LOAD BALANCING BY DOMAIN DECOMPOSITION: THE BOUNDED NEIGHBORS APPROACH

F.BAIARDI A.BONOTTI L.FERRUCCI L.RICCI P.MORI

Dipartimento di Informatica, Universitá di Pisa
via F.Buonarroti, 56125-Pisa (Italy)
baiardi,mori,ricci@di.unipi.it

Abstract: This paper presents a new domain decomposition approach whose main goal is the computation of a load balancing partition while reducing the overhead to compute such a partition. In the proposed approach, the number of neighbours of each sub-domain returned by the decomposition can be bounded by an user supplied value. This reduces the communication overhead of the application. We describe an algorithm implementing our decomposition strategy and apply our approach to WaTOR, a classical dynamical simulation problem. We report also some preliminaries result to prove the effectiveness of our approach.

keywords: Partitioning, Mapping, Load Balancing, Cluster Computing

1 INTRODUCTION

Domain Decomposition is a technique exploited both to balance the load and to reduce communications in parallel applications [Culler and Singh, 1998]. It is applied when the data set of the application can be regarded as a *physical domain* where the computation performed on each element D requires the knowledge of a small subset of data close to D only. Several parallel applications belong to this class, a classical example being that of cellular automata [Sloot and Talia, 1999].

This technique partitions the domain into a set of sub-domains with the same computational load and assigns each sub-domain to a distinct process. Each process updates the elements belonging to its sub-domain S and communicates with the other processes only to update the elements located on the boundary of S.

The domain decomposition problem becomes challenging when *dynamical applications* are considered because in these applications the decomposition of the domain has to be updated during the computation. The update is required to take into account that the number of the elements of the domain and/or their positions are dynamic, i.e. change during the computation.

Several approaches have been proposed in the last years. Each strategy takes into account the trade-off between the overhead introduced by the dynamic partitioning of the domain and the benefits obtained by balancing the load. *Orthogonal Recursive Bisection* [Salmon, 1990] is a domain decomposition strategy exploited in many parallel application. This technique produces an optimal balance of the work, at the expense of a large computational complexity. On the other side, simpler solutions often result in decompositions of the domain characterized by an unsatisfactory load balance.

This work describes the *BoundedNeighbours* approach, a domain decomposition technique whose main goal is to reduce the overhead introduced by partitioning the domain while preserving an acceptable balance of the work. Another interesting feature of our approach is that it allows the user to *bound* the number of neighbours of each sub-domain. Since each sub-domain is assigned to a different process, this bounds also the number of communications of each process and, hence, the overall communication overhead.

Section 2 reviews existing proposals. The main features of our strategy are described in section 3, while section 4 presents the *Bounded Neighbours* implementation. Finally, section 5 shows the application of *Bounded Neighbours* to *WaTOR*, a classical irregular distributed simulation problem. Some significant performance results are described as well.

2 RELATED WORKS

Several domain partitioning techniques have been proposed in the last years. While most of them consider 2-dimensional domains, almost all of them can be easily extended to cover a larger number of dimensions.

Applications defined on irregular and/or dynamical domains generally require a dynamic partitioning of

 ISBN 3-936150-25-7

the domain. Nevertheless, some dynamical applications exploit *scattered decomposition* [Saltz, 1990], a static decomposition technique. Scattered decomposition partitions the domain into a set of rectangular zones, the *templates*. Each template is further divided into a set of rectangular regions, the *granules*. Corresponding granules belonging to different templates are assigned to the same process. The resulting load is balanced only when the domain is characterized by a uniform distribution of the load to the granules. The main advantage of this technique is that the decomposition defines a set of regular communication patterns. On the other way, a satisfactory load balance may be obtained only when the size of the granules is rather small.

Since the communication overhead due to a granule increases with the ratio between the perimeter and the area of the granule, a larger number of granules improves load balancing at the expense of increasing the ratio between the communication overhead and the computational one.

Dynamic decomposition techniques update the domain partition when the number and/or the position of the elements are modified. A well known approach is that of [Salmon, 1990; Simon, 1994], the *Orthogonal Recursive Bisection (ORB)*. *ORB* initially splits the domain into two rectangular sub spaces with the same load. The set of processes is partitioned into two subsets as well, and each subspace of the domain is assigned to a subset of processes. The procedure is recursively applied until a single subspace is assigned to each process. In general, *ORB* achieves a good balance, but its computational cost is high because of the complexity of determining the cuts of the domain. Furthermore, each process records the partitions through a binary tree, built during the load balancing phase. This tree is visited during the computation to detect the neighbours of each process. This visit introduces a further overhead in the computation. Note that, in the worst case, an high number of neighbours may result for each process. *ORB* was originally proposed for an hypercube architecture and its implementation is greatly simplified if the number of processes of the application equals a power of 2.

A simpler approach considers the domain as a grid that is partitioned into blocks of contiguous rows. The boundaries of each block are dynamically re computed to balance the load. The main advantage of this technique is its simplicity. Furthermore, each process has two statically defined neighbours. The main disadvantage is that the balancing is not satisfactory, because of the coarse grain of the partitioning.

The *cost zone* [Singh et al, 1995] or *space filling curves* [Singh et al, 1995; Baden and Pilkington, 1995; Moon et al, 2001] are exploited for applications defining a hierarchical subdivision of the domain .

3 BOUNDING NEIGHBORS

Bounded Neighbours is a domain decomposition strategy whose main goal is *to reduce the overhead* of dynamical domain partitioning while producing an *acceptable load balance*. Furthermore, *Bounded Neighbours* allows the user to *bound* the number *NS* of neighbours of each sub-domain. This implies a reduction of the overhead due to communications. As a matter of fact, the process associated with a sub-domain requires elements belonging to its neighbours when it updates the elements on the border of its partition only. In our approach, the number of processes exchanging data with each process of the application is bounded by NS. Since each communication with a distinct partner implies a new *start-up phase*, this reduces the communication overhead. It is worth noticing that the computational cost of the start-up phase of each communication is high, in particular when considering applications developed on *workstation clusters*. Furthermore, several optimizations can be applied to reduce the communication cost between a single pair of partners.
In *Bounded Neighbours* the value of the parameter NS can be defined by the user. Each decomposition produced by *Boundedn Neighbours* satisfies the *bounded neighbours condition*, i.e. the number of neighbours of each sub-domain does not exceed NS.

Bounded Neighbours generalizes the simple domain decomposition that assigns blocks of consecutive rows of the grid to each process. In our approach, each row can be further subdivided into segments and each segment can be assigned to a different sub-domain. The leftmost part of Figure 1 shows a decomposition produced by *Bounded Neighbours*. This strategy returns an optimal load balancing, but, in general, the bounded neighbours condition is not satisfied. This constraint is considered in a second phase, when the cuts produced by the first one are shifted to produce a legal decomposition. *Bounded Neighbours* defines a set of simple conditions which imply the bounded neighbours one. Consider, for instance, a $n \times m$ grid and suppose $NS = 2$, i.e. the number of neighbour of each domain is bounded by 2. In this case, the following conditions guarantees that the bounded neighbours condition is satisfied.

- each row of the grid includes at most one cut;
- each sub-domain includes at least m points of the grid

These conditions can be easily checked by considering the grid decomposition. For instance, in Figure 1, process P3 includes exactly m points of the grid.

[Bonotti, 2002] defines similar conditions for the more general case. It is worth noticing that the number of cuts that can be applied to each row increases with the value of NS. Furthermore, the balancing is improved by a larger number of cuts. Nevertheless, our experiments show that an acceptable compromise between communication overhead and load balancing may be achieved by low values of NS.

Fig. 1 compares our approach with blocks of rows decomposition (shown in the central part of the figure) and with *orthogonal recursive bisection* (shown in the right part). Our approach produces a better load balancing with respect to the first one because a row can be cut and the resulting subset of the row can be assigned to different processes. In the block of row decomposition, the number of neighbours of each process is equal to to 2. This can be obtained also in our approach, by setting NS to 2.

In general, the *ORB strategy*, achieves a better load balancing. On the other hand, in the worst case, it may result in a large number of neighbours of a given subdomain.

Furthermore, in our approach, the computation of cuts is straightforward. Instead, ORB [Salmon, 1990] requires a parallel median finder algorithm which, in turn, results in a large amount of communications to implement domain decomposition.

4 THE IMPLEMENTATION

This section describes a MPI algorithm to implement the Bounded Neighbours strategy. If n is the number of processes of the application, the algorithm partitions the domain into n sub-domains and assigns Dom_i to process P_i. Each process exploits a data structure, the *cut-array*, to store the initial and the final coordinates of each sub-domain, i.e. the *cuts* of the grid.

This structure is initialized when the elements are distributed to the processes and is updated after each load balancing step. Note that this structure does not store the exact location of the elements in the other sub-domains produced by a partition, it only describes the partition of the domain among the application processes.

The Bounded Neighbours algorithm consists of four phases:

- *Trade off Evaluation*
- *Cuts Computation*
- *Cuts Checking*
- *Data Exchange*

In the first phase, *Trade off* evaluation, processes exchange their current load. This information is exploited to evaluate the trade-off between the overhead introduced by the execution of the algorithm and the unbalance of the computation. The following phases are executed only if the trade-off is significant. In the *Cuts Computation* phase the processes compute the new partitions, i.e. the new cuts to balance the load. This phase can produce an illegal partition of the domain, i.e. a partition where the number of neighbours is larger that NS. In the following phase, *Cuts Checking*, a partition may be updated to produce a legal solution. Finally, in the *Data Exchange* phase, the processes exchange data to build the new partition of the domain. In the following, we will describe each phase in more detail.

4.1 Trade-Off Evaluation

We assume that each process, after each load balancing step, stores in a local data structure the cuts defining the partition of the domain and the current load of any other process. The load may be changed because, during a computational step, the number and/or the position of the elements in the domain may be modified. In this phase, processes exchange their *current load*, i.e. the number of elements currently belonging to a sub-domain. This communication is implemented by a $MPI\ Allgather$ primitive.

After the collective communication, each process computes the optimal load and evaluates the trade-off between the overhead due to the execution of the load balancing algorithm and the benefits of a balancing step. The trade-off is defined by the following criteria:

- *Number of elements*

  The current number of elements can be easily computed by summing the current load of any process. If this value is smaller than a *threshold percentage P* of the total number of positions of the grid, the load balancing step is not executed, because, the overhead of the load balancing is not balanced by the resulting speed up of the computation.

- *Maximum Unbalance*

  The overall execution time of a computation step is determined by the execution time of the slowest process, i.e. the process P_i owning the subdomain Di including $MaxD?$, the largest number of elements. The load balancing algorithm is executed only if the difference between the $MaxD$ and the optimal number of elements of each process is larger than *threshold value V*.

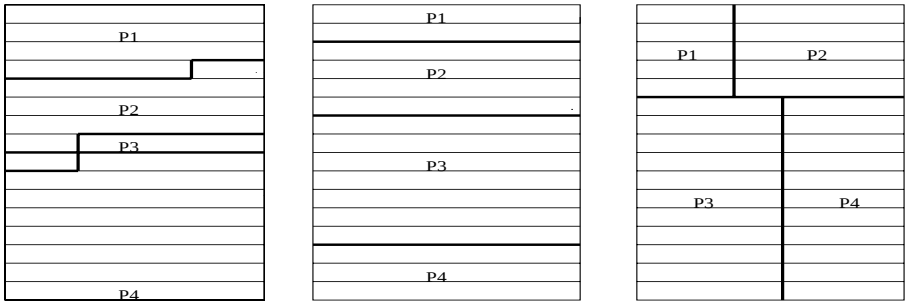

Figure 1. Load Balancing Strategies

The user can modify P and V to tune its application. An example is discussed in Section 5.

4.2 Cuts Computation

The *Cuts Computation* phase computes the new partition of the domain that assigns the optimal number of elements to each process. If the resulting partition does not satisfy the Bounded Neighbours condition, it will be modified in the next phase, *Cut Checking* which always generates a legal partitioning.

The *Cut Computation* phase consists of two steps. Let us denote by *old partition*, the partition computed in the previous load balancing step and by *new partition* that computed in this phase. In the first step, each process computes the *intersections* between the cuts defining the new partition and the sub-domains defined by the old partition. This computation exploits both the information gathered in the *trade-off evaluation* phase and the *cut-array* storing the cuts of the old partition.

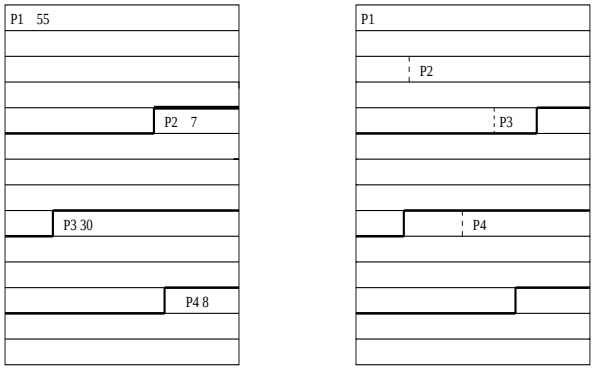

Figure 2. Cuts Computation

Consider, for instance, Figure 2. The leftmost part of the figure shows the old partition of the domain, which is recorded in the *cut-array*. For each sub domain we show a reference to the process owning the sub-domain and the current load of that process. This information, gathered during the *trade off evaluation* step, is exploited to compute the optimal amount of elements to be assigned to each process, 25 in the considered example. In the right part of the figure, the *new cuts* are shown through dashed lines. It is worth noticing that the exact location of a new cut can be determined only by the process P_i owning the domain including the cut. Since any other process P_j, $j \neq i$, does not know the exact location of the elements in Dom_i, P_j can only determine the *number of cuts* of the new partition intersecting Dom_i. During this step, each process P_i builds a list, *ListCuts* and an array, *Receivecuts*. Listcuts stores the coordinates of all the cuts intersecting its domain. *Receivecuts* is an n elements array, where the j-th position records the number of cuts intersecting Dom_j, for any $j \neq i$.

Consider again Figure 2. Process P_1 computes the coordinates of the cuts that intersects the first subdomain of the old partition and stores them in its *Listcuts*. All other processes store the value 2 in the first position of their *Receivecuts* array.

The code implementing this step is shown in Figure 3. We suppose that the variable *TotBalance* and *Optbalance* record, respectively, the total number of elements of the grid and the optimal amount of work to be assigned to each process. The i-th position of the array *ElPart* records the current load of process P_i.

When all the processes have completed this step, the exact location of the new cuts is notified by each process to any other one. This communication step is implemented by a loop executed in parallel by all the processes, according to an $SPMD$ programming style. To notify its cuts to the other processes, at each iteration process P_i executes a distinct MPI broadcast communication as a sender for each of its cuts. Each process P_j $j \neq i$ executes, in turn, *k MPI broadcast*, as a receiver, where k is the value stored in *Receivecuts[i]*.

```
Tot = TotBalance;
Diff= OptBalance;
i =0; CutPos=0;
Listcuts = ∅;
while Tot>0
    if (Elpart[i]<Diff)
        Diff = Diff - Elpart[i];
        Tot = Tot - Elpart[i];
        i = i+1;
    else
        %calcolo nuovo taglio
        ElPart[i] = ElPart[i] - Diff;
        if myrank()== i
            Listcuts=Listcuts ∪NewCutUpdate(CurPos, Diff)
        else
            ReceiveCuts[i]=ReceiveCuts[i]+1;
        endif
        Tot = Tot -Diff;
        Diff= OptBalance
    endif
endwhile
```

Figure 3. Cuts Computation

4.3 Cuts Checking

This phase checks the *Bounded Neighbours* condition for the partition produced by the previous step. If appropriate, it modifies the partition as well. In the following, we show the implementation in the case where $NS = 2$. [Bonotti, 2002] describes the more general case. Each process considers the sub-domains in a sequential order and it checks the following conditions where Dom is the sub-domain that is currently considered.

- Each sub-domain Dom should include at least m grid elements, m is the number of columns of the grid. In this way Dom_i completely separates Dom_{i+1} from Dom_{i-1}. This implies that each domain has 2 neighbours. If this condition is not satisfied, all the processes shifts forward the cuts of the domains following Dom, in order to associate at least m points of the grid to Dom. This operation can introduce a certain amount of unbalance. However, our experiments show that this case is not very frequent and may arise only when the dimension of the grid is small with respect to the number of available processes.

- If Dom includes m or more grid elements, each process checks if the the bounded neighbours condition can be verified by all the sub-domains considered after Dom. This is possible if the number of grid cells from the final cut of Dom to the end of the grid is larger than m times the number of sub-domains still to be considered. If this condition is violated, then this is the first domain violating the condition because the domain are considered one at a time. Then Dom and the following sub-domains can be updated to satisfy the condition.

4.4 Data Exchange

The phase implements, the actual exchange of the elements. Let OP_i be the old sub-domain associated with P_i and NP_j its new sub-domain. Each process:

- sends to P_j each element belonging to the intersection of OP_i with NP_j
- receives from P_j any element belonging to the intersection of NP_i with OP_j

In the example of Figure 2, process P_3 sends some elements to P_4 because some elements in its old sub-domain now belongs to $P4$. It also receives some elements from $P1$ and all the elements of the partition assigned to $P2$.

5 WaTOR: AN IRREGULAR DYNAMICAL SIMULATION

The load balancing strategy defined in Section 4 has been exploited to implement WaTOR, a classical distributed simulation problem. This problem, originally introduced in [Dewdney, 1984], defines an idealized world where fishes and sharks move randomly, feed, breed and die. Plankton is located randomly at the vertices of the grid. Fish eat plankton, while sharks eats fishes.

The exact rules describing the behaviour of fishes and sharks are given in [Dewdney, 1984].

[Fox et al, 1988] observes that, even if these rules are too simple to describe a realistic biological population, the parallel implementation of *WaTOR* presents a number of interesting features characterizing more advanced parallel applications as well. First of all, the application is characterized by a very in homogeneous and dynamic load distribution. As a matter of fact, the distribution of the elements in the domain is not uniform and the number of the elements of the domain changes dynamically due to their death and breeding. Second, a *conflict resolution strategy* has to be defined to solve the conflicts arising among animals. For instance, two animals can decide to move to the same point of the grid, or two sharks can decide to eat the same fish. Since no specific rule is specified in [Dewdney, 1984], any choice is acceptable. In the parallel implementation, conflict resolution is more complex because it can involve several processes. As a matter of fact, conflicts can arise among processes which concurrently try to update the same cell of the grid belonging to a border of the sub-domain. Several strategies

have been proposed in [Fox et al, 1988], ranging from the simplest one which simply eliminates the animal losing the conflict, to that defining a complex *rollback strategy*.

Let us briefly describe the main characteristics of our implementation. Our implementation models the problem domain as a two-dimensional toroidal grid. Fishes and sharks are located at the vertices of this grid and can move only to the four nearest-neighbours vertices. A straightforward implementation stores the grid in a rectangular two-dimensional array. The drawback of this solution is that processes can spend a significant amount of time to examine empty regions of the ocean. For this reason, an application process *P* exploits a *linked list* storing only animals present on the grid. The grid is represented by a one dimensional array of n elements, where n is the number of the rows of the grid. Each element of the array is null if the corresponding row of the grid does not belong to P, otherwise it includes the list of all the animals in the row. This solution reduces the memory required for rows not belonging to the process. Furthermore, rows can be added or deleted during load balancing without modifying the rows not involved in the operation.

The conflicts are solved by avoiding the concurrent update of a grid point. In turn this is achieved by properly ordering the communications among the processes. The computation alternates a computation step and a load balancing step. During the computation, first each process updates the upper border of its domain and sends it, through an immediate MPI communication, to its neighbour. Then, each process updates the inner part of its domain. Before updating the lower border of its domain a process receives the updated upper border from its neighbour, it updates its lower border and it sends it back. After each computation phase, *Bounded Neighbours* is executed.

5.1 Results

This section shows some preliminary performance results. The experiments have been performed on Backus, a cluster of 8 PCs running Linux and connected by a Fast Ethernet network switch [Danelutto, 2003].

We have tested the effectiveness of the Bounded Neighbours algorithm through the Wator implementation and compared the result obtained considering grids of different sizes. For each case, we have compared the speed-up obtained when load is not balanced versus that achieved by exploiting *Bounded Neighbours*. The speed-up achieved for a 600×300 grid, rs. for a 1000×2000 grid are shown in the leftmost, rs. in the rightmost part, of Figure 4. The parameters of the simulation are shown in table 1.

The percentages are referred to the total number of elements of the grid. As far as concerns the 600×300 grid, we have that the overhead introduced by *BoundedNeighbours* equals the benefits obtained by balancing the load. As a matter of fact, the speed-ups achieved in the two experiments are comparable. In the case of the 2000×1000 grid, the speed-up that is achieved by balancing the load is larger that the one that is achieved without load balancing. Furthermore, the former speed-up is close to the optimal one.

number of fishes	20 %
number of sharks	30 %
initial amount of plancton:	30%
fish survival time	7
shark survival time	3
fishes breeding age	10
sharks breeding age	12
number of simulation steps :	200

Table 1. Simulation Parameters

We have also investigated the relation between the percentage of unbalanced load and the execution time, in the case of 8 processors and a 1000×2000 grid. Figure 5 shows the execution times corresponding to different unbalance degrees, i.e. to different values of the threshold value U, see section 4.1. The values shown on the x-axis corresponds to different percentage of unbalance with respect to the size of the grid. The value 0 corresponds to the execution time obtained when *Bounded Neighbours* is not exploited. The results show that, even for low unbalances, the execution time decreases when *Bounded Neighbours* is applied.

6 CONCLUSIONS

This paper has presented the *Bounded Neighbours* approach to domain decomposition. The most important feature of this approach is that the number of neighbours of each process is bounded. An implementation of the *BoundedNeighbours* approach has been developed and its effectiveness has been tested through WaTOR, a classical irregular distributed simulation problem. Current implementation supports two neighbouring sub-domains for each domain produced by the partition. Preliminary results show a good trade off between the overhead introduced by the load balancing algorithm and the reduction of the execution time. We are planning to modify the decomposition procedure in order to support more cuts for each row. Furthermore, we will exploit our algorithm in the implementation of more complex simulation problems, such as real life problems described by cellular automata.

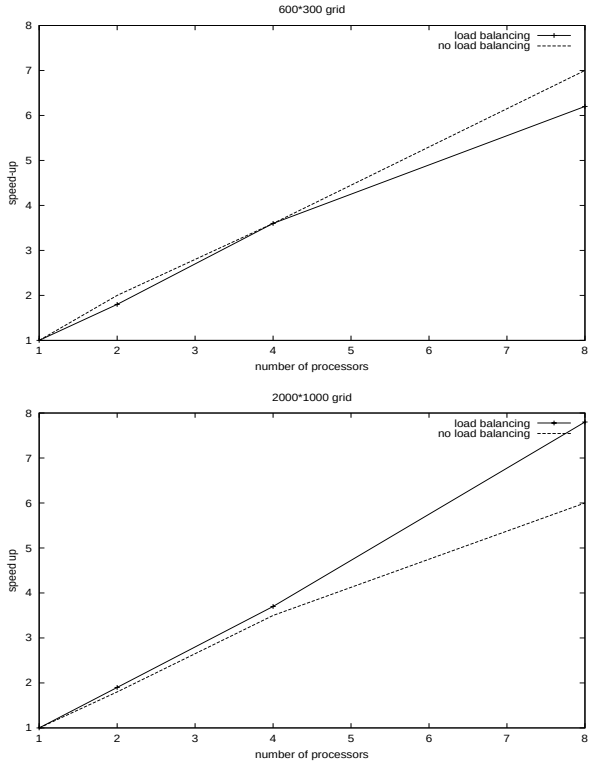

Figure 4. Wator Scalability

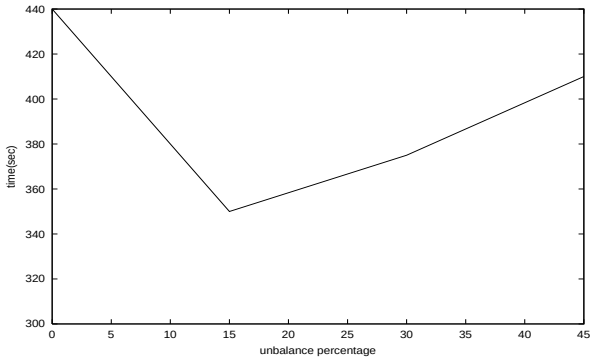

Figure 5. Execution time vs. Unbalance Percentage

References

L.Ferrucci A.Bonotti. 2002 Domain partitioning: The bounded neighbours approach. Technical report, Dipartimento di Informatica.

J.P.Singh D.E.Culler. 1998 *ParallelComputer Architecture: A Hardware/Software Approach*. Morgan Kauffmann.

Dewdney. 1984 Computer recreation. *Scientific American*, December .

D.M.Nicol and J.H.Saltz. 1990 An analysis of scattered decomposition. *IEEE transaction on Computers*, 39.

G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon, and D.Walker. 1988 *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall.

J.K.Salmon. 1990 *Parallel Hierarchical N-Body Methods*. PhD thesis, California Institute of Technology.

J.P.Singh, C.Holt, T.Totsuka, A.Gupta, and J.L.Hennessy. 1995 Load balancing and data locality in adaptive hierarchical n-body methods: Barnes hut, fast multipole and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141.

S.B.Baden J.R.Pilkington. 1995 Dynamic partitioning of non-uniform structured workloads with space-filling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):228–299, March 1995.

M.Danelutto. 2003 The backus environment. Technical report, www.di.unipi.it/ danelutto.

B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. 2001 Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141.

H. D. Simon. 1994 Partitioning of unstructured problems for parallel processing. In *Computing Systems in Engineering*, volume 2, pages 135–148.

J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J.L. Hennessy. 1995 Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141.

P. Sloot and D.Talia. 1999 Cellular automata:promise and prospects in computational science. *Future Generation Computing Systems*, (16).

Laura Ricci is currently an Assistant Professor at the Department of Computer Science, University of Pisa. Her research interests include the parallelization of irregular adaptive applications, the definition of methodologies and tools for teaching concurrency and the design of abstract interpretation based tools for parallelizing compilers.