Bulk Synchronous Parallel Computing - A Paradigm for Transportable Software

Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software

Thomas Cheatham
Amr Fahmy
Dan C. Stefanescu
Leslie G. Valiant

TR-36-94

December 1994

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in the Proceedings of the 28th Annual Hawaii International Conference on System Sciences, Maui, January 1995.

Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software

Thomas Cheatham*
Amr Fahmy†
Dan C. Stefanescu*
Leslie G. Valiant‡
Aiken Computation Laboratory
Harvard University
33 Oxford St, Cambridge MA 02138
Email:cheatham, amr, dan, valiant@das.harvard.edu

Abstract

A necessary condition for the establishment, on a substantial basis, of a parallel software industry would appear to be the availability of technology for generating transportable software, *i.e. architecture independent software which delivers scalable performance for a wide variety of applications on a wide range of multiprocessor computers. This paper describes H-BSP – a general purpose parallel computing environment for developing transportable algorithms. H-BSP is based on the Bulk Synchronous Parallel Model (BSP), in which a computation involves a number of supersteps, each having several parallel computational threads that synchronize at the end of the superstep. The BSP Model deals explicitly with the notion of communication among computational threads and introduces parameters* g *and* L *that quantify the ratio of communication throughput to computation throughput, and the synchronization period, respectively. These two parameters, together with the number of processors and the problem size, are used to quantify the performance and, therefore, the transportability of given classes of algorithms across machines having different values for these parameters. This paper describes the role of* unbundled *compiler technology in facilitating the development of such a parallel computer environment.*

*Research supported in part by ARPA Contract Nr. F19628-92-C-0113.

†Research supported in part by ARPA Contract Nr. F19628-92-C-0113 and a grant from the National Science Foundation, NSF-CDA-9308833

‡Research supported in part by a grant from the National Science Foundation, NSF-CCR-9200884.

1 Introduction

For a parallel software industry to establish itself on a substantial scale a necessary condition would appear to be that the problem of creating transportable software be solved. A solution to this problem has to encompass two vital issues: it has to accommodate a variety of high level programming styles as is found essential in sequential computing, and it has to offer a technology for compiling programs efficiently onto parallel machines as these continue to evolve. Three aspects of parallelism need to be addressed. One is that of providing a computational model to serve as an alternative to the von Neumann Model that has served us so well in transportability with sequential computations. Another is developing programming language constructs that are appropriate for hosting parallel computations. The final one is developing compilers that produce highly efficient code appropriate for a variety of parallel target architectures.

We propose to address these issues as part of a solution to this problem that takes the view that standardization sufficient to ensure success is unlikely to be achieved at either the language or the architecture level, but does appear to be feasible at the level that the von Neumann model plays in sequential computation, one that is intermediate between language and architecture, and tolerates broad variations in both.

Our proposed solution is based on the Bulk Synchronous Parallel Model (the BSP model for short) ([24, 12]), in which a computation involves a number of *supersteps*, each having several parallel computational threads that synchronize at the end of the superstep. The BSP Model deals explicitly with the notion of communication among computational threads

and introduces parameters g and L that quantify the ratio of computational throughput to communication throughput, and the synchronization period, respectively. These parameters, together with the number of processors and the problem size, are used to quantify the performance and, therefore, the transportability of a given class of algorithms. In order to produce efficient code that is transportable to a variety of machines, programmers working in this framework may make explicit how the execution of the program should depend on these parameters. In other respects, the programming style supported may be more or less conventional.

This paper describes H-BSP (see Figure 1) – a proposed general purpose parallel computing environment for transportable software which subscribes to the BSP Model and consists of:

- BSP-L, an experimental higher level programming language, that serves as a testbed for linguistic constructs appropriate to transportable programs, and whose constructs will be usable for extending parallel Fortran, C or other higher level parallel programming languages.
- A collection of compiler tools (optimizers, code generators, etc.) which, based on the parameters of the computational model, will generate efficient code for a large range of parallel computers([4, 23, 6]).
- A collection of library operations for communication and synchronization appropriate for a BSP runtime system.

For a number of significant computational problems algorithms can be found that are provably efficient on the BSP model for specified ranges of the parameters of the model ([12, 24, 3]). For many other algorithms such static analysis may not be feasible because the communication requirements are less predictable. In these cases simulations will be needed ([22]) to determine the algorithms' behavior over a range of parameter values. The efficiency of the algorithms not optimized for the BSP model by the programmer will depend upon the BSP-style optimizations provided by the compiler. We note that in the special case that communication and computation are well balanced in the machine, i.e. g is close to 1, compilation techniques for simulating shared memory models with provable efficiency are known([24, 25]). While these techniques may be used as a default for machines with large values of g, one expects that in many cases better performance can be achieved by explicit use of the parameters by either the programmer or the compiler.

Transportability among machines with widely different values of p, g and L appears to necessitate that these parameters permeate both upwards to the programming language level and downward in the compilation process to the machine level. This is a crucial aspect of what the BSP approach offers when compared with alternative proposals (e.g. [10, 11, 14]).

Recent work ([19]) reports favorable experience with the Oxford BSP Library which provides six basic BSP primitives to be called from standard sequential languages. The goal of H-BSP is to provide a higher level programming environment, in the same vein as the GPL project ([18]).

Since our overall aim is to experiment with a range of alternative language constructs, compilation techniques and library functions, we intend to take advantage of the *unbundled* compiler technology ([9]). The unbundled compiler consists of a family of components, C_i, for $i = 1, \cdots, m$. In this setting the compilation consists of applying C_1 to source text and, in general, applying C_j to the result of applying C_{j-1}. Adding or modifying language constructs, primitives, or target architectures is accomplished by modifying one or more of the C_j. This work is described in detail in [9] and is the basis for compiling BSP-L as well as other parallel programming languages. Furthermore, the unbundled nature of the compiler raises issues of configuration management whose solution is described in [5].

The rest of the paper illustrates our approach by using as a running example the familiar, yet important problem of matrix multiplication. Section 2 describes an example of a BSP algorithm that is efficient over the full spectrum of parameters of the cost model. Section 3 presents its implementation in BSP-L and discusses pertinent language features while the subsequent section discusses and exemplifies optimization strategies.

2 Efficient BSP algorithms

We believe that in order to generate transportable software it is necessary to develop algorithms which behave efficiently (with respect to the chosen bridging model) for the widest possible range of the parameters of the model and thus for the widest range of high performance computers. As an example we briefly describe an optimal transportable algorithm for the multiplication of two n by n matrices A and B, recently devised by W.McColl and L.Valiant.

The algorithm starts by tiling the input matrices A and B as well as the result matrix C into $p^{2/3}$ square

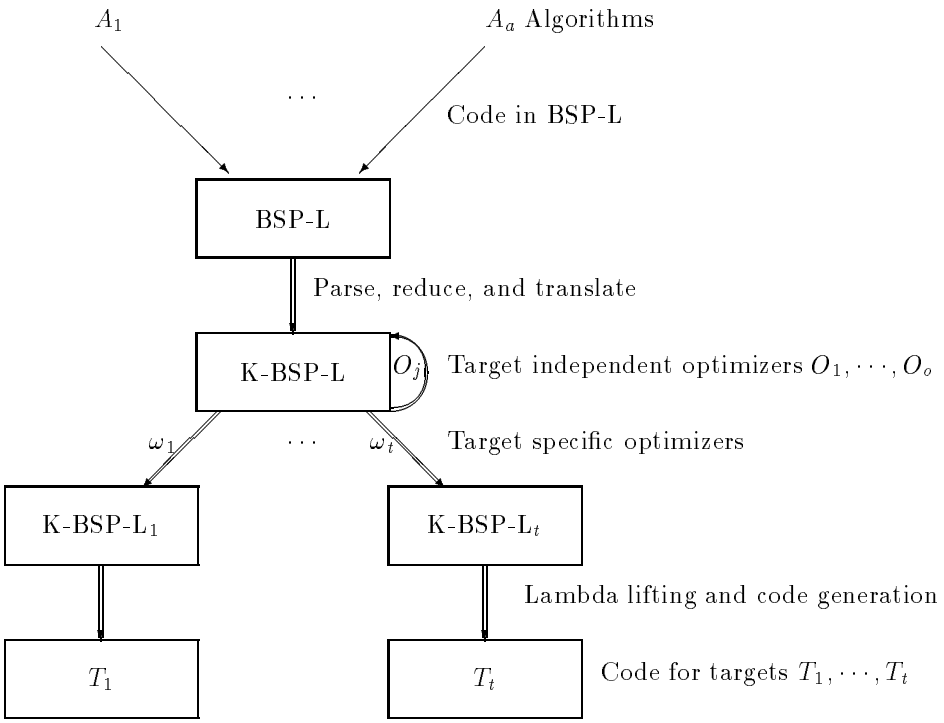

Figure 1: A schematic diagram for H-BSP.

blocks of size $n/p^{1/3}$ each. Each tile in C will then be the sum of $p^{1/3}$ products of pairs of appropriate A and B tiles.

Each processor is assigned the task of performing one of these $p^{1/3}$ products for one of the $p^{2/3}$ tiles of C. Then, assuming that data is initially distributed among the processors equally but possibly arbitrarily, each processor needs to send and receive $2n^2/p^{2/3}$ matrix elements.

Once the p tile products are computed, each tile of C can be obtained by adding the appropriate set of $p^{1/3}$ of these products. Computing each such tile sequentially would provide work for only $p^{2/3}$ processors, corresponding to the current number of C tiles. To ensure full employment without increasing the number of supersteps, we now further partition each tile product into $p^{1/3}$ tiles[1] containing n^2/p elements each, and assign to each processor the task of computing the values of C for a tile of this smaller size. Now each processor sends, receives and sums $p^{1/3}$ of these smaller tiles, each containing n^2/p matrix elements, for a total of $n^2/p^{2/3}$ messages.

The overall algorithm performs $3n^2p^{1/3}$ message transmissions and $2n^3 - n^2$ arithmetic operations. It can be executed on a p-processor BSP machine([24]) in three supersteps[2]. The first performs communication only and takes time $(2n^2/p^{2/3})g$. The second performs the inner products and their final distribution, the latter part being charged as time $(n^2/p^{2/3})g$ and the former as $2n^3/p - n^2/p^{2/3}$. The final superstep performs additions and takes time $n^2/p^{2/3} - n^2/p$. It can be seen that the algorithm is balanced, (i.e. the communication cost does not exceed the computation cost), as long as $g \leq (2n-1)/(3p^{1/3})$. Furthermore, the total synchronization cost is less than the total computation cost provided that $3L \leq (2n^3 - n^2)/p$.

Lower bound proofs ([1, 21, 15]), imply that this algorithm is optimal for communication to small constant factors, independent of n, p, g and L, among algorithms that perform the arithmetic operations of the standard matrix multiplication algorithm. Furthermore, the algorithm is clearly optimal for synchroniza-

[1] The shape of these smaller tiles does not affect the analysis. In the implementation of Figure 2 they are $n/p^{1/3} \times n/p^{2/3}$ rectangles.

[2] See Section 4 for details of how we cost operations.

tion costs since it requires a constant number, namely three, of supersteps. We note that the first of the three supersteps employs the same data distribution as the algorithm given in ([1]) for a different model.

3 Language Constructs

The experience of sequential computing strongly suggests that the advancement of a parallel software industry will crucially depend upon the availability of a host of parallel languages providing a variety of high level programming styles. As such, developing transportable software will require linguistic constructs for exploiting parallelism. We plan to develop and experiment with such constructs as part of the BSP-L language, which will be used in the process of developing transportable software, the ultimate aim being the inclusion of some of these constructs in parallel Fortran, C or other languages of interest to the community.

As an example, we consider the implementation of the efficient matrix multiplication algorithm described in the previous section (see Figure 2). It is important to notice that this implementation depends upon p, the number of processors. This is an elementary example of intentionally allowing the use of model of computation parameters to permeate to the language level (see also [18]). In general the parameters L and g may be used in programs in a similar way.

The BSP-L language [8] is a classically sequential language to which we add several constructs to support parallel processing. For example, the implementation in Figure 2 features sequential constructs like declarations and initializations of variables (e.g. tsize, tsize1 which denote tile sizes) and arrays (e.g. A, B, C and D) and **For** iterators.

3.1 Data partitioning constructs

Parallel programs frequently need to transfer subarrays of data. In order to avoid tedious and mistake prone index calculations, it is helpful to define different *views* of arrays. The construct

Let `Q: R tiled t by s`

generates Q as a $n/t \times m/s$ array of rectangular tiles which represents a new view of the $n \times m$ array R. According to this construction, the element Q[i, j] represents the appropriate $t * s$ elements of array R. The size of the dimensions of the new array Q are given by the built-in function dim(Q,k) which returns the size of the k-th dimension of array Q.

Some other constructs for defining different views on arrays are particularly applicable to programs solving PDEs and are described in [16].

3.2 Constructs for Specifying Parallelism

A process can start new processes using the **Forall** construct. For example, in Figure 2 the code:

```
Forall
  i in 1 to d
    j in 1 to d
      k in 1 to d do
        mat(<i,j,k>;
            d,tsize,TA[i,j],TB[j,k];
            C[i, k+(j-1)*d])
```

generates d^3 processes which are indexed in the iterator set by the triple $< i, j, k >$, $i, j, k \in [1, d]$. As such the processes are organized as a cube of size $d = p^{1/3}$ in each dimension so that we can talk about the process P[i,j,k].

Each of the p processes executes the supersteps associated with the *mat* thread call that corresponds to it. In general the thread construct is defined as follows:

DEF_THREAD `name (index;input;output)`
`body`
END_THREAD

The execution of a thread call:

`name (index ; input ; output)`

proceeds in three stages, each of which may involve several supersteps. In the first stage data *input(i)* is sent to process i. The cost of this pure communication stage depends upon the current data distribution and any repetition among *input(i)*, for the various values of i. For example, in the matrix multiplication case, if arrays A and B are uniformly distributed on the p available processors, the cost of this superstep is $(2n^2/p^{2/3})g$ which matches the analysis in Section 2 of the communication cost of the algorithm. If, on the other hand, arrays A and B reside on the same processor, the cost of this communication is $2n^2g$.

The next stage consists of executing the supersteps prescribed by the *body* of the thread.

Finally, in the last stage, each process i sends the data described by *output(i)* to locations prescribed by the data distribution. The cost of this stage depends on the data distribution and any repetition among *output(i)*. In the matrix multiplication example, the cost

```
Let A,B: array(< 1..N, 1..N >, int)
Let tsize be N/exp(p,1/3)
Let TA: A tiled tsize by tsize
Let TB: B tiled tsize by tsize
Let d be dim(TA,1)                              /* d= p^(1/3) */
Let C: array(< 1..N, 1..N >, int) tiled tsize by tsize/d

DEF_THREAD mat(index:tuple ;
           d:int, tsize:int, A:array(<1..tsize, 1..tsize>, int),
           B:array(<1..tsize, 1..tsize>,int);
           D:array(<1..tsize, 1..tsize/d>, int) initially 0)

            Let C0: array(<1..tsize, 1..tsize>, int)
            Let tsize1 be tsize/d            /* tsize1 = N/p^(2/3) */
            Let TC0: C0 tiled tsize by tsize1
            Let C: array(<1..tsize, 1..tsize1, 1..d>, int)

/*superstep 2: multiplication of tiles and redistribution of smaller tiles */

            For s in 1 to tsize do
              For q in 1 to tsize do
                For r in 1 to tsize do
                   C0[s, r] <- C0[s, r] + A[s, q] * B[q, r]
            For r in 1 to d
                put(<index[1], r, index[3]>, TC0, <1..1, r..r>, _)
            For q in 1 to d
               get(_, C, <1..tsize, 1..tsize1, q>, _)

            synch

/* superstep 3: final summations */

            For q in 1 to tsize
              For r in 1 to tsize1
                For s in 1 to d do
                   D[q,r] <- D[q,r] + C[q,r,s]
END_THREAD

/* Main program: start p threads and name them according to a cube */

Forall i in 1 to d
        j in 1 to d
          k in 1 to d do

/* superstep 1: distribution of tiles of size N/p^(1/3) by thread call*/

            mat(<i,j,k>;d,tsize,TA[i,j],TB[j,k];C[i, k + (j-1)*d] )

/* superstep 4: write back tiles via thread return */
```

Figure 2: BSP-L implementation of the efficient matrix multiplication algorithm

of the last stage depends on the distribution of array C. It would be 0, for a particular uniform distribution of C, and $n^2g(1-1/p)$ if all elements of C reside on the same processor.

The **Forall** construct ends with an implicit barrier synchronization. Thus the statement following the **Forall** is not executed until the execution of all its parallel offspring processes is complete. BSP-L also provides for explicit synchronization of some collection of processes and for that the **synch** primitive is introduced. When executed it blocks the executing process until all its siblings have also issued **synch**.

The transfer of data between processes is done by the communication primitives **put** and **get**.

3.3 Communication

The **put** and **get** primitives are nonblocking and take the form `put(p,A,d,t)` and `get(p,A,d,t)`, where:

- `p` — a process name that identifies the process that is the source (sink) of the data being communicated,
- `A` — an array local to the process executing the put (get),
- `d` — a specification of the sub-array (*slice*)[3] of `A` to be sent (received), and
- `t` — a tag that is used by the sender (receiver) to distinguish among several different messages that might be sent (received) by process `p`.

Each process has an associated *communications buffer* in which data sent to it is stored. The semantics of a **put** issued by `myproc` is then as follows: the slice of `A` specified by `d` is sent to process `p` along with the tag `t`. It is assumed that the data arrives within the same superstep.

The semantics of **get** is the following: when the communications buffer of process `myproc` has data commensurate with `d` received from process `p` and with tag `t`, then this data is removed from the buffer and the subarray A is modified appropriately within the same superstep.

[3] A slice is a set of tuples that specifies a rectangular subarray. For example, $< 1..n, 1..n, q >$ describes the set of tuples $< i, j, k >$ where $1 \leq i, j \leq n$ and $k = q$.

4 Optimizations

There has been considerable work done in developing optimizations for parallel target architectures. For example, [14] discusses message aggregation, message pipelining as well as various optimizations of communications. In [13] there is an algorithm for labeling statements with sync ranks which are used in producing optimized programs with less barrier synchronization. Reference [2] gives an algorithm for communication optimization by solving a set of inequalities.

As described in [24, 17] a BSP program is a sequence of sequential supersteps separated by barrier synchronizations. This organization induces a natural dichotomy on the performance evaluation of a BSP program, and, as a result, on the optimization opportunities for BSP programs.

At a first level of abstraction (the BSP level) the cost of a program is given as the sum of the costs of its supersteps. Each superstep cost is defined[4] to be CMP+COMM+L, where:

- CMP is the maximum computation cost for any process assuming a one-level memory
- COMM is the maximum communication and is computed as hg, where h is the maximum number of messages sent or received by any process and g is a machine parameter denoting the ratio of the number of computation steps/communication steps.
- L, a machine parameter, is the barrier synchronization cost.

The next level of abstraction (the sequential level) details the computation cost of CMP in terms of a register/cache/local memory hierarchy model for the respective platform.

This "separation of concerns" view can be extended naturally to optimization opportunities by distinguishing between BSP-style and sequential-style optimizations.

Superstep explosion is an example of a BSP level optimization. This optimization is typical for code implementing dispersing/combining operations. For example, consider the matrix multiplication implementation and suppose that arrays A and B reside on the same processor. Than the compiler may generate the following code for the first stage of the execution of thread **mat**:

[4] This definition is a slight variant of the ones used in [24, 17, 20].

```
/* distribute data from master to workers */
For i in 1 to d do
  For j in 1 to d do
    For k in 1 to d do
      put(P[i,j,k], TA, <i..i, j..j>, 100)
      put(P[i,j,k], TB, <j..j, k..k>, 200)
```

In the innermost loop the master broadcasts the TA[i, j] tile to d processes P[_, _, k]. One possibility is to use straight message sending at a cost of $C_1 = L + mdg$, for messages of size m, which in this case equals $N^2/p^{2/3}$. An alternative to this approach is to broadcast using a binary tree in which case the cost is $C_2 = log_2(d)(L + 2mg)$. The compiler can choose which of these codes to generate by comparing C_1 and C_2. For example, if $L/g = 200$, as reported in [20] for one measurement, the compiler will choose code implementing straight message broadcast for the case that $d = p^{1/3} = 8$ and $N \leq 112$. Other BSP-style optimizations are described in [7].

As an example of sequential-style optimization, consider the tile multiplication performed in the second superstep of the matrix multiplication program:

```
For s in 1 to tsize do
  For q in 1 to tsize do
    For r in 1 to tsize do
      C0[s, r] <- C0[s, r] + A[s, q] * B[q, r]
```

Based on the cache size of the sequential platform, the compiler can choose a block size b and generate the following code:

```
For qq in 1 to tsize step b
  For rr in 1 to tsize step b
    For s in 1 to tsize do
      For q in qq to qq+b-1 do
        For r in rr to rr+b-1 do
          C0[s, r] <- C0[s, r] + A[s, q] * B[q, r]
```

whose use induces a significant enhancement in performance ([27]).

5 Conclusions

One of the major challenges in parallel computing is the creation, on a substantial scale, of an industry for general purpose parallel software. As a result of intensive efforts, there is now a continuous stream of new parallel computers that provide decreasing price performance ratios. Much work is left to be done, however, before parallel software that is efficient, architecture independent and scalable will be available to fully utilize these machines.

This paper describes an approach to this problem that provides for a variety of high level programming styles and that promises technologies for efficient compilation on a wide variety of existing and evolving machines.

Our tenet is that the optimal area for activity ensuring broad transportability is neither the language nor the architecture levels, but rather the in-between *bridging* level which tolerates significant variations in both. Our approach proposes to adopt the BSP model as the computational model for this level and to generate transportable software by permeating its features in the areas of efficient algorithms, linguistic constructs and compilation techniques.

References

[1] A.Aggarwal, A.Chandra and M.Snir *Communication Complexity of PRAMs*, Theoretical Computer Science, 71(1990), pp 3-28.

[2] S. P. Amarasinghe and M. Lam *Communication Optimization and Code Generation for Distributed Memory Machines* Proceedings of the ACM SIGPLAN'93, Conference on Programming Language Design and Implementation, June 1993

[3] R.H. Bisseling and W. F. McColl *Scientific Computing on Bulk Synchronous Parallel Architectures* Preprint 836, Dept. of Mathematics, Utrecht University, December 1993

[4] T. Cheatham, H. Gao, and D. Stefanescu *A Suite of Analysis Tools Based on a General Purpose Abstract Interpreter*, Proceedings of the International Conference on Compiler Construction, Edinburgh, April 1994

[5] T. Cheatham, A. Fahmy, and D. Stefanescu *Supporting Multiple Evolving Compilers*, SEKE'94, Riga, June 1994

[6] T. Cheatham, *Models, Languages, and Compiler Technology for High Performance Computers*, Workshop on Mathematical Foundations of Computer Science, Kosice, Slovakia, Lecture Notes on Computer Science, Springer Verlag, August 1994.

[7] T. Cheatham, A. Fahmy, and D. Stefanescu *H-BSP – A General Purpose Parallel Computing Environment*, Proceedings of IFIP World

Congress, Vol. 1, pp 515-520, Hamburg, August 1994

[8] T. Cheatham, A. Fahmy, and D. Stefanescu *A Compiler for BSP-L, A Programming Language for the Bulk Synchronous Processing Model*, Proceedings of IEEE TENCON'94, Singapore, August 1994

[9] T.Cheatham *The Unbundled Compiler*, Technical Report, Harvard University, 1993

[10] D. E. Culler, et al. *Introduction to Split-C* EECS, UC Berkeley, Berkeley, CA 94720, April 1993

[11] A. Geist, et al. *PVM3 Users Guide and Reference Manual* ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee, May 1993

[12] A.V.Gerbessiotis and L.G.Valiant *Direct bulk-synchronous parallel algorithms*, Third Scandinavian Workshop on Algorithm Theory, vol. 621, pages 1-18, Springer Verlag, 1992. Extended version in Journal of Parallel and Distributed Computing, 22, pp. 251-267, 1994

[13] E. Heinz, M. Phillipson *Synchronization Barrier Elimination in Synchronous FORALLs* TR13/93 University of Karlsruhe, April 1993

[14] S. Hiranandani, K.Kennedy, C.Tseng *Compiling Fortran D for MIMD Distributed-Memory Machines* Communications of the ACM, August 1992

[15] J.W.Hong and H.T.Kung *I/O Complexity: The Red-Blue Pebble Game* Proceedings of the 13-th ACM Symposium on Theory of Computing, pp 326-333, 1981

[16] V.Kathail and D. Stefanescu *A Data Mapping Parallel Language* TR-21-89, Center for Research in Computing Technology, Harvard University, December 1989

[17] W. F. McColl *General Purpose Parallel Computing*, In A.M. Gibbons and P.Spirakis, editors, Lectures on Parallel Computation, Proc. 1991 ALCOM Spring School on Parallel Computation, vol 4 of Cambridge International Series on Parallel Computation, Cambridge University Press, 1993

[18] W. F. McColl *BSP Programming* In DIMACS Series of Discrete Mathematics and Theoretical Computer Science, 1994. To appear.

[19] W. F. McColl *Scalable Parallel Computing: A Grand Unified Theory and its Practical Development* Proceedings of IFIP World Congress, Vol. 1, pp 539-546, Hamburg, August 1994

[20] R. Miller and J. Reed *The Oxford BSP Library. Users Guide. Version 1.0*, Oxford University, 1994

[21] M. Paterson. Manuscript. 1994

[22] J. P. Singh, E. Rothberg, and A. Gupta *Modeling Communication in Parallel Algorithms: A Fruitful Interaction Between Theory and Systems* Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, 1994.

[23] D. Stefanescu and Y. Zhou *An Equational Framework for the Abstract Analysis of Functional Programs*, Proceedings of ACM Conference on Lisp and Functional Programming, Orlando, 1994.

[24] L. G. Valiant *A Bridging Model for Parallel Computation* Communications of the ACM, 33(8):103-111, 1990

[25] L. G. Valiant *A combining mechanism for parallel computers.* In Parallel Architectures and Their Efficient Use, Proceedings of First Heinz Nixdorf Symposium, Paderborn,Germany, November 1992. Lecture Notes in Computer Science Vol678, Springer-Verlag, 1-10.

[26] L. G. Valiant *Why BSP Computers?* Proceedings of the 7-th International Parallel Processing Symposium, pp 2-5, IEEE Computer Society Press, Los Alamitos, CA, 1993

[27] M.E.Wolf and M.Lam *A Data Locality Optimizing Algorithm*, Conference on Programming Languages Design and Implementation'91, 1991