CHAPTER 3

Practical Uses of Domain Reduction Methods

Craig C. Douglas*

Abstract. A model of computation that is correct for all digital computers made to date is defined. This includes serial, parallel, and distributed environments. Use of this model for designing highly portable software using the Linda system is discussed. A sparse matrix library for solving partial differential equations using domain reduction or decomposition techniques for parallel or distributed machines is mentioned.

1. Introduction. In this paper, a theoretical model of computation is defined, which is currently being applied to the design of sparse matrix software for domain reduction and decomposition methods. This software is probably not optimal in any computing environment, but is close enough to be of interest. However, it is highly portable, and can be fine tuned to a particular machine according to the parameters of the theoretical model.

In §2, a model of computation is defined which realistically fits all conventional computers made to date, serial or parallel. In §3, a brief description of the Linda system is presented. In §4, a brief description of the domain reduction method is presented. In §5, a sparse matrix package, Tupleware, is briefly discussed.

2. A Statistical Model of Computation. One of the standard models for parallel computation is the PRAM model. This model assumes a global clock, that all of the processors communicate through a shared memory, and that all processors can access an element of memory anywhere in the memory in constant time. Various memory access methods have been characterized (see [33]), including concurrent read and write (see [24]), concurrent read and exclusive write (see [6] and [23]), exclusive read and write (see [31]), common concurrent read and write (alternating read and write cycles, see [28]), concurrent read and owner write (part of memory is assigned to a processor, see [21]), and fetch and add models (see [25], [27], and [32]).

There are numerous variations on the PRAM model, including asynchronous models (APRAM, see [12]) where local clocks are assumed, and distributed models (DRAM, see [30]) where a local network is assumed. However, all of these models are still quite simple in nature.

Some computer manufacturers "guarantee" access times for simple data elements or a block of data (e.g., an array or a character string). Unfortunately, these guarantees never take into account the software interface, possible hardware interrupts, direct memory access capabilities that steal memory cycles, if all of the processes are actually running simultaneously, poor compilers, or a variety of similar pitfalls.

* Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 and Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520. E-mail: *bells@ibm.com, bells at yktvmv.bitnet,* or *douglas-craig@cs.yale.edu*

One of the advantages of a PRAM model is that its parameters can usually be determined by looking up the claims of the manufacturers. This simplifies theoretical analysis of complicated algorithms since a real machine never has to be used in order to predict the behavior. This argument has been used to show that PRAM models are simple enough to use and still quite powerful.

The other extreme are models which try to quantify how long every machine operation will take. This includes the time it takes for an add, to move a byte from main memory to a register, an interrupt, paging or swapping, and input or output to an external device. These models are too complicated to be of practical use to most people who want to design highly portable software.

A much more realistic model makes assumptions on how long operations take based on numeric intervals, not absolute numbers. This is actually a compromise between the two extremes in models of computation just described. There are two types of intervals of interest.

DEFINITION 2.1. *Define the set of all simple real intervals by*

$$\mathcal{I} = \{[m, n] \mid m \leq n,\ m, m \in \mathbb{R}\}.$$

Define the set of all function based real intervals by

$$\mathcal{F} = \{[\min f(\cdot), \max f(\cdot)] \mid f : \mathbb{R} \to \mathbb{R},\ f \text{ possibly multivalued}\}$$

DEFINITION 2.2. *The statistical model of computation is defined in terms of the following parameters and sets of intervals.*

p	*tasks (possibly processors)*
$1 \leq C \leq p$	*clocks*
$\{e_i \mid e_i \in \mathcal{I}\}_{i=1}^{d}$	*time to access a simple data element*
$\{\ell_i(N) + t_i(N) \mid \ell_i + t_i \in \mathcal{F}\}_{i=1}^{b}$	*time to access a block of data of length N*
$\{s_i(N) \mid s_i \in \mathcal{F}\}$	*startup time for a task of size N*

For a real computer, many of these intervals can range from infinitesimal to infinite. Hence, we also want to know an X *percent confidence level set of intervals* in order to model "normal" behavior (common values of X are 90, 95, and 98).

To be practical, the number of intervals in each set cannot be very large. Otherwise, the model becomes too complicated to be of value for most people. However, it should be flexible enough to allow many intervals in a set for those who wish to be more precise in their modeling.

2.1. Major components of this model. Since some systems allow threads, a task oriented instead of processor oriented model is preferable. The number of clocks is clearly dependent on how many clusters of how many independent processors are used, and whether or not they are tightly coupled by clock cycles.

The time to access a simple datum is clearly dependent on if it is stored in a machine register, cache, local or global memory bank, and how many lines run in or out of the various memory hierarchies. For simple data elements, there might only be 3 intervals (register, local memory, or global memory). It could be larger, if caching and paging to an external memory device (e.g., a disk drive) are present (see [2] for a data flow model). It could also be as small as one (just memory movement), which is similar to the PRAM models.

Accessing blocks of data can incur both a latency time delay $\ell_i(N)$ (possibly multiple times) as well as time for a string of bytes (which may also be modulo some number).

Finally, consider modeling the startup time of a task. This is much more complicated than modeling data movement. On some machines, all local memories are loaded once with identical copies of an initial memory set, and these must be re-used. This makes predicting the startup time relatively easy.

On other machines, a "fork" system call is used to start another processor. This may cause the entire memory set (program and data space) to be duplicated. Depending on the machine environment, this may cause memory to be copied to local memory, another memory box, or across a communications network to another machine's memory. This can take a very long time if the memory set has grown to a large size and is certainly a function of the amount of space being

duplicated. However, the memory set associated with a completed task can be re-used without duplication if care is exercised. Then the startup time may be much less than it was the first time for an identically sized memory set. Hence, the startup time can only be modeled using multivalued functions.

2.2. Typical bad examples for modeling. Consider the example of a communications network, combining network, or data switch that is accessed. In each case, machines exist which do not guarantee a useful upper bound on when data passes through a bottleneck, independent of the communications topology. For example, suppose multiple pieces of data must pass through a single data path. Assume the method of deciding which datum will be moved along the path is to choose a random one. Then, it is quite possible to have a datum stuck indefinitely, causing all of the processors to eventually halt waiting for that one datum to get to its destination.

A similar bad case is access by multiple processors to the same shared memory bank. On some machines, if p processors try to access the same row of memory chips (or a collection of rows of chips), a p–fold slow down in executions occurs. This situation arises most often when a compiler optimizes the wrong loop with respect to parallelism. The standard solution offered by manufacturers is to suggest that the programmer rewrite their program using a better algorithm (specifically for that particular machine, of course). This hinders portability rather severely.

3. Linda Approach to Computation. A number of semi-portable parallel programming environments now exist. Some of these systems mimic execution time libraries supplied by some hardware manufacturer, others are complete languages. The Linda system is one of the few which is language independent in concept (see [5], [9], [10], and [11], and [29]). Implementations are available for ADA, C, FORTRAN–77, LISP, and PostScript from a variety of sources, including commercial and academic.

For each Linda implementation, there is a preprocessor and execution time system. The preprocessor adds a new data type (a *tuple*) and six operators which manipulate tuples or *tasks*. As a result, Linda can be interpreted as a global database system, rather than as a language. The data is maintained in *tuple space*, which can be distributed across a number of processors. The processors may be part of a tightly coupled system (e.g., a shared or distributed memory machine), or a loosely coupled system (e.g., where the processors are connected via a network).

Data flow is analyzed both at compile time and during execution. On distributed memory systems, efforts are made to prefetch tuples based on this analysis. There can be a penalty for using this type of system. When transferring small tuples (e.g., a few words of memory) between processors, the extra overhead of Linda can be as high as 40%. However, when transferring large tuples (e.g., several thousand words of memory), the penalty is less than 1%.

The actual Linda operators are as follows.

eval	Evaluate a tuple in parallel.
in	Move a tuple from tuple space (wait for one if necessary).
inp	Move a tuple (if it exists) from tuple space.
out	Copy a tuple into tuple space.
rd	Copy a tuple from tuple space (wait for one if necessary).
rdp	Copy a tuple (if it exists) from tuple space.

The operators ending in "p" return a success or failure value, depending on if a tuple matches the pattern or not. In either case, the routines return immediately.

A tuple can have any number of entries, and each entry can be almost anything. Tuples with mixed data types (i.e., character strings, arrays, reals, integers, and constants) are allowed. In the case of the eval() operator, one of the tuple entries typically is a function. This function is evaluated independently of the current task, either as a thread or separate process on this processor, or on another processor (depending on the Linda implementation).

Accessing tuples in tuple space (by in(), inp(), rd(), or rdp()) is complicated by a pattern matching feature: some arguments may be formal (i.e., completely specified) and some may be informal (anything in tuple space of the same data type can match these). Informal arguments are preceded by a question mark. For example, consider the following Linda–C code fragment.

```
in(4, "tupleware", ? x);
```

would input *any* tuple which has as its first entry an integer 4, the character string (of length 9)

tupleware as its second entry, and the same specific data type as x. In case multiple tuples match, a (pseudo) random one is chosen to match. A similar example is

in(4, "*tupleware*", ? $x : n$);

In this case, an array of the same specific data type as x would match. This array would be stored in x and its length in n. Finally, suppose x is of data type *float*. Then,

in(4, "*tupleware*", ? float *:);

would cause a tuple to be deleted from tuple space.

Accessing a simple data element or a block of data cannot be characterized by the PRAM model or its simple variants. Unless the preprocessor data flow analyzer has eliminated a tuple space operator, there is always software involved in accessing tuple space (for either input or output). Further, there is possible communications on distributed or networked memory systems. On an Intel iPSC or a network of machines, the tuple header is hashed first to determine which processor holds the tuple (see [5]). The tuple is fetched, if necessary, and then decomposed to the requested places in local memory. Alternately, it is just transmitted to the correct processor, which must store the tuple, thus causing an unfriendly interruption of another processor's computations.

We determined the intervals for a particularly bad case, namely when a number of processors simultaneously try to get the same tuple from tuple space (using rd). The tuple has a character string key and a vector of double precision (64 bit) words. The size of the vectors are contained in Tables 3.1–3.4. The tuple was requested 2000 times by each processor. The maximum and minimum are given in the tables along with 95% and 98% confidence intervals.

Due to possibly unexpected interrupts on each processor, it is almost impossible to assume that tasks are running exactly in parallel. On distributed systems, a large amount of hidden input/output may occur. On time shared systems (e.g., Sequent or Encore computers), competition with other users occurs, including background jobs run by the operating system. On a simulator running on a single processor, there is no hope of real parallelism. These situations can be modeled with intervals rather easily, but cannot be modeled with absolute numbers in a reasonable fashion.

TABLE 3.1
Intervals for an 8 processor Sequent S-81 Symmetry system

				95%		98%	
Size	Average	Min	Max	Min	Max	Min	Max
1	2.5e–4	0	0.01	0	0.01	0	0.01
100	2.1e–3	0	0.02	0	0.01	0	0.01
1000	2.1e–2	0	0.19	0	0.07	0	0.09
10000	3.0e–1	0.04	6.25	0.04	2.01	0.04	2.92

TABLE 3.2
Intervals for an 16 processor Encore Multimax (APC–02) system

				95%		98%	
Size	Average	Min	Max	Min	Max	Min	Max
1	9.0e–4	3.2e–4	1.7e–2	3.5e–4	3.4e–3	3.2e–4	4.9e–3
100	6.2e–3	6.8e–4	8.7e–2	7.4e–4	2.7e–2	7.0e–4	3.2e–2
1000	5.6e–2	4.1e–3	9.0e–1	6.7e–3	2.4e–1	6.6e–3	3.0e–1
10000	7.7e–1	9.0e–2	9.6e+0	9.7e–2	3.1e+0	9.5e–2	3.8e+0

TABLE 3.3
Intervals for a networked 8 SUN SparcStation/1 system

				95%		98%	
Size	Average	Min	Max	Min	Max	Min	Max
1	8.3e–02	1.4e–04	1.5e+01	1.4e–04	4.9e–01	1.4e–04	6.5e–01
100	1.1e–01	2.5e–04	9.6e–01	2.5e–04	3.7e–01	2.5e–04	5.1e–01
1000	1.1e–01	1.3e–03	5.1e+01	1.3e–03	2.2e–01	1.3e–03	3.1e–01
10000	9.3e–01	1.4e–02	1.5e+00	1.4e–02	1.3e+00	1.4e–02	1.3e+00

TABLE 3.4
Intervals for an 64 processor Intel iPSC/2 system

Processors	Size	Average	Min	Max	95% Min	95% Max	98% Min	98% Max
1	1	3.6e–4	0	3.0e–3	0	1.0e–3	0	1.0e–3
2	1	1.4e–3	1.0e–3	2.0e–3	1.0e–3	2.0e–3	1.0e–3	2.0e–3
4	1	2.1e–3	1.0e–3	3.0e–3	2.0e–3	3.0e–3	2.0e–3	3.0e–3
8	1	4.9e–3	1.0e–3	6.0e–3	4.0e–3	5.0e–3	4.0e–3	5.0e–3
16	1	1.1e–2	6.0e–3	1.1e–2	1.0e–2	1.1e–2	1.0e–2	1.1e–2
32	1	2.1e–2	1.0e–3	2.2e–2	2.0e–2	2.2e–2	2.0e–2	2.2e–2
64	1	2.1e–2	1.0e–3	2.3e–2	2.1e–2	2.2e–2	2.1e–2	2.2e–2
1	100	5.2e–4	0	5.0e–3	0	1.0e–3	0	1.0e–3
2	100	2.0e–3	1.0e–3	3.0e–3	1.0e–3	2.0e–3	1.0e–3	2.0e–3
4	100	4.8e–3	2.0e–3	5.0e–3	4.0e–3	5.0e–3	4.0e–3	5.0e–3
8	100	8.3e–3	2.0e–3	9.0e–3	8.0e–3	9.0e–3	8.0e–3	9.0e–3
16	100	1.8e–2	2.0e–3	1.9e–2	1.7e–2	1.8e–2	1.7e–2	1.9e–2
32	100	3.6e–2	2.0e–3	3.7e–2	3.5e–2	3.7e–2	3.5e–2	3.7e–2
64	100	3.6e–2	2.0e–3	3.7e–2	3.6e–2	3.7e–2	3.5e–2	3.7e–2
1	1000	4.3e–3	4.0e–3	6.0e–3	4.0e–3	5.0e–3	4.0e–3	5.0e–3
2	1000	8.0e–3	5.0e–3	9.0e–3	8.0e–3	9.0e–3	8.0e–3	9.0e–3
4	1000	1.4e–2	6.0e–3	1.5e–2	1.4e–2	1.5e–2	1.4e–2	1.5e–2
8	1000	1.0e–2	8.0e–3	1.9e–2	5.0e–3	1.9e–2	5.0e–3	1.9e–2
1	10000	3.0e–2	3.0e–2	3.1e–2	3.0e–2	3.1e–2	3.0e–2	3.1e–2
2	10000	4.1e+0	3.1e–2	1.3e+1	3.1e–2	1.2e+1	3.1e–2	1.3e+1
4	10000	9.4e+0	3.1e–2	1.3e+1	3.1e–2	1.2e+1	3.1e–2	1.3e+1

On a Sequent or Encore parallel processor, all tasks eval()'ed are forked, which causes the entire current memory set to be copied. On an Intel iPSC (any model) or network of workstations, all processors start with an identical copy of an initial memory set; all tasks eval()'ed use either the initial or a used memory set. The Intel model is very useful since it encourages defensive programming, and does not allow a fork to occur after the memory set has grown (by memory pages being touched as data is changed or generated) to an unreasonable size. Smart implementations of Linda force re-use of memory sets when possible, not regeneration.

4. Domain reduction methods. In this paper, the linear problem

$$Lx = b \tag{4.1}$$

is posed on a linear space V. A collection of independent problems on subspaces of V is used to solve (4.1).

A decomposition of V into subspaces $\{V_i\}_{i=1}^m$ such that

$$V = \bigoplus_{i=1}^{m} V_i \tag{4.2}$$

induces a family of projections $\{\Pi_i\}_{i=1}^m$ given by

$$u = \sum_{i=1}^{m} u_i, \quad u_i = \Pi_i u \in V_i. \tag{4.3}$$

It is known [19] that if the projections Π_i commute with L, then $LV_i \subset V_i$ and (4.1) decomposes into independent problems

$$L_i u_i = f_i \equiv \Pi_i f,$$

where L_i is the restriction of L onto the subspace V_i,

$$L_i = L|_{V_i} = L\Pi_i.$$

The projections and subspaces studied in [8], [15], [18], and [20] are induced by various symmetries of the domain. Use of related group theory approaches is well known in bifurcation theory (see [1]).

It is therefore natural to study the decompositions from a systematic group theory point of view [17]. Using Abelian groups, we show that in decomposing a space V, the projections commute with L if and only if there exists a commutative group $\mathcal{G}$ of automorphisms of V which generates the decomposition and commutes with L. This covers the the constructive interfence theory in [19] and the domain reduction theory in [20]. By using the correct representations, this also covers the high way domain reductions in [8], [15], and [18], which appear at first to be noncommutative.

In applications of symmetry to differential equations, it is usually assumed that the automorphism group $\mathcal{G}$ is obtained by symmetries of the underlying physical domain. In the Abelian group theory, we made no such assumption, but assumed that the group was commutative. In general, the symmetry groups are not commutative.

Let V be a functional space on the domain Ω and $\gamma : \Omega \to \Omega$, one-to-one and onto. Define $Gu = u \circ \gamma$. If the range of G lies in V, then we say that the mapping $G : V \to V$ is induced by γ. This mapping is always one-to-one. However, it may or may not be an automorphism of V, depending on the definition of V and smoothness of γ. For example, if γ is an isometry, then G will be an automorphism in all usual spaces of continuous and differentiable functions and in Sobolev spaces. However, not all automorphisms obtained by the use of the Abelian group technique in [17] can be expressed as induced by a one-to-one, onto mapping of the domain.

In the noncommutative cases, problems on rectangular (in 2 dimensions) domains are reduced to a collection of similar problems on subdomains, some of which are triangular in nature. The analysis in [17] allows us to determine the correct boundary conditions for reducing problems on triangles to ones on subtriangles. This allows a recursive definition similar to multigrid (see [3], [4], [7], [13], [16], [22], and [26]). Unlike multigrid, we can easily produce a direct method instead of iterative.

5. Tupleware. A software package has been produced using the Linda system for solving large sparse linear systems of equations. Details of this package can be found in [14].

A layered design is used in the package: there are serial, ugly parallel, and clean parallel routines. An ugly parallel routine runs on an individual processor and communicates with other processors running the same routine. A clean parallel routine is called serially by the user and interfaces with a set of ugly parallel routines. The ugly parallel routines are faster than the clean routines, but for large enough problems, there is no noticeable difference.

For a constant s, vectors x, y, and z, and a sparse matrix A, routines exist to compute $x^T y$, $z = y + sx$, and sparse matrix-vector multiplication (actually, $y = Ax$, $y = A^T x$, $y = y \pm Ax$, or $y = y \pm A^T x$). Routines also exist to analyze a matrix, build diag(A), redistribute a vector across processors, and print distributed objects. These routines are used to implement conjugate gradients, conjugate gradients squared (see [34]), multigrid, and some simple domain reduction methods (see [8] and [18], [19], and [20]).

We store a distributed sparse matrix as a collection of blocks. The blocks can be subdivided into overlapping subblocks. A matrix A is stored hierarchically in tuple space. All tuples are keyed using a character string. To access a subblock b on processor p, a description is first read. Getting the coefficients and column indices requires a second access to tuple space. Special matrices, like ±Identity matrix, are treated specially, and the coefficients and column indices are not stored. Certain routines need some global information about A, requiring a possible third access to tuple space.

Unlike many sparse matrix packages available today, no array of pointers into the column index array is required. Further, for an $N \times N$ matrix with M rows of all zeros, information is stored only for the $N - M$ rows which have nonzeros, not for all N rows. This format does not guarantee, however, that only nonzero coefficients are stored: any diagonal element $a_{ii} = 0$ is stored. This format really assumes that the matrices will be used row by row sequentially rather than in a random fashion.

The principal application so far has been to the domain reduction method. Here, a number of boundary value problems are solved which only differ in the boundary conditions. The matrices are sparse and can be thought of identical to the others up to an extremely sparse matrix. Asymptotically in the order of the matrices, Tupleware's storage format allows a p–fold reduction in storage (for p similar problems), and a p–fold reduction in the matrix generation time (see [14]).

Tupleware is designed to access a small set of arrays which characterize the machine it is running on. These arrays include information on how long it takes to move certain size objects (e.g., a sparse matrices or vectors of certain sizes). These are intervals and can be used to determine how to distribute data across a machine and how to schedule computation. Very simple programs generate the intervals which can be included in the Tupleware codes.

6. Conclusions. Before embarking on any new software package, knowledge about how a number of computer architectures operate should be considered essential. There are now a large number of computers: serial (simple, vector, very long instruction word), parallel (serial, vector, cluster, MIMD, and/or SIMD), and distributed (all of the above). Unfortunately, very few packages do more than try to adapt to parallel or distributed environments poorly. It should be obvious that a good model of computing is essential to producing anything resembling portable codes. It is not good enough to require a (nearly) complete rewrite of a very large code (possibly in machine language) every time a new machine comes along.

Finally, spending large amounts of intellect trying to parallelize small or moderate sized problems is a waste of time. People who need parallel computers sufficiently to worry about the time spent on problems which can be solved in a few seconds on an inexpensive workstation typically have a huge number of problems to solve. Most parallel methods are not close to 100% efficient in processor utilization. Hence, it is more efficient to solve individual problems on separate (single) processors, guaranteeing approximately 100% processor efficiency.

REFERENCES

[1] E. ALLGOWER, K. BÖHMER, AND M. ZHEN, *A generalized equilbranching lemma with applications in* $D_4 \times Z_2$ *symmetric elliptic problems: part 1*, Tech. Report 9, Philipps-Universität Marburg, Marburg, Germany, 1990.

[2] B. ALPERN, L. CARTER, AND E. FEIG, *Uniform memory heirarchies*, Tech. Report 16069, IBM Research Division, Yorktown Heights, New York, 1990.

[3] G. P. ASTRAKHANTSEV, *An iterative method of solving elliptic net problems*, Z. Vycisl. Mat. i. Mat. Fiz., 11 (1971), pp. 439–448.

[4] N. S. BAKHVALOV, *On the convergence of a relaxation method under natural constraints on an elliptic operator*, Z. Vycisl. Mat. i. Mat. Fiz., 6 (1966), pp. 861–883.

[5] R. BJORNSON, N. CARRIERO, AND D. GELERNTER, *The implementation and performance of hypercube Linda*, Tech. Report 690, Department of Computer Science, Yale University, New Haven, 1989.

[6] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York, 1975.

[7] A. BRANDT, *Multi-level adaptive solution to boundary-value problems*, Math. Comp., 31 (1977), pp. 333–390.

[8] F. BREZZI, C. C. DOUGLAS, AND L. D. MARINI, *A parallel domain reduction method*, Numer. Meth. for PDE, 5 (1989), pp. 195–202.

[9] N. CARRIERO, *Implementation of tuple space machines*, PhD thesis, Yale University, December 1987. Also, Computer Science Department, Yale University, Technical Report 567.

[10] N. CARRIERO AND D. GELERNTER, *The S/Net's Linda kernel*, ACM Trans. Comp. Sys., (1986).

[11] ———, *How to write parallel programs: a guide to the perplexed*, Tech. Report 628, Department of Computer Science, Yale University, New Haven, 1989. To appear in ACM Comp. Surveys.

[12] R. COLE, *The APRAM: incorporating asynchrony into the PRAM model*, in Opportunities and Constraints of Parallel Computing, J. L. C. Sanz, ed., Springer–Verlag, New York, 1989, pp. 25–28.

[13] C. C. DOUGLAS, *Multi-grid algorithms with applications to elliptic boundary-value problems*, SIAM J. Numer. Anal., 21 (1984), pp. 236–254.

[14] ———, *A tupleware approach to domain decomposition methods*, Tech. Report 15360, IBM Research Division, Yorktown Heights, New York, 1990. To appear in the second special issue of *Applied Numerical Mathematics* on domain decomposition methods (1990 or 1991).

[15] ———, *A variation of the Schwarz alternating method: the domain decomposition reduction method*, in Domain Decomposition Methods for Partial Differential Equations III, T. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 191–201.

[16] C. C. DOUGLAS AND J. DOUGLAS, *Abstract multilevel convergence theory requires almost no assumptions*, Tech. Report 15853, IBM Research Division, Yorktown Heights, New York, 1990. Submitted.

[17] C. C. DOUGLAS AND J. MANDEL, *A group theoretic approach to the domain reduction method*. In preparation.

[18] ———, *The domain reduction method: high way reduction in three dimensions and convergence with inexact solvers*, in Fourth Copper Mountain Conference on Multigrid Methods, J. Mandel, S. F. McCormick, J. E. Dendy, C. Farhat, G. Lonsdale, S. V. Parter, J. W. Ruge, and K. Stüben, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1989, pp. 149–160.

[19] C. C. DOUGLAS AND W. L. MIRANKER, *Constructive interference in parallel algorithms*, SIAM J. Numer. Anal., 25 (1988), pp. 376–398.

[20] C. C. DOUGLAS AND B. F. SMITH, *Using symmetries and antisymmetries to analyze a parallel multigrid algorithm*, SIAM J. Numer. Anal., 26 (1989), pp. 1439–1461.

[21] P. W. DYMOND AND W. L. RUZZO, *Parallel random access machines with owned global memory and deterministic context free language recognition*, in Proceedings of 13th International Colloquium on Automata, Languages, and Programming, L. Kott, ed., Springer-Verlag, 1986, pp. 95–104.
[22] R. P. FEDERENKO, *A relaxation method for solving elliptic difference equations*, Z. Vycisl. Mat. i. Mat. Fiz., 1 (1961), pp. 922–927.
[23] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in 10th ACM Symposium on Theory of Computing, San Diego, CA, ACM, 1978, pp. 114–118.
[24] L. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, in 10th ACM Symposium on Theory of Computing, San Diego, CA, ACM, 1978, pp. 89–94.
[25] A. GOTTLIEB, R. GRISHMAN, C. P. KRUSKAL, K. P. MCAULIFFE, L. RUDOLPH, AND M. SNIR, *The NYU Ultracomputer - designing an MIMD parallel machine*, IEEE Trans. on Comp. TC-32, (1983), pp. 175–189.
[26] W. HACKBUSCH, *Multigrid Methods and Applications*, Springer–Verlag, Berlin, 1985.
[27] C. P. KRUSKAL, *Algorithms for replace-add based paracomputers*, in Proceedings of International Conference on Parallel Processing, 1982, pp. 219–223.
[28] L. KUCERA, *Parallel computation and conflicts in memory access*, Info. Proc. Lett., 14 (1982), pp. 93–96.
[29] J. S. LEICHTER, *Shared tuple memories, shared memories, buses, and LAN's - Linda implementations across the spectrum of connectivity*, PhD thesis, Yale University, New Haven, CT, 1989. Computer Science Department Technical Report 714.
[30] C. E. LEISEERSON AND B. M. MAGGS, *Communication-efficient parallel algorithms for distributed random-access machines*, Algorithmica, 3 (1988), pp. 53–77.
[31] G. LEV, N. PIPPENGER, AND L. G. VALIANT, *A fast parallel algorithm for routing permutation networks*, IEEE Trans. on Comp. TC-30, (1981), pp. 93–100.
[32] L. RUDOLPH, *Software structures for ultraparallel computing*, PhD thesis, New York University, New York, 1982.
[33] M. SNIR, *On parallel searching*, SIAM J. Comp., 14 (1985), pp. 688–708.
[34] P. SONNEVELD, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 10 (1989), pp. 36–52.