Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer

Kevin Hammond, Jim S. Mattson Jr. and Simon L. Peyton Jones *

University of Glasgow, Glasgow, UK

Abstract. This paper considers the issue of dynamic task control in the context of a parallel Haskell implementation on the GRIP multiprocessor. For the first time, we report the effect of our task control strategies on task granularity, as measured in terms of dynamic heap allocations. This gives a concrete means of measuring the effectiveness of these strategies other than wall-clock timings, which are notoriously uninformative.

1 Introduction

Compared with most conventional approaches to parallel programming, it is comparatively easy to write parallel functional programs. Deadlock is impossible. Because programs are deterministic, they may be debugged on sequential hardware. Programs can be highly scalable, running without change on different processor configurations or even architectures. However this flexibility places a high load on the functional compiler and runtime system. Our approach is to exploit parallelism which is notionally fairly fine-grained, but which can be dynamically combined to form much larger grained programs. Fine granularity allows greater flexibility when programming, and has better worst-case scheduling properties than coarse granularity [3], but carries much greater overhead on an architecture built from conventional CPUs. We use a state-of-the-art sequential functional language compiler, the Glasgow Haskell Compiler [16], modified with support for parallel language primitives and a sophisticated runtime system.

Previous research has considered problems of determining appropriate throttling and scheduling strategies for particular applications [7, 8]. In this paper we study the effect of two simple spark control strategies ("global" and "local" sparking) on performance and task granularity.

1.1 GRIP

GRIP is a MIMD parallel machine, designed to execute parallel functional programs [15]. Logically, the architecture comprises a number of processing elements (PEs) and intelligent memory units (IMUs) connected by a fast network,

* This work is supported by a Royal Society of Edinburgh Research Fellowship, and the SERC AQUA Project. **Authors' address:** Computing Science Dept, Glasgow University, 17 Lilybank Gdns., Glasgow, Scotland. **Email:** kh@dcs.glasgow.ac.uk

as shown by figure 1. The IMUs are designed to handle high level memory operations including dynamic allocation and garbage collection, as well as maintaining pools of executable tasks and sparks.

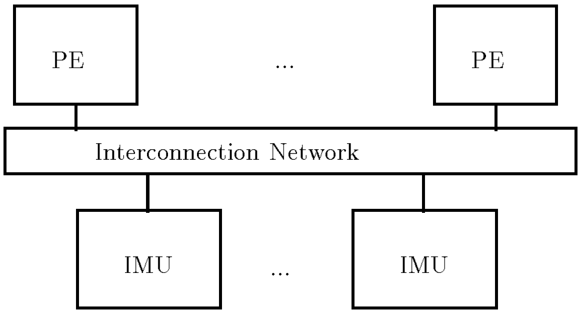

Fig. 1. Logical GRIP Architecture

The physical implementation of this architecture is as a number of identical boards connected by a 100MB/s IEEE P896 Futurebus bus. Each board is a multi-layer PCB connecting either 2 or 4 PEs, one IMU, fast packet-switching hardware, and the Futurebus interface through a 20MB/s local bus. The PEs are 16MHz MC68020s with floating-point units and local memory. The IMUs are built from microcoded AMD 29000s. While the GRIP CPUs are no longer state-of-the art as far as raw performance concerned, GRIP is still a useful testbed for research into parallel execution having low communication latency (10μs), tolerably high bandwidth and great flexibility.

2 Exploiting Parallelism

2.1 The Parallel Execution Model

Our functional execution model is one of cooperating agents each running multiple concurrent tasks. To minimise overhead, tasks are non-preemptive and are scheduled unfairly. Incoming messages are processed at regular intervals, when the task's registers etc. are in a known state.

Heap We use a 2-level model of our dynamic heap, dividing it into local and global areas. Global heap is globally addressed and can be accessed by any PE; local heap is private to a PE. Shared program graph is held in global heap. Local heap may contain pointers to global heap, but there are no pointers from global heap to local heap. This model allows each PE to perform independent garbage collection, at the expense of requiring expressions to be made global (or *flushed*) if they are exported or used to update globally shared expressions.

2.2 Sparks and Tasks

Parallelism is introduced through the use of `par` pseudo-functions. `c 'par' p` sparks a child expression `c` and continues executing the parent expression `p`. Sparks are entered into a global spark pool which is distributed amongst the IMUs. This is the *evaluate-and-die* model of task creation [15]. This model has the advantage of being asymmetric: notification is required only when one task demands a result which is still being computed by another task; and it is safe to throttle by discarding sparks since all sparks are either subsumed by their parent task or are speculative. This allows much more runtime flexibility than either conventional load balancing [21] or throttling [17, 14]: the former is essentially limited to redistributing work once created; the latter is limited to correcting a local glut of parallelism, but can fail if sparking patterns are uneven over time. By storing potential parallelism until it is needed we can discard excess work while avoiding starvation [7, 8]. Mohr, Krantz and Halstead have recently described a similar scheme for Lisp called *lazy task creation* [13].

PEs request tasks by polling an IMU. The IMU responds with an executable task if it has one available, or else with a spark which the PE can use to create a new task. If the IMU has no spark or task available, then it passes on the request to its neighbour. If the request returns to the originating IMU having passed every other IMU, then the PE is placed in a queue of known idle PEs.

2.3 Local Sparking

Several variations can be made to the basic spark strategy described above. One of the most effective is to maintain local spark pools in each PE, and to select sparks from these pools ahead of sparks from the global pool. The IMUs monitor the overall state of the global task pool. If the spark pool falls below a certain threshold, then messages are sent to all PEs asking them to export sparks from their local pools. Sparked global nodes ("global sparks") are exported before any sparked local nodes ("local sparks"). When the number of sparks in the global pool subsequently rises above another threshold, control messages are sent to prevent the PEs exporting any more sparks. In extremis, if no PE has any sparks, then executable tasks may be exported instead. This happens infrequently, in practice, but task migration can be important for relatively coarse-grained programs [8, 3].

3 Granularity

Information about task granularity is vital if parallel compilation techniques are to be improved, and automatic analyses developed. Unfortunately such information is lacking in the literature for compiled functional programs running on real parallel machines. Some useful simulation studies have been performed by Sargeant and Watson [19, 20], but the information provided is coarse: only figures for average granularity are presented, for example.

We have chosen to measure granularity as a function of dynamic heap allocation. This is a reasonable, though not perfect, estimate of execution time for a pure functional program since such programs tend to allocate heap at a fairly constant rate [10]. heap allocations are credited to the task at each garbage collection and context switch, as the difference between the current and previous values of the PE's heap pointer. Note that some extremely short-lived tasks may perform no allocation at all, so a granularity of 0 allocations is possible.

3.1 Results

We ran several programs on four different processor configurations. For space reasons, this version of the paper considers only one: a simple parallel ray tracer. The ray tracer is taken from Kelly's book [12], where it is used as a simple example of process placement in Caliban. The re-implementation in parallel Haskell and some detailed performance results are described elsewhere [9].

The Ray Tracer The ray tracer works by determining the closest impact between a number of rays (representing light from a source at the viewpoint) and a scene composed of a number of 3-dimensional surfaces. The distance of the impact determines the intensity of light that is reflected, and therefore how dark a pixel should be used to represent the visible object. A ray which intersects no object gives a pixel the same colour as the background. We compared three versions of our simple ray tracing program. The first version contains two spark sites, each of which sparks a parallel map function, *parmap*, over a list of arguments. The "outer" spark maps the function which determines the first impact between a ray and the scene, *firstImpact*, over the list of rays. The "inner" spark maps the function which determines impacts between a ray and an object *testForImpact* over the objects which comprise the scene to be viewed. The second version of the ray-tracer uses only the inner of these two spark sites, and the third version used the outer of the two spark sites. Several versions of the `parmap` function were tried: the overall best performance results were obtained with the version shown below, which sparks the list component and recurses over the tail list, whilst returning the result list immediately (so as to reduce synchronisation conflicts with other tasks which are also working on the list structure). These experiments were not exhaustive, however, so it is conceivable that alternative versions of *parmap* could yield better performance in some cases. While, in theory, the *earliest* function could also be evaluated in parallel, in practice, it proved better to use it sequentially to drive the parallel evaluation of *testForImpact*.

```
parmap f l = parmap' l
     where parmap' [] = []
           parmap' (x:xs) = pmxs `par` fx `par` (fx:pmxs)
                              where fx = f x
                                    pmxs = parmap' xs
```

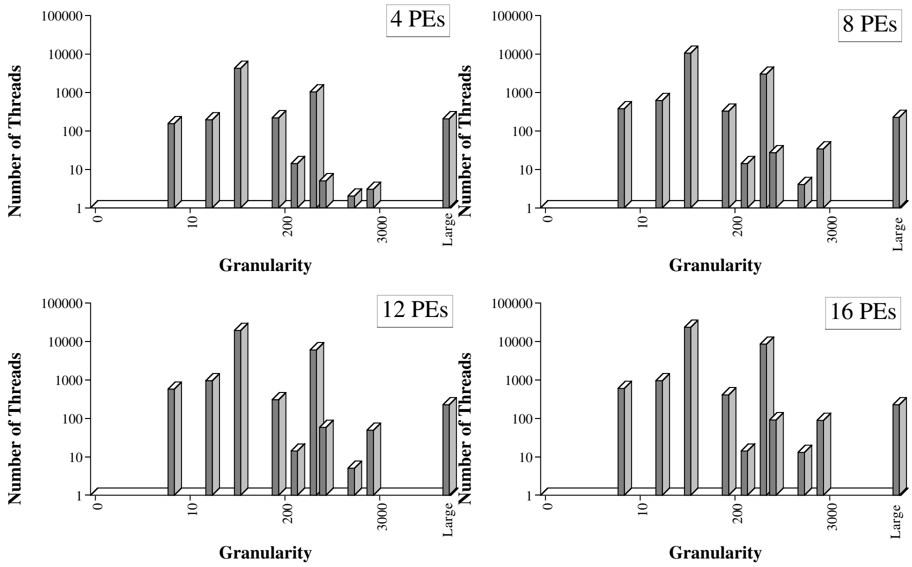

Fig. 2. Granularity Profile: Ray Tracer, Both Sparks, Global Sparking

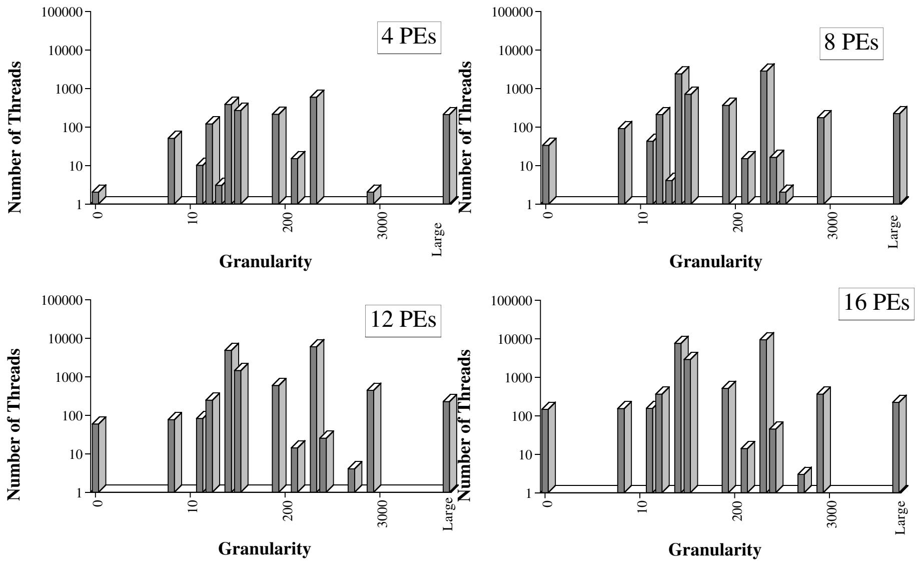

Fig. 3. Granularity Profile: Ray Tracer, Both Sparks, Local Sparking

```
firstImpact :: [Object] -> Ray -> Impact
firstImpact os r =  earliest (parmap (testForImpact r) os)
                    where earliest = foldr earlier NoImpact

findImpacts :: [Ray] -> [Object] -> [Impact]
findImpacts rays objects = parmap (firstImpact objects) rays
```

Figures 2 and 3 give granularity profiles for between 4 and 16 PEs. So that small values are not unduly obscured, these results are plotted on a log scale. The horizontal axis is discrete: all values between 10 and 19 allocations are recorded in the 10 bar, and all values between 900 and 999 allocations are recorded in the 900 bar. All tasks allocating 10000 or more heap cells are aggregated as "large" tasks. To permit comparison with the other results in this paper, the same scales are used for all granularity results throughout.

The similarity between the profiles for differing numbers of PEs is striking. The same task granularities appear in roughly the same proportions regardless of the number of PEs used. Observe, however that more reasonably small tasks are created as the number of PEs increases. Also observe that local sparking causes several trivial-sized tasks to be created. This may be an artefact of our implementation, rather than of local sparking per se.

Since the tasks measured in our results are created from only a few different sparked expressions, it is not surprising to find patterns of task length. It may seem surprising, though that they should vary even for the same spark, but this is clearly true in this case: there are only two spark sites in the entire program (though the *parmap* adds some complication). This variance is a function both of the size of the data and the degree to which it has already been evaluated when a new spark is created. Obviously, certain data sets will generate large tasks and others will generate small tasks. The degree of evaluation is a characteristic only of lazy demand-driven languages, however, and is the means by which results can be shared across parallel tasks. If the sparked function must evaluate its argument, then a larger task will probably result than if the argument were already evaluated. Given the uneven distribution of task size, it seems likely that the program has certain characteristic data sets which are sparked. Identifying these, and determining which can most profitably be sparked would be also an interesting exercise.

Ray Tracer, Inner Spark The inner spark version also reveals a characteristic pattern of sparking, with some elements in common with that obtained for the version using both sparks. This is shown in Figures 4–5. Considering first the global sparking profiles, notice that there no *large* tasks are created, but many more tasks of length 60 and 500 are produced, and several more of length 600 and 2000. In mitigation, far fewer tasks of length 8 and 30 are generated. Thus, the profile has moved towards medium granularity at the expense of the large-grained tasks which existed before. This change actually degrades performance, by a small factor for 4 PEs, but by a factor of 2 for 16 PEs.

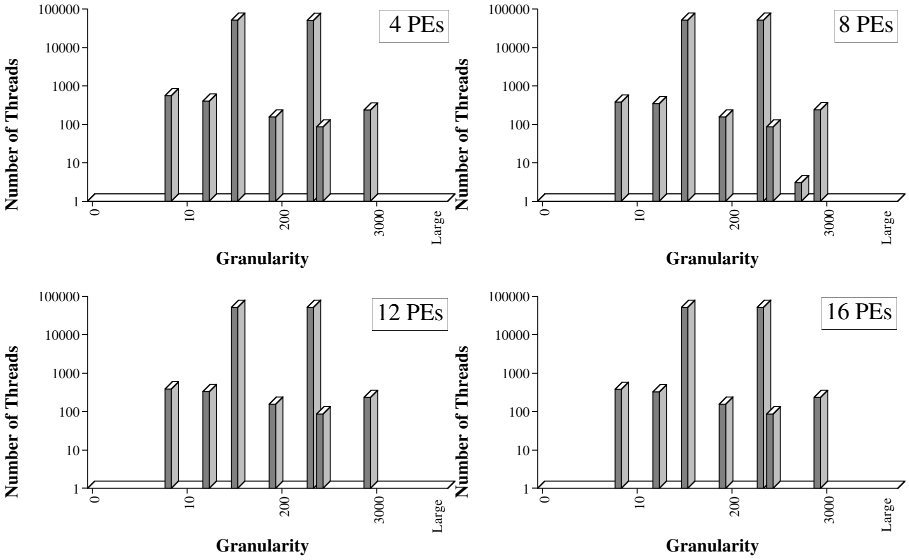

Fig. 4. Granularity Profile: Ray Tracer, Inner Spark, Global Sparking

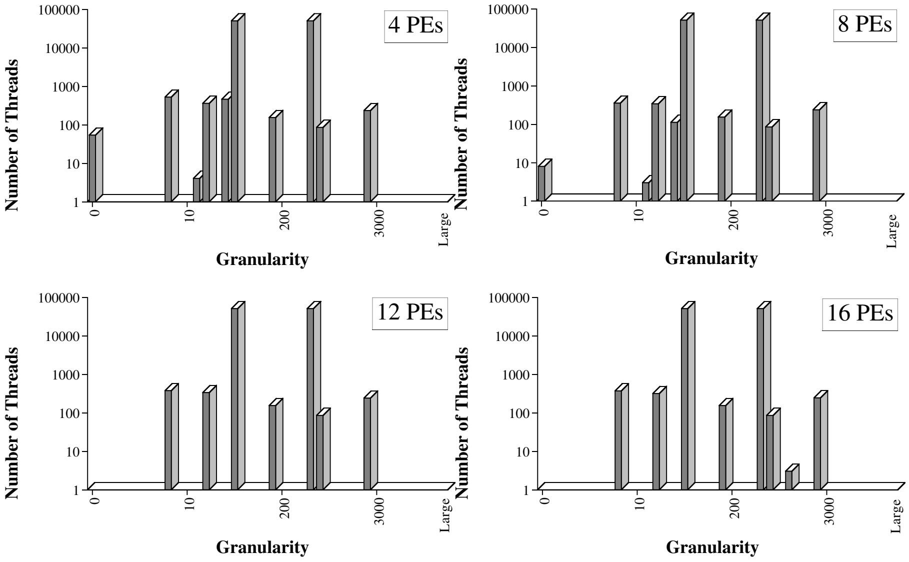

Fig. 5. Granularity Profile: Ray Tracer, Inner Spark, Local Sparking

The results for local sparking are similarly disappointing. In fact, the time performance for local sparking is worse than that for global sparking in all cases, being only 92% as efficient in the 16-PE case. Compared with local sparking in the "both sparks" case, there are many more small and medium grained tasks. In fact, the profiles resemble those for the global sparking case, but with several more small tasks. This is clearly disastrous: local sparking has failed to locate profitable local tasks. Studying the parallelism profile for a typical case (Figure 6) shows that the communication costs are extremely high even with local sparking. This suggests that for this example, significant parts of the data structures must be shared between different tasks, and that therefore there is no advantage to keeping tasks on local data structures.

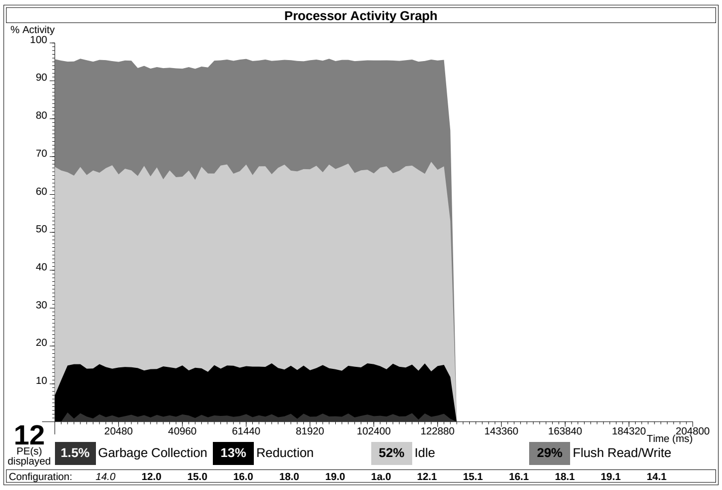

Fig. 6. Parallel Profile: Ray Tracer, Inner Spark, Local Sparking

Ray Tracer, Outer Spark It proved impossible to run the outer-spark-only version of the ray tracer on GRIP without also enabling local sparking, and indeed we initially rejected this as a viable strategy based on our experience with the global sparking case. This was probably due to excessive synchronisation costs through the shared data, and perhaps also to high garbage collection costs which are aggravated by the small GRIP dynamic heap (around 200KB is available after allocation of program and static data). Even with local sparking, garbage collection costs account for around 30% of the total execution time. In spite of this, overall execution times compare very favourably with those for the other two cases, as Table 1 shows. Times are in seconds. This dramatic improvement in execution time compared with the alternative spark strategies

PEs	Both Sparks (Global/Local)	Inner Spark (Global/Local)	Outer Spark (Local Only)
1	331.6/142.8	331.6/331.6	104.8
2	176.9/97.3	204.5/215.1	53.2
4	86.7/54.1	106.1/114.1	27.0
8	46.0/33.8	60.8/65.4	15.0
12	34.0/27.9	58.6/63.9	10.4
6	27.2/25.1	58.4/63.6	8.3

Table 1. Comparison of Execution Times for all Ray Tracer Versions, Local Sparking

is reflected in the granularity profiles, which show very few small tasks, some medium tasks, and several large tasks, for all processor configurations.

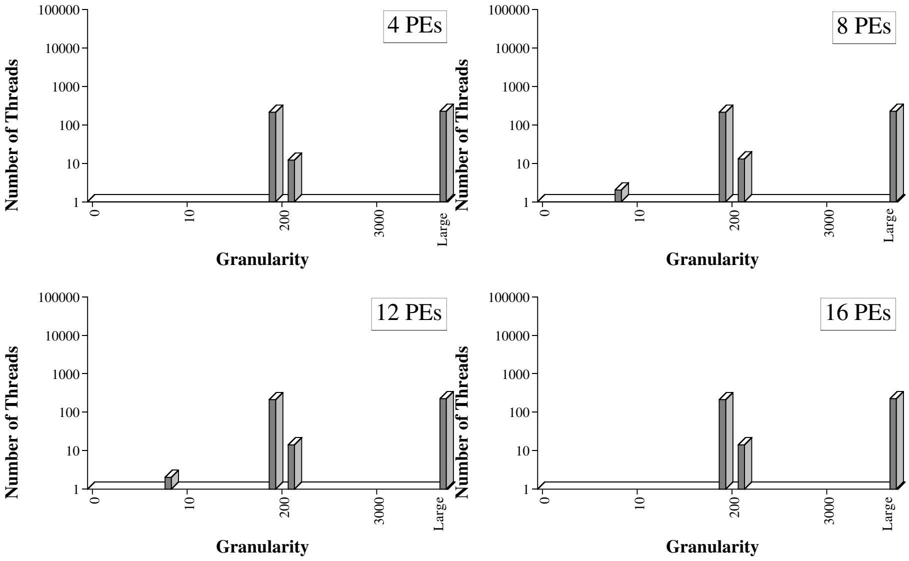

Fig. 7. Granularity Profile: Ray Tracer, Outer Spark, Local Sparking

Summary of Ray-Tracing Results One interesting result to emerge from our study of the ray tracing example is that particular spark sites have distinctive "signatures", or patterns of grain size, even across varying numbers of processors. Given that our first ray tracing spark strategy (spark both) is a combination of the other two strategies (spark inner and spark outer), we might expect that its signature would also be a rational combination of these. In fact, the combination is a fairly good predictor, as Figure 8 shows. The graph on the left shows the result of using both sparks; the graph to the right shows the sum of using just the inner spark or just the outer spark. The combined graph overestimates the medium-sized tasks at the expense of certain types of small task.

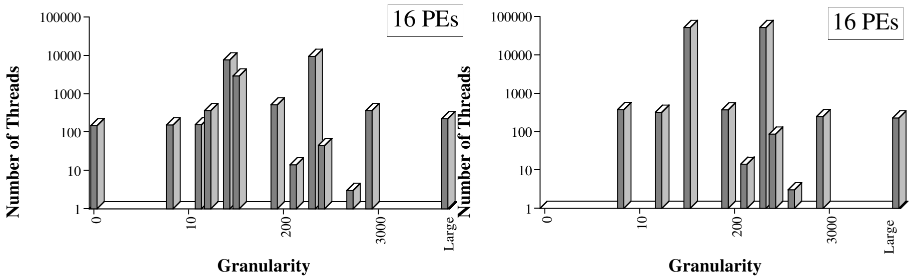

Fig. 8. Ray Tracing, 16 PEs: Both Sparks v. Merged Results

3.2 Using Granularity

Granularity information can be exploited in several ways. The most obvious is in determining which sparks should be chosen as tasks if throttling occurs. Unfortunately, even this apparently simple use is complicated by several factors:

- The same spark may generate tasks of different sizes due to different concrete arguments, or through lazy evaluation causing some part of a task to be executed elsewhere.
- A spark may generate a small task, but that task may then introduce many more sparks directly or indirectly.

Hence a purely static technique such as a static analysis of grain size, or even a feedback mechanism based on execution profiles, such as that proposed by Sargeant [20], seems unlikely to yield extremely good results in general. Continuous dynamic performance monitoring, as suggested by e.g. Aharoni, Farber and Barak [2], and as explored by our work on GRIP [8], also seems likely to react too slowly to changes in the global runtime situation for some programs.

A combination of static and dynamic approaches thus appears likely to be necessary for optimum performance. Ideally a granularity analysis could be used to detect potentially good sparks at compile time. Such an analysis need not be as complicated as the complexity models proposed by e.g. Sands [18] since an occasional inaccurate cost estimate will not normally have a significant impact on overall performance unless only a few scheduling decisions are made [6, 3]. Of course, lazy evaluation does complicate the situation, but perhaps not significantly more than non-determinism and back-communication in parallel logic programs [5]. Per-processor granularity information can be used to detect load imbalances. For example, the processor which runs the initial task frequently executes one very large task, and a few smaller ones.

4 Conclusions

We have presented detailed granularity results for several versions of a parallel functional program running on the GRIP machine. Space constraints prevent similarly detailed studies of other programs, but these results are not unrepresentative. In the best case, local sparking carries an estimated 13% overhead over sequential execution, a remarkably low figure. Compared with global sparking, local sparking costs between 42% and 109% as much.

Since these results are obtained from a physical machine rather than a simulation, they are to some extent architecture-dependent. However, the distinction between global and local spark pools is likely to be important across a range of architectures, if they support a split local/global data model. Our results show clearly that for the program studied (and for others whose results are not shown here), there can be a significant time advantage in maintaining local spark pools for small numbers of PEs. The advantage disappears as the number of PEs increases. The exact cross-over point obviously depends on the speed of the PEs and the communications systems as well as the amount of data which must be shared. We note that while GRIP's PEs are now relatively slow, the communication subsystem is relatively fast, so we might expect results similar to ours to hold for more recent designs such as the Meiko CS/2, or ICL Goldrush machines. We are in the process of porting our software to such a machine at present. In the medium-term, we therefore expect to be able to provide direct comparisons between GRIP and other architectures, but this is not practical at present. There are many other approaches to parallel implementation of functional languages, e.g. *skeletons* [4], explicit parallel control functions [12, 11], or load distribution annotations [1]. Our approach has the advantages of program simplicity, runtime flexibility, and automatic scalability, at the cost of losing direct programmer control.

Clearly, large-grained tasks are tremendously important. While local sparking gave a much better granularity profile for the inner-spark version of the Ray Tracer as far as the elimination of small-grained tasks was concerned, the overall time performance did not improve. This was probably due to the simultaneous elimination of the large-grained tasks introduced by the outer sparks. Executing only these large outer sparks, with a local sparking strategy, gives a significant increase in performance. For a global sparking strategy, however, executing these sparks was terminally catastrophic. This was probably due to extremely high synchronisation costs.

An interesting result to emerge from this research is the identification of granularity profiles for particular spark placements across varying numbers of PEs. This suggests that scheduling strategies based on analysis of granularity patterns for one processor configuration should also apply to other configurations. As further research, it would be a valuable exercise to consider granularity profiles across a range of data set sizes and programs. If granularity profiles are broadly similar for varying sizes of data set, then they may well prove useful for feedback into static compiler analyses.

References

1. Achten P, "Annotations for Load Distribution", *Proc 3rd. Intl. Workshop on Parallel Impl. of Funct. Langs.*, Glaser H (ed.), Univ. of Southampton, (June 1991).
2. Aharoni G, Feitelson DG, and Barak A, "A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs", *Journal of Functional Programming* **2:4**, (October 1992), pp. 387–405.
3. Burton FW and Rayward-Smith VJ, "Worst Case Scheduling for Parallel Functional Programming", *To appear in Journal of Functional Programming*, (1994).
4. Cole MI, *Algorithmic Skeletons*, Pitman/MIT Press, (1988).
5. Debray, SK, Lin N-W, and Hermenegildo, "Task Granularity Analysis in Logic Programs", *Proc. ACM Conf. on Prog. Lang. Design and Impl.*, (June 1990).
6. Eager DL, Zahorjan J and Lazowska ED, "Speedup versus efficiency in parallel systems", *Dept of Computational Science, University of Sasketchewan*, 1986.
7. Hammond K, and Peyton Jones SL, "Some Early Experiments on the GRIP Parallel Reducer", *Proc 2nd. Intl. Workshop on Parallel Implementation of Functional Languages*, Plasmeijer MJ (ed.), University of Nijmegen, (July 1990).
8. Hammond K, and Peyton Jones SL, "Profiling Scheduling Strategies on the GRIP Multiprocessor", *Proc 4th. Intl. Workshop on Parallel Implementation of Functional Languages*, Küchen H (ed.), Aachen University, (September 1992).
9. Hammond K, "Implementation on a Parallel Machine: GRIP", In *Functional Languages Applied to Realistic Examplars: the FLARE Project*, Runciman C and Wakeling D (eds.), Cambridge University Press, to appear (1994).
10. Hartel P, and Vree WG, "Benchmarking Lazy Functional Languages", *Proc. FPCA 93*, Copenhagen, Denmark, (June 1993).
11. Hudak P, "Para-functional programming", *IEEE Computer* **19(8)**, (August 1986).
12. Kelly PHJ, *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman/MIT Press, (1989).
13. Mohr E, Kranz DA, and Halstead RH, "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems*, **2(3)**, (July 1991).
14. Ostheimer G, *Parallel Functional Programming for Message-Passing Multiprocessors*, PhD Thesis, University of St. Andrews, (March 1993).
15. Peyton Jones SL, Clack C, Salkild J, and Hardie, M, "GRIP – a high-performance architecture for parallel graph reduction", *Proc FPCA '87*, Portland, Oregon, Kahn G (ed.), Springer-Verlag LNCS 274, (1987).
16. Peyton Jones SL, Hall CV, Hammond K, Partain WD and Wadler PL, "The Glasgow Haskell Compiler: a Technical Overview", *Proc. JFIT 93*, Keele, (March 1993).
17. Ruggiero CA, and Sargeant J, "Control of parallelism in the Manchester dataflow machine", *Proc FPCA '87*, Springer Verlag LNCS 274, (1987), pp. 1-15.
18. Sands DA, "A Naive Model of Time Complexity and its Theory of Cost Equivalence", Internal Report, DIKU, Copenhagen, (June 1993).
19. Sargeant J, and Watson I, "Some Experiments in Controlling the Dynamic Behaviour of Parallel Functional Programs", *Proc 3rd. Intl. Workshop on Parallel Impl. of Funct. Langs.*, Glaser H (ed.), Univ. of Southampton, (June 1991).
20. Sargeant J, "Improving Compilation of Implicit Parallel Programs by Using Runtime Information", *Proc. Workshop on Compilation of Symbolic Langs. for Parallel Computers*, Argonne National Lab. Technical Report ANL–91/34, pp. 129–148.
21. Siedl H, and Wilhelm R, "Probabilistic Load Balancing for Parallel Graph Reduction", *Proc IEEE TENCON 89*, Bombay, India, (November 1989), pp. 879-884.