Optimising Data-Parallel Programs Using the BSP Cost Model

D.B. Skillicorn[1] M. Danelutto, S. Pelagatti and A. Zavanella[2]

[1] Department of Computing and Information Science
Queen's University, Kingston, Canada
[2] Dipartimento di Informatica
Universitá di Pisa, Corso Italia 40
56125 Pisa, Italy

Abstract. We describe the use of the BSP cost model to optimise programs, based on skeletons or data-parallel operations, in which program components may have multiple implementations. BSP's view of communication transforms the problem of finding the best implementation choice for each component into a one-dimensional minimisation problem. A shortest-path algorithm that finds optimal implementations in time linear in the number of operations of the program is given.

1 Problem Setting

Many parallel programming models gain expressiveness by raising the level of abstraction. Important examples are skeletons, and data-parallel languages such as HPF. Programs in these models are compositions of moderately-large building blocks, each of which hides significant parallel computation internally.

There are typically multiple implementations for each of these building blocks, and it is straightforward to order these implementations by execution cost. What makes the problem difficult is that different implementations require different arrangements of their inputs and outputs. Communication steps must typically be interspersed to rearrange the data between steps.

Choosing the best global implementation is difficult because the cost of a program is the sum of two different terms, an execution cost (instructions), and a communication cost (words transmitted). In most parallel programming models, it is not possible to convert the communication cost into comparable units to the execution cost because the message transit time depends on which other messages are simultaneously being transmitted, and this is almost impossible to determine in practice.

The chief advantage of the BSP cost model, in this context, is that it accounts for communication (accurately) in the same units as computation. The problem of finding a globally-optimal set of implementation choices becomes a one-dimensional minimisation problem, in fact with some small adaptations, a shortest path problem.

The complexity of the shortest path problem with positive weights is quadratic in the number of nodes (Dijkstra's algorithm) or $\mathcal{O}((e+n)\log n)$, where e is the

number of edges, for the heap algorithm. We show that the special structure of this particular application reduces the problem to shortest path in a layered graph, with complexity linear in the number of program steps.

BSP's cost model assumes that the bottleneck in communication performance is at the processors, rather than in the network itself. Thus the cost of communication depends on the fan-in and fan-out of data at each processor, and a single architectural parameter, g, that measures effective inverse bandwidth (in units of time per word transferred). Such a model is extremely accurate for today's parallel computers. Other programming models can use the technique described here to the extent that the BSP cost model reflects their performance. In practice, this requires being able to decide which communication actions will occur in the same time frame. This will almost certainly be possible for P3L, HPF, GoldFish, and other data-parallel models.

In Section 2 we show how the BSP cost model can be used to model the costs of skeletons or data-parallel operations. In Section 3, we present an optimisation algorithm with linear complexity. In Section 4, we illustrate its use on a non-trivial application written in the style of P3L. In Section 5, we review some related work.

2 Cost Modelling

Consider the program in Figure 1, in which program operations, BSP supersteps [7], are shown right to left in a functional style. Step A has two potential implementations, and we want to account for the total costs of the two possible versions of the program.

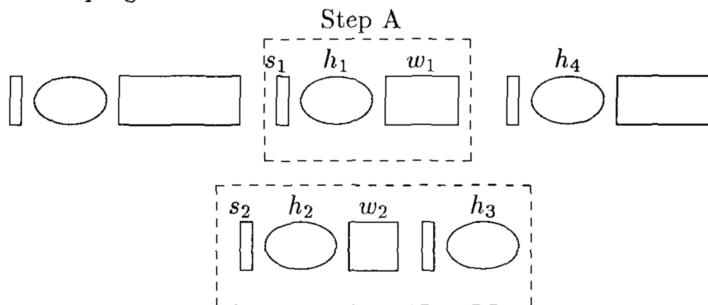

Fig. 1. Making a Local Optimisation. Rectangles represent computation, ovals communication, and bars synchronisation. Each operation is labelled with its total cost parameter. Both implementations may contain multiple supersteps although they are drawn as containing one. The alternate implementation of step A requires extra communication (and hence an extra barrier) to place data correctly. In general, it is possible to overlap this with the communication in the previous step.

The communication stage of the operation preceding step A arranges the data in a way suitable for the first implementation of step A. Using the second imple-

mentation for step A requires inserting extra data manipulation code to change this arrangement. This extra code can be merged with the communication at the end of the preceding superstep, saving one barrier synchronisation, and potentially reducing the total communication load. For example, identities involving collective communication operations may allow some of the data movements to 'cancel' each other.

We can describe the cost of these two possible implementations of Step A straightforwardly in the BSP cost framework. The cost of the original implementation of Step A is $w_1 + h_1 g + s_1 l$ and the cost of the new implementation is $w_2 + h_2 g + s_2 l$ plus a further rearrangement cost $h_3 g + l$ To determine the extent to which the $h_3 g$ cost can be overlapped with the communication phase of the previous step we must break both costs down into fan-out costs and fan-in costs. So suppose that h_4 is the communication cost of the preceding step and

$$h_3 = \max(h_3^{out}, h_3^{in}) \qquad h_4 = \max(h_4^{out}, h_4^{in})$$

since fan-ins and fan-outs are costed additively. The cost of the combined, overlapped communication is

$$\max(h_3^{out} + h_4^{out}, h_3^{in} + h_4^{in})g$$

The new implementation is cheaper than the old, therefore, if

$$w_2 + h_2 g + s_2 l + \max(h_3^{out} + h_4^{out}, h_3^{in} + h_4^{in})g < w_1 + h_1 g + s_1 l + h_4 g$$

3 Optimisation

The analysis in the previous section shows how a program is transformed into a directed acyclic graph in which the nodes are labelled with the cost of computation (and synchronisation) and the edges with the cost of communication. Alternate implementation corresponds to choosing one path between two points rather than another. In what follows, we simply assume that different implementations for each program step are known, without concerning ourselves with how these are discovered.

Given a set of alternate implementations for each program step, the communication costs of connecting a given implementation of one step with a given implementation of the succeeding step can be computed as outlined above. If there are a alternate implementations for each program step, then there are a^2 possible communication patterns connecting them. If the program consists of n steps, then the total cost of setting up the graph is $(n-1)a^2$.

We want to find the shortest path through this layered graph. Consider the paths from any implementation block at the end of the program. There are (at most) a paths leading back to implementations of the previous step. The cost of a path is the sum of the costs of the blocks it passes through and the edges it contains. After two communication steps, there are a^2 possible paths, but only (at most) a of them can continue, because we can discard all but the cheapest

of any paths that meet at a block. Thus there are only a possible paths to be extended beyond any stage. Eventually, (at most) a paths reach the start of the program, and the cheapest overall can be selected. The cost of this search is $(n-1)a^2$, and it must be repeated for each possible starting point, so the overall cost of the optimisation algorithm is $(n-1)a^3$. This is linear in the length of the program being optimised. The value of a is typically small, perhaps 3 or 4, so that the cubic term in a is unlikely to be significant in practice.

4 An Example

We illustrate the approach with a non-trivial program, Conway's game of Life, expressed in the intermediate code used by the P3L compiler[5,1]. The operations available to the P3L compiler are:

distribute :: Distr_pattern $\rightarrow$ SeqArray α $\rightarrow$ ParArray (SeqArray α)
gstencil :: ParArray α $\rightarrow$ Distr_pattern $\rightarrow$ Stencil $\rightarrow$ Distr_pattern $\rightarrow$ ParArray α
lstencil :: ParArray α $\rightarrow$ Stencil $\rightarrow$ Distr_pattern $\rightarrow$ ParArray (SeqArray α)
map :: ParArray α $\rightarrow$ $(\alpha \rightarrow \beta)$ $\rightarrow$ ParArray β
reduce :: ParArray α $\rightarrow$ $(\alpha \rightarrow \alpha \rightarrow \alpha)$ $\rightarrow$ α
reduceall :: ParArray α $\rightarrow$ $(\alpha \rightarrow \alpha \rightarrow \alpha)$ $\rightarrow$ ParArray α
gather :: ParArray α $\rightarrow$ Distr_pattern $\rightarrow$ SeqArray α

where distribute distributes an array to a set of worker processes according to a distribution pattern (Distr_pattern), gstencil fetches data from neighbour workers according to a given stencil, lstencil arranges the data local to a worker to have correct haloes in a stencil computation, map and reduce are the usual functional operations, reduceall is a reduce in which all the workers get the result, and gather collects distributed data in an array to a single process. Possible intermediate P3L code for the game of life is:

```
1. d1 = (block_block p)
2. W = distribute d1 World
3. while NOT(C2)
4.        W1 = gstencil W d1 [(1,1),(1,1)] d1
5.        W2 = lstencil W1 [(1,1),(1,1)] d1
6.        (W3,C)= map (map update) W2
7.        C1 = map (reduce OR C)
8.        C2 = reduceall OR C
9. X = gather d1 W3
```

where `d1` is the distribution adopted for the `World` matrix (block,block over a rectangle of p processors), and `W` is the ParArray of the distributed slices of the initial `World` matrix. Line 4 gathers the neighbours from non-local processors. Line 5 creates the required copies. Line 6 performs all the update in parallel. Array `C` contains the boolean values saying if the corresponding element has changed. Line 9 gathers the global results. We concentrate on the computation inside the loop. It may be divided into three phases: Phase A gathers the needed values at each processor (Line 4). Phase B computes all of the updates in parallel (Lines 5–7). Phase C checks for termination: nothing has changed in the last iteration (Line 8). There are several possible implementations for these steps:

Operation	Computation Cost	Comment
A_1	l	direct stencil implementation
A_2	$2l$	gather to a single process then distribute
A_3	$3l$	gather to a single process then broadcast
B	$(9 + t_{\oplus})N^2 + l$	local update and reduce
C_1	$(p-1)t_{\mathrm{OR}}$	total exchange of results, then local reductions
C_2	$(\log p - 1)(l + t_{\mathrm{OR}})$	tree reduction

where $t_{\oplus}$ is the cost of a local update, t_{OR} is the time taken to compute a logical OR, and N is the length of the side of the world region held by each processor.

In A_1, all workers in parallel exchange the boundary elements $(4(N+1))$, requiring a single barrier (costing l). In A_2, all the results are gathered on a single processor and redistributed giving each worker its halo; this requires much more data exchange $(N^2p + 4(N+1))$ and two synchronisations $(2l)$. In A_3, data are collected as in A_2 and then broadcasted to all the workers. This requires three synchronisations $(3l)$, as optimal BSP broadcast is done in two steps [7]. The cost of the communication between different implementations is:

Pair	Communication Cost	Comment
(A_1, B)	$4(N+1)g$	send the halo of an $N \times N$ block
(A_2, B)	$N^2pg + 4(N+1)g$	gather and distribute the halo
(A_3, B)	$3N^2pg$	gather and broadcast, no haloes
(B, C_1)	$(p-1)g$	total exchange
(B, C_2)	$2g$	
$(C_1, *)$	0	data already placed as needed
$(C_2, *)$	$2(\log p - 1)g$	

where p is the number of target processors, and N the length of side of the block of `World` on each processor). The optimal solution depends on the values of g, l and s of the target parallel computer. In general, for this example, A_1 is better than A_2 and A_3, but C_1 is better than C_2 only for small numbers of processors or networks with large values of g. For instance, if we consider a CRAY T3D (with values of $g \approx 0.35\mu s$/word, $l \approx 25\mu s$ and $s \approx 12Mflops$ as in [7]), C_1 is faster than C_2 when $p \leq 128$, so the optimal implementation is A_1-B-C_1. For larger numbers of processors, A_1-B-C_2 is the best implementation. For architectures with slower barriers (Convex Exemplar, IBM SP2 or Parsytec GC) the best solution is always A_1-B-C_1.

5 Related Work

The optimisation problem described here has, of course, been solved pragmatically by a large number of systems. Most of these use local heuristics and a small number of program and architecture parameters, and give equivocal results [4,3,2]. Unpublished work on P3L compiler performance has shown that effective optimisation can be achieved, but it requires detailed analysis of the

target architecture and makes compilation time-consuming. The technique presented here is expected to replace the present optimisation algorithm. A number of global optimisation approaches have also been tried. To [8] gives an algorithm with complexity n^2a to choose among block and cyclic distributions of data for a sequence of data-parallel collective operations. Rauber and Rünger [6] optimise data-parallel programs that solve numerical problems. Different implementations are homogeneous families of algorithms with tunable algorithmic parameters (the number of iterations, the number of stages, the size of systems), and the cost model is a simplification of LogP.

6 Conclusions

The contribution of this paper is twofold: a demonstration of the usefulness of the perspective offered by the BSP cost model in simplifying a complex problem to the point where its crucial features can be seen; and then the application of a shortest-path algorithm for finding optimal implementations of skeleton and data-parallel programs.

The known accuracy of the BSP cost model in practice reassures us that the simplified problem dealt with here has not lost any essential properties. Reducing the problem from a global problem with many variables to a one-dimensional problem that can be optimised sequentially makes a straightforward optimisation algorithm, with linear complexity, possible.

Acknowledgement. We gratefully acknowledge the input of Barry Jay and Fabrizio Petrini, in improving the presentation of this paper.

References

1. B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwer, Dordrecht, The Netherlands, 1997.
2. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.
3. S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based p3l compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, Australian National University, Canberra, 1997.
4. M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer-Verlag, August 1997.
5. S. Pelagatti. *Structured development of parallel programs*. Taylor&Francis, London, 1997.
6. T. Rauber and G. Rünger. Deriving structured parallel implementations for numerical methods. *The Euromicro Journal*, (41):589–608, 1996.
7. D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
8. H.W. To. *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Imperial College, 1995.