Bisseling, Rob H.

Sparse Matrix Computations on Bulk Synchronous Parallel Computers

The Bulk Synchronous Parallel (BSP) programming model is studied in the context of sparse matrix computations. As a case study, a BSP algorithm is developed for sparse Cholesky factorisation.
(This paper appeared in: G. Alefeld, O. Mahrenholtz, and R. Mennicken (Eds.), Proceedings ICIAM'95. Issue 1. Numerical Analysis, Scientific Computing, Computer Science, Akademie Verlag, Berlin, 1996, pp.127-130.)

1. Introduction

Sparse matrix computations are at the heart of many scientific computing applications. Much could be gained if we were able to accelerate such computations by efficiently using parallel computers. This is a difficult task, however, because these computations are mostly irregular. Therefore, sparse matrix computations can benefit from better parallel programming models, but they also form a litmus test for any new model: sparse matrix computations will separate the useful models from the useless ones.

The Bulk Synchronous Parallel (BSP) model was recently proposed by Valiant [7]. It attempts to simultaneously achieve portability and efficiency in parallel computations and thereby to enable general purpose parallel computing [3]. This goal is in sharp contrast with the current state of affairs, which can be characterised by: a parallel software industry that is virtually non-existent; parallel hardware vendors that disappear at an alarming rate; promises of high performance computing that are fulfilled for only a few applications.

Recent developments indicate that the goal of general purpose parallel computing can be achieved. An important development is that shared memory primitives become available for parallel computers with distributed memory. Examples of these are remote write ('put' or 'store') operations and remote read ('get' or 'fetch') operations. These are one-sided communication operations that only involve the initiating processor, and therefore they are more efficient and conceptually simpler than traditional message passing, which involves an active sender and an active receiver. A set of independent one-sided communications must be followed by global synchronisation of the processors to ensure memory integrity. The bulk-synchronisation required by one-sided communications is identical to that of the BSP model. Therefore, we may view the BSP model as providing a theory for the use of one-sided communications.

The aim of this paper is to show how the BSP model can be used in developing and analysing an algorithm for parallel sparse Cholesky factorisation. The *Cholesky factor* L of a real symmetric positive definite matrix A is defined as the lower triangular matrix that satisfies

$$A = LL^T. \tag{1}$$

We assume that A is sparse, and that A has been ordered to maintain sparsity during the factorisation. We also assume that all structural information, such as the sparsity pattern of L and the corresponding elimination tree, is available at the start of the computation. Therefore, we are only concerned with the numerical part of the factorisation.

2. The BSP model

A *BSP computer* consists of a number of processors, each with its own memory, a communication network that provides access to other processor's memories, and a mechanism for global synchronisation. Reading from or writing to memory is fast if the operation is local and it is slower if the memory location belongs to a different processor. There is no distinction in access time between different non-local memories. This implies that the communication network can be viewed as a black box, where the network topology is hidden in the interior. This property is essential for achieving portability. Previous work [2] has shown that direct control over data distribution is crucial to achieving efficiency for sparse matrix computations on BSP computers with realistic system parameters. Therefore, we will ignore the alternative approach based on memory hashing [7].

A *BSP algorithm* consists of a number of supersteps. A *superstep* is either a number of computation steps or an h-relation, both followed by a global synchronisation. An *h-relation* is a communication procedure where each processor sends at most h data to other processors and receives at most h data. Note that two different

communication patterns may have the same h; in that case, the cost function of the BSP model does not distinguish between them. The cost function of the BSP model is the basis for complexity analysis of algorithms and for performance prediction of implementations. There exist a few variants of the cost model, which differ by at most a small constant factor. The variant presented here was proposed in [2]. Its main virtue is simplicity.

The cost of an h-relation, including the cost of synchronisation, is

$$T_{\text{comm}}(h) = hg + l, \tag{2}$$

where g and l are machine dependent parameters and the cost unit is the time of a floating point operation (flop). This cost is charged because of the expected linear increase of communication time with h. The processor that sends/receives the maximum number of elements determines h and hence the communication cost. Asymptotically, for large h, the time of an h-relation is the product of the maximum number of elements sent into or received from the communication network and the time g needed to send or receive one element. We can also view g as the ratio between the global computation throughput and the global communication throughput. The linear cost function includes a nonzero constant because initiating an h-relation incurs a fixed cost. This fixed cost includes: the cost of global synchronisation; (part of) the cost of ensuring that all communicated data have arrived at their destination (processors must do this before they can declare themselves ready for synchronisation); and communication startup costs. We lump all these costs into one parameter l. This parameter l is similar to but not identical with the latency L of the original BSP model [7]. We call l, by a slight abuse of language, the *synchronisation cost* of a superstep. Approximate values for g and l of any particular machine can be obtained by benchmarking a range of *full* h-relations (i.e., h-relations where each processor sends and receives exactly h data), with reals as data. This method of benchmarking produces an upper bound on the cost of actual h-relations.

The cost of a computation superstep with an amount of work w, including the cost of synchronisation, is

$$T_{\text{comp}}(w) = w + l. \tag{3}$$

The amount of work is defined as the maximum number of flops performed by any processor in the superstep. The value of l is taken to be the same as that of a communication superstep, despite the fact that the fixed cost is less; global synchronisation is still necessary, but the other associated costs disappear. The advantage of having one parameter l is simplicity: the total synchronisation cost of an algorithm can be determined by simply counting the supersteps.

The total cost of a BSP algorithm is an expression of the form $a + bg + cl$ [2]. A BSP computer can be characterised by four parameters [3]: the number of processors p, the single-processor speed s, the computation/communication throughput ratio g, and the synchronisation cost l. By analysing the complexity of an algorithm and, independently, benchmarking a computer for its BSP performance, we can predict the execution time of an implementation of the algorithm on that computer. Of course, the accuracy of the prediction depends on how the BSP cost function reflects reality, and this may differ from machine to machine.

Efficient implementations of the BSP model are currently being developed. One such implementation is the Oxford BSP library [5]. This public-domain library is available for many architectures, including clusters of UNIX workstations, shared memory multiprocessors such as the Silicon Graphics Challenge, and massively parallel computers with distributed memory such as the Cray T3D. To give an impression of the wide range of BSP performance: we recently benchmarked a cluster of 16 SUN 4/20 workstations using the Oxford BSP library as: $p = 16$, s=0.41 Mflop/s, $g = 524$, $l = 35011$. Miller [4] benchmarked a Cray T3D as: $p = 256$, s= 10 Mflop/s, $g = 4$, $l = 360$.

3. Algorithm

To derive a parallel algorithm, it is necessary to start with a suitable sequential algorithm. We start with a so-called submatrix Cholesky algorithm, since it exhibits more potential parallelism than other algorithms. Each step of a sequential sparse Cholesky algorithm contains little work, since the number of nonzeros involved is small. Several steps must be combined to achieve bulk in a computation and hence to obtain more potential parallelism. One method of doing this is layered defoliation of the elimination tree [1]. (The nodes of this tree correspond to the columns of L; a child in the tree must be computed before its parent.) The leaves of the tree are numbered first; they form the first layer. The leaves are then deleted from the tree, and the new set of leaves forms the second layer, and so on. The computations in one layer can be taken as one basic step of the sequential algorithm. Since these computations are independent and involve a relatively large amount of work, they can be used to design BSP supersteps.

Figure 1 presents a layered sequential algorithm. For each layer l, m_l columns of L are computed and then used to update the current matrix A. The current total number of columns computed is K. The algorithm expresses

Algorithm SEQ-CHOL. Input A, output $A = L$.
$K := 0$;
for $l := 0$ **to** $nlayer - 1$ **do**
(0) $m := m_l$;
for all $k : K \leq k < K + m$ **do** $a_{kk} := \sqrt{a_{kk}}$;
(2) **for all** $k : K \leq k < K + m$ **do**
for all $i : K + m \leq i < n \wedge a_{ik} \neq 0$ **do** $a_{ik} := a_{ik}/a_{kk}$;
(0') **for all** $k : K \leq k < K + m$ **do**
for all $j : K + m \leq j < n \wedge a_{jk} \neq 0$ **do**
for all $i : j \leq i < n \wedge a_{ik} \neq 0$ **do** $a_{ij} := a_{ij} - a_{ik}a_{jk}$;
$K := K + m$;

Figure 1: Layered sequential algorithm for sparse Cholesky factorisation

sparsity by statements of the form '$a_{ik} \neq 0$'. In an implementation, such testing is avoided by using a suitable sparse datastructure, e.g. a collection of sparse column vectors. The statement labels correspond to supersteps of the parallel algorithm.

The parallel algorithm, see Fig. 2, is derived as follows. (It is a based on a previous algorithm for a square mesh of processors [1].) First, we choose a data distribution. Assume a two-dimensional numbering $P(s,t)$ of processors, with $0 \leq s < M$ and $0 \leq t < N$, where $p = MN$ is the number of processors. A *Cartesian* distribution

$$a_{ij} \longmapsto P(\phi_0(i), \phi_1(j)) \tag{4}$$

limits the amount of communication in most linear algebra computations, since it partitions rows and columns among processor sets of limited size: the sets contain at most $\max(M,N)$ processors. This distribution scheme is sufficiently general to allow optimisation for load balancing and communication reduction. The computation supersteps are obtained by distributing the work according to the data distribution. This gives the computation supersteps (0), (2), and (0'). (Superstep (0') can be combined with superstep (0) of the next layer, to save one superstep.)

The communication supersteps are obtained by following a need-to-know principle. For example, in superstep (1), the pivot element a_{kk} is fetched by the processors that need it for divisions in superstep (2). This is expressed by using the boolean variable col-empty$_s(k)$ which is true if the set of local column nonzeros $\{a_{ik} \mid k < i < n \wedge \phi_0(i) = s \wedge a_{ik} \neq 0\}$ is empty. In a sparse computation, the information about communication requirements may be available only at the sender, or only at the receiver. For this reason, the initiator may sometimes be the sender and sometimes the receiver. For example, the receiver initiates in superstep (1). This is an improvement over previous work [1] where the pivot element is sent to all processors that might need it, i.e. to the processors $P(*,t)$.

In superstep (5), parts of rows k and columns k are fetched, but only if *both* row and column are needed; this improves on the indiscriminate broadcast of the previous algorithm [1]. Here, only the boolean information on the emptiness of rows and columns must be broadcast. This is done in superstep (4). The row and column distribution may be different, in particular since this prevents *diagonal load imbalance* [2,6]. (If $\phi_0 = \phi_1$, processors $P(s,s)$ are overloaded, e.g. in superstep (0).) In superstep (3), the columns k are transposed, to gather sets of nonzeros according to the distribution function ϕ_1, instead of ϕ_0.

The BSP model guides us in developing algorithms, but it also provides us with a tool for complexity analysis. For example, we can roughly estimate the total cost of the Cholesky algorithm by

$$T_p \approx \frac{2nc^2}{p} + \frac{2nc}{\sqrt{p}}g + \frac{6n}{m}l, \tag{5}$$

where n is the matrix size, c the average number of nonzeros per column of L, m the average number of columns per layer. Here, we haven taken $M = N = \sqrt{p}$. The cost estimate is based on the contributions of the most expensive supersteps, (5) and (0'), and on a count of the total number of supersteps.

4. Conclusion

A generic BSP algorithm has been presented which performs communication on the basis of the need-to-know: the only values sent are nonzeros, and they are sent only to processors that need them. Suitable preprocessing can produce a distribution that requires less communication during the subsequent factorisation. The algorithm can

Algorithm BSP-CHOL for processor $P(s,t)$
$K := 0$;
for $l := 0$ **to** $nlayer - 1$ **do**
(0) $m := m_l$;
for all $k : K \le k < K + m \wedge \phi_0(k) = s \wedge \phi_1(k) = t$ **do** $a_{kk} := \sqrt{a_{kk}}$;
(1) **for all** $k : K \le k < K + m \wedge \phi_1(k) = t$ **do**
if not col-empty$_s(k)$ **then** fetch a_{kk} from $P(\phi_0(k), t)$;
(2) **for all** $k : K \le k < K + m \wedge \phi_1(k) = t$ **do**
for all $i : K + m \le i < n \wedge \phi_0(i) = s \wedge a_{ik} \neq 0$ **do** $a_{ik} := a_{ik}/a_{kk}$;
(3) **for all** $k : K \le k < K + m \wedge \phi_1(k) = t$ **do**
for all $i : K + m \le i < n \wedge \phi_0(i) = s \wedge a_{ik} \neq 0$ **do**
store a_{ik} at $P(\phi_0(k), \phi_1(i))$;
(4) **for all** $k : K \le k < K + m \wedge \phi_1(k) = t$ **do** store col-empty$_s(k)$ at $P(s, *)$;
for all $k : K \le k < K + m \wedge \phi_0(k) = s$ **do** store row-empty$_t(k)$ at $P(*, t)$;
(5) **for all** $k : K \le k < K + m$ **do**
if not col-empty$_s(k) \wedge$ not row-empty$_t(k)$ **then**
fetch $\{a_{ik} \mid k < i < n \wedge \phi_0(i) = s \wedge a_{ik} \neq 0\}$ from $P(s, \phi_1(k))$;
fetch $\{a_{ik} \mid k < i < n \wedge \phi_1(i) = t \wedge a_{ik} \neq 0\}$ from $P(\phi_0(k), t)$;
(0') **for all** $k : K \le k < K + m$ **do**
for all $j : K + m \le j < n \wedge \phi_1(j) = t \wedge a_{jk} \neq 0$ **do**
for all $i : j \le i < n \wedge \phi_0(i) = s \wedge a_{ik} \neq 0$ **do** $a_{ij} := a_{ij} - a_{ik}a_{jk}$;
$K := K + m$;

Figure 2: BSP algorithm for sparse Cholesky factorisation

fully benefit from this. As a default distribution, we can use the function $a_{kk} \longmapsto P(k \bmod p)$ and then renumber the processors to obtain two-dimensional processor identifiers. The distribution of the matrix then determines the distribution of the complete matrix. This method is expected to work well because it effectively randomises the computations. Furthermore, the distribution function can be computed by a simple formula, which implies that all processors can compute the location of any data. For irregular distributions, such information must be stored in a table which may be distributed or replicated. Work on an implementation of the algorithm is in progress. Experimental results will be published elsewhere.

Acknowledgements

I would like to thank Frank van der Stappen for numerous discussions on the BSP model. I thank Tom Cheatham, Amr Fahmy, Satish Rao, Dan Stefanescu, Pilar de la Torre, and Leslie Valiant for their hospitality and for many interesting discussions during my recent visit to the US. Furthermore, I acknowledge partial support of this work by the NCF/Cray Research University Grants Program.

5. References

1 Bisseling, R.H., Doup, T.M., Loyens, L.D.J.C.: A parallel Interior Point algorithm for linear programming on a network of transputers; Ann. OR **43** (1993), 51–86.
2 Bisseling, R.H., McColl, W.F.: Scientific computing on bulk synchronous parallel architectures; preprint 836, Dept. Mathematics, Utrecht University, Dec. 1993.
3 McColl, W.F.: General purpose parallel computing; in: Gibbons, A., Spirakis, P. (eds.): Lectures on Parallel Computation; Cambridge University Press, Cambridge 1993, 337–391.
4 Miller, R.: A library for bulk synchronous parallel programming; in: General Purpose Parallel Computing; British Computer Society Parallel Processing Specialist Group 1993, 100–108.
5 Miller, R., Reed, J.: The Oxford BSP library users' guide, version 1.0; Oxford Parallel, Oxford 1993.
6 Rothberg, E., Schreiber, R.: Improved load distribution in parallel sparse Cholesky factorisation; in: Supercomputing '94; IEEE Computer Society 1994.
7 Valiant, L.: A bridging model for parallel computation; Comm. ACM **33** (1990), 103–111.

Addresses: Rob H. Bisseling, Mathematics Department, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands. E-mail: `Rob.Bisseling@math.ruu.nl`.
WWW: `http://www.math.ruu.nl/people/bisseling`