A New Metric for Controlling Granularity for Parallel Execution

Kish Shen, Vítor Santos Costa and **Andy King**

1 Introduction

The estimation or deduction of the time-complexity of goals has been used for granularity control [3, 5]: a particular complexity is chosen as a threshold, and goals deduced to have complexity less than this threshold are not executed in parallel. This method of granularity control have been shown to be useful in improving the performance of some parallel Prolog programs. However, it does not work so well for some other programs, such as quick-sort. In this paper, we propose a new metric for measuring granularity, and discuss why this metric might be more suitable for granularity control than the time-complexity of goals. We present some initial results from two very simple methods of using this metric for controlling granularity, and discuss how more sophisticated methods might be devised.

The original motivation for controlling granularity is not to execute task in parallel when the overhead for making it run in parallel outweigh the actual amount of work done in parallel, thus negating the benefits of parallelism. Such overheads, which we would refer to as *parallel task execution overheads*, include the cost of scheduling the task for parallel execution, the cost for initialising the new parallel task, and any hardware dependent costs such as the extra lockings and interprocess communication costs during the execution of the task. However, the parallel task execution overheads do not include an important source of possible extra parallel overheads: the sequentialisation of the task not only prevents the task itself from running in parallel, it also prevents the creation of *further* parallelism within the task (i.e. potential parallel tasks which would have been created if the task was not sequentialised). For a system with finite number of processors, and in which there is a cost associated with creating potentially parallel work, and which would be incurred even if the task is not actually executed in parallel, then the exclusion of this parallel work creation cost within sequentialised task can contribute significantly to the reduction of overall overheads. Indeed, as shall be discussed later, the experimental results from [5] strongly suggest that, at least for those two configurations and programs studied, it is the reduction of this parallel work creation overhead, of generating largely superfluous parallel work for which the system do not have the resources to exploit, that is major factor for the increase in performance of the programs with granularity control.

Using size of a task as the metric for controlling granularity is suitable for accounting for the parallel execution overheads, because these are generally to a first approximation either of a fixed size (such as making a task visible to other processors), or are approximately proportional to the size of the task (such as the interprocess communication overheads). However, for systems that do not create parallel work at regular intervals, this overhead is in general not very dependent on the size of the task being executed in parallel, and thus in the general case, using size of tasks alone as a metric for granularity control is, we believe, not sufficient to capture the full complexities of reducing overheads with granularity control.

Figure 1: The “Distance” Metric

In this paper, we argue for a somewhat different metric for measuring the “granularity” for improving parallel performance on system: the “distance” metric, defined as the amount of work performed between successive points in which parallel work creation overheads are incurred. This metric, along with the difference from the conventional metric of the complexity of a task, is illustrated in Figure 1. It shows the time line of execution of a task, with Figure 1a showing the task executing in parallel, with the overheads for creating new parallel work shown as gray boxes. The “distance” between successive creation overheads are indicated by the double arrows. In Figure 1b, the same task is constrained to run sequentially, as it would be in the traditional method of controlling granularity if it’s time-complexity is below the threshold: all the extra task creation overheads within the task has now disappeared.

The above general description applies to any parallel system in which creation of parallel work incurs overheads. For example, in conventional parallel programming environment, the parallel work creation overhead might correspond to the creation of a (lightweight) process. With parallel Prolog, for traditional or-parallel systems such as Aurora [6], Muse [1] and PEPSys [2], where a task corresponds to the execution of a new or-parallel goal, 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 [4] and &ACE [7], a task corresponds to an and-goal, and the overhead for creating parallel work corresponds to the overhead of handling CGEs and so “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, because this allows us to use specific examples for illustration, and also because we have ready access to an IAP system (the IAP part of DASWAM [10]) that allows us to perform experiments.

There are several important considerations with 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. Another important consideration is that in constraining tasks to execute sequentially, potential parallelism would be removed. This would not be a problem if the parallelism cannot be exploited by the system because of a lack of resources. In fact, the minimal execution time for a particular configuration is achieved when the amount of parallelism produced is just sufficient to keep the processors in the system as busy as possible doing useful work, while minimising the amount of parallel overheads. However, this is difficult to achieve in practice, because it depends on many 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 [5]). In addition, because different sources of parallelism (CGEs in an &-Prolog like language) leads to different amounts of parallelism, producing just the right amount of parallelism while minimising the amount of parallel overheads (which to first approximation would be equivalent to minimising the number of sources of parallelism) depends significantly on which sources of parallelism are removed. Thus, ideally, the “correct” sources of parallelism has to be removed such that the required amount of parallelism delivered is achieved by minimising the number of sources, i.e. with the maximum average distance. The important point is that for a given average distance, the amount of parallelism that is available can defer significantly, depending on which actual sources are removed. In the ideal case, we want to remove those sources of parallelism that leads to the least amount of parallelism.

Note that the ideal is almost certainly impossible to achieve in practice, but a system which seeks to perform close to the ideal will need both run-time and compile-time controls: 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 of the parallel system) is not known. Both these factors affect what the ideal amount of parallelism should be. On the other hand, it is impossible to determine at run-time what the "worth" of a particular source of parallelism would be when the decision has to be made about if it should be allocated or not – this decision has to be made before the parallelism is actually exploited. Indeed the ideal decision would depend not just on the particular source, but also on what further sources of parallelism might be available further into the execution. 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 would probably be quite complex, and has not yet been designed. However, as a first step to using the distance metric for controlling granularity, we will consider two simple schemes (one compile-time, one run-time) which 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.

2 New Granularity Control Methods

2.1 Compile-time method

This method tries to determine the distances between individual run-time CGEs (the sources of parallelism in a marker based and-parallel Prolog scheme) at compile-time, and if the distance is too small, it increases the distance by source-to-source transformation, by removing some CGEs. In many cases, the distance between CGEs can be easier to determine that the complexity of a goal. However, the exact method of determining the distance would not be considered here.

Consider the Boyer benchmark developed by Tick [13]: this program has significant amounts of (non-strict) IAP, and most of the parallelism comes from one clause in the program:

```
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.
```

However, this program has very small granularity by the distance measure, because successive

CGEs occur very close together at run-time.[1] This is shown by the very significant overhead of running Boyer with the parallel annotation on one worker and running it without annotations – on the DASWAM it is 43% slower, the largest such overhead for any IAP program examined in [10]. The program is not suitable for the granularity control based on time-complexity analysis of [3, 5], because of the difficulties of deriving an invariant relationship between input arguments and the resulting complexity. With the distance metric, however, there is a very simple way to increase the distance, by performing a source-to-source transformation on the program, so that and-parallelism is generated only for every second call to `rewrite_args/3`:[2]

```
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. The effect of this is shown in Table 1. The '#' column shows the number of workers for each row, the '(g)' indicate cases with the granularity control, 'ideal' is the 'ideal' amount of parallelism, obtained from the DASWAM simulator [10], 'sym' are the results for a 10 processor Sequent Symmetry with 25MHz 80386, running Dynix; 'sun' is a 10 processor SPARCcenter-2000 running Solaris, 'cgi' is a 10 processor Silicon Graphics Challenge running IRIX, and 'pc' is a 2 processor PC with 200MHz Pentium Pros, running Linux. All times are in seconds. The timings measure the duration from the start of 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, which was compiled with MIPS' cc), with optimisation turned on. Note the extremely low speedup obtained for 2 workers on the Challenge for the non-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 granularity 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.

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

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

#w	ideal	ideal(g)	sym	sym(g)	sun	sun(g)	chal	chal(g)	pc	pc(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×)	2.482 (1.00×)	2.045 (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×)	1.396 (1.77×)	1.109 (1.84×)
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×)		
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×)		
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

As expected, the amount of parallelism available is reduced by granularity control, as shown by the lower speedups from ideal(g) compared to ideal. Thus, in the absence of overheads, the performance should be better without granularity control. With realistic systems, the granularity control would be effective if it gives performances than the non-granularity controlled program. The results shows that this form of granularity control is indeed effective in all cases. With 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, and even for the Sequent, the agreement with the ideal parallelism is better for the controlled case. This shows that the granularity control was able not only to remove the parallel creation overheads (cost of allocating CGEs), but also some parallel task execution overheads, especially for the faster machines. The main component that leads to the increased performance is clearly due to the removal of parallel creation overheads, as shown by the difference in performances for the one worker cases.

However, the speedups achieved by the non-controlled case is still reasonable and generally not very significantly lower than the controlled case: this shows that the parallel execution overhead is not that significant, because if it dominates, one would expect the speedups achieved in the non-controlled case to be much lower. The main improvement is achieved through the reduction of the number of CGEs, as shown by the faster execution times with one worker for the controlled case.

The results also show that reducing the number of allocated CGEs does also reduce the amount of parallelism, but with the configurations tested, this reduction in parallelism is more than compensated for by the reduction in parallel overheads. There certainly may be a point where the continually more pronounce reduction of parallelism may offset the gains from the reduction of parallel overheads, but this point is beyond the maximum of 10 processors tested in these experiments.

The results also shows the weakness of measuring distance between individual CGEs, as the reduction in parallelism becomes 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 with 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 run-time method

In this second method for controlling the granularity, a simple counter is used to decide if a CGE should be allocated or not. The counter is used to measure the work done since the last CGE was allocated and, if it is below a pre-determined threshold, then the new CGE would not be allocated, and the goals in the CGE run sequentially. Conceptually, this can be thought of as:

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

The actual system implemented the above conceptual model at the abstract machine level to minimise the overhead that would be incurred. With 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[3] 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 that the presence of *allocate_pcall*, *push_goal* and *parallel_proceed* instructions in the parallel code. Thus, the execution of these instructions have to be avoided if the execution is to be sequential. To accomplish this with the minimal impact on performance, two new abstract machine instructions were added, and the granularity controlled code for the CGE in `rewrite_args/3` of boyer is:

```
      skip_par_if(label(_1),5,2)
      allocate_pcall(2,1,6)
      push_goal(label(_1388),1,0,0)
label(_1):
      put_y_value(5,0)
      put_y_value(0,1)
      call(rewrite/2,5)
      gran_par_proceed(label(_1386),0)
label(_1388):
      put_y_value(2,0)
      put_y_value(3,1)
      put_y_value(4,2)
      call(rewrite_args/3,2)
      gran_par_proceed(label(_1386),1)
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, and the following code is to be executed sequentially. Label is placed after the *allocate_pcall* and *push_goal* instructions. N is the number of and-goals in the CGE.

[3] see [8, 9] for details of DASWAM's instruction set.

ng_orig	ng	g(0)
6195.6	6239.2	6331.8

Table 2: Execution times (ms) for Boyer

This 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 takes the place of the par_proceed instruction, and has exactly the same arguments. The only difference is that this instruction determines if it is inside a sequentialised CGE or not, and if so, the code associated with the normal *par_proceed* is not executed. In fact, the extra functionality of this instruction can be incorporated into *par_proceed*, but this was not done in order to not 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[4] or not: a new counter (PS) is added to all the slots representing and-goals in a Parcall marker,[5] 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 current (i.e. the last previously allocated) CGE's slot representing the current task. In this sequentialised case, no new Parcall marker will be allocated, and the procedure calls in the sequentialised CGE still has the previous Parcall marker as the current Parcall marker. Later, when the execution returns from the call and executes *gran_par_call*, the same Parcall marker will again be current. The PS counter is check to see if it is zero, and 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 would allow multiple successive CGEs to be sequentialised: the PS counter is simply incremented by the appropriate amount each time.

As for the instruction threshold counter, this is implemented as a new register of the worker. As the instructions executed by a worker is already counted by an existing register, this new register records the instruction count at the last parallel CGE or, if a new task is started on a worker, the instruction count at the start. This is then used with the instruction count to calculate if a CGE should be sequentialised or not.

Thus the overheads introduced by this scheme in cases where granularity control is not applied is minimal: the update of the threshold counter at each CGE, and the initialisation of the PS counters for each slot in the CGE. With granularity control, the extra cost is the execution of the *skip_par_if* instruction, and the check performed at each *gran_par_proceed* instruction. This overhead is still reasonably small, as shown in Table 2. The execution times are obtained on the SGI Challenge for the execution of the parallelised boyer benchmark on 1 worker, with DASWAM compiled with the MIPS C compiler, under the same optimisation level (-O2). The timings are the mean of 5 executions. ng_orig is the timing for DASWAM without the granularity scheme compiled in; ng is the timing for DASWAM with the scheme compiled in, but without using it in the code; and finally, g(0) is with the granularity control code in the CGE, but with zero threshold, so that all CGEs will be parallelised, thus effective behaving like the other two timings,

[4]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.

[5]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.

#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

but with the granularity control overheads. The overheads are reasonable: 2.2% for g(0) over no_orig for boyer, a program that has an unusually low average distance (and thus a unusually high overhead in this case) as already discussed.

Another source of overhead of this scheme is the the parallel call code is sometimes not as optimised as the equivalent sequential code for sequential execution, in terms of register allocations, and some of the instructions used (e.g. 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), but the most important difference is the lost of last call optimisation, in particular in a non-strict IAP code like boyer, where there is now an additional unification at the end of the CGE. However, this effect is still relatively small.

This scheme can also be used for executing sequentialised CGE in other situations, such as when a CGE condition fails. This will avoid the cost of the relatively more expensive if-then-else construct, which is compiled into extra procedure calls for the then and else parts.

Another feature of this scheme to note is that the number of CGE that will be sequentialised is actually non-determinate for more than one worker. This is due to the way the threshold is counted from the start of a new and-goal. Given that which goal are executed in parallel is non-determinate, so the number of sequentialised CGEs can vary. It is expected that this variation would be relatively small, and that as more tasks are actually executed in parallel, the number of sequentilised CGE would increase slightly.

Table 3 shows the results of using this runtime method on boyer, with the threshold set at 64, 80, and 96 abstract machine instructions (these thresholds were chosen because the implementation of *skip_par_if* allowed the threshold to be set in increments of 16 units only). The results are the best timings of at least 5 runs, 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.

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 there.

Even with the loaded machine, the results shows that the granularity control can improve the performance, and the improvements becomes greater with the higher thresholds. Further experiments with larger thresholds are necessary.

Table 4 shows a program which does not need granularity control, running on the SGI Chal-

#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

lenge when it was lightly loaded. Times are in ms, and are again the best of at least 5 timings. The program is orsim [12, 11]. which is quite a complex program that is very difficult to analyse, but it has quite high granularity, and does not need granularity control. 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 nearly the same as without the control.

2.3 Discussion

Using average distance between CGEs, we can explain the effectiveness of granularity analysis on boyer, and also for the previous programs studied in [5]. 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 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 [3, 5].

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 does not contain any CGEs, and is recursive, so a simple 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 will then decide not to add compile-time granularity control in this case.

#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

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 [5], 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 ms, the best of at least 5 executions. qs(3200) was performed on the 10 processor Sequent, while hanoi(16) and fib(23) performed on the SGI Challenge. The compile-time and run-time control methods, along with the traditional complexity threshold methods are presented. The results for the complexity threshold method of qs(3200) are taken from [5], and are average timings, rather than the best times. For the run-time control method, thresholds of 64, 80 and 96 abstract machine instructions were used, with the addition of 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) (because many of the calls terminates in the base case, with no CGEs). The results for the complexity method are show in the columns with headings of size(X), where X is the size threshold (in resolution steps).

The programs have quite different characteristics, with qs(3200) having a high average distance, and relatively low parallelism; while hanoi(16) and fib(23) have very low average distances and quite significant parallelism. The size of the two goals that are run in parallel in a CGE are identical for Hanoi, but are slightly less well balance for Fib, where one goal is about 60% larger than the other. Another difference is that CGEs occurs regularly and at constant distances for Hanoi and Fib, 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 to 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 method.

2.3.1 Comparison of run-time and compile-time methods

For quick-sort, the results shows that for the run-time control, some CGEs are removed, resulting in some slight improvement in execution times for the one worker cases. The differences are not very significant, because the original average distance was already relatively large. In addition, because CGE occurs between recursive partitions of the list, which means that each successive CGE distance must be smaller than the last (because smaller and smaller lists are partitioned), then sequentialising CGEs with small distances would mean removing those CGEs that results in small tasks, which would not greatly affect the available parallelism, and so the speedups were only reduced very slightly. The slight reduction of speedups tend to cancel out the slight decrease in overhead, but nevertheless, some small improvements were still obtained, at least up to the 9 workers tested. On the other hand, the compile-time approach shows that an inappropriate application of the transformation to remove CGEs can have great impact on the performance: the speedup was greatly reduced, and the resulting performance is significantly worse than the other cases. Another point illustrated by the results is that importance of removing the right CGEs: the number of CGEs for the 96 run-time threshold case is only slightly more than the compile-time transformation case, and yet the speedup obtained from the 96 threshold case is significantly better, because only the CGEs that produced the least parallelism were removed. Another point to note about the results is that the one worker case for the compile-time transformation is noticeably better than the other cases, and this is because the compile-time transformation carries less overheads than the run-time control, at least for this case.

For hanoi(16), both compile-time and run-time methods were effective in increasing execution efficiency and, in fact, the improvement is quite significant. Part of this can be attributed to the use of the Challenge, which is a much faster machine than the Sequent, where the parallel execution overheads are therefore comparatively more significant.[6] The compile-time method is more effective in this case, because again there is less overhead to pay for using the compile-time control. In addition, the speedups have been reduced, but because parallelism is plentiful, the reduction does not have a very large impact.

The results for the Fib case is similar to the Hanoi case, though the increase in performance is somewhat less, and the reduction in speedups is more marked, especially when the distance

[6] 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.

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, and their performances 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. This suggests that in this case where CGEs occurs regularly, neither the compile-time or run-time methods are better at removing the "right" CGEs.

2.3.2 Comparison to the complexity granularity method

As already seen from the quick-sort example for the compile-time method, choosing the "right" CGEs can have very profound impact on the number of CGEs that can be removed without a serious impact on parallel performance. The results obtained for the 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 [5] on using complexity analysis for granularity control illustrates this point for a larger set of programs and query sizes: the number of

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

CGEs remaining after performing the granularity control with the various granularity thresholds for the various programs are shown in Table 9, with the mean speedup obtained with 9 workers on a Sequent Symmetry shown in brackets. The results again shows that although the number of CGEs were drastically reduced, 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, with fib(19), the results does show that 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: the query a program will run with cannot in general be determined at compile-time, and run-time information is needed to ensure good performances in these cases.

For all the programs tested, the complexity of a goal is a reasonable guide to the amount of parallelism that would be produced. This is because the and-goals in the CGEs are in all cases recursive and, in the general case, the two goals would both lead to significant parallelism if they are sufficiently large. This, however, is not true of all programs — consider the case where only one of the and-goals is recursive:

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

```
b.
```

In such a case, the distance measure 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 for sufficiently large `X`, even though this would probably not result in a great amount of speedups.

However, in the programs studied, both Hanoi and Fib produces much better results, both in speedups and actual execution times, than the two new methods. This is in particular true for the Challenge results, where the speedups obtained for the non-granularity controlled case of Hanoi(16) is significantly lower than those obtained in [5] for the same program on the Sequent. This illustrates (as we would expect) that the parallel task execution overheads is much more significant for the faster Challenge machine, and that granularity control would therefore be much more effective on the Challenge than the slower Sequent. For quick-sort, the complexity method does slightly less well than the run-time threshold method. The main reason for this seems to be the higher overheads for the granularity control: the one worker performance for a complexity threshold of 64 resolution steps removes the same number of CGEs as the run-time distance threshold of 80, but the latter executes significantly faster. The cost for determining the size of the input list for complexity granularity control is relatively high, but even for the Hanoi and Fib, where a numeric value is checked for determining the threshold, the cost for the complexity method is still higher than either of the two new methods: this can be seen most clearly for Fib(23), where the 112 run-time threshold case removes about the same number of CGE as the 16 complexity threshold case, but the execution time for the run-time method case is faster, and in fact even for the 2 workers case the performance is better, before the lack of parallelism due to removal of "good" CGEs makes the 112 run-time threshold case perform worse with more workers.

The main reason for the performance improvements using the complexity method can thus be seen to be the great reduction of CGEs. This is in particular true for the results in [5], where the parallel task execution overheads is very low. Even for the Challenge, the effect of the reduction in CGEs seems to outweigh the effect of reduction in parallel task execution overheads (as measured by its effect on speedups obtained), although this latter effect is significant, especially for the 9 workers cases of Hanoi and Fib. However, as already noted, the number of CGEs that occurs does not have a program independent dependency on the complexity of a goal, but rather, this method of granularity control succeeds because these particular programs do have a large number of CGEs that occur regularly, and with the shape of the execution (especially in the case of hanoi, where each CGE leads to two goals which are about the same size, and are highly recursively parallel), so that sequentialising the execution of goals below particular complexity happens to remove many CGEs, and furthermore, it removes precisely those CGEs that leads to the least parallelism. This method would not work so well with other distributions of CGE, or when the average distance is larger, as in the case for quick-sort, where although the proportion of CGEs removed for the 16 and 64 resolution steps are similar to the fib benchmark, there is much less effect (and in fact, there was a negative impact on the 16 resolution steps threshold case, because the gain from reduction of CGEs is not sufficient to compensate for the cost of the relatively expensive list length threshold test needed). Another 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 exist in many programs and, even where it exists, the relationship may be difficult to deduce in some of the more complex programs.

2.4 Improving the methods

While in the quick-sort case, it was the compile-time method that removed "good" CGEs, the run-time method can also remove "good" CGEs under certain conditions: consider the case where a program creates several pieces of parallel work of significant size in quick succession early in the execution, and then stops creating parallel work. In this case (which occurs in the bt_cluster benchmark first used by Andorra-I [14]), the actual distances between each CGE will be short, but the average distance (defined ratio of total work per CGE) would be much larger, and the CGEs should not be sequentialised, but the run-time method may remove some of them because the individual distances may be below the threshold. This illustrates a weakness of any purely run-time scheme: they cannot predict what may happen further into the execution, e.g. when and where new CGEs will be created, and how much parallelism these CGEs 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. A somewhat more sophisticated run-time scheme can for example only start to sequentialise CGEs when all the workers are busy *and* the distance is below the threshold. However, as we have seen even for the simple scheme presented here, any run-time scheme does carry some overheads, and the more sophisticated the scheme, the more expensive the overhead and, such systems would still suffer from not having specific information about the execution still to come. Such information can be provided by a compile-time analysis, which is then combined with run-time information to provide the best load for the system.

2.5 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, for dependent and-parallelism (DAP), while the distance between parallel work creation is a factor, there are other parallel overheads that should be considered. In the DASWAM [9] implementation, there are cost associated with variables being dependent, and for suspensions, and probably other costs. However, even here, just increasing distances between CGEs may be helpful, because it would serve also to reduce the number of dependent variables and suspension, in addition to reducing the overhead of creating parallel work.

#w	no_gran	96
1	6286 (6400)	6252 (5662)
2	3515 (1.79×)	3013 (2.08×)
4	1641 (3.83×)	1657 (3.77×)
8	964 (6.52×)	949 (6.59×)

Table 10: Run-time granularity control for DAP qs(6400), on Sun SPARCcenter

Figure 10 show the effect of applying the run-time granularity control to quick-sort (with 6400 element list to allow reasonable timing measurements to be made). The results are again the best of at least 5 runs, in ms, on the Sun SPARCcenter. The granularity controlled case does yields slightly better performance in execution times (as in the IAP case), and perhaps more significantly, in speedups (unlike the IAP case). Note that relatively fewer CGEs have been removed in this DAP case than the equivalent IAP case, this is because the partition predicate is included in the CGE, and thus counted one recursion level earlier. Note that because the average distance is quite high, the improvement would not be expected to be significant in this case, because few CGEs would be removed. However, the reduction in the DAP cost does seem to have an effect even in this case. More programs, especially those with low average distances, needs to be examined. In addition, methods need to be develop that takes more direct account of DAP overheads in the definition of granularity.

3 Conclusions

In this paper, we presented a new metric — distances between sources of parallelism — for measuring granularity, and how this metric can be used to control granularity to improve parallel execution performance. We presented the results from two very simple schemes that make use of this metric.

We believe that to fully make use of this distance metric, a combined run-time and compile-time granularity control scheme would be needed. We are actively researching ways to achieve this.

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] P. L. 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.

[4] 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.

[5] A. King, K. Shen, and F. Benoy. Lower-bound Time-complexity Analysis of Logic Programs. To appear in Proceedings of ILPS'97, 1997.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.