Alcom-FT Technical Report Series
ALCOMFT-TR-01-3

A new way to divide and conquer

Alexandre Tiskin *

Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
tiskin@dcs.warwick.ac.uk

Abstract

Valiant's model of bulk-synchronous parallel (BSP) computation does not allow the programmer to synchronise a subset, rather than the complete set of parallel computer's processors. This is perceived by many to be an obstacle to expressing divide-and-conquer algorithms in the BSP model. We argue that the divide-and-conquer paradigm fits naturally into the BSP model, without any need for subset synchronisation. The proposed method of divide-and-conquer BSP programming is fully compliant with the BSP computation model. The method is based on sequentially interleaved threads of BSP computation, called *superthreads*.

1 Introduction

The model of *bulk-synchronous parallel (BSP) computation* (see [12, 8, 9]) provides a simple and practical framework for general-purpose parallel computing. Its main goal is to support the creation of architecture-independent and scalable parallel software. Key features of BSP are its treatment of the communication medium as an abstract fully connected network, and strict separation of all interaction between processors into point-to-point asynchronous data communication and barrier synchronisation. Synchronisation barriers break up the BSP program into a sequence of *supersteps*, each of which is a parallel asynchronous computation. This top-level sequential structure of BSP algorithms does not compromise their efficiency to any significant degree, but simplifies greatly their design, analysis and implementation. Among BSP programming models, the most prominent one is *the BSPlib standard* [6]. BSPlib follows the BSP superstep discipline by

*Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

defining only one synchronisation primitive: a barrier at which the communication network completes the delivery of any outstanding messages, and all processors reach a synchronised, well-defined global state.

Barrier synchronisation is perceived by many to be too restrictive. Valiant's seminal paper [12] mentions briefly the possibility of synchronising a subset, rather than the complete set of processors, but this possibility is not explored any further in that work. Subset synchronisation is a feature of various other parallel computation models, either independent of BSP (e.g. LogP [3]), or claimed to be extensions of BSP (e.g. dBSP [1], BSPWB [4]). Programming primitives for synchronising processor subsets are defined in the MPI programming standard, and in several existing or proposed libraries with BSP-like functionality (e.g. PUB [2], miniBSP [10]).

Subset synchronisation is usually justified by the following concerns:

- adapting the algorithm to network topology;
- reducing processors' idle time, especially in irregular applications;
- overlapping computation and communication;
- recursively decomposing the computation into independent subproblems — a technique commonly known as *divide-and-conquer*.

The first two concerns are analysed in detail in [11, 5, 7]. It is shown that for typical applications, exploiting network topology has little or no effect on program efficiency, while making programming and debugging much more difficult. The question of reducing processors' idle time is in fact independent from the synchronisation mechanism: both barrier and subset-synchronising approaches require careful load balancing. The third concern is addressed in [11]. It is argued that the loss of efficiency due to the lack of computation-communication overlap is more than outweighed by the advantages of bulk data transfer. Furthermore, for sufficiently large problems it is possible to achieve computation-communication overlap and bulk data transfer simultaneously by controlling the size of communication buffers (see [2]).

The final concern is perhaps the most serious one. Divide-and-conquer is a natural method to design and implement many important algorithms. The method partitions the data set of a problem into a fixed number of subsets, each subset defining a smaller independent problem. The partitioning is applied recursively to subproblems, until each subproblem is small enough to be solved directly. When divide-and-conquer is used on a parallel computer, it is natural to execute the recursive calls on each recursion level as parallel threads. However, writing such a recursive program in the BSPlib standard is virtually impossible: the programming style of BSPlib is SPMD (single program, multiple data), so there is no mechanism even to start several threads in parallel, let alone to synchronise them during execution.

The PUB library [2] extends BSPlib with the concept of a "BSP object". A BSP object is effectively a subset of processors acting as an isolated BSP submachine within a BSP computer. As suggested in [2], BSP objects can be used to interleave several threads of BSP computation running concurrently on a group of processors. However, BSP threads are not an integral part of the PUB programming model, and there is no easy mechanism for thread interleaving.

In this work, we develop the idea of threads and thread interleaving as an integral part of a BSP programming model. We argue that the problems with implementing divide-and-conquer algorithms in BSPlib are due to the SPMD programming style, rather than the underlying "flat" BSP computation model. Therefore, we do not deviate from pure BSP, and do not provide subset synchronisation. Instead, we provide a more sophisticated programming model, that allows the BSP computation to be split into interleaved parallel threads. Our aim is to preserve the simplicity and elegance of BSP, without compromising the BSP discipline of barrier synchronisation. The second aim is to facilitate load balancing, and to allow efficient use of the communication-synchronisation overlap. The next section describes briefly our proposed model.

2 Superthreads: divide-and-conquer BSP programming

From the discussion in the previous section, it is clear that a model for generic divide-and-conquer programming must dispense with the SPMD approach by allowing multiple threads of computation. Each thread is an SPMD-style BSP computation; because such computation consists of supersteps, we call our threads *superthreads*. Our programming model is language-independent; however, throughout our discussion it will be convenient to think of a superthread as an object similar to C++ or Java objects.

Being a self-contained BSP part of the whole BSP computation, a superthread must include (or have access to) all necessary data of the subproblem being solved by the superthread. It must also have a **run** method, describing the computation in terms of standard computation and communication primitives (e.g. C operators and BSPlib library calls). The **run** method consists of supersteps, defined as segments of computation between two successive calls of the barrier synchronisation primitive (e.g. **bsp_sync**).

Since the BSP model does not allow subset synchronisation, every superthread must be either local (i.e. run sequentially on a single processor), or fully-distributed (i.e. run on all p processors of a BSP computer). In the course of its execution, a superthread may switch many times between the local and the fully-distributed mode. The current mode of a superthread must be available by calling a method **is_local**. This method may be invoked by the **run** method, so the process of computation by a superthread may depend on its current mode. To determine the processor on which a local superthread is executed, it should have a method **local_proc**, returning an integer in the range from 0 to $p-1$.

For switching between the local and fully-distributed modes, s superthread must provide a method **switch_proc**. This method is called with an integer argument between 0 and $p-1$ to switch to local execution on the corresponding processor, and with a negative argument to switch to fully-distributed execution. The **switch_proc** method takes charge of all communication required to rearrange the data from one mode of execution to another.

A superthread may create and start (*spawn*) other superthreads. By default, both the parent and the child superthreads run concurrently; however, there should be a mechanism allowing the programmer to suspend the parent superthread until all its children have terminated. At the time of spawning, the locality mode of the child superthread is determined by the mode of the parent superthread. These modes can be subsequently changed, independently of each other, by the superthreads' **switch_proc** methods.

Concurrent execution of superthreads is controlled by the *superthread manager*. This is a distributed BSP program running in SPMD style. Every superthread registers with the manager upon creation, and deregisters upon termination. In one superstep of BSP computation, the manager runs one superstep of each superthread (except the ones waiting for children's termination). While a particular superthread is being executed, other superthreads are suspended at a synchronisation barrier. The manager may also check the internal state of any superthread, and may decide to switch the superthread's locality mode, or to move a local superthread to a different processor. The code of a superthread's **run** method must be sensitive to the possibility that a suspended thread may be resumed in a new locality mode. This can be achieved either by testing the mode at the beginning of every superstep, and then branching the computation according to the result of the test, or by implementing high-level library functions invariant with respect to the locality mode.

A typical divide-and-conquer computation starts as a single, fully distributed main superthread. Suppose the problem of size n is decomposed into two independent subproblems. The main superthread spawns two children, each solving a problem of size $n/2$. The main thread is then suspended until both children have terminated. Each of the children, in its turn, spawns two further superthreads. This recursive process continues under the supervision of the superthread manager, until p superthreads, each solving a problem of size n/p, are created. Reaching this stage would typically require $\log p$ supersteps. At this point, the superthread manager may decide that p independent superthreads of equal sizes should be made local, each running on a separate processor. While all the superthreads are suspended at the synchronisation barrier, the manager will call their **switch_proc** methods, each with a unique processor number. From the next superstep, the execution of superthreads resumes in the local mode. Depending on the nature of the problem, the threads may decide to continue replicating, populating the local processors with their descendants, until n superthreads of size 1 are created. Alternatively, the threads may switch to a flat, non-recursive local algorithm, thus saving on the overhead of superthread spawning and recursion.

Divide-and-conquer algorithms often require combining the subproblems' solutions to obtain the main problem's solution. Such combining can be done in a process symmetric to the described process of decomposition. The local threads perform the combining, recursively or non-recursively, until there are p local threads with their subproblems' solutions. At this point, the superthread manager may decide that the superthreads should now be made fully distributed. While all the superthreads are suspended at the synchronisation barrier, the manager will call their **switch_proc** methods with a negative value, instructing them to switch to the fully distributed mode. From the next superstep, the superthreads will terminate in pairs, resuming the execution of their waiting parent, who will then combine their children's solutions. After another $\log p$ supersteps, the main superthread will terminate with the solution of the original problem.

The superthread manager has complete freedom in deciding when to switch the mode of superthreads' execution. In real-life applications, it must aim both to minimise the amount of data being communicated, and to balance the workload of the processors. In the above example, it is likely that the number of processors p will not be an exact power of two. For better load balancing, the manager may be programmed to *overpartition* the problem by creating, say, a number of subproblems close to $p \log p$. After that, even a random assignment of problems to processors will achieve a load balancing close to optimal.

3 Examples

As an example application of superthreads, consider the computation of the inverse of a nonsingular square matrix (without pivoting). We analyse two cases: a general matrix and a triangular matrix.

Let A be a nonsingular $n \times n$ real matrix. To find the inverse A^{-1}, we use recursive block Gauss–Jordan elimination. For convenience we assume that the resulting matrix A^{-1} must replace the original matrix A. The algorithm works by dividing the matrix into square blocks of size $n/2$,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{1}$$

and then applying block Gauss–Jordan elimination:

$$\begin{aligned}
\bar{A}_{11} &\leftarrow A_{11}^{-1} & \bar{\bar{A}}_{22} &\leftarrow \bar{A}_{22}^{-1} \\
\bar{A}_{12} &\leftarrow \bar{A}_{11} A_{12} & \bar{\bar{A}}_{21} &\leftarrow -\bar{\bar{A}}_{22} \bar{A}_{21} \\
\bar{A}_{21} &\leftarrow A_{21} \bar{A}_{11} & \bar{\bar{A}}_{12} &\leftarrow -\bar{A}_{12} \bar{\bar{A}}_{22} \\
\bar{A}_{22} &\leftarrow A_{22} - A_{21} \bar{A}_{11} A_{12} & \bar{\bar{A}}_{11} &\leftarrow \bar{A}_{11} + \bar{A}_{21} \bar{\bar{A}}_{22} \bar{A}_{12}
\end{aligned} \tag{2}$$

after which every $\bar{\bar{A}}_{ij}$ overwrites A_{ij}. The procedure can be applied recursively to find A_{11}^{-1} and $\bar{A}_{22}^{-1}$. The resulting matrix is

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1} A_{12} \cdot G^{-1} \cdot A_{21} A_{11}^{-1} & -A_{11}^{-1} A_{12} \cdot G^{-1} \\ -G^{-1} \cdot A_{21} A_{11}^{-1} & G^{-1} \end{pmatrix} \tag{3}$$

where $G = A_{22} - A_{21} A_{11}^{-1} A_{12}$.

Due to data dependency, there is no substantial parallelism between block inversion and block multiplication tasks in (2); we can only exploit the parallelism within block multiplication. However, it is still convenient to implement the recursive calls by spawning a superthread computing the assignment $\bar{A}_{11} \leftarrow A_{11}^{-1}$, and, later in the program, another superthread for the assignment $\bar{\bar{A}}_{22} \leftarrow \bar{A}_{22}^{-1}$. For large block sizes, these superthreads are executed in fully-distributed mode. When blocks become sufficiently small, the superthread manager is programmed to switch the superthreads to the local mode, thus avoiding unnecessary synchronisations. The processor for local execution can be chosen arbitrarily, since only one superthread is active at any given time. Although in this application we cannot exploit superthread parallelism, superthreads enable us to organise the recursion in a synchronisation-efficient manner.

Let us now consider the case of a triangular matrix. Let A be an $n \times n$ nonsingular lower triangular matrix. As before, we solve the problem of computing the matrix inverse A^{-1} by recursive block Gauss–Jordan elimination. We assume that the resulting lower triangular matrix A^{-1} must replace the original matrix A. The algorithm works by dividing the matrix into square blocks of size $n/2$,

$$A = \begin{pmatrix} A_{11} & \\ A_{21} & A_{22} \end{pmatrix} \tag{4}$$

and then applying block Gauss–Jordan elimination:

$$\begin{aligned}
\bar{A}_{11} &\leftarrow A_{11}^{-1} \\
\bar{A}_{22} &\leftarrow A_{22}^{-1} \\
\bar{A}_{21} &\leftarrow -\bar{A}_{22} A_{21} \bar{A}_{11}
\end{aligned} \tag{5}$$

after which every $\bar{A}_{ij}$ overwrites A_{ij}. The procedure can be applied recursively to find A_{11}^{-1} and A_{22}^{-1}. The resulting matrix is

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} & \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{pmatrix} \tag{6}$$

An important difference from the case of general matrices is that the block inverses A_{11}^{-1} and A_{22}^{-1} in (5) can be computed concurrently. In order to exploit this feature, the main superthread spawns two children, both running in fully distributed mode. The main superthread then suspends its execution until both children are completed. Each of the children spawns further superthread, so the number of active superthreads double on every recursion level. The superthreads are spawned concurrently, and are therefore interleaved by the superthread manager. On every superstep of the BSP computer, the manager executes one superstep of every active superthread. Similarly to the case of general matrices, the manager avoids unnecessary synchronisations by switching all active superthreads to the local mode, when blocks become sufficiently small. At this point, p concurrent superthreads are active. For reasons of load balancing, it may be necessary to overpartition the problem, and to postpone mode switching by a few levels of recursion. In this case, the assignment of local superthreads to processors may be randomised, although a balanced, deterministic assignment could be more efficient, and just as easy to implement.

4 Conclusion

We have presented a new programming model based on the BSP model of computation. We have argued that the elegant, semantically clean basis of BSP makes our model preferable to subset synchronisation. The situation here is not unlike the one in programming languages: the discipline of structured programming, though restrictive, has clear theoretical and practical advantages over the seemingly more liberal "goto" programming style.

Our model adheres strictly to the BSP discipline of barrier synchronisation, yet at the same time it allows us to program divide-and-conquer algorithms by creating interleaved concurrent threads. Each thread is an independent sequence of BSP supersteps, and is therefore called a superthread. We have illustrated the use of superthreads by some examples. The examples show that superthreads can serve as a tool for developing communication and synchronisation-efficient, load-balanced BSP programs.

References

[1] M. Beran. Decomposable bulk synchronous parallel computers. In *Proceedings of SOFSEM '99*, volume 1725 of *Lecture Notes in Computer*

Science, pages 349–359. Springer-Verlag, 1999.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library — design, implementation, and performance. In *Proceedings of IPPS/SPDP '99*, 1999.

[3] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of ACM*, 39(11):78–85, November 1996.

[4] C. Rodríguez, J. L. Roda, F. Sande, D. G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26:753–767, 2000.

[5] G. Hains. Subset synchronization in BSP computing. In H. R. Arabnia, editor, *Proceedings of PDPTA '98*, volume 1, pages 242–246. CSREA Press, 1998.

[6] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24:1947–1980, 1998.

[7] J. M. R. Martin and A. V. Tiskin. BSP modelling of two-tiered parallel architectures. In B. M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems. Proceedings of WoTUG '99*, volume 57 of *Concurrent Systems Engineering Series*, pages 47–55.

[8] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1995.

[9] W. F. McColl. Universal computing. In L. Bougé et al., editors, *Proceedings of Euro-Par '96 (Part I)*, volume 1123 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 1996.

[10] D. B. Skillicorn. miniBSP: A BSP language and transformation system. 1996.

[11] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[12] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.