Distance: a New Metric for Controlling Granularity for Parallel Execution

Kish Shen
University of Manchester
M13 9PL, U.K.
kish@cs.man.ac.uk

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

Andy King
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 [6]. Granularity control have been recently applied to Prolog/Logic Programming [20, 9, 8] and Functional Programming [7]. In these methods, 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 thus boils down to a two-phase process. At compile time, a global analysis system calculates the time-complexity of a potentially parallel goal[1]. At run-time, a pre-determined threshold for time-complexity is used to determine if a task is to run in parallel or not: if its 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 key motivation for controlling grain size is to reduce overheads. *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. Furthermore, in many systems, as long as there is parallel execution, the direct overheads probably do not vary (on average) greatly from program to program, so the constant of proportionality would not change greatly between programs. Task size, as estimated

[1] Or more precisely, an upper or lower bound on the goal's complexity. In [9, 8], 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.

by using a goal's time complexity, is therefore a suitable metric for accounting this source of overheads across a wide range of programs.

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 [8] 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. We thus believe that using task size 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: The "Distance" Metric

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 shows this metric, along with the difference from the conventional complexity metric. Figure 1a illustrates a task executing sequentially, without any overheads. Figure 1b 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 1c 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 these creation points occur at irregular intervals.

Figure 1c also shows the new distance metric. The metric measures distances (amounts of work performed) between points where work was created.

The above general description 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. With parallel Prolog, for traditional or-parallel systems such as Aurora [10], 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 independent and-parallel (IAP) systems such as &-Prolog [5] and &ACE [11], a task corresponds to an and-goal, the overhead for creating parallel work corresponds to the overhead of handling CGEs, hence “distance” corresponds to the distance between successive CGEs. For concreteness, in the rest of this paper, we will discuss the issues within the context of IAP. This allows us to use specific examples for illustration. We also we have ready access to an IAP system (the IAP part of DASWAM [14]) that allows us to perform experiments.

There are several important considerations when using the distance metric. 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. Therefore, it should be the *average* distance over the whole program (or over parts of the program in which granularity control is applied) that should be considered.

A second consideration is that 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 as busy as possible doing useful work, while minimising the amount of parallel overheads. This is difficult to achieve, as it depends 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 [8]). In addition, different sources of parallelism (such as CGEs in an &-Prolog like languages) 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. We would want to 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 not known. 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. However, and as a first step towards using the distance metric for controlling granularity and studying its usefulness, we will consider two simple schemes, one compile-time and the other run-time. These metrics do not take the quality of parallelism into account, i.e. they simply try to increase the distance between *individual* sources of parallelism by removing those sources whose distances are considered too small.

The amounts and distributions of the various types of overheads will of course vary with different machines. This will have an impact on granularity control. For example, for distributed non-shared memory machines or networks, the direct parallel execution overheads would be much more significant, and although it is more appropriate to ignore indirect parallel overheads in such cases, using the distance measure is still appropriate as it also takes into account direct parallel execution overheads, and can thus effectively reduce such overheads as our results shall show. In addition, it will apply in situations where it is difficult to estimate the complexity of a goal.

2 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 methods 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.

2.1 The Compile-Time Method

This method tries to determine the distances between individual run-time CGEs (the sources of parallelism in a goal-level and-parallel Prolog scheme) at compile-time, and if the distance is too small, it increases the distance by source-to-source transformations that remove CGEs. 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 [8] except quick-sort, then the distance can simply be calculated by counting the number of resolutions between successive CGEs. As we already stated, we do not believe that these simple methods would lead to the best results, and our interest here is simply to study the effectiveness of distance as a granularity control metric in order to guide our development of more sophisticated methods. Therefore, we would not consider the exact method that would be needed to determine the distance.

To explain the compile-time method, consider first the Boyer benchmark as ported to Prolog by Tick [18]. This 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:[2]

```
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.[3] 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 [14]. Moreover, the program is not suitable for the granularity control based on time-complexity analysis as described in [9, 8], because of the difficulties of deriving a relationship between input arguments and the complexity. The distance

[2] The &-Prolog notation is used: & indicates a parallel conjunction.

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

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 generated only for every second call to `rewrite_args/3`:[4]

```
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.

2.1.1 Evaluation

To evaluate the impact of this optimisation 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 [14]; '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 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 seem 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.

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

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

#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

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 is found to remove CGEs, 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. One can of course reduce the numbers of CGEs by a greater or lesser amount by transforming the program to generate CGEs more or less often.

2.2 The Run-Time Method

This 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 predetermined 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(...))
```

We first discuss the implementation issues in implementing this technique, and then give a brief performance analysis.

2.2.1 Implementation Issues

We implemented the above conceptual model at the abstract machine level to minimise overheads. Within the abstract machine, the natural unit to use for the counter 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[5] to use the same code for parallel and sequential execution, thus removing a potential source of overhead. The parallel code differs from the sequential code mainly in the addition of `allocate_pcall`, `push_goal`, and `par_proceed` instructions. Thus, execution of these instructions must

[5] see [12, 13], for details of DASWAM's instruction set.

be avoided if the CGE is to be run sequentially. To accomplish this with minimal impact on performance, we added two new abstract machine instructions, `skip_par_if` and `gran_par_proceed`. We demonstrate their application through the granularity controlled code for the CGE in boyer's program (see section 2.1):

```
      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[6] or not. First, the slots representing and-goals in a Parcall marker have as an extra field a new counter (`PS`)[7] which is initialised to 0. 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 representing 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

[6] 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.

[7] 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 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.

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.

2.2.2 Overheads

The run-time scheme introduces some overhead to the execution. One always has to update of 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.

ng_orig	ng	g(0)
6195.6	6239.2	6331.8

Table 2: Execution times (ms) for Boyer

Table 2 gives the single processor execution times for boyer under the original scheme, ng_orig; under a scheme where dynamic granularity control enabled but not actually used in CGE code, ng; and under a scheme where dynamic granularity control enabled code being generated, but not actually sequentialising CGEs, g(0). The latter is obtained by the use of a zero threshold so that all CGEs are parallelised. 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 they are just 2.2% for g(0) over ng_orig for boyer, a program that has an unusually low average distance (and thus an unusually high overhead) as already discussed.

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 that 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, as Table 2 shows, the 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.

2.2.3 Performance

Table 3 shows the results of using the runtime method on boyer, with the threshold set at 64, 80, and 96 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
1	6189 (94056)	5456 (48511)	5370 (44104)	5040 (32230)
3	2514 (2.46×)	2158 (2.53×)	2099 (2.56×)	1920 (2.63×)
4	1959 (3.16×)	1717 (3.18×)	1651 (3.25×)	1514 (3.33×)
5	1641 (3.77×)	1492 (3.66×)	1444 (3.72×)	1301 (3.87×)
6	1436 (4.31×)	1347 (4.05×)	1292 (4.16×)	1168 (4.32×)
7	1285 (4.82×)	1242 (4.39×)	1190 (4.51×)	1083 (4.65×)

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 the higher thresholds, probably indicating that direct parallel execution overheads were being reduced[8]. Further experiments with even larger thresholds are necessary.

#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

Table 4 shows a worst-case program for traditional granularity analysis. The program is orsim [17, 15]. 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

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

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.

3 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 [8]. 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

The average distances 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 [9, 8].

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 granularity-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 [8], 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 [8], 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 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

3.1 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. 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.[9] 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.

3.2 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 [8] on using complexity analysis for granularity control illustrates this point for a larger set of programs and query sizes.

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

[9] 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.

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

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 benchmark 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. However, this method would not work as well for several classes of programs, such as programs with other distributions of CGEs, for example where the sizes of the goals in a CGE are less well balanced, or programs where the average distance is larger, as it is the case for quick-sort. In this case, although the proportion of removed CGEs for the 16 and 64 resolution steps are similar to the Fib benchmark, the positive impact is much less. In fact, there was a negative impact on the 16 resolution steps threshold case, because the gain from reducing CGEs cannot compensate the cost of the relatively expensive list length threshold test.

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, 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 overheads to derive the size parameter, as in the case for quick-sort.

4 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 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 [19]), 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 the execution history to try and anticipate future behaviour [4]. 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 of controlling granularity.

We would also like to extend granularity analysis to dependent and-parallelism. In a previous working paper [16], we described some initial experimentations with DAP granularity control. Our results showed that granularity control for DAP is more complicated than for IAP. the definition of distance needs to be modified to take into account such overheads as task suspension and, in addition, the actual time a task spends suspended would also need to be considered.

5 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 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, the two distance-based methods were able to obtain significant performance improvements in complex programs that are 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 to fully make use of this distance metric, combined run-time and compile-time granularity control scheme 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.

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. 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] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] K. Shen, V. Santos Costa, and A. King. A New Metric for Controlling Granularity for Parallel Execution. In *1997 Post ILPS Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming*, 1997.

[17] 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.

[18] 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.

[19] 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.

[20] 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.