Analysing an SQL Application with a *BSPlib* Call-Graph Profiling Tool

Jonathan M.D. Hill, Stephen A. Jarvis,
Constantinos Siniolakis, and Vasil P. Vasilev

Oxford University Computing Laboratory, UK.

Abstract. This paper illustrates the use of a post-mortem *call-graph* profiling tool in the analysis of an SQL query processing application written using *BSPlib* [4]. Unlike other parallel profiling tools, the architecture independent metric of imbalance in size of communicated data is used to guide program optimisation. We show that by using this metric, *BSPlib* programs can be optimised in a portable and architecture independent manner. Results are presented to support this claim for unoptimised and optimised versions of a program running on networks of workstations, shared memory multiprocessors and tightly coupled distributed memory parallel machines.

1 Introduction

The Bulk Synchronous Parallel model [8,6] views a parallel machine as a set of processor-memory pairs, with a global communication network and a mechanism for synchronising all processors. A BSP program consists of a sequence of *supersteps*. Each superstep involves all of the processors and consists of three phases: (1) processor-memory pairs perform a number of computations on data held locally at the start of a superstep; (2) processors communicate data into other processor's memories; and (3) all processors barrier synchronise.

The BSP cost model [6] asserts that globally balancing computation and communication is the key to optimal parallel design. The rationale for balancing computation is clear as the barrier that marks the end of the superstep ensures that all processors have to wait for the slowest processor before proceeding into the next superstep. It is therefore desirable that all processes enter the barrier at about the same time to minimise idle time. In contrast, the need to balance communication is not so clear-cut. The BSP cost model implicitly asserts that the dominating cost in communicating a message is the cost of crossing the boundary of the network, rather than the cost of internal transit. This is largely true for today's highly-connected networks, as described in detail in [2]. Therefore, a good design strategy is to ensure that all processors inject the same amount of data into the network and are consequently equally involved in the communication.

This paper explores the use of a call-graph profiling tool [3] that exposes imbalance in either computation or communication, and highlights those portions of the program that are amenable to improvement. Our hypothesis is that by

minimising imbalance, significant improvements in the algorithmic complexity of parallel algorithms usually follow. In order to test the hypothesis, the call-graph profiler is used to optimise a database query evaluation program [7]. We show how the profiling tool helps identify problems in the implementation of a randomised sample sort algorithm. It is also shown how the profiling tool guides the improvement of the algorithm. Results are presented which show that significant performance improvements can be achieved without tailoring algorithms to a particular machine or architecture.

2 Profiling an SQL Database Application

A walk-through of the steps involved in optimising an SQL database query evaluation program is used to demonstrate the use of the call-graph profiling tool.

A series of relational queries were implemented in BSP. This was done by transcribing the queries into C function calls and then linking them with a *BSPlib* library of SQL-like primitives. The test program was designed to take as input a sequence of relations (tables), it would then processes the tables and yield as output a sequence of intermediate relations. The input relations were distributed among the processors using a simple block-cyclic distribution. Three input relations `ITEM`, `QNT` and `TRAN` were defined. Six queries were evaluated which created the following intermediary relations: (1) `TEMP1`, an aggregate sum and a "group-by" rearrangement of the relation `TRAN`; (2) `TEMP2`, an equality-join of `TEMP1` and `ITEM`; (3) `TEMP3`, an aggregate sum and group-by of `TEMP2`; (4) `TEMP4`, an equality-join of relations `TEMP3` and `QNT`; (5) `TEMP5`, a less-than-join of relations `TEMP4` and `ITEM`; and (6) a filter (`IN` "low 1%") of the relation `TEMP5`.

2.1 Using the Profiler to Optimise SQL Queries

Figure 2 shows a screen shot of a call-graph profile for the SQL query processing program running on a sixteen processor Cray T3E. The call-graph contains a series of interior and leaf nodes. The interior nodes represent procedures entered during program execution, whereas the leaf nodes represent the textual position of the end of a superstep, that is, the line of code containing a call to the barrier synchronisation function `bsp_sync`. The path from a leaf to the root of the graph identifies the nesting of procedure calls that were active when `bsp_sync` was executed. This path is termed a *call-stack* and a collection of call-stacks comprise a *call-graph*. The costs of shared procedures can be accurately apportioned to their parents via a scheme known as *inheritance* [5]. This is particularly important when determining how costs are allocated to library functions. Instead of simply reporting that all the time was spent in a parallel sorting algorithm for example, a call-graph profile also attributes costs to the procedures which contained calls to the sort routine.

In line with superstep semantics, the cost of all communication issued during a superstep is charged to the barrier that marks the end of the superstep. Similarly, all computation is charged to the end of the superstep. Leaf nodes

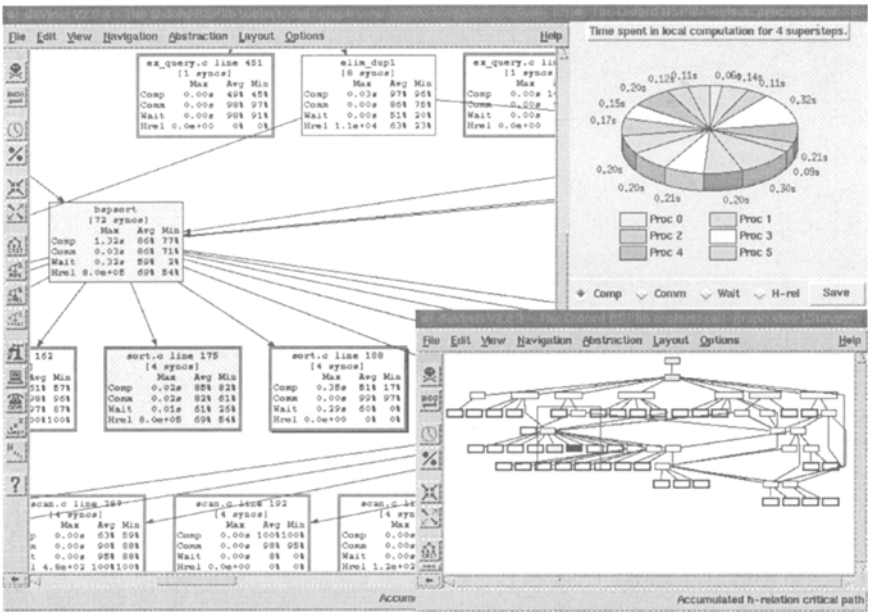

Fig. 1. Screen shot of the call-graph profiling tool

record the costs accrued during a superstep as: (i) the textual position of the **bsp_sync** call within the program; (ii) the number of times a particular superstep is executed; and (iii) summaries of computation cost, communication cost, idle time, and the largest amount of data either entering or leaving any process (a h-relation). These costs are given as a maximum, average, and minimum cost over p processors (the average and minimum costs are given as a percentage of the maximum). Interior nodes record similar information to leaf nodes, except that the label of the node is a procedure name and the accumulated cost is inherited from each of the supersteps executed during the lifetime of that procedure.

Critical paths are made visible by shading each of the nodes in the graph with a colour ranging from white to red. A red node corresponds to a bottleneck (or 'hot-spot'!) in the program. In the graphs that follow, the shading for each node is determined by the largest difference between the maximum and average cost, and also the largest percentage-wise deviation between maximum and average cost [3]. Terms of the form, (12% | 7%), are used to quantify this balance. This states that the average cost of executing a piece of code on p processors is 12% of the cost of the processor that spent the longest time performing the task; whereas the process with the least amount of work took only 7% of the time of the most loaded process. In the following sections we refine the algorithms used in the SQL program until there is virtually no imbalance, which is identified by terms of the form (100% | 100%).

Query Evaluation Stage 1 A portion of the call-graph for the original program (version 1) is shown in Figure 2. The three input relations were initially unevenly distributed among the processors. It is to be expected that an irregular

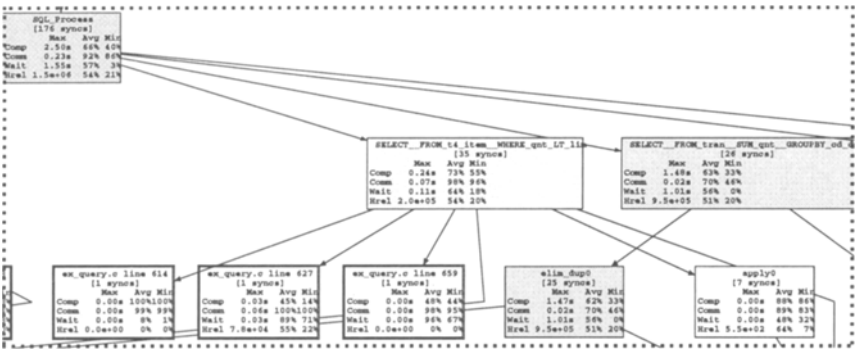

Fig. 2. Query version 1: SQL query evaluation.

distribution of input relations would produce a considerable amount of imbalance in computation and communication when performing operations using these data structures. For example, Figure 2 shows a (54%| 21%) imbalance in h-relation size. To remedy this imbalance, load balancing functions were introduced into the code which ensured that each processor contained approximately an equal sized partition of the input relation.

Query Evaluation Stage 2 Load balancing the input relations reduced the amount of communication and computation imbalance by 26%. Further profiles revealed that the imbalance had not been completely eradicated. It appeared that the SQL primitives had inherent imbalances in communication even for perfectly balanced input data. This can be seen in Figure 3.

The critical paths identifying the imbalance in communication were followed from the select function to `elim_dup0`. At this point it was easy to identify that the major cause of the communication imbalance, (69%| 54%), was attributed to the function `bspsort` at line `175`. This is shown in the screen-shot in Figure 1.

In the same way, following the imbalance in the computation critical paths, (51%| 17%), it was also clear that the primary cause of computation imbalance was attributed to the same function `bspsort`, this time at line `188` in Figure 1. Figure 1 also shows a pie-chart that gives a breakdown of the accumulated computation time at the superstep at line `188` for all the processes. To give some idea of the type of computation that may cause problems, the underlying algorithm of the `bspsort` function is (briefly) described. The function `bspsort` implements a refined variant of the optimal randomised BSP sorting algorithm of [1].

The algorithm consists of seven stages: (1) each processor locally sorts the elements in its possession; (2) each processor selects a random sample of $s \times p$ elements (where s is the oversampling factor) which are gathered onto process zero; (3) the samples are sorted and p regular pivots are picked from the $s \times p^2$ samples; (4) the pivots are broadcast to all processors; (5) each processor partitions the elements in its possession into the p blocks as induced by the

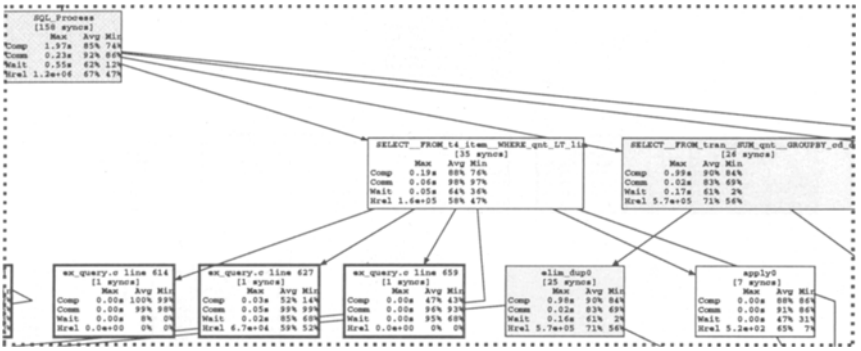

Fig. 3. Query version 2: SQL query evaluation after load balance.

pivots; (6) each processor sends partition i to processor i; and (7) a local multi-way merge produces the desired effect of a global sort.

If stages (6) and (7) are not balanced, then this can only be attributed to a poor selection of splitters in stage (2). Since the random number generator that selects the sample had been extensively tested prior to usage the only possible cause for the disappointing performance of the algorithm was the choice of the oversampling factor s. The algorithm had however been previously tested and the oversampling procedure had been fine tuned by means of extensive experimental results using simple timing functions. The experimental results had suggested that the oversampling factor established by the theoretical analysis of the algorithm had been a gross overestimate and, therefore, at the implementation level it had been decided to employ a considerably reduced factor.

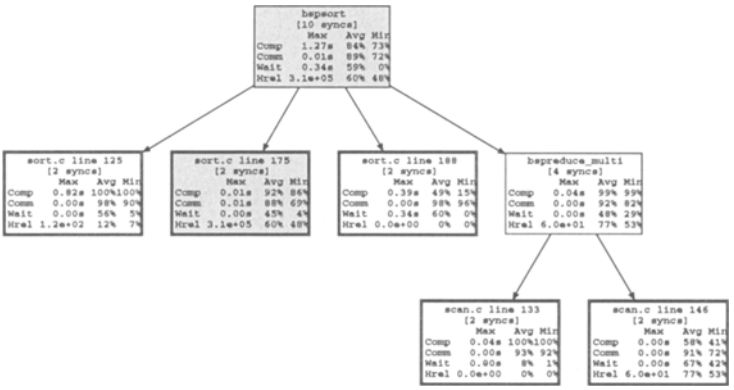

Fig. 4. Sorting version 1: experimental oversampling factor.

Query Evaluation: Improving Parallel Sorting Experiments continued by varying the oversampling factor for the sorting algorithm. A portion of the call-graphs for the optimal experimental and theoretical parameters are exhibited in figures 4 and 5 respectively. The original experimental results were confirmed by the profile, that is, the algorithm utilising the theoretical oversampling factor (sort version 2) delivered performance that was approximately 50% inferior to that of the algorithm utilising the experimental oversampling factor (sort version

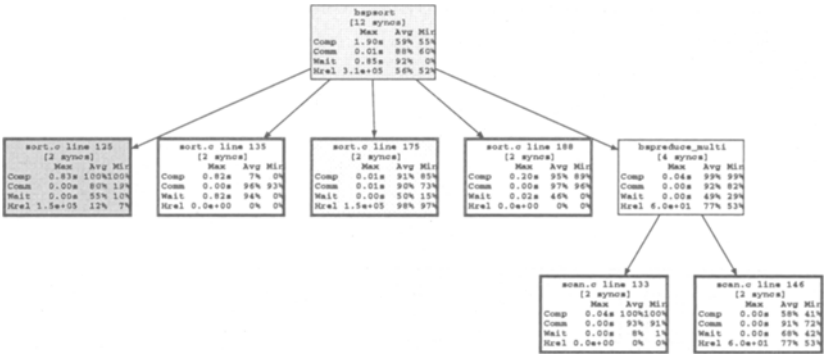

Fig. 5. Sorting version 2: theoretical oversampling factor.

1). The computation imbalance of (49%| 15%) in stage (7) of sort version 1, shown at line 188 in Figure 4, shifted to a computation imbalance of (7%| 0%) in stage (2) of sort version 2, shown at line 135 of Figure 5. Similarly, the communication imbalance of (60%| 48%) in stage (6) of sort version 1 shifted to an imbalance of (12%| 7%) in stage (3) at line 125 of sort version 2.

It was noted however that the communication and computation requirements of stages (6) and (7) in sort version 2 of Figure 5 are highly balanced, (98%| 97%) and (95%| 89%) respectively. Therefore, the theoretical analysis had accurately predicted the oversampling factor value required to achieve load balance. Unfortunately, the sustained improvement gained by balancing the communication pattern of stage (6) of the underlying sorting algorithm – and consequently, the communication requirements of the algorithm – had been largely overwhelmed by the cost of communicating and sorting a larger sample in stages (2) and (3).

To remedy this problem the ideas of [1] were adopted. In particular, the unbalanced communication and computation algorithm of stages (2) and (3), which collected and sorted a sample on a single process, were replaced with a simple and efficient parallel sorting algorithm, which sorted the sample set among all the processes. As noted in [1], an appropriate choice for such a parallel sorting algorithm is presented by an efficient variant of the bitonic-sort network.

The introduction of the bitonic sorter caused the data presented in the `bspsort` node in Figures 4 and 5 to: Computation (99%| 98%), Communication (83%| 70%), and h-relation (99%| 98%). This improved the wall-clock running time of the sorting algorithm by 8.5%.

Query Evaluation Stage 3 As the sorting algorithm is central to the implementation of most of the SQL queries, a minor improvement in the sorting algorithm brings about a marked improvement in the performance of the query evaluation as a whole. This can be clearly seen from the lack of any shading of critical paths in Figure 6 (contrast with Figure 2)–all the h-relations in the SQL queries are almost perfectly balanced.

To reinforce the conclusion that these improvements are architecture independent, Table 2.1 shows the wall-clock times for the original and optimised pro-

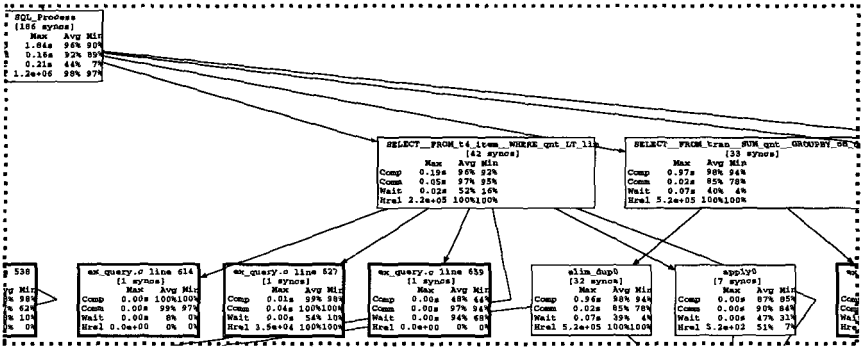

Fig. 6. Query version 3: SQL query evaluation, final version.

grams running on a variety of parallel machines. The size of the input relations was chosen to be small, so that the computation time in the algorithms would not dominate. This is particularly important if we are to highlight the improvements in communication cost. A small relation size also enabled the Network of Workstations (NOW) implementation of the algorithm to complete in a reasonable amount of time. For all but one of the results, the optimised version of the program provides an approximate 20% improvement over the original program. This evidence supports our hypothesis that the call-graph profiling tool provides a mechanism that guides architecture independent program optimisation. As an aside, the parallel efficiencies in the table are marginal for the T3E and Origin 2000, due to the communication intensive nature of this query processing application, in combination with the small amount of data held on each process (750 records per process at $p = 16$). In contrast, a combination of the slow processors, fast communication, and poor cache organisation on the T3D gives super-linear speedup even for this small data set. However, the communication intensive nature of this problem makes it unsuitable for running over loosely coupled distributed memory machines. For example, the results for a NOW, exhibit super-linear speedup at $p = 2$ in the optimised program, yet *super-linear slowdown* in all other configurations.

3 Conclusions

The performance improvements resulting from the analysis of the call-graph profiles demonstrate that the tool can be used to optimise programs in a portable and architecture independent manner. Unlike other profiling tools, the architecture independent metric – h-relation size – guides the optimisation process. The major benefit of this profiling tool is that the amount of information displayed when visualising a profile for a parallel program is no more complex than that of a sequential program.

[1] A 10Mbps Ethernet network of 266MHz Pentium Pro processors.

Table 1. Wall-clock time in seconds for input relations containing 12,000 records

Machine	p	Unoptimised		Optimised		gain
		time	speedup	time	speedup	
Cray T3E	1	6.44	1.00	5.37	1.00	17%
	2	4.30	1.50	3.38	1.59	21%
	4	2.48	2.60	1.85	2.90	25%
	8	1.23	5.23	1.04	5.17	15%
	16	0.68	9.43	0.67	8.02	1%
Cray T3D	1	27.18	1.00	22.41	1.00	18%
	2	13.18	2.06	10.88	2.06	17%
	4	6.89	3.94	5.70	3.93	17%
	8	3.29	8.25	3.07	7.30	7%
	16	1.66	16.34	1.89	11.88	-14%
SGI Origin 2000	1	2.99	1.00	2.42	1.00	19%
	2	1.65	1.81	1.27	1.91	23%
	4	1.26	2.37	1.11	2.16	12%
	8	0.88	3.39	0.77	3.15	13%
Intel NOW[1]	1	4.96	1.00	4.04	1.00	18%
	2	78.21	0.06	1.65	2.45	98%
	4	174.77	0.03	88.76	0.05	49%
	6	216.42	0.02	101.33	0.04	53%

References

1. A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the* 8^{th} *ACM Symposium on Parallel Algorithms and Archictures*, Padova, Italy, June 1996. ACM Press.
2. J. M. D. Hill, S. Donaldson, and D. Skillicorn. Stability of communication performance in practice: from the Cray T3E to networks of workstations. Technical Report PRG-TR-33-97, Programming Research Group, Oxford University Computing Laboratory, October 1997.
3. J. M. D. Hill, S. Jarvis, C. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.
4. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, to appear 1998. see `www.bsp-worldwide.org` for more details.
5. S. A. Jarvis. *Profiling large-scale lazy functional programs.* PhD thesis, Computer Science Department, University of Durham, 1996.
6. D. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.
7. K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *5th EuroMicro Workshop on Parallel and Distributed Processing (PDP'97)*. IEEE Computer Society Press, Jan. 1997.
8. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.