new alternative, and the overhead for creating parallel work corresponds to the overhead for creating an or-node which allows or-parallelism to be exploited (sometimes referred to as a branch-node). Thus, "distance" in this case is the distance between successive or-parallel nodes.

For concreteness, we shall mainly discuss the distance metric in the context of goal-level independent and-parallel (IAP) systems for now on. Examples of such systems are &-Prolog [7] and &ACE [14]. Work on granularity control has been shown to be particularly effective for IAP systems, and it is indeed for these systems that previous granularity systems have been originally developed. We chose a stable implementation of an IAP system (the IAP component of DASWAM [20]) designed by one of us. The system's stability and portability allowed us to perform extensive and repeated timing experiments on several different multi-processors, thus giving the additional benefit of observing the effect of granularity control on several different machines.

We next give a brief overview of IAP as exploited in DASWAM. We refer the reader to [6, 16] for more details. DASWAM is a prototype parallel implementation of the DDAS [16] parallel Prolog execution model. DDAS is an extension of the IAP scheme first proposed by DeGroot [4] and extended by Hermenegildo [6]. In terms of actual implementations (e.g. [8, 24, 14]), this scheme is possibly the most successful among various IAP scheme proposed. The notation for declaring IAP evolved through several syntactic changes. The syntax used in this paper follows &-Prolog's if-then-else notation, as presented by Hermenegildo and Greene [8].

In IAP schemes, parallelism is exploited by running independent sub-goals in parallel. In strict IAP schemes, sub-goals are considered independent if they do not share variables. Non-strict IAP schemes allow the same variable to occur in the same goal if the goals do not communicate through the variable. In &-Prolog, sources of IAP parallelism are indicated by annotations known as *Conditional Graph Expressions* (CGEs). These annotations can either be generated as part of the compilation process, or be provided by the programmer. Note that the annotations are used only to indicate *where* parallelism should be exploited, not how. The CGEs corresponds directly to the sources of parallelism discussed in the last section.

In a CGE, if two consecutive body goals are to be run in parallel, this is indicated by a '`&`' instead of Prolog's standard '`,`', now reserved for sequential conjunction. For example, in the following quick-sort program fragment,

```
qsort([X|L],R-R0) :-
    partition(L,X,L1,L2),
    (qsort(L1,R-[X|R1]) &
     qsort(L2,R1-R0)
    ).
qsort([],R0-R0).
```

there is a CGE enclosing the two `qsort/2` goals, with the `&` indicating that these two goals can be run in parallel. The `partition/4` goal, which occurs before the CGE, is not run in parallel. It is executed first, as in sequential Prolog, and when it is successful, the two `qsort` goals are run in parallel. These two goals can recursively execute the first clause of `qsort/2`, thus introducing more CGEs and sources of parallelism.

The DDAS model extends the IAP scheme to allow goals with dependencies to execute in parallel. In the previous example, the `partition/4` goal, partitions the incoming list `L` into two lists (`L1`,`L2`), which are then sorted by the two `qsort` goals. The `partition/4` goal shares the variables `L1` and `L2` with the `qsort` goals, and cannot be run in parallel with these two goals in an IAP scheme. The DDAS model extends IAP by exploiting dependent and-parallelism (DAP) [16]. This allows body goals with dependencies to be run in parallel, so in the quick-sort example all three body goals inside the CGE can be run in parallel.

The design of the DDAS model and of its DASWAM implementation is discussed in detail by Shen [17, 18]. In order to describe dependent and-parallelism, DDAS expands the CGE notation to describe dependent goals that can run in parallel and share variables. Such dependencies are indicated via the `dep` annotation. This will cause the execution to synchronise at the level of unification to ensure correct behaviour. The quick-sort example can be annotated as:

```
qsort([X|L],R-R0) :-
    (dep([L1,L2]) ->
       partition(L,X,L1,L2) &
       qsort(L1,R-[X|R1]) &
       qsort(L2,R1-R0)
    ).
qsort([],R0-R0).
```

Note that IAP is an instance of DDAS where the goals run in parallel just so happen to have no dependencies. In particular, the DASWAM implementation behaves very similarly to an IAP implementation if no DAP is exploited. DASWAM can thus be used to study the distance metric in IAP. In this paper, DASWAM is used mostly in such a context, but we also briefly examine the implications of using the distance metric in DAP in section 5.1.

A detailed initial evaluation of the DASWAM system was presented by Shen [19]. On IAP programs, DASWAM (without granularity control) has an overhead between 1.5 to 2.5 over the original sequential WAM implementation the DASWAM was based on.[3] As a result, and even without granularity control, performance improvements were obtained for all programs over sequential execution, including the modestly parallel Pentium Pro machines. As shall be shown, the distance metric enabled us to improve parallel performance further.

3 The New Granularity Control Methods

We next discuss and compare the two basic methods for increasing grain-size as estimated by measuring distances between points of creation of work. The first method uses a simple source-to-source transformation to increase distance between CGEs. The second uses a run-time counter to estimate the distance between CGEs. We believe that these simple methods demonstrate the utility of the distance metric:

- the distance metric can be applied in circumstances where the time-complexity measure breaks down;
- although compile-time analysis will enhance the utility of the distance metric, the simple run-time scheme (which does not need any compile-time support) demonstrates the feasibility of run-time controls, and can reduce the reliance on the precision of compile-time techniques.

3.1 The Compile-Time Method

This method applies at compile-time sequential variants of parallel procedures such that the distance between two parallel CGEs is guaranteed to always exceed a given threshold at run-time.

The algorithm considers every procedure P_0 with parallel CGEs. The transformation is applied to an initial CGE in procedure P_0, receives a minimal distance m, and works over an initially empty list of sub-goals L, representing the sub-goals that may be at a distance less than m:

1. Initialise by setting the list of sub-goals to the calls in the CGE.
2. Terminate if the list L is empty.
3. Extract a sub-goal G from list L and find the defining procedure P. If the distance through G from a CGE in P to the original CGE is smaller than m, then create a new version of P, say P', otherwise go to step 2. To obtain procedure P', transform all CGEs whose distance is smaller than m in sequential conjunctions. Replace the original invocation of sub-goal G by a call to P'. Last, add the sub-goals in the transformed CGEs to the list L and go to step 2.

Note that after applying the algorithm to P_0 one will obtain new procedures P', that must also be considered by the algorithm. The new procedure P' differs from the original procedure P in that it always contain strictly less CGEs than P. The algorithm therefore can only create a finite number of new procedures, and thus must terminate.

In this work we shall use a particularly simple application of this algorithm: the *alternate* transformation. In this transformation the metric we use is the number of parent goals, and the threshold distance is two. In other words: sub-goals of a CGE cannot create CGEs. For recursive procedures this results in an alternation between parallel and sequential procedures.

We give an example application of th is method to the Boyer benchmark as ported to Prolog by Tick [23]. Note that the Boyer program has significant amounts of (non-strict) IAP. Most of the parallelism arises from parallel execution of term rewrite rule, that is a part of the `rewrite_args` procedure:

[3] Many of the programs studied here have very low granularity and unusually high parallel overheads, thus making them suitable for granularity control. Most typical programs will have lower parallel overheads.

```
rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        N1 is N-1,
        (rewrite(OldArg,MidArg0)
          &
         rewrite_args(N1,Old,Mid)),
        MidArg0 = MidArg.
```

Experience has shown that this program has very small granularity by the distance measure, because successive CGEs occur very close together at run-time.[4] This results in a very significant overhead for running Boyer with the parallel annotation on a single worker versus running it without annotations – on the DASWAM running on the Sequent, it is 43% slower, the largest such overhead for any IAP program examined in [20]. Moreover, the program is not suitable for the granularity control based on time-complexity analysis as described in [12, 11], because of the difficulties of deriving a relationship between input arguments and the complexity. The distance metric, however, suggests a very simple way to increase the distance, by performing a source-to-source transformation on the program such that and-parallelism is only generated for every second call to `rewrite_args/3`:[5]

```
rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        N1 is N-1,
        (rewrite1(OldArg,MidArg0)
         &
         rewrite_args1(N1,Old,Mid)),
        MidArg0 = MidArg.

rewrite_args1(0,_,_) :- !.
rewrite_args1(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        N1 is N-1,
        rewrite(OldArg,MidArg),
        rewrite_args(N1,Old,Mid).
```

This simple transformation reduces the number of CGEs allocated by half, regardless of what query is run. We shall discuss in the next section in some more detail how to generalise this transformation.

3.1.1 Evaluation

To evaluate the impact of the alternating method we experimented with the original and the transformed program in several parallel machines. The results are shown in Table 1. The '#w' column shows the number of workers for each row. We have results for 'ideal', giving the 'ideal' amount of parallelism as obtained from the DASWAM simulator [20]; 'seq', giving results for a 10 processor Sequent Symmetry with 25MHz 80386, running Dynix; 'sun', giving results for a 10 processor SPARCcenter-2000 running Solaris; 'chal' for a 10 processor Silicon

[4] In this case, the distance can be very simply determined by tracing a possible execution and counting the number of unifications between CGEs.

[5] The modification to `rewrite1/2` is not shown to avoid cluttering the presentation; but much the same transformation is performed there.

Graphics Challenge running IRIX; and 'pc' for a 4 processor PC with 200MHz Pentium Pros, running Linux. The '(g)' column indicates cases with granularity control, and the '(no-g)' without. All times are in seconds. The timings measure the duration from the start of query execution to the production of the (only) solution, and are the best of at least 5 executions. DASWAM is compiled with `gcc` (except for the Challenge, where it was compiled with MIPS' `cc`, because that resulted in faster code), with optimisation turned on. Note the extremely low speedup obtained for 2 workers on the Challenge for the granularity controlled case: this problem seems to be isolated to 2 workers on some programs (note for example the problem does not occur for the non-controlled case) on this particular machine; another SGI Challenge that we have access to (with 4 slower processors) does not appear to have this problem. Note also that the SPARCcenter has a performance problem in that the cache for some of the processors are not correctly implemented, which effectively renders these processors slower than the rest. Thus results obtained here (and throughout the rest of this paper) with higher number of processors can be expected to be lower than the predicted speedups.

#w	ideal		seq		sun		chal		pc	
	(no-g)	(g)	(no-g)	(g)	(no-g)	(g)	(no-g)	(g)	(no-g)	(g)
1	–	–	97.092 (1.00×)	81.409 (1.00×)	10.951 (1.00×)	9.034 (1.00×)	6.189 (1.00×)	5.051 (1.00×)	2.494 (1.00×)	2.025 (1.00×)
2	1.99×	1.98×	49.178 (1.97×)	41.039 (1.98×)	6.039 (1.81×)	4.787 (1.89×)	3.588 (1.73×)	4.389 (1.15×)	1.354 (1.84×)	1.074 (1.89×)
3	2.98×	2.93×	32.624 (2.98×)	27.970 (2.91×)	4.329 (2.53×)	3.466 (2.61×)	2.514 (2.46×)	1.955 (2.58×)	0.971 (2.57×)	0.777 (2.63×)
4	3.95×	3.85×	24.624 (3.94×)	21.316 (3.82×)	3.276 (3.34×)	2.862 (3.16×)	1.959 (3.16×)	1.547 (3.27×)	0.799 (3.12×)	0.608 (3.33×)
5	4.92×	4.74×	19.954 (4.86×)	17.427 (4.67×)	2.804 (3.91×)	2.251 (4.01×)	1.641 (3.77×)	1.312 (3.85×)		
6	5.87×	5.58×	16.697 (5.81×)	14.863 (5.48×)	2.426 (4.51×)	1.978 (4.57×)	1.436 (4.31×)	1.151 (4.39×)		
7	6.82×	6.39×	14.474 (6.71×)	13.095 (6.22×)	2.088 (5.25×)	1.811 (4.99×)	1.285 (4.82×)	1.064 (4.75×)		
8	7.76×	7.16×	12.846 (7.56×)	11.640 (6.99×)	1.997 (5.48×)	1.660 (5.44×)	1.187 (5.21×)	0.976 (5.18×)		
9	8.69×	7.86×	11.537 (8.42×)	10.661 (7.64×)	1.823 (6.01×)	1.574 (5.74×)	1.112 (5.57×)	0.928 (5.44×)		
10	9.60×	8.58×	10.528 (9.22×)	9.929 (8.20×)	1.732 (6.32×)	1.537 (5.88×)	1.020 (6.07×)	0.894 (5.65×)		

Table 1: Compile-time granularity control results for Boyer

Granularity control is effective if it gives better performances than the non-granularity controlled program. The ideal columns show that, as expected, granularity control reduces the amount of available parallelism. Thus, in the absence of overheads, performance should be better without granularity control. Instead, the results show that this form of granularity control gives better execution times, and is indeed effective in all cases. The main component that leads to the increased performance is clearly the removal of parallel creation overheads, as shown by the difference in performances for the one worker cases. Even so, and for all systems except the Sequent, the *speedup* (which is relative to their respective 1 worker case) is better for the lower number of workers for the granularity controlled cases. Even for the Sequent, the agreement with the ideal parallelism is better for the controlled case. The good speedups show that the granularity control was able not only to remove the parallel creation overheads (cost of allocating CGEs), but also some direct parallel task execution overheads, especially for the faster machines.

The major weakness of this method is that the reduction in parallelism can become quite significant with larger number of workers. It should be possible to achieve better performance if a more selective way to remove CGEs is found, but this would need some form of analysis (at compile-time) to determine the 'worth' of the CGEs. Additionally, this purely compile-time transformation does not take into account run-time situations: for example, with smaller number of workers, better results can be achieved by producing less CGEs.

The example for Boyer transformed the program to create and not create CGEs alternately, thus reducing the number of CGEs generated at run-time by about half and duplicating average distance. One advantage of this method is that, in many cases, the distance between CGEs can be easier to determine than the time-complexity of a goal: e.g. if no recursive goals are executed sequentially between successive CGEs, as is the case for every example program in [11] except quick-sort, then the distance can simply be calculated by counting the number of resolutions between successive CGEs, and this can be performed at compile-time. As we are interested in *average* distances, then even for quick-sort, we can see that the partition goal, which is run sequentially between successive CGEs will generally have a relatively high complexity, and thus make average distances large (it will only be small when the input list is very small). In general, we would expect that this method could be improved by compile time analysis. Such work could be used to estimate average distance between CGEs and thus approximate the number of CGEs that should be sequentialised in order to improve distance up to a desired factor. The number of sequentialised CGEs can be controlled to some extent by changing the way the program is transformed. However, in this work, our interest is to study the effectiveness of distance as a granularity control metric, and not to propose the most effective control methods based on the metric. Thus, to start with we would like to examine how well we can perform with the simplest compile-time heuristic, and we will not consider the issue of the actual compile-time transformation in more detail, except to note that the simple transformation presented here can be readily improved upon.

3.2 The Run-Time Method

The second granularity control method uses a simple counter in order to decide whether a CGE should be allocated or not. The counter measures the work done since the last CGE was allocated. If its value is below a pre-determined threshold, then the new CGE would not be allocated, and the goals in the CGE will be run sequentially instead. Conceptually, this method can be thought of as:

```
(above_threshold(Counter) ->   reset(Counter), g1(...) & ... gn(...)
                             ; g1(...), ... gn(...))
```

In other words, if the condition `above_threshold` is fulfilled, reset the counter back to zero and execute in parallel. Otherwise, execute the goals sequentially. We first discuss implementation issues, and then give a brief performance analysis.

3.2.1 Implementation Issues

We implemented the above conceptual model at the abstract machine level in order to minimise overheads. Within the abstract machine, the natural unit to count is the number of abstract machine instructions executed. In addition, the system takes advantage of the fact that, in DASWAM, the code for making calls within a CGE is very similar to the sequential code[6]. We generate the same code for the parallel and sequential execution, thus removing a potential source of overhead. The parallelised code differs from traditional WAM code mainly in the addition of `allocate_pcall`, `push_goal`, and `par_proceed` instructions. Briefly, the `allocate_pcall` instruction creates a new CGE, the `push_goal` instruction creates the parallel goals in the new CGE, and `par_proceed` is executed when a goal in a parallel execution completes. It informs the parallel scheduler the goal has completed and performs the necessary operations on the CGE.

One method to sequentialise CGE execution is through Prolog source level constructs such as if-then-else as presented above. Instead, in order to accomplish this with minimal impact on performance, we added two new abstract machine instructions, `skip_par_if` and `gran_par_proceed`. The instructions are designed to allow the same piece of code to be executed both for parallel and sequential CGEs. We demonstrate their application through the granularity controlled code for the CGE in Boyer's program (see section 3.1):

```
(rewrite(OldArg,MidArg0) &
 rewrite_args(N1,Old,Mid))
```

[6]See [15, 19], for details of DASWAM's instruction set.

```
      skip_par_if(label(_1),5,2)           ;CGE has 2 and-goals.
      allocate_pcall(2,1,6)
      push_goal(label(_1388),1,0,0)
label(_1):                                 ;skip here if sequentialised
      put_y_value(5,0)
      put_y_value(0,1)
      call(rewrite/2,5)                    ;rewrite_args(OldArg,MidArg0)
      gran_par_proceed(label(_1386),0)     ;no-op if sequentialised
label(_1388):
      put_y_value(2,0)
      put_y_value(3,1)
      put_y_value(4,2)
      call(rewrite_args/3,2)               ;rewrite_args(N1,Old,Mid))
      gran_par_proceed(label(_1386),1)     ;no-op if sequentialised
label(_1386):
      put_y_value(0,0)
      get_y_value(1,0)
      deallocate
      execute(true/0)
```

The two new instructions are:

- `skip_par_if(Label, Threshold, N)`: this instruction causes execution to branch to `Label` if the counter is below `Threshold`, so that the following code will be executed sequentially. The `Label` should follow the `allocate_pcall` and `push_goal` instructions. `N` is the number of and-goals in the CGE, and is used to determine the number of `gran_par_proceed` instructions to skip for the execution of code in this (sequentialised) CGE.

- `gran_par_proceed(Label,N)`: This instruction replaces the `par_proceed` instruction, and has exactly the same arguments. The difference is that this instruction first checks whether it is inside a sequentialised CGE or not, and if so, skips the execution of the rest of the instruction. The extra functionality of this instruction could be incorporated into `par_proceed` but, we preferred to keep two instructions in order not to incur the overhead of determining if the CGE is sequentialised or not in non-granularity controlled code.

The following method is used to determine if a *gran_par_proceed* is in a sequentialised CGE[7] or not. First, each and-goal in the parallel conjunction implements a new counter (`PS`)[8] which is initialised to 0. The counter is stored in the Parcall marker, the data-structure that describes a new and-parallel task [18]. If the `skip_par_if` instruction decides that the CGE immediately following is to be sequentialised, then the number of and-goals in that CGE (`N`) is added to the `PS` counter of the current (i.e. the last allocated) CGE slot that represents the current task. In this sequentialised execution, no new Parcall marker will be allocated, and the procedure calls in the sequentialised CGE still have the previous Parcall marker as the current Parcall marker. Later, when the execution returns from the call and enters the `gran_par_proceed` instruction, the same Parcall marker will again be current. The PS counter is checked to see if it is zero. If so, the CGE has not been sequentialised, and the normal `parallel_proceed` actions take place. Otherwise, the counter is decremented, and the next instruction is executed. This method allows for multiple successive CGEs to be sequentialised, as the `PS` counter is simply incremented by the appropriate amount each time.

[7]Recall that this notion of sequentialisation is different from the traditional granularity control: it only applies to the immediate CGE. Subsequent CGEs can still be executed in parallel.

[8]In fact in the actual implementation, the counter reuses a location in the slot that would not be used until the CGE has completed, thus introducing no extra memory overheads.

The instruction threshold counter is implemented as a new worker register. The current DASWAM implementation already keeps the count of the total number of instructions executed by a worker in a DASWAM register, hence this new register needs only record the instruction count at the last parallel CGE or, if a new task is started on a worker, the instruction count at the start of work. The value of the register is then used together with the instruction count to calculate if a CGE should be sequentialised or not.

3.2.2 Overheads

The run-time scheme introduces some overhead to the execution. One always has to update the threshold counter at each CGE, and to initialise the PS counters for each slot in the CGE. When granularity control is used, the extra cost also includes the execution of the `skip_par_if` instruction, and the check performed at each `gran_par_proceed` instruction. We would expect these overheads to be reasonably small.

prog.	ng_orig	ng	g(0)
Boyer	6195.6	6239.2	6331.8
fib(23)	1917.2	1859.0	1956.6
hanoi(16)	2839.6	2661.0	2773.4
Orsim	7676.6	7747.0	7758.4

Table 2: Comparison of Execution times (ms) for Scheme

To study the impact of the run-time scheme's overheads, we compared the execution times for several programs on the original and the run-time schemes. The programs chosen for this study are benchmarks taken from [11], except for Boyer and Orsim. Orsim is an application program developed by one of us [22, 21] to study or-parallelism. All the programs except for Orsim have small distances between CGEs (see section 4), and so the run-time scheme's overheads should show up most clearly. The results are shown in Table 2, where ng_orig column represents the single processor execution times for the original scheme; ng represents the timings for a scheme where dynamic granularity control enabled but not actually used in CGE code; and g(0) represents timings under a scheme where dynamic granularity control enabled code being generated, but not actually sequentialising CGEs. The latter is obtained by the use of a zero threshold so that all CGEs are parallelised. Note that this represents the maximum overheads that would be incurred by the scheme. Execution times were obtained on a SGI Challenge. DASWAM was compiled with the MIPS C compiler, under the same optimisation level as before (`-O2`). The timings are the mean of 5 executions.

The overheads are quite reasonable. In fact, other factors, such as what other tasks the machines was performing while these results where obtained, may have more significant effect on the performance, such that the performance of g(0) is actually measured to be slightly faster than ng_orig in hanoi(16). The biggest overhead is just 2.2% for g(0) over ng_orig for Boyer, suggesting that, even for programs where an unusually high overhead for the run-time scheme is expected, it is still not significant.

Another possible source of overhead of this scheme is that the code for parallel conjunctions may not be as optimised as the equivalent sequential code. There are differences in terms of register allocation, and of instructions semantics, as unlike the sequential code, the parallel code cannot assume that an environment variable has been initialised by a previous call in the CGE, because it cannot be assumed that the call would be executed first. The most important difference is the loss of last call optimisation. This is most severe in non-strict IAP code like Boyer, where parallelisation forces an additional unification at the end of the CGE but, the programs in Table 2 are also highly tail recursive, and thus the results suggest that this source overhead is still relatively small.

One interesting application of this new scheme is in executing sequentialised CGEs in other situations, such as when a CGE condition fails. The scheme avoids the cost of the relatively more expensive if-then-else construct, for which most compilers need to generate extra procedure calls for the then and else parts.

Lastly, one should note that in parallel execution the number of CGEs that will be sequentialised is non-determinate. This is due to the fact that the threshold is counted from the start of a new and-goal. Given that which goals are selected first to execute in parallel is non-determinate, the calculations of distance can vary, and the number of sequentialised CGEs can vary. Our results suggest that this variation is relatively small, and that as more tasks are actually executed in parallel, the number of sequentialised CGE would increase slightly.

3.2.3 Performance

Table 3 shows the results of using the runtime method on Boyer, with the threshold set at 64, 80, 96, 112 and 128 abstract machine instructions. These particular thresholds were chosen because the implementation of `skip_par_if` allowed the threshold to be set in increments of 16 units only. Again, the results are the best timings of at least 5 runs. The system was run on the 10 processor SGI Challenge. The numbers in brackets in the 1 worker case are the number of CGEs in each of the 1 worker case. The bracketed numbers in the other cases are the speedups relative to their respective 1 worker case.

#w	no_gran	64	80	96	112	128
1	6189 (94056)	5456 (48511)	5370 (44104)	5040 (32230)	4961 (32092)	4868 (29127)
3	2514 (2.46×)	2158 (2.53×)	2099 (2.56×)	1920 (2.63×)	1887 (2.63×)	1902 (2.56×)
4	1959 (3.16×)	1717 (3.18×)	1651 (3.25×)	1514 (3.33×)	1548 (3.20×)	1664 (2.93×)
5	1641 (3.77×)	1492 (3.66×)	1444 (3.72×)	1301 (3.87×)	1340 (3.70×)	1406 (3.46×)
6	1436 (4.31×)	1347 (4.05×)	1292 (4.16×)	1168 (4.32×)	1185 (4.19×)	1286 (3.79×)
7	1285 (4.82×)	1242 (4.39×)	1190 (4.51×)	1067 (4.72×)	1083 (4.58×)	1204 (4.04×)

Table 3: Run-time granularity control for Boyer

As the machine was loaded at the time the experiments were performed, no results were gathered beyond 7 workers, because the system did not have the resources to allow DASWAM to utilise these processors. Indeed, it is not clear how much impact the load has on the presented results. The results for no_gran, the case with no granularity, is taken from the experiment with the compile-time method presented in Table 1, when the machine was not busy. The results for 2 workers are not included because of the same problem as discussed previously.

Even on a loaded machine, the results shows that the granularity control can improve the performance. Base speed (performance on one worker) improves, as one would expect. Speedups are quite close to the one processor version. Interestingly, the improvements becomes greater with higher thresholds until 112, probably indicating that direct parallel execution overheads were being reduced[9]. At even higher thresholds, the reduction in overheads cannot compensate for the lost in parallelism, especially for larger number of workers.

#w	no_gran	96
1	7893 (2100)	8034 (2100)
2	4062 (1.94×)	4074 (1.97×)
4	2035 (3.88×)	2063 (3.89×)
8	1053 (7.50×)	1060 (7.58×)
10	835 (9.45×)	859 (9.35×)

Table 4: Run-time granularity control for Orsim

[9] The speedups are better for the no granularity controlled case probably because they were obtained when the machine was not loaded.

Table 4 shows a worst-case program for traditional granularity analysis. The program is Orsim [22, 21]. This is quite a complex program that is very difficult to analyse. The program has quite high granularity, hence does not need granularity control. We obtained the results on a the SGI Challenge when it was lightly loaded. Times are in milliseconds, and are again the best of at least 5 timings. One advantage of the run-time method is that it does not require any analysis, and the overhead of using it is very small, so that in this case where no CGEs are sequentialised, the performance of the program with granularity control is quite close to the original performance.

4 Discussion

The average distance between CGEs is a good metric to explain the effectiveness of granularity analysis on Boyer, and other programs as studied in [11]. Table 5 shows the average distances for these programs with no granularity control as a function of abstract machine instructions: i.e. the average number of instructions executed per CGE.

prog	Σinst	ΣCGE	av. distance
Boyer	5106759	94056	**54.3**
hanoi(10)	47083	1023	**46.0**
hanoi(16)	3014635	65535	**46.0**
Fib(17)	95593	2583	**37.0**
Fib(19)	250290	6764	**37.0**
qs(300)	55452	300	**184.9**
qs(3200)	823168	3200	**257.2**
Orsim	9069226	1200	**7557.7**

Table 5: Average distances

Table 5 clearly show that all the programs except quick-sort and Orsim have very small average distances: there are only about 3 to 5 unifications between each CGE (using the approximation that each unification is approximately made up from about 10 abstract machine instructions). We would expect that granularity control in these cases would be very effective in reducing the parallel overheads. On the other hand, we would expect that granularity control to be less effective for quick-sort. This is indeed the case from experimental results [12, 11].

The reason that quick-sort, running in IAP, has a large average distance is because the partition predicate is executed between successive CGEs:

```
qsort([], Rest, Rest).
qsort([X|Unsorted], Sorted, Rest) :-
   partition(Unsorted, X, Smaller, Larger),
   (qsort(Smaller, Sorted, [X|Sorted1]) &
    qsort(Larger, Sorted2, Rest)),
   Sorted2 = Sorted1.
```

The partition predicate is recursive and does not contain any CGEs. Analysis should be able to determine that it is likely that this predicate will perform a significant amount of work, thus making the distance between CGEs large on average. Such an analysis would decide not to add compile-time granularity control in this case.

It is interesting to compare the new methods with complexity-based approaches. Tables 6, 7 and 8 show the effect of the using granularity controls on qs(3200), hanoi(16), and fib(23) respectively. The programs are taken from [11], with 23 used as the parameter for Fib to get results that would take over a second to execute on the Challenge. The results are again in milliseconds, the best of at least 5 executions. The qs(3200) query was performed on the 10 processor Sequent, while hanoi(16) and fib(23) were performed on the SGI Challenge. We present the compile-time and run-time control methods, along with the traditional complexity threshold methods. The results for the complexity threshold method of qs(3200) are taken from [11], and are average timings, rather than the best times, but in the context of these experiments, the differences from best times are small. For the run-time control method, thresholds of 64, 80 and 96 abstract machine instructions were used, with the addition of a 112 threshold for fib(23). These are shown in the column with the threshold as the headings. The 'compile-time' columns are for the program transformed to call CGE only on alternate recursive calls, which reduces the CGEs by half in qs(3200) and fib(23), and by two-thirds for hanoi(16). The reduction is two-thirds for hanoi(16) because many of the calls terminates in the base case, and generate no CGEs. The results for the complexity method are show in the columns with headings of size(X), where X is the size threshold, measured in resolution steps.

The programs have quite different characteristics. As explained before, qs(3200) has a high average distance, and relatively low parallelism. The hanoi(16) and fib(23) queries have very low average distances and significant parallelism. The size of the two goals that are run in parallel in a CGE are identical for hanoi(16), but are slightly less well balance for fib(23), where one goal is about 60% larger than the other. Another difference is that CGEs occurs regularly and at constant distances for hanoi(16) and fib(23), but irregularly for quick-sort. Note that although the compile-time transformation was applied to quick-sort, in an actual system, no control would be applied because as discussed in section 3.1, the analysis would have concluded that the average distance is sufficiently large and the query does not need granularity control. The results are presented here mainly to show the effect of an inappropriate application of the method.

We shall first discuss and compare the results obtained from the two new methods, and then compare them to the more traditional complexity-based approach.

#w	no_gran	64	80	96	compile-time	size(64)	size(256)
1	13486 (3200)	13377 (2462)	13298 (2093)	13232 (1724)	13035 (1601)	14084 (2093)	13197 (557)
2	7453 (1.81×)	7399 (1.81×)	7376 (1.80×)	7364 (1.80×)	8151 (1.60×)	7701 (1.83×)	7338 (2.77×)
4	4818 (2.80×)	4767 (2.83×)	4762 (2.79×)	4818 (2.75×)	7091 (1.83×)	4926 (2.86×)	4772 (2.77×)
9	3722 (3.62×)	3699 (3.62×)	3703 (3.59×)	3698 (3.58×)	6988 (1.87×)	3767 (3.74×)	3742 (3.53×)

Table 6: Granularity control for qs(3200) on Sequent

#w	no_gran	64	80	96	compile-time	size(16)	size(256)
1	2745 (65535)	2325 (34952)	2190 (26760)	2073 (21536)	1857 (21845)	1646 (8191)	1374 (256)
2	1671 (1.64×)	1276 (1.82×)	1176 (1.86×)	1097 (1.89×)	1012 (1.83×)	836 (1.97×)	676 (2.03×)
4	939 (2.92×)	703 (3.31×)	637 (3.44×)	591 (3.51×)	542 (3.43×)	434 (3.79×)	338 (4.07×)
9	567 (4.84×)	376 (6.18×)	337 (6.50×)	313 (6.62×)	278 (6.68×)	201 (8.19×)	157 (8.75×)

Table 7: Granularity control for hanoi(16) on SGI Challenge

#w	no_gran	64	80	96	112	compile-time	size(16)	size(1024)
1	1939 (46367)	1578 (20735)	1355 (14674)	1303 (13928)	1255 (11577)	1586 (23184)	1379 (10945)	945 (232)
2	1190 (1.63×)	820 (1.92×)	732 (1.85×)	710 (1.84×)	685 (1.83×)	870 (1.82×)	722 (1.91×)	474 (1.99×)
4	671 (2.89×)	446 (3.54×)	391 (3.47×)	387 (3.37×)	364 (3.45×)	463 (3.43×)	376 (3.67×)	235 (4.02×)
9	405 (4.79×)	254 (6.21×)	265 (5.11×)	282 (4.62×)	298 (4.21×)	251 (6.32×)	179 (7.70×)	107 (8.83×)

Table 8: Granularity control for fib(23) on SGI Challenge

4.1 Run-Time versus Compile-Time Methods

Table 6 shows the results for quick-sort. The run-time control removes some CGEs resulting in slight improvement in execution times for one worker. These differences are not very significant, because the original average distance was relatively large. In addition, and because in this benchmark CGEs occurs between recursive partitions of the list, each successive CGE distance must be smaller than the last (because smaller and smaller lists are partitioned). Sequentialising CGEs with small distances therefore removes those CGEs that result in small tasks, and does not significantly affect the available parallelism, and so the speedups were only reduced very slightly. When compared to no_gran, the slight reduction of speedups with run-time control tends to cancel out the slight decrease in overhead, but nevertheless, small improvements were still being obtained up to the 9 workers.

On the other hand, the compile-time approach shows that an inappropriate application of the CGE removal transformation can have negative impact on the performance[10]. In fact, the speedup was greatly reduced, and the parallel performance is significantly worse than the other cases. Another point illustrated by the results is the importance of removing the right CGEs. Whereas the number of CGEs for the 96 run-time threshold case is only slightly more than the compile-time transformation case, the speedup obtained from the 96 threshold case is significantly better, because only the CGEs that produced the least parallelism were removed. A final point to note about the results is that the one worker case for the compile-time transformation are noticeably better than the other cases. We believe this is because the compile-time transformation carries no run-time overheads as the parallelism is removed at compile-time.

Table 7 shows results for the hanoi(16) query. Both compile-time and run-time methods were effective in increasing execution efficiency and, in fact, their improvements are quite significant. Part of this can be attributed to the use of the Challenge, which is a much faster machine than the Sequent, and therefore where the parallel execution overheads are more significant.[11] The compile-time method is more effective in this case, because again there is less overhead to pay for using the compile-time control.

The results for the Fib benchmark, as shown in Table 8, are similar to the ones for Hanoi, although the increase in performance is somewhat less, and the reduction in speedups is more marked, especially when the distance threshold is set high. The result is that the best run-time method performance for 9 workers is obtained for the lowest threshold value of 64, although the performance is better in the higher thresholds for smaller number of workers. The compile-time method removes slightly less CGEs than the 64 threshold case. Performance in terms of speedups and actual exaction times are similar, although again the performance on one worker is slightly better in the compile-time method, even though it retains somewhat more parallel CGEs. At least in this case where CGEs occurs regularly, neither the compile-time or run-time methods are better at removing the "right" CGEs.

4.1.1 Determining suitable thresholds

As already stated, our interest in this paper is to use the two methods presented here as a guide to further developments of methods based on the distance metric, and as such we are not interested in developing these two methods as actual fielded methods. However, if we were to field either methods, we would need to determine the appropriate threshold for a particular system, at a particular configuration (machine and number of workers used). More systematic results would be needed for a particular system than what we presented here, but it can be seen already that the thresholds would be about 100 abstract machine instructions (or about 10 unifications) for most configurations – less for Sequent because it has relatively less significant parallel overheads; and larger when a small number of workers are used in any machine.

[10]While we do not believe any analyser that is developed would make the incorrect decision in the quick-sort case, we cannot be sure it will always perform correctly, especially in more complex programs.

[11]This can be seen by the lower speedups obtained by programs on this machine compared to the Sequent, especially for low granularity programs. The speedup differences are much less marked for large granularity programs such as Orsim, or the granularity controlled executions.

4.2 Distance-Based versus Complexity-Based Methods

We have seen choosing the "right" CGEs can have very profound impact on how many CGEs that can be removed without decreasing parallel speedups. The results obtained for the time-complexity granularity method illustrates this point even more dramatically: there has been a great reduction in the number of parallel CGEs in all three programs (but most especially for Hanoi), but the resulting speedups are not greatly affected. Our previous results [11] on using complexity analysis for granularity control illustrates this point for a larger set of programs and query sizes.

size	fib(17)	fib(19)	hanoi(10)	hanoi(16)	qs(300)	qs(3000)
0	2583 (8.37×)	6764 (8.66×)	47083 (8.28×)	65535 (8.93×)	300 (3.34×)	3200 (3.62×)
16	986 (8.40×)	4180 (8.74×)	127 (8.17×)	8191 (8.95×)	124 (3.32×)	2093 (3.74×)
64	376 (8.53×)	986 (8.72×)	31 (7.69×)	2047 (8.94×)	63 (3.20×)	557 (3.53×)
256	88 (8.38×)	232 (8.65×)	7 (7.63×)	511 (8.98×)		
1024	20 (6.79×)	54 (8.07×)	1 (2.02×)	127 (8.59×)		

Table 9: Number of CGEs with complexity granularity control

Table 9 shows the number of CGEs remaining after performing the granularity control with the various granularity thresholds for the various programs. The mean speedup obtained for 9 workers on a Sequent Symmetry is shown in brackets. The results again show that although the number of CGEs was drastically reduced, in most cases the available parallelism is not greatly reduced. This is because the CGEs that are removed are the ones that do not contribute much extra parallelism at the relatively small number of workers used. However, results for the hanoi(10) query show that this is not always the case: if the parallelism is constrained to be less than what the processor resource can exploit, there can be a dramatic effect on the speedup. This again illustrates the importance of run-time considerations. As the query a program will run with cannot in general be determined at compile-time, run-time information is needed to ensure good performances in these cases.

The reason for the much better performance of the complexity-based method than the two distance methods for the Hanoi and Fib benchmarks can be attributed to the great reduction in CGEs with granularity control. These two programs are ideal for the complexity method because the and-goals in the CGEs are highly recursive and the two goals in each CGE would both lead to significant parallelism if they are sufficiently large. In addition, the distances between successive CGEs are very small. Thus, using even small complexity thresholds, a very large number of CGEs are removed, and those are precisely the CGEs which lead to least parallelism. Very few CGEs are needed to keep the workers busy, because the parallel tasks are large and well balanced in size.

These conditions are arguably not usual for application programs, such as Orsim, where the average distance is very large, and the parallel tasks are not as well balanced. Another example where the average distance is relatively large is Quick-sort. In this case, although the *proportion* of CGE removed is about the same as Fib, this does not result in great reduction of overheads *with respect to* the time spent on performing useful work. Thus, the complexity granularity control is not very effective in this case. Quick-sort still has relatively well balanced amount of work performed by the goals, and again the CGEs that lead to the least parallelism are removed. With programs which are less balanced, the complexity-based control can be even less suitable. Consider:

```
a(0) :- !.
a(X) :- X1 is X - 1,
        (a(X1) & b).

b.
```

In this example, the distance metric would show that the program has very low granularity (and probably should not be parallelised), but the complexity method could execute `a(X)` in parallel with `b` for sufficiently large `X`, even though this would probably not result in great speedups. It is possible, in the complexity method, to measure the complexities of both goals (as proposed in [12]), and only execute in parallel if both goals are above a threshold. Unfortunately, this solution still has problems. First, our example is an extreme case of a situation where parallel tasks may have very different amounts of work. In general, the removal of CGEs in such cases will be less ideal than for Hanoi and Fib. Hence, more CGEs will be needed to keep the load balanced, resulting in less dramatic improvements. Second, the run-time cost of doing multiple thresholding tests per CGE will be even more expensive than performing a single threshold test. The parallelisation conditions will also be more complex, especially with more than two goals in a CGE.

In general, consider equation 2 again:

$$H \simeq kW + f + Nq \tag{3}$$

An effect of thresholding is to prevent the creation of CGEs below the threshold, and this would generally have the effect of reducing N, thus reducing the parallel overhead H. However, as discussed previously, the number and the actual temporal distribution of subtask creation events (Nq, creation of CGEs) are highly program dependent, using W alone for thresholding does not give very good control over H, the effect is highly program dependent, and thus choosing a universal threshold for a particular system is very difficult.

The run-time overheads for performing complexity granularity control can also be a factor, like in Quick-sort. Here the run-time distance control method performs better because the overheads are lower – testing the size threshold of the input list in the complexity method is much more expensive. Even for the Hanoi and Fib programs, where the system just checks a numeric value against the threshold, the cost for the complexity-based method is still higher than either of the two new methods. The point is brought home clearly comparing the 112 run-time case and the 16 complexity case for fib(23). Both remove about the same number of CGEs, but the execution time for the run-time method case is faster for 1 worker.

A different problem with using the complexity threshold is that it relies on being able to derive a relationship between the input arguments' sizes and the resulting complexity. Such a relationship does not always exist, as in Boyer, and in cases where it does, it may not always be easy to derive. Even where it can be estimated, we have seen it might require relatively large run-time overheads to determine the size parameter, as in the case for quick-sort, where the length of the input list is the norm, and thus determining the size would require some sort of run-time test on the length of lists, which is relatively expensive.

5 Improvements and Extensions

The quick-sort example showed a situation where the compile-time method (when applied blindly) performed worse than the run-time method because it removed "good" CGEs. There are cases where the run-time method can also remove "good" CGEs. As an example, consider the case where, early in the execution, a program creates several pieces of parallel work of significant size in quick succession, and then stops creating parallel work. In this case (which does occur in the bt_cluster benchmark first used by Andorra-I [25]), the actual distances between each CGE will be short, but the average distance (defined as the ratio of total work per number of CGEs) would be much larger. The CGEs should therefore not be sequentialised.

This situation illustrates a weakness of any purely run-time scheme. These methods cannot look ahead into what may happen further into the execution. In this case, when and where new CGEs will be created, and how much parallelism they might provide. Of course, more sophisticated systems than the very simple counter method can be designed, for example, systems which takes into account if the workers are currently busy or not; and heuristics based on execution history to try and anticipate future behaviour [5]. For example, a more sophisticated run-time scheme can only start to sequentialise CGEs when all the workers are busy *and* the distance is below the threshold. However, and as we have seen even for the simple scheme presented here, any run-time scheme does carry some overheads: the more sophisticated the scheme, the more expensive the overhead. In the end, run-time systems will always suffer from not having specific information about the execution still to come. This information can only be provided by compile-time analysis. On the other hand, a compile-time only method cannot take into account run-time conditions such as the load of the machine, and furthermore there will probably always be programs that cannot be analysed.

The ideal distance also needs to be adjusted for different parallel systems and machines, as system architecture would affect the relative importance of various sources of overheads. In fact, distributed network systems, where Debray et al. [3] have recently shown some promising results with complexity-based granularity control, would require larger average distances than shared-memory machines in order to obtain performance improvements. Moreover, the penalty for parallelising the "wrong" CGE may be very high, suggesting that they may require more sophisticated distance granularity control schemes than the two we presented here. We believe that the notion of distance, under the appropriate granularity control scheme, will be a suitable method for granularity control: equation 2 still holds for such systems – between points where distance is measured, the overheads, to a first approximation, still have a simple relationship to the amount of work performed.

5.1 Dependent And-parallelism

As already stated, the distance metric can be readily applied to or-parallelism, and indeed to any parallel system where the parallel work creation introduces overheads. However, in cases where the parallel execution entails additional overheads to those considered so far in this paper, the situation may be more complex. In some cases, they can simply be taken into account as additional sources of indirect parallel execution overheads, and thus taken into account when calculating the distance.

Consider the case of dependent and-parallelism (DAP), specifically DDAS [15] as exploited by DASWAM. In this implementation, costs are associated with variables being dependent and with suspensions, among others. These can be expected to have different implications on the distance metric: suspension can be considered as an additional source of indirect parallel overhead, and thus distance should be measured to when a suspension occurs. However, suspension is dynamic and varies from run to run of a program even with the same configuration, thus a statistical approach would probably be needed to adequately account for it. As for dependent variables, which can be accessed throughout the parallel execution, should be considered as part of the direct parallel overhead. However, a complication now arises because this new source of overhead breaks an important assumption about direct parallel overheads as expressed in equation 1, that is, k is no longer a constant that does not vary from program to program, or even within the same program. With dependent variables, more dependent variables are likely to lead to higher overheads, i.e. k *depends* on the number of dependent variables present during a particular parallel execution. Thus, we can no longer simply use the amount of work performed between points where distance is measured (W in equation 1) as indicating the time-complexity, but need to take the number of dependent variable into account: the more dependent variables there are in a particular part of a parallel execution, the higher the direct parallel overhead (i.e. a larger k in equation 1). However, between points where distance would be measured, it is still reasonable to assume that the overhead would be approximately proportional to the amount of work done, and equation 2 can be modified by splitting the k of equation 2 into two different components:

$$H \simeq (k + d)W + f + Nq$$

for the whole task, and thus the overhead would be $(k + d)W$ for each portion of task between points where distance is measured, where k represents the same original proportional overheads, while d is a factor that is dependent on the number of dependent variables found in that portion of the task.

Thus, the distance measure becomes more complex in a DAP setting than in previously considered situations. It should certainly be possible to devise schemes which make use of this more complex distance measure but, even the original distance measure may be helpful in a DAP setting, because increasing the amount of work performed between CGEs would also reduce the number of dependent variables and suspension, in addition to reducing the overhead of creating parallel work.

#w	no_gran	pred.	96	240	compile-time	pred.
1	6286 (6400)	–	6221 (5662)	5709 (3448)	4586 (3205)	–
2	3177 (1.98×)	1.99×	3017 (2.06×)	2713 (2.10×)	2335 (1.96×)	1.98×
4	1654 (3.80×)	3.91×	1725 (3.61×)	1661 (3.44×)	1560 (2.94×)	3.22×
8	974 (6.45×)	7.45×	1350 (4.61×)	1396 (4.09×)	1401 (3.27×)	4.15×
1	9956 (10099)	–	9775 (9900)	9058 (7732)	6989 (5050)	–
2	5112 (1.95×)	2.00×	3985 (2.45×)	3742 (2.42×)	3566 (1.96×)	1.98×
4	2715 (3.67×)	3.99×	2903 (3.37×)	2717 (3.33×)	2700 (2.59×)	3.44×
8	1404 (7.09×)	7.97×	2986 (3.27×)	3004 (3.02×)	3020 (2.31×)	3.44×

Table 10: Granularity control for DAP qs(6400)(top) and pascal(200), on Sun SPARCcenter

To see how well the simple original distance measure would work for DAP programs, we applied the run-time and compile-time granularity control to two DAP programs. Table 10 show the results, the two ‘pred.’ columns give the predicted speedups for their respective configurations (no_gran and compile-time) as obtained on the DASWAM simulator; no pred. figures are given for the run-time results because the simulator had not been extended for the run-time control scheme.

The two programs chosen showed quite high DAP overheads during studies of DASWAM performance [19]. They are: a DAP version of quick-sort[12], sorting a 6400 element list, and Pascal(200), generating in this case the coefficients for the 200^{th} row of the so-called Pascal’s triangle. Compared to the IAP programs for which granularity control was effective, the initial (i.e. before granularity control) average distances (measured in abstract machine instructions) are quite high: 281 for Pascal, and 307 for quick-sort. The results are again the best of at least 5 runs, in milliseconds, on the Sun SPARCcenter. Thresholds of 96 and 240 instructions were used for the run-time method. The 240 threshold was chosen because in the quick-sort case, it sequentialised approximately half the CGEs as in the compile-time method.

Perhaps the most striking result is that the granularity control greatly reduces the available parallelism, much more so than might be expected from the IAP experience.[13] In particular, the number of sequentialised CGE should be quite small for the 96 run-time threshold, but the reduction in speedups were still significant at 8 workers. The predicted speedups for the compile-time cases, which is significantly less than the predicted speedups for the non-controlled cases, show that the reduction in speedup is due mostly to the great reduction in parallelism, rather than some performance problem with DASWAM. This was somewhat unexpected, and we think the likely reason is that the sequentialisation of even some CGEs causes the delay of the production of bindings for dependent variables, thus causing longer suspensions and hence loss of parallelism. This suggest that simply measuring the distance (however defined) may be insufficient, if some of the overheads have other effects other than introducing extra overheads to the execution (in this case, suspension causes computation to stall).

Nevertheless, both granularity control methods do reduce the parallel overheads on one worker, although the reductions with the compile-time method are much more significant with smaller number of workers. At a run-

[12]In addition to the parallel execution of the two recursive calls to quick-sort, which give rise to IAP, the DAP version executes the partition of the list in parallel as well.

[13]Note that the apparent super-linear speedups seen in the 2 workers cases of the run-time method is probably due to a greater number of sequentialised CGEs for 2 workers compared to 1 worker, resulting in overall faster execution.

time threshold of 96 instructions, few CGEs are sequentialised, and the improvement in execution time is small but, nevertheless, the parallelism is still reduced significantly with larger number of workers. The result is that the improvement in performance with granularity control is unable to compensate for the decreased parallelism with this larger number of workers. However, it should be noted that the reduction in overheads are much more significant for the DAP case than the IAP case. For example, the DAP version of the quick-sort has the same average distance between CGEs as the IAP version, but the reduction in parallel overhead is much more significant. This is to be expected as DAP incurs significant extra overheads when compared to IAP.

In summary, the simple granularity control schemes were still effective in providing better performances for smaller (4 or less in the examples) number of workers; but it is clear that additional factors need to be considered in controlling granularity for DAP to achieve better performance. From the significant reduction of overheads, it also seems likely that with the right method, DAP can benefit greatly, perhaps more than IAP, from granularity control. Much more research is needed in this area.

6 Conclusions

In this paper, we presented a new metric — distances between sources of parallelism — for measuring granularity, and explained how this metric can be used to control granularity to improve parallel execution performance. We believe that this new metric is better able to account for why granularity control is more effective in some programs than in others. We presented results for two very simple schemes that make use of this metric, one run-time based and the other compile-time based. Results show that the new metric is quite promising. In fact, we showed that the two distance based methods were able to obtain significant performance improvements for a complex program that was hard to improve with traditional, complexity-based, systems. Initial results suggest that even the simple run-time method can obtain surprisingly good performance for some of the benchmarks, even though no consideration have been given to the "worth" of CGEs being sequentialised. On the other hand, better performance requires compile-time analysis in order to remove the "correct" CGEs. This point was demonstrated by the fact that a complexity-based system could outperform the new methods for benchmarks that create parallelism regularly, frequently, and have well balanced parallel tasks, as in such cases it was able to greatly increase the average distance through removing the "correct" CGEs.

We believe that in order to take full advantage of the distance metric, combined run-time and compile-time granularity control will be needed. Hard problems are still open, such as how to best set thresholds, how to more efficiently combine run-time and compile-time techniques, how to combine the complexity-based and the distance based metrics, and how to use these principles to improve execution for other parallel systems. We are actively researching ways to achieve these goals.

Finally, as discussed in the introduction, we believe that this metric has wider application than to parallel Logic Programming systems. It applies to any parallel system that dynamically creates parallelism, and has a cost associated with this.

Acknowledgements

The authors would like to acknowledge and thank for the contribution and support that Manuel V. Hermenegildo and Pedro López-García gave to this work, including the use of the SGI and Sun multiprocessors. We also thank the anonymous referees of this paper for their helpful comments. Vítor Santos Costa has been supported by the PRAXIS PROLOPPE project, the JNICT MELODIA project, and by the NATO MAPLE project.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Approach to Or-Parallel Prolog. Technical Report SICS/R-90/R9009, Swedish Institute of Computer Science, 1990.

[2] U. C. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Volume 3*, pages 841–850, 1988.

[3] S. K. Debray, P. López-García, and M. V. Hermenegildo. Lower Bound Cost Estimation for Logic Programs. In J. Małuszyński, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, pages 291–306. The MIT Press, 1997.

[4] D. DeGroot. Restricted And-Parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 471–478, 1984.

[5] I. Dutra. Strategies for Scheduling And and Or Work in Parallel Logic Programming Systems. In *Logic Programming: Proceedings of 1994 Internation Symposium*. The MIT Press, 1994.

[6] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, The University of Texas At Austin, 1986.

[7] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 253–268. The MIT Press, 1990.

[8] M. V. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[9] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, 1988.

[10] L. Huelsbergen, J. R. Larus, and A. Aiken. Using the Run-Time Sizes of Data Structures to Guide Parallel-Thread Creation. In *LFP'94*. ACM Press, 1994.

[11] A. King, K. Shen, and F. Benoy. Lower-bound Time-complexity Analysis of Logic Programs. In J. Małuszyński, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, pages 261–276. The MIT Press, 1997.

[12] P. López-García, M. V. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computing, Special Issue on Parallel Symbolic Computation*, 11(3–4):217–242, 1996.

[13] E. L. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Vol. 3*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[14] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[15] K. Shen. An Overview of DASWAM — An Implementation of DDAS. Technical Report CSTR-92-08, Computer Science Department, University of Bristol, 1992.

[16] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731. The MIT Press, 1992.

[17] K. Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[18] K. Shen. Implementing Dynamic Dependent And-parallelism. In *Logic Programming: Proceedings of the Tenth International Conference*, pages 167–183. The MIT Press, 1993.

[19] K. Shen. Initial Results from the Parallel Implementation of DASWAM. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1996.

[20] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, Oct.–Dec. 1996.

[21] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 135–151. The MIT Press, 1991.

[22] K. Shen and D. H. D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proceedings of the Fourth Symposium on Logic Programming*. Computer Society Press of the IEEE, Sept. 1987.

[23] E. Tick. Memory Performance of Lisp and Prolog Programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 642–649. Springer-Verlag, 1986. Published as Lecture Notes in Computer Science 225.

[24] A. R. Verden. *And-Parallel Implementation of Prolog on Distributed Memory Machines*. PhD thesis, Department of Electronics and Computer Science, University of Southampton, 1991.

[25] R. Yang, A. J. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. Technical report, Department of Computer Science, University of Bristol, 1993.

[26] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A. V. S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992, Volume 2*, pages 809–816. Institute for New Generation Computing, June 1992.