Recursive Individually Distributed Object

Z. George Mou

Applied Physics Laboratory
Johns Hopkins University, USA
george.mou@jhuapl.edu

Abstract. Distributed Objects (DO) as defined by OMG's CORBA architecture provide a model for object-oriented parallel distributed computing. The parallelism in this model however is limited in that the distribution refers to the mappings of different objects to different hosts, and not to the distribution of any individual object. We propose in this paper an alternative model called Individually Distributed Object (IDO) which allows a single large object to be distributed over a network, thus providing a high level interface for the exploitation of parallelism inside the computation of each object which was left out of the distributed objects model. Moreover, we propose a set of functionally orthogonal operations for the objects which allow the objects to be recursively divided, combined, and communicate over recursively divided address space. Programming by divide-and-conquer is therefore effectively supported under this framework. The Recursive Individually Distributed Object (RIDO) has been adopted as the primary parallel programming model in the Brokered Objects for Ragged-network Gigaflops (BORG) project at the Applied Physics Laboratory of Johns Hopkins University, and applied to large-scale real-world problems.

1 Introduction

A methodology to decompose a computational task into a sequence of concurrent subtasks is the essence of any parallel programming model. Function-level parallelism takes advantage of the partial ordering of the dependencies whereas data-level parallelism [13] exploits the parallelism inside a single operation over a large data set. The OMG's *distributed objects* should be viewed as the result of function-level decomposition. Each object is responsible for a set of operations over a given collection of data, and the messages between objects realize the dependencies between functions. It should be noted that in the distributed-objects (DO) model of the OMG's Common Object Request Broker Architecture

(CORBA) [12, 26] or Microsoft's DCOM [25], each object is assigned to one and only one host. Therefore, a collection of distributed objects is distributed, but any object in the collection is not.

The *individually distributed object* (IDO) model we propose is not at all the same as one of the distributed objects in the OMG's model. A single data field of an IDO can be distributed over multiple host processors, and each operation over the distributed data field(s) is implemented by the collective operations of the hosts. The IDO implementation preserves the object semantics to give the programmers the illusion of working with a single object. The distributed nature of the object therefore is hidden from the programmers. IDO's are particularly well suited for computations with operations over large data sets as often found in scientific computing.

A data structure is *recursive* if it can be recursively divided and combined. The *list* in the language Lisp and other functional programming languages is an example of a recursive data structure. Recursive data structures can undoubtedly contribute to an elegant high-level programming style, particularly for programming by divide-and-conquer. The importance of divide-and-conquer as a programming paradigm has been discussed in many textbooks on algorithms [1, 6] or numerical analysis [9]. The significance and potential of divide-and-conquer in parallel processing have also been emphasized over the years by Smith [27], Preparata [24], Mou [23, 21, 20, 22], Axford [2], Dongarra [8], Gorlatch and Christian [11], and many others. Unfortunately, balanced recursive data structures (i.e., those with balanced division trees) are unsupported in most, if not all, existing major programming languages. Research efforts to address the problem have been made over the years, including David Wise's block arrays [30], Mou's recursive array [21], and Misra's Powerlist [19]. A *recursive individually distributed object* (RIDO) proposed here is an IDO that supports divide, combine, and communication operations.

In response to the demand for high-level and high-performance computing, the Applied Physics Laboratory of Johns Hopkins University has launched a research project under the name of the Brokered Objects for Ragged-network Gigaflops (BORG). The RIDO model is adopted as the foundation of the BORG programming system. The BORG system architecture (Figure 5) is described in Section 5.

In Section 2, we introduce the notion of distributed function and individually distributed object, and show why and how functions and objects can be dis-

tributed. In Section 3, we focus on recursive array objects, and their functionally orthogonal operations. Higher order programming frameworks that work with RIDO's are discussed in Section 4 with some examples. A systematic method for the implementation of the RIDO's and the BORG architecture are described in Section 5. An application to the problem of radar scattering over ocean surface is presented as a test case in Section 6.

2 Distributed Function and Individually Distributed Object

Let $f : X \to Y$ be a function from X to Y, and H a partition of the domain X and Y, such that $Hx = [x_1, \ldots, x_m]$ and $Hy = [y_1, \ldots, y_m]$ for any $x \in X$ and $y \in Y$. We say $F : X^m \to Y^m$ is a *distribution* of f if and only if $f = H^{-1}FH$ (Figure 2).

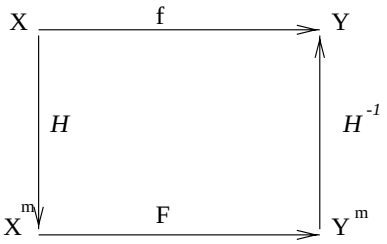

Fig. 1. The relationship between a function f and its distribution F with respect to a partition of H.

For example, let f be the function that maps an array of integers to their squares, then its distribution is simply the construct $(f, \ldots f)$ which applies to a tuple of arrays of the same length $(x_1, \ldots, x_m)$, and returns $(fx_1, \ldots, fx_m)$. As another example, let us consider the inner product ip of two vectors. Its distributed form is $IP = \sum_{i=1}^{m}(ip, ip, \ldots, ip)$. The second example shows that inter-block[1] dependencies or communications is generally a component of the distribution of a function except for trivial cases.

Formally, a *communication* $C : S \to S \times S$ defines a directed graph of the form (S, A), where an arc $a = (x, y) \in A$ if and only if $C(x) = (x, y)$ for $x, y \in S$.

[1] Blocks here refer to the blocks generated by a partition.

Note that an arc (x, y) implies that the element x receives a value (message) from element y. Given a partition H over S, a communication C is said to be *intra-* (*inter-*)*block* if none (all) of the arcs in C crosses the boundaries between the blocks. A communication which is neither intra- nor inter block is *mixed*. Obviously, a mixed communication can always be decomposed into inter- and intra-block communications.

A function $L : X \to Y$ is *local* with respect to a partition H over X, if there there exists a tuple of functions $(L_1, \ldots, L_k)$ such that

$$\begin{aligned} L\ x &= (L_1, \ldots, L_k)\ H\ x \\ &= (L_1, \ldots, L_k)\ (x_1, \ldots, x_k) \\ &= (L_1\ x_1, \ldots, L_k\ x_k) \end{aligned}$$

A function is *strongly local* if it is local with respect to the maximum partition by which each element of X is a block. Note that a strong local function is necessarily local with respect to any partition.

It is easy to see that any function can be decomposed into a sequence of communications and local functions. Since any mixed communications can in turn be decomposed into inter and intra-block communications, we can re-group the sequence so that each group in the sequence is either a inter-block communication or a local operation. For example, if function f is decomposed into the sequence of $f = \cdots L_i C_i L_{(i-1)} C_{(i-1)} \cdots$ where C_j is decomposed into C_{j1} followed by C_{j2} which are respectively its inter- and intra-communication components. The above sequence can be rewritten as

$$\begin{aligned} f &= \cdots L_i C_{i2} C_{i1} L_{(i-1)} C_{(i-1)2} C_{(i-1)1} \cdots \\ &= \cdots (L_i C_{i2}) C_{i1} (L_{(i-1)} C_{(i-1)2}) C_{(i-1)1} \cdots \end{aligned}$$

We conclude that any function f over a given set has a distribution F that is a sequence of inter-block communications and local operations.

An object is no more than a collection of functions over a collection of data sets. Given that its data sets are all partitioned, we say an object is individually distributed if all its functions are distributed with respect to the partitions of its data set. The above discussion therefore shows that an object can always be individually distributed. Observe that the components of the local operations can be done in parallel, and data movement implied by the arcs in the inter-block communication can be performed in parallel by definition.

3 Array Recursive Individually Distributed Object

There are three types of *functionally orthogonal operations* that can be performed over data: manipulations of the structure, data exchanges or communications between the nodes, and mappings of the nodes to new values. There is no reason to think any type of the operations is more important than others and none can replace another since they are mutually *orthogonal* in functionality. However, there is apparently an operation orientation in the design of most programming languages. As a result, communication functions are treated as second class citizens buried in other operations. Structural operations (e.g., array division in merge-sort) are often absent, and can only be "simulated". Functional programming languages provide structural operations (e.g., car, cdr, and cons in Lisp) for its main data structure of lists. However, the division is fixed and unbalanced, and the communication operation is still implicit.

According to many textbooks on algorithms [1, 6, 9], divide-and-conquer is the single most important design paradigm for both sequential and parallel algorithms. The ubiquity of divide-and-conquer indeed cannot be overstated. Moreover, many algorithms and paradigms not always considered as divide-and-conquer can be naturally modeled by divide-and-conquer, including greedy algorithms, dynamic programming, recursive doubling [28], and cyclic reduction [14, 16]. All of the above share a need for balanced recursive data structures, which are supported in few, if any, programming languages.

We will focus our discussion on the *array recursive individually distributed object* (ARIDO), which is an individually distributed array that can recursively divide and combine itself, communicate over a recursively divided address space, and perform local operations in parallel to all the entries.

In the following we list the operations of an ARIDO array, which are supported in the BORG Recursive Individually Distributed Object Library (BRIDOL). We use A[s:k:e] to denote the sub-array of A that contains entries of index s, s+k, s+2k, and so on, but excluding any entries with index greater than e.

dlr: divides an array of even length into its left and right sub-arrays:

$$dlr\ A = (A_l, A_r),\ \text{where } A_l = A[0:1:n/2-1],\ A_r = A[n/2:1:n-1]$$

deo: divides an array of even length into its even and odd sub-arrays:

$$deo\ A = (A_e, A_o),\ \text{where } A_e = A[0:2:n-1],\ A_o = A[1:2:n-1]$$

dht: head-tail division of an array:

$$dht\ A = (A_h, A_t),\ \text{where}\ A_h = A[0:1:1],\ A_t = A[1,\ 1,\ n-1]$$

clr: Inverse of dlr[2].
ceo: Inverse of deo.
cht: Inverse of dht.

Higher dimensional arrays can divide along one or more of its dimensions. For example, [dlr, dlr] divides a matrix into four sub-blocks as expected.

A communication can be either *intra-array* or *inter-array*. In the case of inter-array communication, we assume the two arrays have the same shape.

corr: *Correspondent* inter-array communication which sends the value of every entry to the entry of the same index in the other array, i.e.,

$$corr\ (A, B) = (A', B'),\ \text{where}\ A'(i) = (A(i),\ B(i)),\ B'(i) = (B(i),\ A(i))$$

mirr: *Mirror-image* inter-array communication which sends the value of every entry to the entry with "mirror-image" index in the other array, i.e.,

$$mirr\ (A, B) = (A', B'),\ \text{where}\ A'(i) = (A(i),\ B(n-1-i)),\quad B'(i) = (B(i),\ A(n-1-i))$$

br: *Broadcast* inter-array communication which sends the value of last entry to all entries in the other array, i.e.,

$$br\ (A, B) = (A', B'),\ \text{where}\ A'(i) = (A(i),\ B(n-1)),\ B'(i) = (B(i),\ A(n-1))$$

Similar to br, the broadcast inter-array communication *brr* sends values only from left to right, *brl* sends values only from right to left, and $(br\ k)$ sends the k-th last value instead of the very last value.

pred: *Predecessor* intra-array communication, i.e.,

$$pred\ (A) = A'\ \ \text{where}\ A'(i) = (A(i),\ A(i-1)),\ (i \neq 0)$$

succ: *Successor* intra-array communication, which is symmetric to predecessor.

[2] The inverse of the division in fact is not unique [23]. We here refer to the minimum inverse, i.e., the one with the smallest domain. The same applies to the inverses of other divide functions.

By overloading, an intra-array communication can turn into an inter-array communication when supplied with two, instead of one, array as arguments, and vice versa. For example, mirr(A) can be used to reverse a one-dimensional array[3].

For an array of type T, an unary (binary) operator $\oplus$ for T can be converted to a *local operator* for array of type T before (after) the communication by prefixing the operator with a "!". For example

$$!+.corr\ ([1,2],[3,4]) = ([4,6],[4,6])$$
$$!second.mirr\ ([1,2,3,4]) = !second\ ([(1,4),(2,3),(3,2),(4,1)]) = [4,3,2,1]$$

where "." denotes *function composition*, and second(a,b) = b.

4 The Divide-and-Conquer Frameworks

In object-oriented terminology, *frameworks* are equal to components plus patterns [15]. A framework is therefore an abstract pattern that can be instantiated by binding to given components. This is not an idea inherently tied to object technology, and is referred to as *high order constructs* [3, 23] in functional programming. Divide-and-conquer is a powerful framework which can be instantiated to a wide range of algorithms.

The BORG library currently provides the following DC constructs:

Pre-DC: Premorphism [21, 20, 22] divide-and-conquer, defined as

```
f = PreDC (d, c, g, p, f_b)
      where
        f A = if p A then return f_b A
              else c.(map f).g.d A
```

where g is the *pre-function* applied before the recursion and is a composition of communication followed by a local operation, d and c respectively the divide and combine operators, p the base predicate, f_b a base function for f, and (map f)$(x_1, x_2) = (fx_1, fx_2)$.

Post-DC: Postmorphism divide-and-conquer, defined as

[3] More precisely, the reversed array after the communication only exists as the second element of the pairs generated by the communication.

f = PostDC (d, c, h, p, f_b)
where
f A = if P A then return f_b A,
else c.h. (map f). d A

where h is the *post-function* applied after the recursion and is a composition of communication followed by a local operation.

RD: Recursive doubling which is divide-and-conquer with even-odd division, and intra-array communication, defined as

f = RD ($\oplus$)
where
f A = if (SizeOne? A) return A
else ceo.(map f).deo.!$\oplus$.pred.A

For example, the function scan $\oplus$ (associative binary operator) defined by

$$scan \oplus A = A', \text{ where } A(i) = \oplus_{j=0}^{i} A(j) \text{ for } i = 0 \text{ to } |A| - 1$$

can be programmed as a postmorphism or a recurisve doubling:

scan $\oplus$ = PostDC (dlr, clr, $\oplus$.brr, SizeOne?, id) = RD ($\oplus$)

where *id* is the identity function. The PostDC algorithm[4] is illustrated in Figure 2.

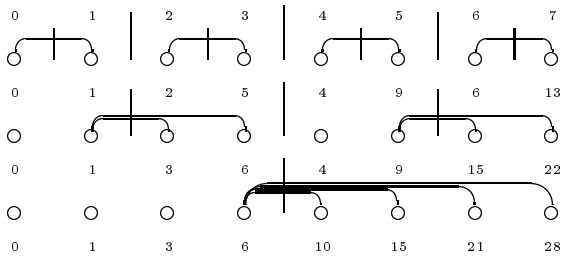

Fig. 2. Scan with broadcast communication

[4] This algorithm is simple but not most efficient, and can be improved by replacing its *broadcast* communication by *correspondent* communication [21]. In terms of total number of operations, a cyclic reduction algorithm is more efficient [17].

A divide-and-conquer with other divide-and-conquer as its components is a *higher order divide-and-conquer*. Figure 3 gives the *monotonic sort* as an example of second order divide-and-conquer, which resembles but differs from Batcher's bitonic sort [4].

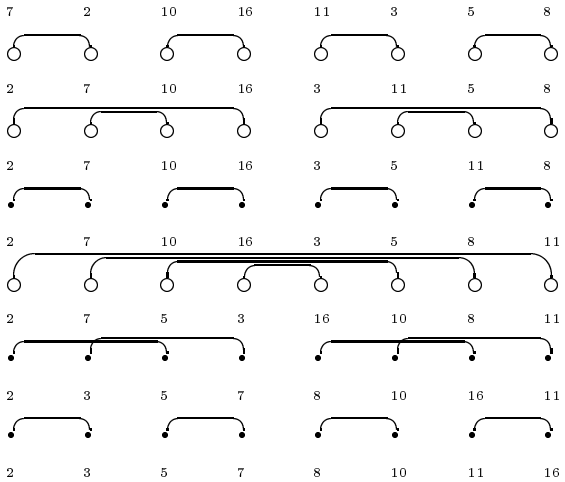

Fig. 3. Monotonic sort is a second order divide-and-conquer algorithm where the top level is PostDC with mirror-image communication (circles), and the nested level is PreDC with correspondent communication (solid dots).

5 Implementation

An ARIDO array is distributed over multiple processors. By default, the most significant $\log p$ bits of an index are used to map the entry with the index to a host processor.

Observe that the left-right (even-odd) division recursively partitions an array from left to right (right to left) by the remaining most (least) significant bit. The implementation of balanced division (left-right and even-odd) uses a "pointer" of type integer, whose only non-zero bit points to the current dividing bit. Each divide operation thus reduces to a shift of the pointer by one-bit. Left-right division shifts to right, even odd to left (Figure 4). Note that an implementation with this approach involves no copying or sending of sub-arrays as often suggested in literature (e.g., [18]), and takes O(1) time. A combine operation is simply to shift towards the opposite direction from that of the corresponding division[5].

[5] If you noticed that the odd-even division is the same as left-right combine, you are right. In fact, an isomorphism can be formalized which allows DC algorithms with

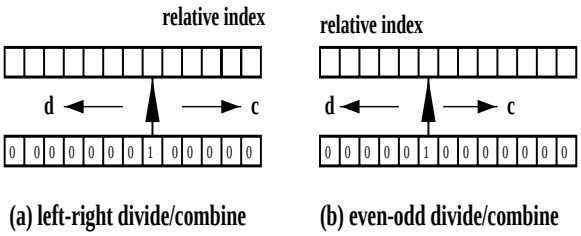

Fig. 4. Division and combine operations implemented with a pointer. The bottom integer in binary contains exactly one non-zero bit, which points to the current dividing bit. For the left-right (even-odd) division, the relative index is given by the bits to the right (left) of the dividing bits. This example shows an array of size $8k$ divided at the eighth level.

To support high-level programming, the indexing of the recursively divided arrays is relative. Therefore, every sub-array starts with the relative index of zero at all recursion levels. This scheme allows the system communication operators to be specified with very simple functions. For example, the *correspondent* communication is specified by the identity function[6]. The implementation however needs to translate the relative indices to global indices. It turned out that the relative index is given precisely by the remaining bits of the index at each level of the division (Figure 4). To translate a given relative index to its global index, all we need to do is to concatenate the remaining bits with the masked bits. If the communication is inter-array, the dividing bit needs to be complemented. It follows that the translation time between relative and global indices is O(1), the actual time to carry out the communication depends on the communication pattern and the underlying communication network platform.

A clear distinction should be drawn between the communications over recursive distributed arrays and the inter-processor communications over the platforms. The former is defined over the logic domain, and the latter the physical domain. The programmers should only be concerned with the former, and only the system implementors are concerned with the latter. Also, observe that

---

the two types of divisions to be transformed into each other under certain conditions. This topic will be left for future discussion elsewhere.

[6] The seemingly complicated butterfly communication used in FFT and many other applications is therefore no more than the recursive applications of communications generated by the identity function.

communications over recursive arrays may and may not involve inter-processor communications. Let n be the size of a recursive array, p the number of processors, then inter-processor communications only occur for $\log(p)$ times in a first-order DC algorithm. In the remaining $\log(n/p)$ times, communications over the recursive arrays are mapped to memory accesses. Whether a given recursive array communication maps to inter-processor communication can be easily determined by a comparison of the dividing pointer and the bits used to index the processors. If the pointer points to a bit used to index the processors, the communication will be translated into an inter-processor communication.

Unlike Beowulf systems [5], BORG's implementation of inter-processor communication is based on the Common Object Request Broker Architecture (CORBA). The benefits of this approach in comparison with MPI or PVM include:

- An open environment that is neutral to different programming languages, operating systems, and hardware platforms.
- Host location transparency.
- Portability.
- Capability of dynamically adding computational resources available to the system through the Internet.
- Higher level system programming.

The BORG hardware platform currently consists of 17 Pentium boxes, of which 12 has single 300MHz Pentium II processor, and the rest dual 233Mhz Pentium II processors. Each machine is equipped with a Fast Ethernet card with 100 Mb/s bandwidth. All the nodes are connected by a high bandwidth Ethernet switch with an aggregate bandwidth of 3 Gb/s. The inter-processor communications are implemented with CORBA ORB as illustrated in Figure 5. BORG allows a mixture of Unix and NT workstations to work together. We have made a distinction between the individually distributed object proposed here and the distributed objects defined in literature. Our CORBA implementation indeed means that an individually distributed object is implemented, not surprisingly, with a set of distributed objects. Our preliminary experience also showed that a communication transaction using CORBA takes time in the order of millisecond, which we consider acceptable relative to the performance of MPI or PVM.

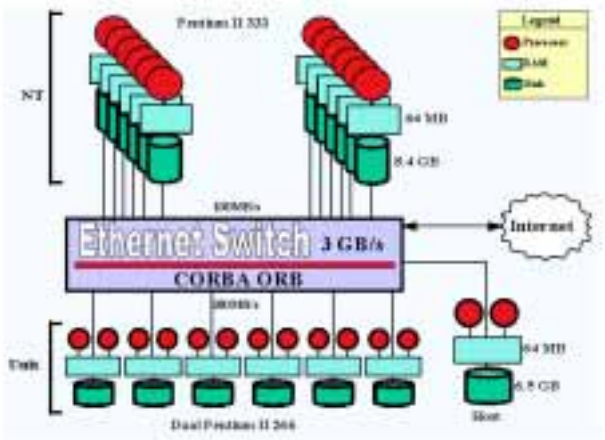

Fig. 5. The Architecture of the Brokered Objects for Ragged-network Gigaflops (BORG).

6 Applications

A number of real world applications are being developed with the recursive individually distributed object model on BORG system. In this section, we will focus on one – the ocean surface radar scattering problem [7]. A multi-grid method for the problem iteratively solves the following equation on each grid level:

$$ZX^{n+1} = ZX^n + \alpha(C - ZX^n)$$

where n is the iteration number, α a free parameter, Z a preconditioner which can be approximated by a banded matrix. To solve for X^{n-1}, the inverse of Z is computed, and a matrix-vector multiplication (ZX^n) needs to be performed.

We use the N-merge algorithm illustrated in Figure 6 to solve the banded linear system (with a bandwidth of hundreds). The division used is left-right along the rows, and the communication used is broadcast. This algorithm is a generalization of the algorithm presented in [10], which is similar but different from Wang's algorithm in [29].

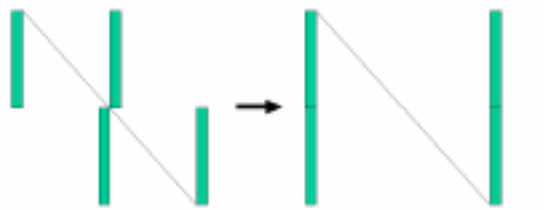

Fig. 6. N-Merge algorithm for banded linear systems merges two N-shaped non-zero bands of height m to one N-shape of height 2m, and thus eliminates all the non-diagonal elements in $\log(n)$ parallel steps for a system of size n.

The matrix-vector multiplication can be solved by DC in two ways. The first divides the matrix along the column dimension, the other the row dimension (Figure 7). Although the total operation counts of the two algorithms are the same, the first is more memory efficient than the second. The performance of the two algorithms can be quite different depending on the layout method of two dimensional arrays of the underlying programming language.

Let the sample space size be n, the total number of operations for the above two key procedures is in the order of $O(n^2)$. Since this is to be performed for many iterations with complex double precision floating point numbers, the total number of operation for a moderate value of $n = 10^4$ is in the order of Teraflops (10^{12} floating point operations per second) for just one (of the hundreds of) realizations in the Monte Carlo simulation.

The problem was solved on Sun Ultra workstations. Since the memory is not large enough to hold the entire preconditioner matrix, the preconditioner was recalculated at each realization. The total time taken is in the order of 36 hours for one realization. The BORG version of the algorithm is currently being implemented, and the initial performance data suggests that the same computation can be completed in a couple of hours with 16 Pentium II processors.

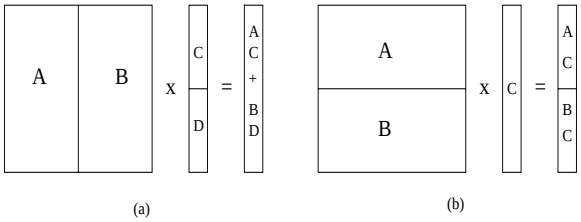

Fig. 7. Two DC algorithms for matrix vector multiplication. In (a), the recursive application results in column vector and scalar operations, and the final result is obtained by row reduction over all rows. On the other hand, (b) leads to many inner products between the matrix rows and the input vector.

7 Conclusion

The recursive individually distributed object model can effectively exploit the parallelism in the computation internal to an object, and support high level programming by divide-and-conquer. The implementation of the model under the BORG project at the Applied Physics Laboratory is built on top of the Java language and the CORBA architecture. The resultant system is neutral to programming languages, operating systems, and hardware platforms. A high performance scientific computing library with individually distributed object model is being developed, and the preliminary results have shown that the model can be effectively applied to large-scale real problems with acceptable parallel performance.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addision-Wesley, 1974.
2. Tom Axford. *The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design*, pages 26–65. John Wiley and Sons, Inc., 1992. edited by Kronsjo, Lydia and Shumsheruddin, Dean.
3. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
4. K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.
5. CESDIS. Beowulf project at cesdis. *http://www.beowulf.org*.

6. T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
7. D.J.Donohue, H-C. Ku, and D.R.Thomson. Application of iterative moment-method solution to ocean surface radar scattering. *IEEE Transactions on Antennas and Propagation*, 46(1):121–132, January 1998.
8. Jack Dongarra. Linear library liberaries for high-performance computer: A personal perspective. *IEEE Parallel & Distributed Technology*, (Premiere Issue):17–24, February 1993.
9. W. H. Press *et al. Numerical Recipes – The Art of Scientific Computing*. Cambridge University Press, 1986.
10. Y. Wang *et al.* The 3dp: A processor architecture for three dimensional applications. *Computer, IEEE*, 25(1):25–38, January 1992.
11. Sergei Gorlatch and Christian Lengauer. Parallelization of divide-and-conquer in the bird-meertens formalizm. Technical Report MIP-9315, Universitat Passau, Dec. 1993.
12. Object Management Group, editor. *CORBA Specifications*. www.omg.org.
13. W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.
14. R. W. Hockney. The potential calculation and some applications. *Methods in Computational Physics*, 9:135 – 211, 1970.
15. Ralph E. Johnson. Frameworks = components + patterns. *Communications of the ACM*, 40(10):39–42, October 1997.
16. S. Lennart Johnsson. Cyclic reduction on a binary tree. *Computer Physics Communication 37 (1985)*, pages 195–203, 1985.
17. R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
18. V. Lo and *et. al.* Mapping divide-and-conquer algorithms to parallel architectures. Technical Report CIS-TR-89-19, Dept. of Computer and Info. Science, University of Oregon, January 1990.
19. Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6), November 1994.
20. Z. G. Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461. IEEE Computer Society Press, October 1990.
21. Z. G. Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization*. PhD thesis, Yale University, May 1990.
22. Z. G. Mou. The elements, structure, and taxonomy of divide-and-conquer. In *Theory and Proactice of Higher-Order Parallel Programming, Dagstul Seminar Report 169*, pages 13–14, February 1997.

23. Z. G. Mou and Paul Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *The Journal of Supercomputing*, 2(3):257–278, November 1988.
24. F. P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 8(5):300–309, May 1981.
25. Roger Sessions. *Com and Dcom: Microsofts's Vision of Distributed Objects*. John Willey and Sons, 1987.
26. Jon Siegel. Omg overview: Corba and the oma in enterprise computing. *Communications of the ACM*, 41(10):37–43, October 1998.
27. D. R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.
28. Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4), 12 1975.
29. H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.
30. David S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, pages 134–153. Springer, 1987.