Data Distribution at Run-Time: Re-Using Execution Plans

Olav Beckmann and Paul H J Kelly

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, U.K.
{ob3,phjk}@doc.ic.ac.uk

Abstract. This paper shows how data placement optimisation techniques which are normally only found in optimising compilers can be made available efficiently in run-time systems. We study the example of a delayed evaluation, self-optimising (DESO) numerical library for a distributed-memory multicomputer. Delayed evaluation allows us to capture the control-flow of a user program from within the library at run-time, and to construct an optimised execution plan by propagating data placement constraints backwards through the DAG representing the computation to be performed.
In loops, essentially identical DAGs are likely to recur. The main concern of this paper is recognising opportunities where an execution plan can be re-used. We have adapted both conventional parallelising compiler techniques and hardware dynamic branch prediction techniques in order to ensure that our run-time optimisations need not perform any more work than a parallelising compiler would have to do unless there is a prospect of better performance.

1 Introduction

Parallel libraries have two major advantages as a parallel programming model. Firstly, they are convenient because user programs simply call library operators which hide all aspects of parallelism internally. Secondly, there is ample evidence [6] that compiled programming models do not yet get close to purpose-built libraries in terms of performance. It is therefore often worthwhile to invest in a highly optimised, machine-specific library of common numerical subroutines. Hitherto, the disadvantage with such libraries has been that opportunities for optimisation across library calls have been missed.

In this paper, we study a delayed evaluation, self-optimising (DESO) vector-matrix library for a distributed-memory multicomputer. The idea of DESO is to delay actual execution of function calls for as long as possible. Evaluation is forced by the need to access array elements[1]. Delayed evaluation then provides the opportunity at run-time to construct an execution plan which minimises

[1] The most common reasons for accessing array elements are output and conditional tests. We will refer to statements where this happens as *force points*.

redistribution by propagating data placement constraints backwards through the DAG representing the computation to be performed. We will study a detailed example in Section 2.

Key issues. The main challenge in optimising at run-time is that the optimiser itself has to be very efficient. We achieve this by

- Working from aggregate loop nests, which have been optimised in isolation and which are not re-optimised at run-time. It is precisely the point of the library approach that we have invested in an implementation of selected operators which have been pre-optimised offline.
- Using a purely mathematical formulation for data distributions, which allows us to calculate, rather than search for optimal placements. We will not expand on this aspect of our methodology here.
- Re-using execution plans for previously optimised DAGs. A value-numbering scheme is used to capture cases where this may be possible. The value numbers are used to index a cache of optimisation results, and we use a technique adapted from hardware dynamic branch prediction for deciding whether to further optimise DAGs we have previously encountered.

Context and Structure of this Paper. This paper builds on related work in the field of automatic data placement [8], run-time parallelisation [4, 9, 10] and conventional compiler and architecture technology [1, 5]. In our earlier paper [3] we described the basic idea of a lazy, self-optimising parallel vector-matrix library. In this current paper, we extend that work by presenting the techniques we use to avoid re-optimisation of previously encountered problems. Below, we begin in Section 2 by discussing two alternative strategies for run-time optimisation. Following that, Section 3 presents our techniques for avoiding re-optimisation where appropriate. Section 4 shows performance results and Section 5 concludes.

2 Issues in Run-Time Optimisation

This section discusses and compares two different basic strategies for performing run-time optimisation. We will refer to them as "Forward Propagation Only" and "Forward And Backward Propagation". We use the conjugate gradient iterative algorithm for solving linear systems $Ax - b = 0$ to illustrate both strategies. The pseudocode for this algorithm can be found in Figure 2. We use the following terminology: n is the number of operator calls in a sequence, a the maximum arity of operators, m is the maximum number of different methods per operator. If we work with a fixed set of data placements, s is the number of different placements, and in DAGs, d refers to the degree of the shared node (see [8]) with maximum degree.

Forward Propagation Only. This is the only strategy open to us if we perform run-time optimisation of a sequence of library operators under *strict evaluation*. The strategy is illustrated in Figure 1: we optimise the placement of each node

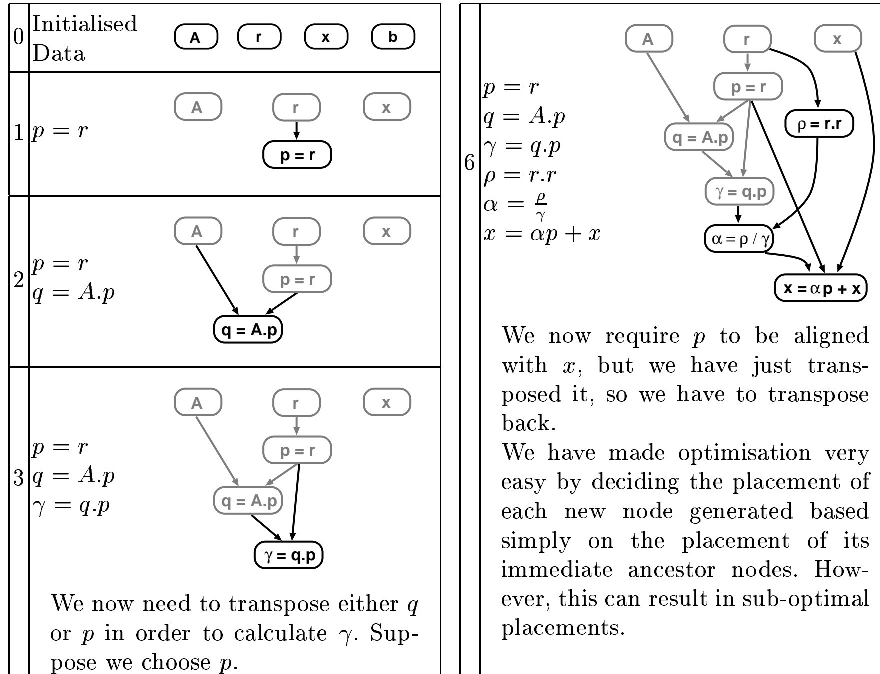

Fig. 1. Run-time optimisation of the first iteration of the CG algorithm (see Figure 2) with *Forward Propagation Only.*

based on information purely about its ancestors. The total optimisation time for a sequence of operator calls under this strategy has *linear* complexity in the number of operators. However, as we have illustrated in Figure 1, the price we pay for using such an algorithm is that it may give a significantly suboptimal answer. This problem is present even for trees, but it is much worse for DAGs since shared nodes are not taken into account properly.

Forward And Backward Propagation. Delayed evaluation gives us the opportunity to propagate placement constraint information backwards through a DAG since we accumulate a full DAG before we begin to optimise. This type of optimisation is much more complex than *Forward Propagation Only.* Mace [8] has shown it to be NP-complete for general DAGs, but presents algorithms with complexity $O((m + s^2)n)$ for trees and with complexity $O(n \times s^{d+1})$ for a restricted class of DAG. The point to note here, though, is that *Forward and Backward Propagation* does give us the opportunity to find the optimal solution to a problem, provided we are prepared to spend the time required.

3 Re-Using Execution Plans

The previous section has shown how delayed evaluation gives us the opportunity to derive optimal execution plans, but potentially at not insignificant cost. In

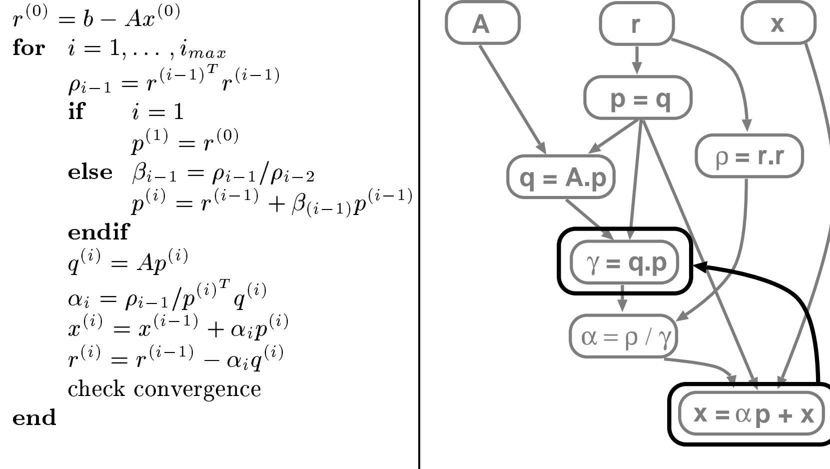

Fig. 2. Left: Pseudocode for the conjugate gradient iterative algorithm. **Right:** Optimisation with *Forward And Backward Propagation*: we take account of the use of p in the update of x in deciding the correct placement for the earlier calculation of γ.

real programs, essentially identical DAGs often recur. In such situations, our delayed evaluation, run-time approach is set to suffer a significant performance disadvantage over compile-time techniques unless we can reuse the results of previous optimisations we have performed.

This section shows how we can ensure that our optimiser does not have to carry out any more work than an optimising compiler would have to do, unless there is the prospect of better performance than the compiler could deliver. We discuss first the problem of how to recognise a DAG, i.e. optimisation problem, which we have encountered before and then the issue of whether to optimise further, or, re-optimise, such a DAG.

Recognising Opportunities for Reuse. The full optimisation problem is a large structure. To avoid having to traverse it for comparing with previously encountered DAGs, we derive a hashed "value number" [1, 5] for each node.

- Our value numbers have to encode data placements and placement constraints, not actual data values. For nodes which are already evaluated, we simply apply a hash function to the placement descriptor of that node. For nodes which are not yet evaluated, we have to apply a hash function to the *placement constraints* on that node.
- The key observation is that by seeking to take account of all placement constraints on a node, we are in danger of deriving an algorithm for calculating value numbers which has the same O-complexity as *Forward and Backward Propagation* optimisation algorithms: each node in a DAG can potentially exert a placement constraint over every other node.

- Our algorithm for calculating value numbers is therefore based on *Forward Propagation Only*: we calculate value numbers for unevaluated nodes by applying a hash function to the placement constraints deriving from their immediate ancestors.
- Since we do not store the "full DAG" information, we cannot easily detect hash conflicts. We return to this point shortly.

When to Re-Use and When to Optimise. Because our value numbers are calculated on *Forward Propagation Only* information, we have to address the problem of how to handle those cases where nodes which have identical value numbers are used in a different context later on; in other words, how to to avoid the drawbacks of *Forward Propagation Only* optimisation. This is a branch-prediction problem, and we use a technique adapted from hardware dynamic branch prediction (see [7]) for predicting heuristically whether identical value numbers will result in identical future use of the corresponding node and hence identical optimisation problems.

Caching Execution Plans. Value numbers and 'dynamic branch prediction' together provide us with a fairly reliable mechanism for recognising the fact that we have encountered a node in the same context before. Assuming that we optimised the placement of that node when we first encountered it, our task is then simply to re-use the placement which the optimiser derived. We do this by using a "cache" of optimised placements, which is indexed by value numbers. Each cache entry has a valid-tag which is set by our branch prediction mechanism.

Competitive Optimisation. As we showed in Section 2, full optimisation based on *Forward And Backward Propagation* can be very expensive. Each time we invoke the optimiser on a DAG, we therefore only spend a limited time optimising that DAG. For a DAG which we encounter only once, this means that we only spend very little time trying to eliminate the worst redistributions. For DAGs which recur, our strategy is to gradually improve the execution plan used until our optimisation algorithm can find no further improvements.

The final point we need to address is how to handle hash conflicts. The result of a hash conflict on a value number will be that we use a sub-optimal placement for a node which we had previously optimised. In order to detect this, our system has been instrumented to record the communication cost of executing a DAG under an optimised execution plan. An increase in this cost on a "re-use" of that plan indicates a hash conflict.

Summary. We use the full optimisation information, i.e. *Forward and Backward Propagation*, to optimise. We obtain access to this information by delayed evaluation. We use a scheme based on *Forward Propagation Only*, with linear complexity in program length, to ensure that we re-use the results of previous optimisations.

	P	N	*Comp.*	*Memory*	*Overh.*	*Comms.*	*Opt.*	*Total*	*Speedup*
N	1	256	51.14	0.81	3.97	0.00	0.00	55.93	·
O	1	256	51.22	0.81	3.88	0.00	3.59	59.50	0.94
C	1	256	51.13	0.79	3.25	0.00	0.30	55.47	1.01
N	4	512	51.30	1.03	3.47	30.51	0.00	86.30	·
O	4	512	51.79	0.86	3.06	22.35	3.73	81.79	1.06
C	4	512	51.69	0.94	2.45	21.94	0.31	77.32	1.12
N	9	768	51.72	1.12	3.46	34.78	0.00	91.10	·
O	9	768	51.96	0.91	3.08	26.06	3.74	85.75	1.06
C	9	768	51.90	0.98	2.46	25.91	0.31	81.55	1.12
N	16	1024	52.05	1.03	3.48	45.37	0.00	101.93	·
O	16	1024	52.29	0.88	3.16	35.99	3.82	96.14	1.06
C	16	1024	52.13	0.95	2.49	35.65	0.31	91.53	1.11

Table 1. Time in milliseconds for 20 iterations of Conjugate Gradient, with a convergence test every 10 iterations, on the AP3000, with 300MHz UltraSparc-2 nodes (average figures, outlying points were omitted). N denotes timings without any optimisation, O timings with optimisation but no caching, and C timings with optimisation and caching of optimisation results. *Memory* shows time spent in `malloc()` and `free()`, *Overhead* the cost of maintaining data distribution descriptors and suspended library calls at run-time. *Speedup* is the speedup due to optimisation and execution plan re-use.

4 Performance

In this Section, we show performance figures for our library on the Fujitsu AP3000 multicomputer here at Imperial College. As a benchmark we used the conjugate gradient iterative algorithm. The pseudo-code for the algorithm and the source code when implemented using our library were shown in Figure 2 and the timing data are in Table 1.

- Our optimiser avoids two out of three vector transpose operations per iteration. This can not be seen from the data in Table 1, it was determined analytically and by tracing communication.
- Optimisation achieves a reduction in communication time of between 20% and 30%. We do not achieve more because a significant proportion of the communication in this algorithm is due to reduce-operations which are unaffected by our current optimisations.
- Run-time overhead and optimisation time are independent of the amount of parallelism used. We suspect that the slight difference is due to cache effects.
- The reason why run-time overhead is reduced by optimisation is that performing fewer redistributions also results in spending less time inspecting data placement descriptors. Caching achieves a further reduction in overhead; this is because it is cheaper to read placement descriptors from cache than to generate them by function calls.
- Without caching of optimisation results, we achieve an overall speedup of around 6%. On platforms which have less powerful processors than the 300MHz UltraSparc-2 nodes we use here, the cost of optimising afresh each time can easily outweigh the benefit of reduced communication.

– With caching of optimisation results, the time we spend optimising is negligible, and we achieve overall speedups of around 12%.

5 Conclusions

We have presented a technique for interprocedural data placement optimisation which exploits run-time control-flow information and is applicable in contexts where the calling program cannot be analysed statically. We present preliminary experimental evidence that the benefits can easily outweigh the run-time costs.

Related Work. There is a huge body of work on data mappings for regular problems, [2] is but one example. Our work relies on this in producing optimised implementations for library operators. However, the problem we seek to address in this paper is different — how to perform interprocedural optimisation over a sequence of such pre-optimised operators.

Saltz *et al.* [10] address the basic problem of how to parallelise loops where the dependence structure is not known statically. Loops are translated into an *inspector* loop which determines the dependencies at run-time and constructs a schedule, and an *executor* loop which carries out the calculations planned by the inspector. Saltz *et al.* discuss the possibility of reusing a previously constructed schedule, but rely on user annotations for doing so. Ponnusamy *et al.* [9] propose a simple conservative model which avoids the user having to indicate to the compiler when a schedule may be reused. Benkner *et al.* [4] describe the reuse of parallel schedules via explicit directives in HPF+: `REUSE` directives for indicating that the schedule computed for a certain loop can be reused and `SCHEDULE` variables which allow a schedule to be saved and reused in other code sections.

Value numbering schemes were pioneered by Ershov [5], who proposed the use of "convention numbers" for denoting the results of computations and avoid having to recompute them. More recent work on this subject has been done, e.g., by Alpern *et al.* [1].

Run-Time vs. Compile-Time Optimisation. The particular example studied in this paper would have been amenable to compile-time analysis. We refer back to Section 1 and the arguments of convenience and efficiency we outlined there for why we parallelise this application via a parallel library. using a library then forces us to optimise at run-time. Further, we have shown that a runtime optimiser need not actually perform any more work than a compiler.

To compare the quality of run-time and compile-time schedules, consider the loop opposite, assuming that there are no force-points inside the loop and that the loop is encountered a number of times, evaluation being forced after the loop each time.

```
for(i = 0; i < N; ++i) {
  if <unknown condition>
    <do A>
  else
    <do B>
}
```

This loop can potentially have 2^N control-paths. A compile-time optimiser would have to find one compromise execution plan for all invocations of the

loop. With our approach, we optimise the actual DAG which is generated on each occasion. If the number of different DAGs is high, compile-time methods would probably have the edge over ours, since we cannot reuse execution plans. If, however, the number of different DAGs is small, our execution plans for the *actual* DAGs will be superior to compile-time compromise solutions, and by reusing them, we limit the time spent optimising.

Future work. The most exciting next step is to store cached execution plans persistently, so that they can be reused subsequently for this or similar applications. Although we can derive some benefit from exploiting run-time control-flow information, we have the opportunity to make run-time optimisation decisions based on run-time properties of data; we plan to extend this work to address sparse matrices shortly. The run-time system has to make on-the-fly data placement decisions. An intriguing question raised by this work is to compare this with an optimal off-line schedule.

Acknowledgements. This work was partially supported by the EPSRC, under the Futurespace and CRAMP projects (refs. GR/J 87015 and GR/J 99117). We thank the Imperial College Parallel Computing Centre for the use of their AP3000 machine.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equalities of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, Jan. 1988.
2. J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. *SIGPLAN Notices*, 30(8):166–178, Aug. 1995.
3. O. Beckmann and P. H. J. Kelly. Runtime interprocedural data placement optimisation for lazy parallel libraries (extended abstract). In Lengauer *et al.*, editor, *Proceedings of Euro-Par '97, Passau, Germany*, number 1300 in LNCS, pages 306–309. Springer Verlag, Aug. 1997.
4. S. Benkner, P. Mehrotra, J. V. Rosendale, Zima, and Hans. High-level management of communication schedules in HPF-like languages. Technical Report TR-97-46, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, USA, Sept. 1997.
5. A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958. Three figures from this article are in CACM 1(9):16.
6. W. D. Gropp. Performance driven programming models. In *MPPM'97, Proceedings of the* 3rd *International Working Conference on Massively Parallel Programming Models*, London, U.K., Nov. 1997. To appear.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantative Approach.* Morgan Kaufman, San Mateo, California, 1st edition, 1990.
8. M. E. Mace. *Storage Patterns in Parallel Processing.* Kluwer, 1987.
9. R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 361–370, New York, NY 10036, USA, Nov. 1993. ACM Press.
10. J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.