Portable and architecture independent parallel performance tuning

S.A. Jarvis [a] J.M.D. Hill [b] C.J. Siniolakis [c] V.P. Vasilev [b]

[a] *Department of Computer Science, University of Warwick, UK*

[b] *Sychron Ltd., One Cambridge Terrace, Oxford, UK*

[c] *The American College of Greece, Athens 15342, Greece*

Abstract

A call-graph profiling tool has been designed and implemented to analyse the efficiency of programs written in *BSPlib*. This tool highlights computation and communication imbalance in parallel programs, exposing portions of program code which are amenable to improvement.

A unique feature of this profiler is that it uses the BSP cost model, thus providing a mechanism for portable and architecture-independent parallel performance tuning. In order to test the capabilities of the model on a real-world example, the performance characteristics of an SQL query processing application are investigated on a number of different parallel architectures.

Key words: Profiling, Bulk Synchronous Parallel, Program Efficiency.

1 Introduction

The role of a profiling tool is to associate the execution costs of a program with identifiable segments of the underlying source code. How useful a profiling tool is depends on how easy it is for programmers to employ this information so as to alleviate computational bottlenecks in their code.

Three criteria need to be satisfied when designing successful profiling tools for sequential programming languages. The first relates to *'what'* the profiler measures; it is desirable for the percentage of execution time spent in each part of the program and/or the amount of memory used to be identified. The second criterion concerns *'where'* in the code these costs should be attributed; in order to improve the program implementation, costs should be associated with functions or libraries within the code. The third criterion relates to *'how'* the profiling information can be used to best effect; program code should be optimised in a quantifiable manner, an example of which might be rewriting problematic portions of code using an algorithm with improved asymptotic complexity.

Profiling parallel programs as opposed to sequential programs is made more complex by the fact that program costs are derived from a number of processors. As a result, each part of the program code may be associated with up to p costs, where p is the number of processors involved in the computation. One of the major challenges for developers of profiling tools for parallel programming languages is to design tools which will identify and expose the relationships between the computational costs accrued by the processors and highlight any imbalances. These cost relationships must subsequently be expressed in terms of the three criteria outlined above. Unfortunately many more issues are at stake with parallel frameworks and therefore the criteria are far harder to define and satisfy. In particular:

- *What to cost*: In parallel programming there are two significant cost metrics which may cause bottlenecks within programs; these are *computation* and *communication*. It is not good practice to decouple these two metrics and profile them independently as it is of paramount importance that the interaction between the two is identified and exposed to the user. This is because if programs are optimised with respect to one of these metrics, this should not be at the expense of the other.
- *Where to cost*: Costing communication can be problematic. This is because related communication costs on different processors may be the result of up to p different (and interacting) parts of a program. In message-passing systems, for example, there exist p distinct and independently interacting 'costable' parts of code. Without attention to design, profiling tools developed for such systems may overload the user with results. Too much profiling information can be difficult to interpret; the *upshot* system [6] has been criticised for this very reason.
- *How to use*: When profiling information is used to optimise parallel programs, care has to be taken to ensure that these optimisations are not specifically tailored to a particular machine or architecture. An optimisation is more likely to be portable if program improvements are made at the level of the underlying algorithms. Portable and architecture-independent optimisations to parallel programs are more likely to be achieved if the programming model on which the algorithm is built possesses a supporting, pragmatic cost model.

These three criteria form the basis for the development of a profiling tool for parallel programs whose code is based on the BSP model [18,14,16]. When using this profiling tool the programmer will use the balance of computation and communication as the metric with respect to which their parallel programs are optimised. It is shown that by minimising this imbalance, significant improvements to the complexity of parallel algorithms are often achieved. This is the key to portable and architecture-independent optimisations.

The BSP model, its implementation (*BSPlib*) and cost calculus are introduced in section 2. In section 3 attributes of *BSPlib* which facilitate parallel profiling are described and the call-graph profiling tool is introduced with the analysis of two broadcast algorithms. The profiler is used to optimise of a real-world distributed database query processing application and the results are described in section 4. The architecture-independent properties of this framework are explored in section 5.

2 The BSP model

The exploration of parallel computation within theoretical computer science has been led by the study of time, processor and space complexities of 'ideal' parallel machines which communicate via a shared memory; this is known as the *Parallel Random Access Machine* (PRAM) Model [2]. The PRAM model assumes that an unbounded set of processors shares a global memory. In a single step, a processor can either read or write one data word from the global memory into its local address space, or perform some basic computational operation. The simplicity of the model has, over the past two decades, encouraged the development of a large collection of PRAM algorithms and techniques [10]. Conversely, the model's simplicity also means that it does not reflect a number of important aspects of parallel computation which are observed in practice; these include communication latency, bandwidth of interconnection networks, memory management and processor synchronisation, amongst others.

The *Bulk Synchronous Parallel* (BSP) model – a high-level abstraction of hardware – provides a general-purpose framework for the design and analysis of scalable programs, which may then be run efficiently on existing diverse hardware platforms. In addressing many of the previous limitations, BSP is widely regarded as a bridging model for parallel computing [18,17]. In the BSP model no assumptions are made about the underlying technology or the degree of parallelism. The BSP model thus aims to provide a general-purpose parallel computing platform [12,13,17,18]. A *Bulk Synchronous Parallel Machine* (BSPM) provides an abstraction of any real parallel machine; a BSPM has three constituent parts:

(1) A number of *processor/memory components* (processors);
(2) An *interconnection network* which delivers messages in a point-to-point manner between the processors;
(3) A *facility for globally synchronising* the processors by means of a barrier.

In the BSP model a program consists of a sequence of *supersteps*. During a superstep, each processor can perform computations on values held locally at the start of the superstep, or it can initiate communication with other processors. The model incorporates the principle of *bulk synchrony*; that is, processors are barrier synchronised at regular intervals, each interval sufficient in length for the messages to be transmitted to their destinations [18]. The model does not prescribe any particular style of communication, but it does require that at the end of a superstep any pending communications be completed.

2.1 The BSP cost model

The cost of a BSP program can be calculated by summing the costs of each superstep executed by the program. In turn, the cost of an individual superstep can be broken down into: (i) local computation; (ii) global exchange of data and (iii) barrier synchronisation. The maximum number of messages (words) communicated to or from any processor during a superstep is denoted by h, and the complete set of messages is captured in the notion of an *h-relation*. To ensure that cost analysis can be performed in an architecture-independent manner, cost formula are based on the following *architecture-dependent* parameters:

p – the number of processors;
l – the minimum time between successive synchronisation operations, measured in terms of basic computational operations;
g – the ratio of the total throughput of the system in terms of basic computational operations, to the throughput of the router in terms of words of information delivered; alternatively stated, g is the single-word delivery cost under continuous message traffic.

Intuitively, g measures the permeability of the network to continuous message traffic. A small value for g, therefore, suggests that an architecture provides efficient delivery of message permutations. Similarly, l captures the cost of barrier synchronisation.

Using the definition of a superstep and the two architectural parameters g and l, it is possible to compute the cost of executing a program on a given architecture. In particular, the cost C^k of a superstep $\mathcal{S}^k$ is captured by the formulae [4,18],

$$\begin{aligned} & C^k = w^k + h^k \cdot g + l \\ & \text{where}\, w^k = \max\{ \quad w_i^k \quad \mid 0 \le i < p \} \\ & \qquad h^k = \max\{ \max(h_i^k\text{-in}, h_i^k\text{-out}) \mid 0 \le i < p \} \end{aligned} \tag{1}$$

where k ranges over the supersteps; i ranges over processors; w_i^k is an *architecture-independent cost* representing the maximum number of basic computations which can be executed by processor i in the local computation phase of superstep $\mathcal{S}^k$; h_i^k-in (respectively, h_i^k-out) is the largest accumulated size of all messages entering (respectively, leaving) processor i within superstep $\mathcal{S}^k$.

In the BSP model, the total computation cost of a program is the sum of all the costs of the supersteps, $\sum_k C^k$.

3 Profiling the imbalance in parallel programs

The BSP model stipulates that all processors perform lock-step *phases* of computation followed by communication. This encourages a disciplined approach in the utilisation of computation and communication resources. BSP programs may be written using existing communication libraries which support non-blocking communications. However, these general-purpose libraries are rarely optimised for the subset of operations which are required for the BSP programming paradigm [9,16]. In order to address this problem, the BSP research community has proposed a standard library – *BSPlib* – which can be used for parallel programming within the BSP framework [8,5].

BSPlib is a small communication library consisting of twenty operations for SPMD (Single Program Multiple Data) programming. The main features of *BSPlib* are two modes of communication, the first capturing a BSP-oriented message-passing approach and the second reflecting a one-sided direct remote memory access (DRMA) model.

The applications described in this paper have predominately been written using the DRMA style of communication. They utilise the one-sided *BSPlib* function `bsp_put`. This function transfers data from contiguous memory locations on the processor which initiates communication into contiguous memory locations on a remote processor, *without* the active participation of the remote processor. The function `bsp_sync` identifies the end of a superstep, at which time all processors barrier synchronise. It is at this point that any message transmissions issued by processors during the superstep are guaranteed to have arrived at their destination.

In contrast to programs written in a general message-passing style, *BSPlib* facilitates profiling in a number of ways:

(1) The cost model highlights the use of both computation and communication as profiling cost metrics.
(2) The cost of communication within a superstep can be considered *en*

masse. This greatly simplifies the presentation of profiled results. In particular, communication within a superstep can be attributed to the barrier synchronisation which marks the end of a superstep and not to individual communication actions [7].

(3) BSP cost analysis is modular and *convex*; that is, improvement to the performance of algorithms as a whole cannot be achieved by making one part slower. This is important when profiling, as portions of code may be elided to simplify the presentation of results. In this model this can be done safely; the removed parts of the code will have no adverse effect on the cost of the remaining supersteps.

(4) The treatment of computation and communication engineered by the BSP model (and consequently *BSPlib*) foster a programming style in which processes pass through the same textual `bsp_sync` for each superstep [1]. Consequently, the line number and file name of the code which contains the `bsp_sync` statement provide a convenient reference point in the source code to which profiling costs can be attributed.

3.1 Criteria for good BSP design

In this section, two broadcast algorithms are analysed and the call-graph profiler for *BSPlib* programs is introduced.

A post-mortem *call-graph* profiling tool has been developed to analyse trace information generated during the execution of *BSPlib* programs. The units of code to which profiling information is assigned are termed *cost centres*. For simplicity, each cost centre in the program corresponds to a `bsp_sync` call. Each cost centre records the following information: (i) the accumulated computation time; (ii) the accumulated communication time; (iii) the accumulated idle (or *waiting*) time; and (iv) the accumulated h-relation size. The timing result recorded at a cost centre is simply the sum of the maximum communication and computation times recorded since the last `bsp_sync` call.

The aim of the profiling tool is to expose imbalances in either computation or communication, and to highlight those parts of the program which are amenable to improvement. The hypothesis that balance is the key to good BSP design is supported by the BSP cost formulae:

- Within a superstep the computation should be balanced between processes. This is based on the premise that the maximum value of w_i (for $0 \leq i < p$, see equation 1) will determine the overall cost of local execution time;
- Within a superstep the communication should be balanced between processes. This is based on the premise that the maximum value of h_i (for

[1] This imposes tighter restrictions than the *BSPlib* program semantics.

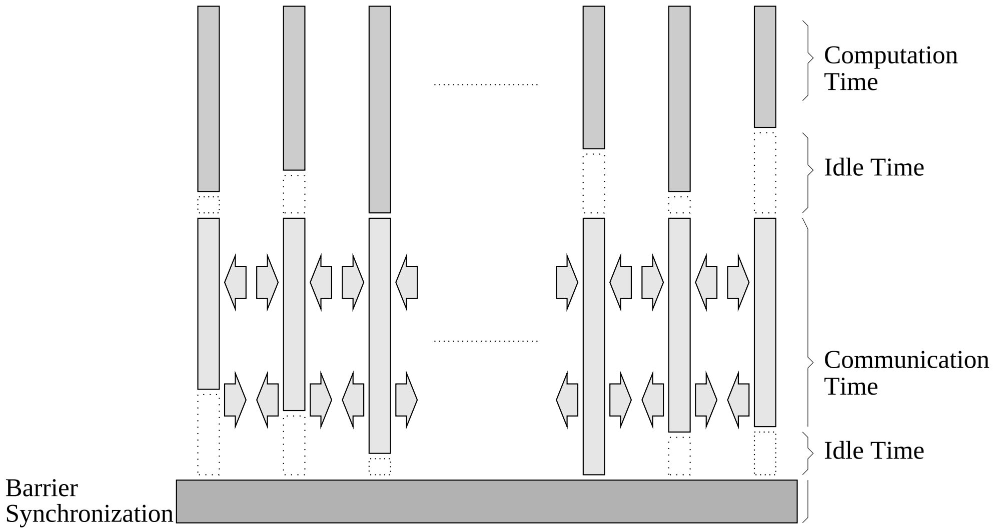

Fig. 1. Superstep structure.

$0 \leq i < p$, see equation 1) will determine the overall cost of the fan-in and fan-out of messages;

- Finally, the total number of supersteps should be minimised. Each contributes an l term to the total execution time (see equation 1).

Figure 1 shows a schematic diagram of a BSP superstep and its associated costs. The figure shows that idle time can arise in either local computation or communication. In local computation, idle time will arise when processes are forced to wait at a barrier synchronisation for the process with the largest amount of computation to be completed. Idle time will occur during the communication phase of a superstep if processes are forced to wait until interprocess communication has been completed [2].

At each cost centre, p costs – corresponding to one cost per process – are recorded. This data is presented to the user in one of two ways:

Summarised data: The cost within a single cost centre can be summarised in terms of maximum (the standard BSP interpretation of cost), average and minimum accumulated costs over each of the p processes. More formally, given that a program may pass through a particular cost centre k times, generating a sequence of costs $\langle C^1, \ldots, C^k \rangle$, the accumulated computation cost for the given cost centre is given by the formulae:

[2] It is noted that idle time during communication depends upon the type of architecture on which *BSPlib* is implemented. For example, on the DRMA and shared memory architectures (e.g. Cray T3D/E and SGI Power Challenge), communication idle time arises as shown in Figure 1. However, with architectures which only support message passing (e.g. IBM SP2), communication idle time is coalesced with the computation idle time of the following superstep; see [9] for details.

$$maximum\ cost = \sum_k \max \{ w_i^k \mid 0 \leq i < p \} \tag{2}$$

$$average\ cost = \sum_k \frac{1}{p} \left(\sum_{0 \leq i < p} w_i^k \right) \tag{3}$$

$$minimum\ cost = \sum_k \min \{ w_i^k \mid 0 \leq i < p \} \tag{4}$$

Similar formulae exist for communication time, idle time and h-relation size.

All data: The costs associated with each of the p processes are presented to the user in the form of a pie chart. The results must be interpreted with some care as the costs are calculated using formulae which differ from the standard BSP cost formulae. This is necessary as a user of this pie chart is typically looking to identify the largest (maximum) segment in the chart. The size of this maximum segment is:

$$\max \{ \sum_k w_i^k \mid 0 \leq i < p \} \tag{5}$$

Equation 5 abstracts the maximum outside the summation, producing a result which may be smaller than that obtained from equation 2. Although this interpretation is not strictly in line with BSP cost analysis, it is a useful method for identifying the process which may be causing an efficiency bottleneck.

3.2 Example: broadcasting n values to p processes

In this example a common broadcast problem is considered; this is the communication of a data structure of size n (where $n \geq p$) from one process to all p processors in a parallel computing system.

A naive algorithm for this task can be implemented in a single superstep if $p-1$ distinct `bsp_puts` are performed by the broadcasting process. This requires the transmission of $p-1$ messages, each of size n; the superstep therefore realises an $n(p-1)$-relation with approximate cost (substituting p for $p-1$):

$$cost\ of\ one-stage\ broadcast = npg + l \tag{6}$$

where l is the cost of performing a single superstep.

This algorithm, captured in equation 6, is not scalable as its cost linearly increases with p.

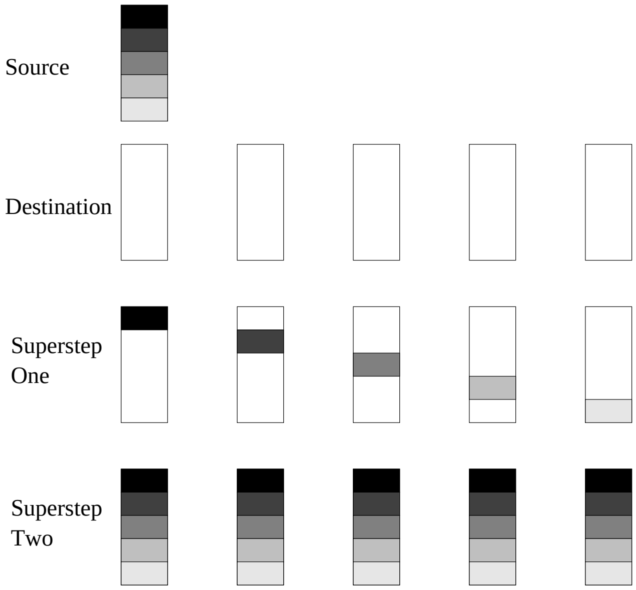

Fig. 2. Two-stage broadcast using total exchange.

An alternative scalable BSP broadcasting algorithm [1,11], with cost $2ng + 2l$, is shown in Figure 2. The algorithm consists of two supersteps: in the first superstep, the data is distributed evenly amongst the p processes; in the second superstep, all processes then broadcast their local data. This results in balanced (system) communication.

The cost of the distribution phase is $(n/p)(p-1)g + l$, as a single message of size (n/p) is sent to each of the $p-1$ processes. In the second superstep, each process sends and receives $p-1$ messages of size (n/p) from each other process. The cost of this superstep [3] is also $(n/p)(p-1)g + l$. The approximate cost of the entire algorithm is determined by summing the cost of the two supersteps (once again, assuming the substitution of p for $p-1$):

$$cost\ \ of\ \ two-stage\ \ broadcast = \left(\frac{n}{p}pg + l\right) + \left(\frac{n}{p}pg + l\right) = 2ng + 2l \quad (7)$$

Using equations 6 and 7 it is possible to determine the size of data for which the two-stage algorithm is superior to the one-stage algorithm:

$$n > \frac{l}{pg - 2g} \quad (8)$$

For example, when l is large and both n and p are small, the cost of the extra superstep outweighs the cost of communicating a small number of short messages. Conversely, for a large n or p, the communication cost outweighs the overhead of the extra superstep.

[3] Note that BSP cost analysis encourages balanced communication.

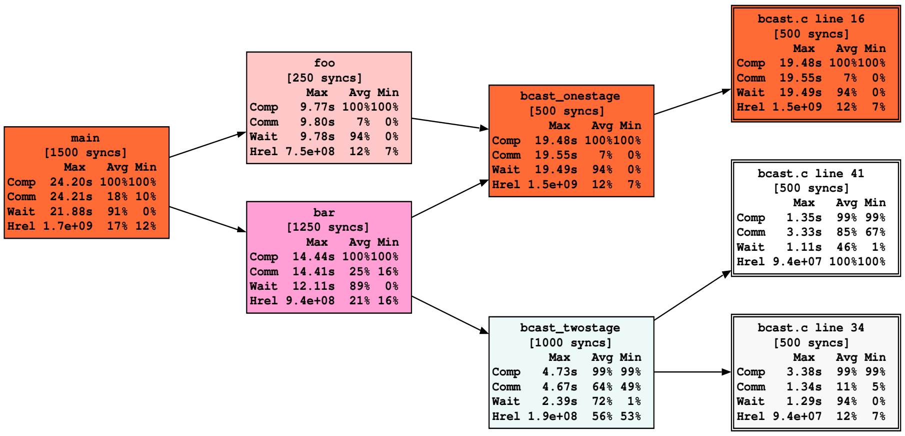

Fig. 3. Sample call-graph profile on a 16 processor Cray T3E.

Figure 3 shows an example call-graph profile for the two broadcast algorithms running on a 16 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 supersteps, i.e. the lines of code containing a `bsp_sync`. The path from a leaf to the root of the graph identifies the sequence of cost centres passed through to reach the part of the code that is active when the `bsp_sync` is executed. This path is termed a *call stack* and a collection of call stacks therefore comprise a *call-graph* profile. One significant advantage of call-graph profiling is that a complete set of unambiguous program costs can be collected at run-time and post-processed. This is a great help when identifying program bottlenecks. Furthermore, the costs of shared procedures can be accurately apportioned to their parents via a scheme known as *cost inheritance*. This allows the programmer to resolve any ambiguities which may arise from the profiling of shared functions[15].

The call-graph in Figure 3 shows the profile results for a program which performs 500 iterations of the one-stage broadcast and 500 iterations of the two-stage broadcast. In order to highlight the features of the call-graph profile, the procedures `foo` and `bar` contain procedure calls to the two broadcasting algorithms. The order of program execution is as follows: in procedure `foo` the one-stage broadcast algorithm is called 250 times; the procedure `bar` then calls the one-stage broadcast algorithm 250 times and makes a further 500 calls to the two-stage broadcast algorithm.

The graph illustrates how the program costs are inherited from the leaves of the graph towards the root. The top-level node `main` displays the accumulated computation, communication and idle costs for each of the supersteps within the program. At the interior nodes in the call-graph, information is displayed which relates to supersteps executed during the lifetime of the procedure identified at that node.

At the leaf nodes the profile records: (i) the textual position of the `bsp_sync` call within the program; (ii) the number of times the superstep is executed; (iii) summaries of the computation, communication and idle times and (iv) the cost of the h-relation. Each of these summaries consists of the maximum, average and minimum cost over the p processors; the average and minimum costs are given as percentages of the maximum.

Interior nodes store similar information to leaf nodes. The interior nodes are also labelled with procedure names and the results displayed at the nodes are the inherited costs from the supersteps executed during calls to that procedure.

In the profiling results of Figure 3, the maximum computation and communication times for the interior node labelled `bcast_onestage` are both 19 seconds. The total execution time for the calls to the one-stage broadcast is therefore 38 seconds (total computation + communication). The interior node also displays the maximum idle time (also 19 seconds) which is the delay caused by the broadcasting process transmitting the data to the remaining $p-1$ processes.

3.4 Identifying critical paths

The scope of the profiling tool is not limited to simply visualising the computation and communication patterns at each cost centre. The tool also allows *critical cost paths* to be identified within parallel programs. Each node in the graph is displayed in a colour ranging from white to red. A red node identifies an efficiency bottleneck (or program 'hot spot') [4] . A sequence of dark-coloured nodes identifies a critical path in the program. There are seventeen different types of program critical path which can be identified by the profiler. The simplest of these is the *synchronisation critical path* which identifies nodes in the graph which contain the greatest number of supersteps. In addition, four different critical paths can be identified for each of computation, communication, idle time and h-relation:

- Absolute – identifies the nodes with the *greatest maximum cost*;
- Absolute imbalance – identifies the nodes with the *greatest difference between the maximum and average cost*;
- Relative imbalance – identifies the nodes with the *greatest percentage deviation between maximum and average cost*;

[4] In this paper colours have been replaced by greyscales ranging from white to dark grey.

- Weighted – identifies the nodes with both the *greatest difference between the maximum and average cost* and the *greatest percentage deviation between the maximum and average cost*, a combination of the previous two critical paths.

The *absolute critical path* identifies those nodes to which the greater part of the program costs are attributed. The *absolute imbalance critical path* highlights those nodes which are amenable to further improvement, as it identifies the underlying imbalance between the maximum and average cost. However, care must be taken when this metric is used, as nodes with large cost values and small deviations may be identified as being 'more critical' than nodes with smaller cost values but larger deviations. The latter are most receptive to significant improvement; the *relative imbalance critical path* is therefore more useful when determining node imbalance, irrespective of the cost size. Finally, the *weighted critical path* combines the advantages of the previous two approaches.

The critical paths identified in Figure 3 highlight the absolute imbalance in h-relation in the program. The profiling tool shows that when the one-stage broadcast algorithm is used, there is a significant communication imbalance; this imbalance is quantified in the h-relation in the form (12% | 7%) — the average cost is 12% of the maximum, whereas the minimum cost is 7%. Results showing such small percentages for the average and minimum costs point to large imbalances in the underlying algorithm. The profiling tool also highlights a similar imbalance in the first superstep (`bcast.c line 34`) of the two-stage algorithm (the initial distribution of data), an imbalance which is unavoidable using this approach. It is interesting that the profiling tool does not rank this imbalance as highly as the imbalance underlying the one-stage algorithm; this is because it is caused by a smaller h-relation, i.e. an $(n/p)(p-1)$-relation rather than an $n(p-1)$-relation. Finally, it can be seen that the last superstep of the two-stage broadcast (`bcast.c line 41`) has no h-relation imbalance i.e. it is (100% | 100%).

4 Profiling an SQL database application

The optimisation of an SQL database query evaluation program provides a persuasive real-world case study to illustrate the effectiveness of the call-graph profiling tool on larger applications.

The SQL database query evaluation program contains a number of relational queries which are implemented in BSP. The program consists of standard SQL database queries which have been transcribed into C function calls and then linked with a *BSPlib* library of SQL-like primitives. The program takes a

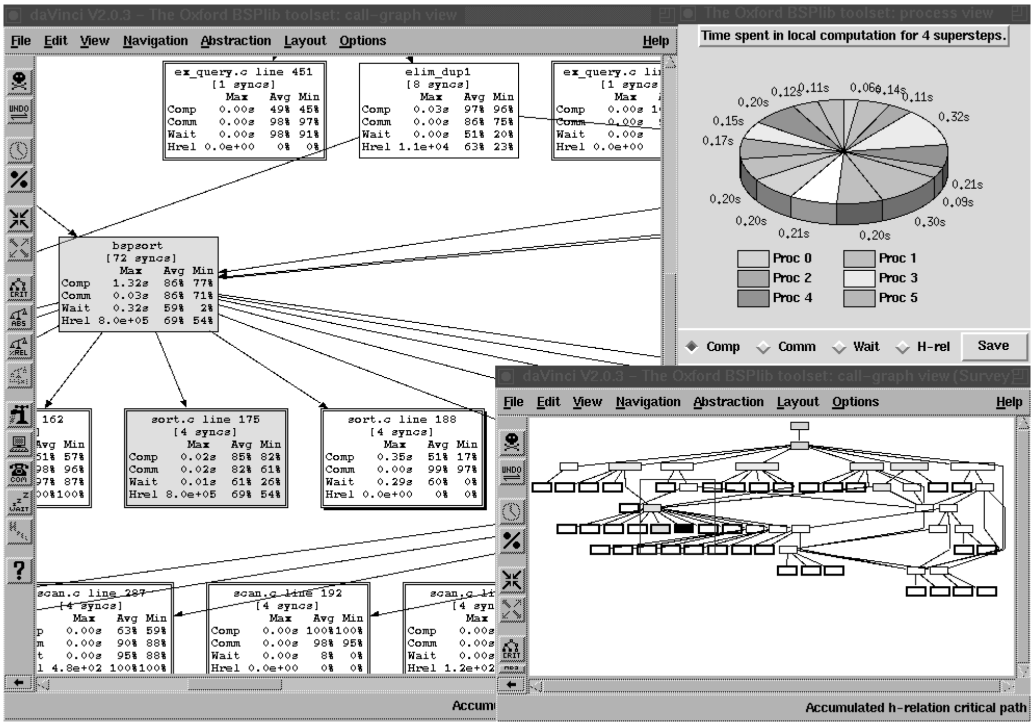

Fig. 4. Screen view of the call-graph profiling tool

sequence of relations, in the form of tables, as its input. It processes the tables and yields, as output, a sequence of intermediate relations.

The program works by distributing input relations among the processors using a simple block-cyclic distribution. Three input relations `ITEM`, `QNT` and `TRAN` are defined. Here the program evaluates six queries which in turn create 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`.

The program was executed on a sixteen processor Cray T3E and the screen view of the profile results is shown in Figure 4. On the basis of these results, a series of optimisations – documented below – is performed with the aim of achieving a balanced program i.e. one whose cost terms are of the form (100% | 100%).

4.1 Optimisation: stage 1

The results found in Figure 5 are a section of the call-graph profile for the original SQL query processing program (*version 1* of the program).

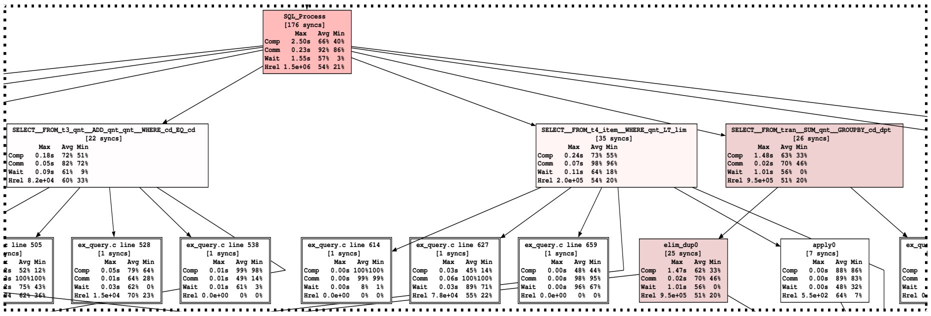

Fig. 5. Detail of the complete call-graph profile found in Figure 4.

The initial results show an uneven distribution of the three input relations amongst the processors. This gives rise to a considerable imbalance in computation and communication when database operations are performed using these data structures. Figure 5 shows a (54%| 21%) imbalance in h-relation size.

As a potential remedy, load balancing functions were introduced into the code to ensure that each processor contained an approximately equal partition of the input relation. The results of load balancing these input relations reduces the communication and computation imbalance by 26%.

4.2 Optimisation: stage 2

Further profiles of the SQL query processing application reveal that the imbalance has not been eradicated. It appears that the SQL primitives had inherent communication imbalance, even when perfectly balanced input data was used. The profiling results which support this observation can be seen in Figure 6.

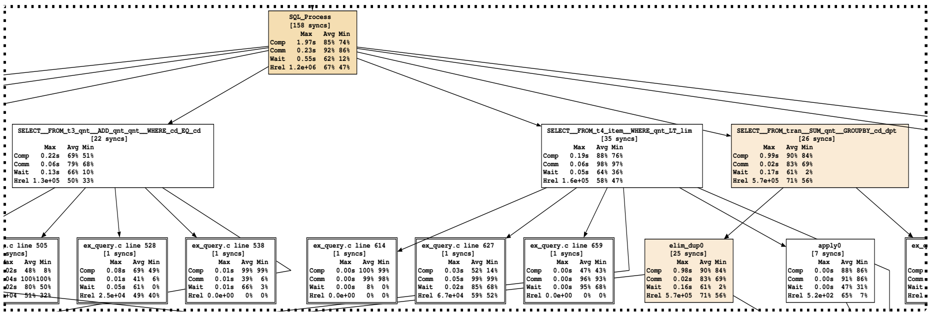

Fig. 6. SQL query evaluation after load balance.

Using the critical paths which the call-graph profiling tool identifies, it is possible to follow the path of execution through the `SELECT_FROM` function to the graph node labelled `elim_dup0`. This highlights a major cause of communication imbalance – (69%| 54%) – which can be attributed to the function `bspsort` at line `175` in the program. This is shown in Figure 4.

Repeating this procedure using the computation critical paths highlights that the same function, `bspsort`, is responsible for similar computation imbalance. The profile results in Figure 4 identify the primary source of this computation imbalance (51%| 17%) as line `188`. The pie chart in Figure 4 presents a breakdown of the accumulated computation time for each of the processes at the superstep found at line `188`.

In order to illustrate the class of computation which may be the cause of this type of problem, the underlying algorithm of the `bspsort` function is described.

The `bspsort` function implements a refined variant of the optimal randomised BSP sorting algorithm of [3]. 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 collected at 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 pivots; (6) each processor sends partition i to processor i and (7) a local multi-way merge results in the intended global sort.

If stages (6) and (7) are not balanced, this can only be attributed to a poor selection of splitters in stage (2). The random number generator which selected the sample had been extensively tested prior to use; therefore the probable cause for the disappointing performance of the algorithm was thought to be the choice of the oversampling factor s. The algorithm had however been previously tested and the oversampling factor fine tuned on the basis of experiments using simple timing functions. The experimental results had suggested that the oversampling factor established during theoretical analysis of the algorithm had been a gross overestimate and, as a result, when it was implemented, a much reduced factor was used.

4.3 Optimisation: improving the parallel sort

The oversampling factor for the sorting algorithm was further tested in a number of profiling experiments. Sections of the call-graph for the optimal experimental and theoretical parameters are presented in Figures 7 and 8 respectively. The original experimental results were confirmed by the profile: the performance of the algorithm utilising the theoretical oversampling factor (Figure 8) was approximately 50% inferior to that of the algorithm utilising the experimental oversampling factor (Figure 7). This can be seen by comparing the computation imbalance of (49%| 15%) in stage (7) of the first sort - found at line 188 in Figure 7 - with the computation imbalance of (7%| 0%) in

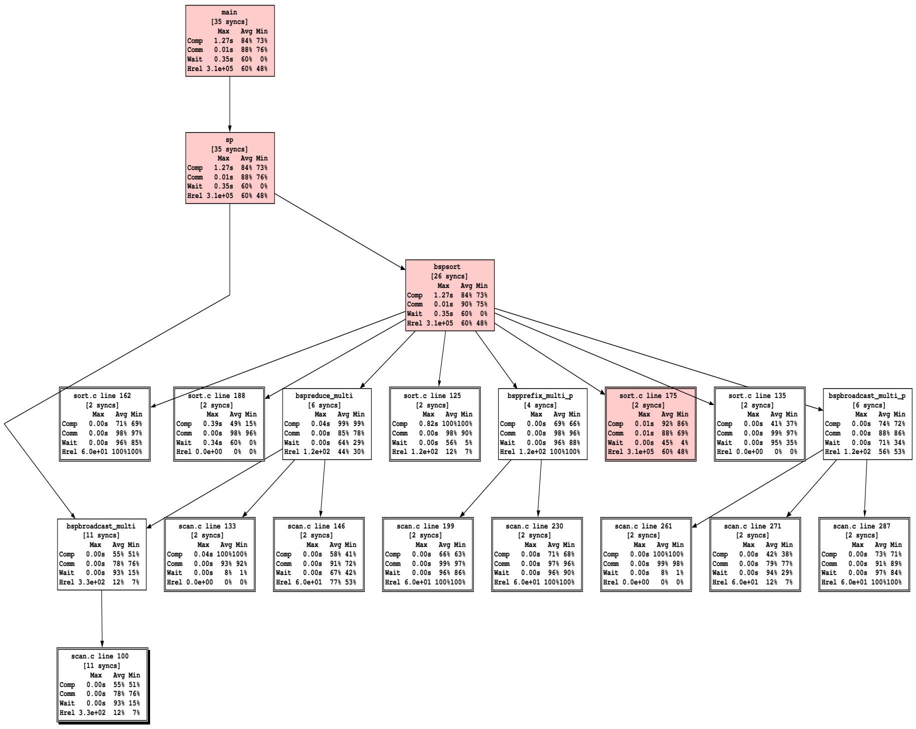

Fig. 7. Parallel sort using experimental oversampling factor.

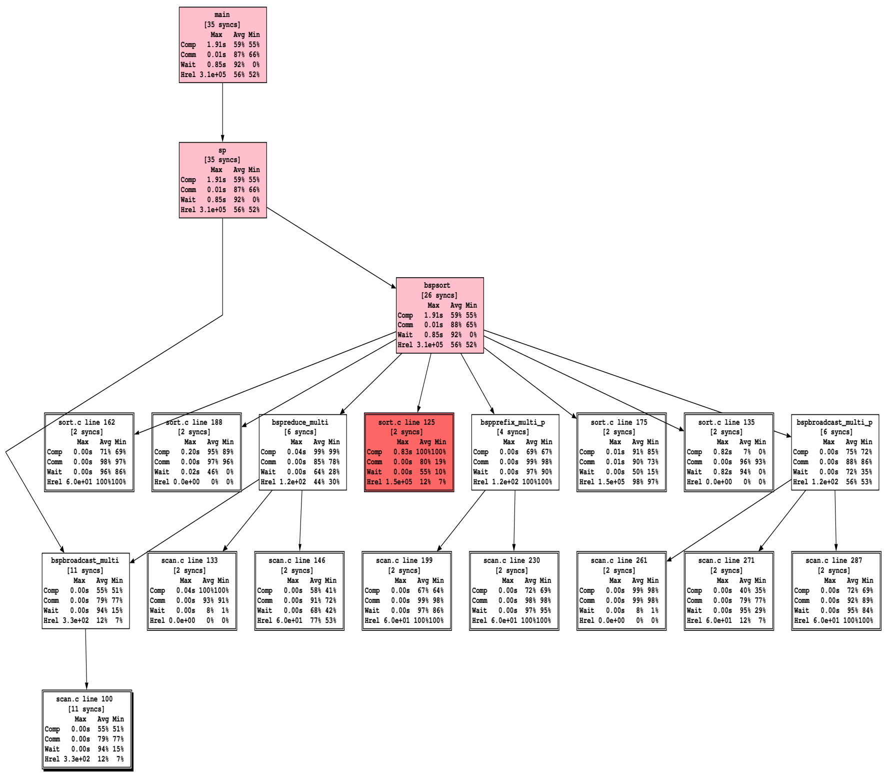

Fig. 8. Sorting version 2: theoretical oversampling factor.

stage (2) of the second sort - found at line 135 of Figure 8. Similarly, the communication imbalance of (60%| 48%) present at stage (6) of the first sort (found at line 175) contrasts with the imbalance of (12%| 7%) present at stage (3) of the second sort (found at line 125).

The communication and computation requirements of stages (6) and (7) in the second sort (Figure 8) were balanced, showing factors of (98%| 97%) and (95%| 89%) respectively. This showed that the theoretical analysis had indeed accurately predicted the oversampling factor required to achieve load balance. Unfortunately, the sustained improvement to the underlying sorting algorithm gained by balancing communication at stage (6) – and consequently, the improved communication requirements of the entire algorithm – had been largely overwhelmed by the cost of communicating and sorting larger samples in stages (2) and (3).

As a solution to this problem the work by Gerbessiotis and Siniolakis [3] was applied. The unbalanced communication and computation algorithms of stages (2) and (3), which collected and sorted a sample on a single process, were replaced by an alternative parallel sorting algorithm. This simple and efficient solution to the problem involves the sample set being sorted among all the processes. An appropriate implementation is an efficient variant of the bitonic-sort network.

The introduction of the bitonic sorter brought a marked improvement to the results. These showed an 8.5% improvement to the overall wall-clock running time of the sorting algorithm; the results also demonstrated corresponding program balance: computation (99%| 98%), communication (83%| 70%), and h-relation (99%| 98%).

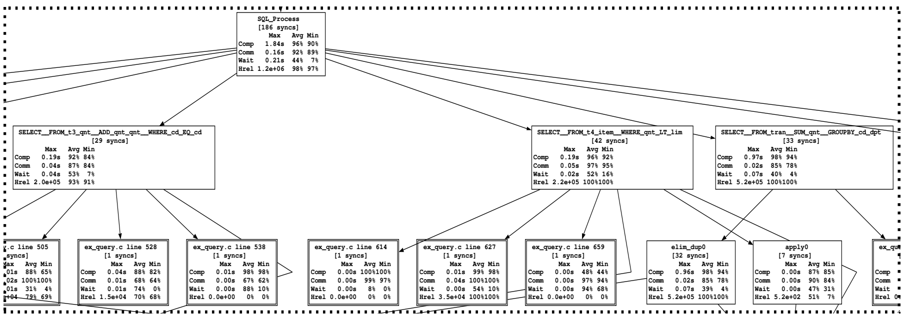

Fig. 9. Final version of the SQL query evaluation.

4.4 Optimisation: stage 3

The sorting algorithm is central to the implementation of most of the queries in the SQL database query evaluation application. Therefore, a minor improvement in the sorting algorithm results in a marked improvement in the performance of the query evaluation program as a whole. This is confirmed by the lack of any shading in Figure 9. In contrast to Figure 5, the h-relations of the SQL queries are almost perfectly balanced.

The overall improvement in performance of the SQL query evaluation program is discussed in the next section.

5 Architecture-independent code optimisation

In the introduction it was stated that care should be taken when optimising parallel programs based on profile information. In particular, using the wall-clock time as a cost metric is not a good basis for scalable and portable program improvements. An important feature of BSP is that the size of an h-relation directly influences the cost of communication. Therefore, instead of using actual communication time as a cost metric, which may be subject to quantification error, the predicted cost of communication ($hg+l$) is used. This method is error-free as the value of h, which is not affected by the choice of the underlying machine or architecture, is accurately recorded at runtime. This forms the basis for the hypothesis that imbalance in maximum and average h-relations can be used as the metric by which BSP programs are optimised and optimal architecture-independent parallel algorithms are developed. This hypothesis is supported by both the BSP cost analysis formulae and experimental results.

Analysis of the two broadcast algorithms also provides support for this hypothesis. In Figure 3 the accumulated values for computation, communication and idle time displayed at the nodes labelled `one_stage_bcast` and `two_stage_broadcast` show that the two-stage broadcast is superior to the one-stage broadcast. The performance of the two-stage broadcast algorithm on a 16 processor Cray T3E shows an improvement over the one-stage algorithm of a factor of:

$$\frac{19.5 + 19.6}{4.7 + 4.7} = 4.16$$

5.1 Architecture-independent optimisation of the query processing application

Table 1 shows the wall-clock times for the original and optimised SQL query processing applications running on a variety of parallel machines. The small size of the input relation was deliberate as this would prevent the computation time in the algorithms from dominating. This allows the improvements in communication cost to be highlighted.

Machine	p	Unoptimised time	Unoptimised speed-up	Optimised time	Optimised speed-up	gain
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 Cluster [5]	1	19.68	1.00	13.01	1.00	51%
	2	9.70	2.02	6.63	1.96	46%
	4	5.75	3.42	4.02	3.24	43%
Athlon Cluster [6]	1	11.94	1.00	7.85	1.0	52%
	2	6.06	1.97	4.08	1.92	49%
	4	3.58	3.34	3.07	2.56	17%

Table 1
Wall-clock time (secs.) for input relations containing 12,000 records

The results show that for up to eight processors, the optimised version of the program yields an improvement of between 7% and 52% over the original program. The relatively small input relation means that the experiment is not scalable beyond this point. Detailed analysis of the results in Table 1 supports this observation. The parallel efficiency of both the T3E and Origin 2000 is limited: increasing the number of processors on the Origin 2000 from 4 to

[6] A 400Mbps clustered network of 450Mhz Intel Pentium II processors, running the Sychron VPS software.
[6] A 300Mbps clustered network of 700Mhz AMD Athlon processors, running the Sychron VPS software.

8, for example, does not double the speed of execution. It is thought that this is due to the communication-intensive nature of the query processing application, together with the small quantity of data which is held on each process (750 records per process at $p = 16$). By contrast, a combination of the slow processors, fast communication and poor cache organisation on the T3D gives super-linear speed-up, even for this small data set.

The absolute timing results for the application running on the Intel and Athlon clusters are within the same approximate range as on more traditional supercomputers such as the Cray T3E and Origin 2000. It is interesting to note that the gains seen using the optimised version of the program on these clusters exceed those achieved on other machines. It is thought that this is due to the relatively small secondary cache size of the Intel processors compared with both the Alpha and Mips R10000 used on the Cray and Origin. A large imbalance in the data set used in the unoptimised version of the program means that one processor contains more data than the others; this processor will therefore have a larger cache overflow. The size of the data set utilised in this experiment is similar to the size of the cache itself. The absolute size of the cache will therefore have an effect on the performance of this benchmark; this is confirmed by the seemingly poor timing results for the Intel and Athlon processors.

6 Conclusions

The call-graph profiling tool, used in the efficiency analysis of *BSPlib* programs, provides the basis for portable and architecture-independent parallel performance optimisation. This hypothesis is tested by profiling an SQL query processing application, a real-world test-case written in *BSPlib*.

Program optimisations made to this application hold on a number of different parallel architectures including Intel and Athlon clusters, shared memory multiprocessors and tightly coupled distributed memory parallel machines.

A major benefit of the BSP call-graph profiling tool is the concise way in which program-cost information is displayed. Visualising the costs for a parallel program is no more complex than for a sequential program. Program inefficiencies are quickly identified with the use of critical cost paths. A scheme of cost inheritance also ensures that accurate profile costs are displayed even when shared functions form a large part of the program.

References

[1] M. Barnett, D. Payne, R. van de Geijn and J. Watts, Broadcasting on meshes with wormhole routing, *Journal of Parallel and Distributed Computing* **35** (1996) 111–122.

[2] S. Fortune and J. Wyllie, Parallelism in random access machines, in: *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing (1978)* 114–118.

[3] 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* (ACM Press, 1996).

[4] A. V. Gerbessiotis and L. G. Valiant, Direct bulk-synchronous parallel algorithms, *Journal of Parallel and Distributed Computing* **22** (1994) 251–267.

[5] M. W. Goudreau, J. M. D. Hill, K. Lang, W. F. McColl, S. B. Rao, D. C. Stefanescu, T. Suel and T. Tsantilas, A proposal for the BSP Worldwide standard library (preliminary version), Technical report, Oxford University Computing Laboratory, April 1996. see `www.bsp-worldwide.org` for more details.

[6] V. Heearte and E. Lusk, Studying parallel program behaviour with upshot, Technical Report ANL91/15, Argonne National Lab, Argonne, Il. 60439, 1991.

[7] J. M. D. Hill, P. I. Crumpton and D. A. Burgess, Theory, practice, and a tool for BSP performance prediction, in: *EuroPar'96*, Lecture Notes in Computer Science **1124** (Springer-Verlag 1996) 697–705.

[8] J. M. D. Hill, W. F. 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* **24** (1998) 1947–1980.

[9] J. M. D. Hill and D. Skillicorn, Lessons learned from implementing BSP, *Journal of Future Generation Computer Systems* **13** (1998) 327–335.

[10] J. JáJá, *An Introduction to Parallel Algorithms.* (Addison-Wesley, 1992).

[11] B. H. H. Juurlink and H. A. G. Wijshoff, Communication primitives for BSP computers, *Information Processing Letters* **58** (1996) 303–310.

[12] W. F. McColl, General purpose parallel computing, in: A. M. Gibbons and P. Spirakis, eds., *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation 337–391 (Cambridge University Press, 1993).

[13] W. F. McColl, Scalable parallel computing: A grand unified theory and its practical development, in: *Proceedings of IFIP World Congress* **1** (1994) 539–546.

[14] W. F. McColl, Scalable computing, in: J van Leeuwen, ed., *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science **1000** (Springer-Verlag, 1995) 46–61.

[15] R. G. Morgan and S. A. Jarvis, Profiling large-scale lazy functional programs, *Journal of Functional Programming* **8** (1998) 370-398.

[16] D. Skillicorn, J. M. D. Hill and W. F. McColl, Questions and answers about BSP, *Scientific Programming* **6** (1997) 249–274.

[17] L. G. Valiant. General purpose parallel architectures, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (North Holland, 1990).

[18] L. G. Valiant. A bridging model for parallel computation, *Communications of the ACM* **33** (1990) 103–111.