Ādhāra: Runtime Support for Dynamic Space-Based Applications on Distributed Memory MIMD Multiprocessors

Immaneni Ashok and John Zahorjan

Technical Report # 93-04-01
April 1993
(Revised March 1994)

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

To appear in the proceedings of the
Scalable High Performance Computing Conference, May 1994

Ādhāra: Runtime Support for Dynamic Space-Based Applications on Distributed Memory MIMD Multiprocessors [*]

Immaneni Ashok and John Zahorjan
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

In this paper we describe Ādhāra, a runtime system specialized for dynamic space-based applications, such as particle-in-cell simulations, molecular dynamics problems and adaptive grid simulations. Ādhāra facilitates the programming of such applications by supporting spatial data structures (e.g., grids and particles), and facilitates obtaining good performance by performing automatic data partitioning and dynamic load balancing.

We demonstrate the effectiveness of Ādhāra by efficiently parallelizing a specific plasma physics application. The development of the parallel program involved the addition of very few lines of code beyond those required to develop a sequential version of the application, and executed at 90% efficiency on 16 nodes of an Intel Paragon.

1 Introduction

Dynamic space-based applications are simulations of objects moving through a closed k-dimensional space subject to mutual forces. There are a wide variety of such applications, differing in the kinds of objects and forces being simulated. Many important scientific applications in plasma physics, molecular dynamics, and applications that use multi-level adaptive grid methods for solving differential equations fall into the category of dynamic space-based applications.

These applications exhibit strong data locality patterns, but these patterns change during the computation as objects change position in the simulated space. To achieve good performance when run on distributed memory MIMD multiprocessors, two conflicting goals must be addressed: the strong spatial locality must be exploited, and the computational load must be balanced across the processors. These optimizations can be done statically, by either the programmer or the compiler, or dynamically, by handwritten application code or a runtime system. Relying on the programmer for these functions imposes an unreasonable burden on her, as it is a hard and time consuming process to develop the code required. At the same time, a general purpose compiler cannot be expected to automatically generate code leading to good performance, since it cannot extract the space-based data dependencies from the program source. Thus, the most appropriate approach to providing the needed functions is through a specialized runtime system that can be used with any dynamic space-based application.

The runtime system that we propose simplifies the process of developing parallel code so that scientists need to worry only about the "physics" of the application, and not the details of parallelizing their programs. *Ādhāra* achieves this objective by providing mechanisms for expressing space-based data objects, such as grids and particles, and operating with them. The programs developed using *Ādhāra* are portable across different types of MIMD architectures, since the programming model does not assume any specific architecture, and since all optimizations are performed by the runtime system.

Ādhāra aims to achieve performance as close to a hand-coded parallel program as possible. This objective is realized by minimizing the overheads of communication and load imbalance, by exploiting spatial locality and balancing the load dynamically. *Ādhāra* aims to optimize the time spent on load balancing by initiating load balancing only when needed, and by using clever schemes for balancing the load.

[*]This material is based upon work supported by the National Science Foundation (Grants CCR-9123308 and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation Systems Research Center and External Research Program. Authors' email addresses: *ashok@cs.washington.edu*, *zahorjan@cs.washington.edu*.

2 Comparison with Existing Systems

We compare *$\bar{A}$dh$\bar{a}$ra* with the relevant existing systems based on the following features that a programming environment must support for simplifying the process of developing efficient parallel code for dynamic space-based applications:

1. *Mechanisms for expressing and operating with space-based data objects*: Without these mechanisms, the user needs to develop complicated code for maintaining distributed spatial data structures. If the system does not support dynamic space-based data, such as particles, the burden of determining the granularity of space partitioning and load balancing falls on the user.

2. *Mechanisms for sharing data along partition boundaries*: These mechanisms relieve the user from managing communication, and increase portability.

3. *Automatic data partitioning*: The performance impact of a partitioning scheme depends not only on the application characteristics (such as communication patterns and load distribution and movement), but also on the number of processors and machine architecture (in particular, the speed of inter-processor communication). To achieve high performance and portability, the system must support automatic data partitioning.

4. *Automatic dynamic load balancing*: The optimal frequency of load balancing depends on the overheads due to load imbalance and load balancing. The load balancing cost depends on the algorithm as well as the machine architecture. Clearly, implementing this feature in each application would be a significant burden on the application developers.

$\bar{A}$dh$\bar{a}$ra supports all the above features. Details are given in the next section.

High-level languages such as Fortran-D [12], Vienna Fortran [9] and HPF [10] provide annotations for controlling data partitioning. The compiler and its associated runtime environment take care of the communication. This type of high-level environment reduces program development time, but may not offer good performance, since not all characteristics of the application can be exploited by a general purpose compiler. There is no convenient way of expressing dynamic space-based data objects and their spatial relationships. The user is responsible for deciding on the partitioning scheme, and when and how to perform dynamic load balancing.

PARTI [6] is a runtime system that has been developed primarily to support compilers for languages such as HPF. PARTI addresses a broader class of applications that includes unstructured mesh codes. Support for the class of space-based applications is limited to irregularly coupled regular mesh computations [1]. There is no convenient way of expressing computations that involve dynamically moving particles, such as plasma physics, molecular dynamics and N-body simulations.

The only other specialized runtime system for space-based applications that we are aware of is LPAR [5] (which is based on genMP [4]). LPAR provides mechanisms for expressing sharing of data along partition boundaries. However, dynamic space-based data objects, such as particles, cannot be conveniently expressed in LPAR. The user needs to keep track of the data relationships by partially sorting the particles into slots. Partition granularity depends on the size of the slot, and must be determined by the user. The user is also responsible for deciding on the partitioning scheme, and how and when to load balance.

3 Design of $\bar{A}$dh$\bar{a}$ra

$\bar{A}$dh$\bar{a}$ra is implemented as a C-library. It assumes simple message passing support, and does not require any specific processor interconnection topology for correctness. *$\bar{A}$dh$\bar{a}$ra* presents a *non-shared memory, data-parallel* (SPMD) programming model to the user.

To make program development easier, *$\bar{A}$dh$\bar{a}$ra* provides high-level executable statements that are specialized for dynamic space-based applications, using only concepts that are natural to this class of scientific applications. A pre-processor converts these specialized statements into a set of C-statements that make calls to the runtime library.

$\bar{A}$dh$\bar{a}$ra partitions the data using a *computation space* (the closed physical space that is modelled by the application) as the basis. To exploit spatial locality, *$\bar{A}$dh$\bar{a}$ra* partitions the data using domain decomposition [15], where the computation space is partitioned into contiguous regions and each region is assigned to a processor. Each region is a rectangular block and forms the basis for exploiting data locality and for balancing the load. Load balance is maintained by balancing the load across the regions. Each node *owns* all the data objects that map into its region, and is responsible for maintaining consistent values of the data that it owns.

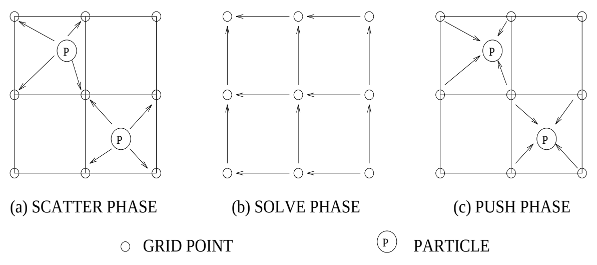

Figure 1: Data Dependencies in various phases in one time step of the EMPIC application

(Only two dimensions are used for simplicity of illustration. The grid points refer to the Electric Field, Magnetic Field or Current Density data, depending on the context.)

3.1 Example Applications

This section outlines two sample dynamic space-based applications that are used in the following section to explain the design and the programming interface of *Ādhāra*.

EMPIC

The EMPIC (plasma physics) application simulates movement of charged particles that interact by exerting electric and magnetic field forces on each other [7, 13, 15]. The force experienced by a particle depends on the current position and velocity of all the particles, and this changes continuously with time. The current implementation uses a particle-mesh method which discretizes space by a grid, and time by updating the position and velocity of the particles only at the boundaries of small time intervals. Time is simulated in a series of steps. Each step consists of three phases. In the *scatter phase*, current-density is assigned to the grid points using the position and velocity of all the particles (Figure 1(a)). In the *solve phase*, new values of electric and magnetic fields are computed at each grid point, using the old field values and the current density assigned in the scatter phase (Figure 1(b)). In the *push phase*, using the field values at the grid points, the force on each particle is computed, and the position and velocity of all the particles are updated (Figure 1(c)).

CGV

CGV is a chemistry application that simulates Crystal Growth from Vapor, which can be used to study the formation of crystals of metals such as platinum and copper [2]. Atoms are dropped, one by one, onto the existing crystal. When a new atom collides with the stable crystal, the existing atoms are disturbed, settling down after a while. For each new atom, the system is simulated for several time steps, until the crystal comes back to a stable state. Each time step has two phases: In the *potential phase*, $potential_{i,j}$ and $electronDensity_{i,j}$ are computed for each atom pair (i, j) such that distance between i and j is less than some cut-off distance. In the *push phase*, using the values computed in the previous phase, the force on each atom is computed, and the position and velocity of all the particles are updated.

3.2 Spatial Data Structures

Ādhāra distinguishes between two types of data structures: *particle* and *regular-grid*. (Support for adaptive multi-level grids is under development.) The data objects of a particle data structure can lie anywhere in the computation space, and can change their positions dynamically. A *regular-grid* is a restricted form of a particle data structure that is regular and static. It is perfectly aligned with the computation space. The mapping of a *regular-grid* onto the *computation space* is implicitly specified by giving the dimensions of the grid, whereas the mapping of a particle data structure is explicitly specified by giving the coordinate of each object of the data structure, and is allowed to change dynamically.

Ādhāra supports the most common access patterns to operate on the spatial data structures, which is to iterate over the sets of their objects. A *regular-grid* is declared and accessed as follows:

```
typedef struct { double X, Y, Z; } tType;
R-GRID tType MagneticField (Nx,Ny,Nz);
FORALL (I,J,K) IN MagneticField DO
    MagneticField[I,J,K] = function of (
                    ElectricField[I,J,K],
                    ElectricField[I,J,K+1], ..)
```

The user can access the objects in a *particle* data structure either one at a time, or in groups of arbitrary size, such that all members in each group are within some fixed distance of each other. (Molecular dynamics applications, for example, iterate over pairs of particles that are within a cut-off distance of each other.) A *particle* data structure is declared and accessed as given below. The coordinates of a particle are given by the implicit fields *coordX, coordY, coordZ.*

```
PARTICLE-DEF atomType {
    int type;
    double velocityX, velocityY, velocityZ;
    double charge; }
PARTICLE atomType Atoms;
```

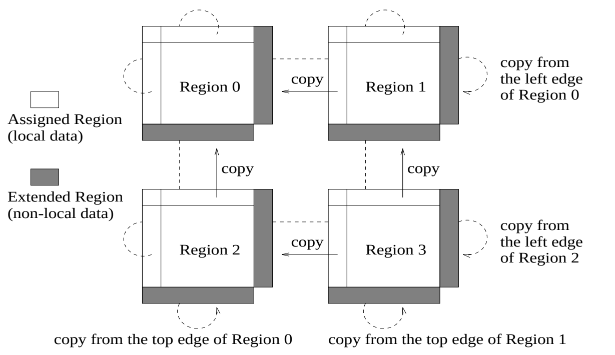

Figure 2: Read Overlap in the Solve Phase of the EMPIC Application

(The space is partitioned into four regions. The figure gives the read overlap for all the regions. Only two dimensions are used for simplicity of illustration.)

```
atomType aa;
aa.coordX = 0.25; aa.coordY = 0.5; ...
ADD aa TO Atoms;
FORALL ($A_1$,$A_2$) IN Atoms CUTOFF $R_{cut}$ DO
    potential = function of (
            $A_1$ →coordX − $A_2$ →coordX,
            $A_1$ →coordY − $A_2$ →coordY)
```

3.3 Sharing Data along the Partition Boundaries

A processor can access that portion of the data which it *owns* (local data), and some overlapped data owned by the other nodes (non-local data). The sharing of data along the boundaries of the regions is described by specifying how each region must be overlapped with the neighboring regions. The user can specify, for each direction, the overlap distance or cells (in case of *regular-grid*), and the boundary condition. The following statement is used in the *solve phase* of the EMPIC application:

```
READ-BOUNDARY ElectricField OVERLAP(
    {X-DIR = (0,1),Y-DIR = (0,1),Z-DIR = (0,1)},
    PERIODIC-BOUNDARY );
```

In the READ-BOUNDARY statement, "X-DIR = (a,b)" means that there is an overlap of 'a' cells in the negative X direction and an overlap of 'b' cells in the positive X direction. The statement given above specifies that the partition of the *regular-grid* ElectricField must be overlapped by one cell along the positive X, Y, and Z directions, and that the data must be *read* into the extended region using a *periodic* boundary condition (Figure 2).

In the *scatter phase* of the EMPIC application, for each particle p, current density is assigned to the grid points enclosing p. If the particle lies near the boundary of its region, then some of the *non-local* values of current density may need to be updated. At the end of the phase, the updated *non-local* data must be sent to its *owner*. This task is accomplished by the following statement:

```
WRITE-BOUNDARY CurrentDensity
    OVERLAP( { ALL-DIR=1 },
        PERIODIC-BOUNDARY)
    OPERATION(DOUBLE-ADD);
```

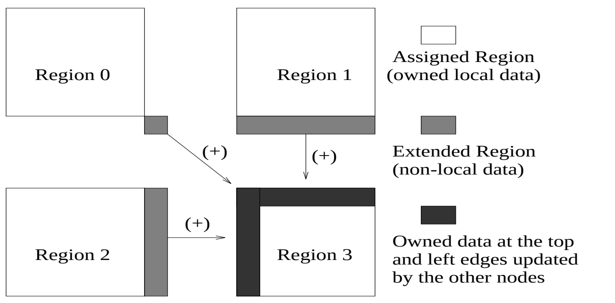

Figure 3: Write Overlap in the Scatter Phase of the EMPIC Application

(The figure gives the write overlap for the data assigned to Region-3 only. Only two dimensions are used for simplicity of illustration.)

More than one node can update the current density at the same grid point. The OPERATION specifies the commutative-associative operation that must be used to combine the values of *CurrentDensity* at the same grid point that are updated by more than one node.

3.4 Automatic Data Partitioning and Dynamic Load Balancing

Ādhāra uses heuristics to dynamically choose a good partitioning scheme from among seven possible schemes (block, beams in three directions and slices in three directions – Figure 4) based on the communication patterns and the distribution and movement of the load. The computation is divided into *phases*, where each phase computes on a particular data structure called its *primary* data structure. (It often corresponds to the computation in a *do-loop* of a sequential algorithm.) This *phase*, in concept, is similar to the phase defined in the Phase Abstractions Model [11]. The compute load in a *phase* is assumed to be proportional to the sum of the computation (measured in terms of iterations) on the objects in the corresponding primary data structure. This information is used

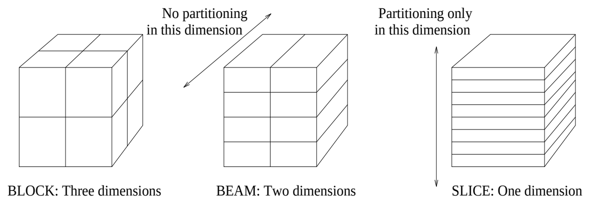

Figure 4: Methods of Partitioning a 3D Space into Eight Rectangular Regions

by *Ādhāra* to balance the load in each phase. Phases are specified by the user, and they are natural to the application. For example, the phases in the EMPIC application can be declared as follows:

```
PHASE GridPhase {
    ( ElectricField, PRIMARY,
        USAGE READ-WRITE )
    ( MagneticField, USAGE READ-WRITE )
    ( CurrentDensity, USAGE READ-ONLY )
}
PHASE PushPhase {
    ( ChargedParticle, PRIMARY,
        USAGE READ-WRITE )
    ( ElectricField, USAGE READ-ONLY )
    ( MagneticField, USAGE READ-ONLY )
}
PHASE ScatterPhase {
    ( ChargedParticle, PRIMARY,
        USAGE READ-ONLY )
    ( CurrentDensity, USAGE WRITE-ONLY )
}
```

In the *GridPhase*, ElectricField is the *primary* data structure, so the computation is proportional to the number of grid points. The data structures MagneticField and CurrentDensity are also accessed in the *GridPhase*. The USAGE specifies how the data is used in this computation. The information about the usage is used by Ādhāra to determine whether to redistribute the data or not, when the execution proceeds from one *phase* to another. The computation is performed by executing a procedure within a phase:

```
void SolveRoutine() { ..... }
void PushRoutine() { ..... }
void ScatterRoutine() { ..... }
main() {
    for T time-steps do:
        EXECUTE SolveRoutine IN GridPhase;
        EXECUTE PushRoutine IN PushPhase;
        EXECUTE ScatterRoutine IN ScatterPhase;
}
```

In the *GridPhase*, the space is partitioned into equal sized regions, since the computation is proportional to the number of elements in ElectricField, a *regular-grid*. In the *PushPhase*, the computation is proportional to the number of particles, which are nonuniformly distributed in space, so the space is partitioned in such a way that each region contains approximately equal number of particles. Since the distribution of particles in space changes dynamically, the space partitioning in *PushPhase* must also change dynamically. In the above fragment of the program, in between the computation of *SolveRoutine* and *PushRoutine*, the data elements of ElectricField and MagneticField are automatically redistributed, if the space partitioning in *PushPhase* is different from that in *GridPhase*.

Ādhāra uses predictive [3] and monitoring [14] schemes for determining when to balance the load. The goal is to minimize the sum of the overheads due to load imbalance and load balancing. *Ādhāra* does not attempt to find an optimal partitioning every time it load balances, but instead uses heuristics to find a reasonably good one. It uses a *non-uniform, adaptive load balancing grid* to discretize the computation space, and maintains the load in each cell of this grid. Each node maintains a portion of this grid and communicates the local information with the other nodes to globally repartition the grid. *Ādhāra* alters this grid dynamically depending on the load density and movement, and uses a *hierarchical* scheme [8] for globally repartioning the grid. This scheme incurs very little overhead, and exploits the information about load movement and distribution. (Traditional orthogonal recursive bisection schemes do not exploit this information.) In this paper we focus only on the runtime support and the programming interface provided by *Ādhāra*. The details of the load balancing scheme are covered in another paper [3].

4 Results

We give the results of parallelizing a specific three-dimensional electromagnetic particle-in-cell (EMPIC) application using the *Ādhāra* runtime system. This application was developed by hand converting a sequential Fortran program (written by David Walker, Oak Ridge National Laboratory) to a sequential C program, and then to an *Ādhāra* program.

Convenience of Programming and Portability

Parallelizing a 1500 line sequential 3D EMPIC application using *Ādhāra* required the addition of less than 50 lines of code (to declare spatial data structures, phases, and overlaps for data sharing), and modification of the *for*-loops to *forall*-loops (for iterating

Table 1: Performance of a three-dimensional EMPIC application

(A 327680 particle, 65536 grid-point application is simulated on 16 nodes of Intel Paragon for 500 time-steps. The numbers given below represent the time as a percentage of the optimal compute time, which is 2325 seconds.)

	Static Load Balancing		Dynamic Load Balancing	
	Block	Beam	Block	Beam
total measured	221.3	141.6	118.7	111.5
optimal	100.0	100.0	100.0	100.0
data locality	11.0	3.6	3.4	2.6
communication	10.3	8.0	11.5	7.0
load balancing	0.0	0.0	1.6	0.6
load imbalance	100.0	30.0	2.2	1.3
efficiency	45%	71%	84%	90%

over the set of objects of the spatial data structures). We estimate that developing an optimized hand-coded parallel program involves writing at least 5000 lines of extra code, based on the amount of *Ādhāra* code exercised by this program.

Using *Ādhāra* there is absolutely no necessity for the programmer to develop any code to handle data partitioning, communication and load balancing. Since the details about communication are completely hidden from the programmer, the code developed on *Ādhāra* can be easily ported to different parallel (MIMD) architectures. Thus, *Ādhāra* provides two advantages: easy porting of an application on to a new parallel machine, and easy development of a new application using the same parallelism management primitives.

Performance

Table 1 gives the performance of the 3D EMPIC application developed using the *Ādhāra* runtime system. The application is executed on 16 nodes of an Intel Paragon. (We are currently porting *Ādhāra* to the KSR1.) A $32 \times 64 \times 32$ grid is used to discretize the physical space, and the movement of 327680 particles is simulated for 500 time steps. The first two columns of Table 1 is for the case where data locality is exploited but no dynamic load balancing is performed (the load is balanced statically at the beginning of the execution). This is representative of the code a user might reasonably custom develop. The first column shows results for a block partitioning scheme, and the second column for a beamX scheme where only Y and Z dimensions are partitioned. The last two columns is for the case where load is balanced dynamically while exploiting data locality. The results show that the performance is sensitive to the partitioning scheme. *Ādhāra* dynamically estimates load movement and density, and uses these estimates to choose a good scheme (the beamX scheme, in this case).

We compare the results in each case to the *optimal compute time*, which is calculated by measuring the time taken by the true sequential program and dividing by 16. The numbers given in Table 1 represent the time as a percentage of this optimal time. The *total observed* time is the measured execution time of the application for 500 simulation time steps. The *data locality overhead* includes the following: the overhead of implementing the *particle* data structure for maintaining the spatial coordinates of the particles (which constitutes more than 80% of the data locality overhead), and the overhead of exchanging particles among the processors. The *communication overhead* gives the time spent on exchanging data for data overlaps and redistributing data between *phases*. The *load balancing overhead* gives the time spent on the load balancing protocol. The processor idle time due to the load imbalance is given by the *load imbalance overhead.*

The bottom line on execution performance is given by the measured processor *efficiency*. (Processor efficiency is the average fraction of time each processor in a parallel execution spends performing work inherent to the computation, rather than overhead. We note that the efficiences reported here are computed using a true sequential implementation of the program as the basis of comparison.) We see that the implementation using *Ādhāra* achieves very high efficiency, and that, in contrast, the more difficult to construct hand-coded version employing static partitioning performs much worse.

While it would be possible to build load balancing by hand into this application, it would require greatly increased programming effort, and when done would be functionally identical to the easily coded *Ādhāra* version. Because the execution overhead of providing this functionality in a run-time system, rather than directly in the application, is small, we conclude that there is no performance benefit to hand coding into each application the parallelism management functions that *Ādhāra* provides. Additionally, because hand coding requires vastly increased effort, *Ādhāra* is clearly the better choice for the development of dynamic, space-based applications.

5 Conclusions

Dynamic space-based applications, those exhibiting spatial locality and operating on data objects whose spatial relationships change dynamically, are an important class of scientific applications. To efficiently parallelize them on distributed memory multiprocessors, spatial locality must be exploited and computation load must be balanced across the processors. In this paper we argued that specialized runtime systems are required for this purpose. We described the design and the programming interface of *Ādhāra*, a system that is specialized for dynamic space-based applications. We showed the effectiveness of *Ādhāra* by efficiently parallelizing a plasma physics application, requiring very little additional code compared to a functionally equivalent sequential program and involving only a very few, simple concepts to deal with the parallel execution, thus minimally distracting the programmer from her primary job of implementing the physics of the application.

Acknowledgements

We would like to thank Ed Felten for introducing us to the class of space-based applications, and David Walker for helpful discussions on the topic of this work and for providing us the sequential code for the EM-PIC application.

References

[1] Gagan Agrawal, Alan Sussman and Joel Saltz. Compiler and Runtime Support for Structured and Block Structured Applications. *Proceedings of the Supercomputing Conference, pp.578-587* (November 1993).

[2] M.P.Allen and D.J.Tildesley. Computer Simulation of Liquids. *Clarendon Press, Oxford* (1987).

[3] Immaneni Ashok and John Zahorjan. Data Partitioning and Dynamic Load Balancing for Dynamic Space-Based Applications. *In Preparation.*

[4] Scott Baden. Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors. *SIAM Journal of Science and Statistical Computation, Volume 12, Number 1, pp.145-157* (January 1991).

[5] Scott Baden and Scott Kohn. Lattice Parallelism: A Parallel Programming Model for Manipulating Non-Uniform, Structured Scientific Data Structures. *SIGPLAN Notices (1992 Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors)* (January 1993).

[6] Harry Berryman, Joel Saltz and Jeffrey Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines. *Concurrency: Practice and Experience, Volume 3(3), pp.159-178* (June 1991).

[7] Charles K. Birdsall and A. Bruce Langdon. Plasma Physics via Computer Simulation. *McGraw-Hill International, New York* (1985).

[8] Philip M. Campbell, Edward A. Carmona and David W. Walker. Hierarchical Domain Decomposition With Unitary Load Balancing for Electromagnetic Particle-In-Cell Codes. *Proceedings of the Fifth Distributed Memory Computing Conference, pp.943-950* (April 1990).

[9] Barbara Chapman, Piyush Mehrotra, Hans Moritsch and Hans Zima. Dynamic Data Distributions in Vienna Fortran. *Proceedings of the Supercomputing Conference, pp.284-293* (November 1993).

[10] B. Chapman, P. Mehrotra and H.Zima. High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (May 1993).

[11] William G. Griswold, Gail A. Harrison, David Notkin and Lawrence Snyder. Scalable Abstractions for Parallel Programming. *Proceedings of the 5th Distributed Memory Computing Conference (April 1990).*

[12] Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer and Chau-Wen Tseng. An Overview of the Fortran D Programming System. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing* (August 1991).

[13] Roger W. Hockney and James W. Eastwood. Computer Simulation Using Particles. *Adam Hilger, Bristol, England* (1988).

[14] David M. Nicol and Joel H. Saltz. Dynamic Remapping of Parallel Computations with Varying Resource Demands. *IEEE Transactions on Computers, Volume 37, Number 9, pp.1073-1087* (1988).

[15] David W. Walker. Characterizing the Parallel Performance of a large-scale Particle-In-Cell Plasma Simulation Code. *Concurrency: Practice and Experience, volume 2, pp.257-288* (1990).

Appendix - Parallel EMPIC Application Using Ādhāra

```
/* specify size of the simulation box */
COMPUTATION-SPACE ( $size_x$, $size_y$, $size_z$ );

typedef struct { double X, Y, Z; } tripleType;
R-GRID tripleType
    MagneticField ($N_x$,$N_y$,$N_z$),
    ElectricField ($N_x$,$N_y$,$N_z$),
    CurrentDensity ($N_x$,$N_y$,$N_z$);

PARTICLE-DEF particleType {
    int type;
    double velocityX, velocityY, velocityZ;
    .........
    double charge;
}
PARTICLE particleType ChargedParticle;

PHASE GridPhase {
    (ElectricField, PRIMARY,
            USAGE READ-WRITE)
    (MagneticField, USAGE READ-WRITE)
    (CurrentDensity, USAGE READ-ONLY) }
PHASE PushPhase {
    (ChargedParticle, PRIMARY,
            USAGE READ-WRITE)
    (ElectricField, USAGE READ-ONLY)
    (MagneticField, USAGE READ-ONLY) }
PHASE ScatterPhase {
    (ChargedParticle, PRIMARY,
            USAGE READ-ONLY)
    (CurrentDensity, USAGE WRITE-ONLY) }

/*********************************************/
main() {
    particleType part;

    ADHARA-INIT();
    /* read initial particle configuration */
    for ( i = 0; i < N; i++ ) {
        read the initial position and velocity of
        the i-th particle into
        part.coordX, ..., part.velocityX,...
        .....
        ADD part TO ChargedParticle;
    }
    ... other initialization ...

    for ( step = 0; step < NumTimeSteps; step++ ) {
        EXECUTE SolveRoutine IN GridPhase;
        EXECUTE PushRoutine IN PushPhase;
        EXECUTE ScatterRoutine IN ScatterPhase;
    }
}
/*********************************************/
```

```
void SolveRoutine() {
    /* This algorithm uses the leap-frog scheme
            for the time integration */
    /* Advance Magnetic Fields from step K to K+1/2 */
    READ-BOUNDARY ElectricField OVERLAP(
        {X-DIR = (0,1), Y-DIR = (0,1), Z-DIR = (0,1)},
        PERIODIC-BOUNDARY );
    FORALL (I,J,K) IN MagneticField DO {
        MagneticField[I,J,K].X = function of (
            ElecField[I,J,K].Y - ElecField[I,J,K+1].Y,
            ElecField[I,J,K].Z - ElecField[I,J+1,K].Z,
            ....... )
        ........
    }
    /* Advance Electric Fields from step K to K+1 */
    READ-BOUNDARY ElectricField OVERLAP(
        {X-DIR = (1,0), Y-DIR = (1,0), Z-DIR = (1,0)},
        PERIODIC-BOUNDARY );
    FORALL (I,J,K) IN ElectricField DO {
        ElectricField[I,J,K].X = function of (
            MagField[I,J,K].Y - MagField[I,J,K-1].Y,
            MagField[I,J,K].Z - MagField[I,J-1,K].Z,
            CurrentDensity[I,J,K].X, ..... )
        ........
    }
    /* Adv. Magnetic Fields from step K+1/2 to K+1 */
    ................
}
/*********************************************/
void PushRoutine() {
    READ-BOUNDARY ElectricField ....
    READ-BOUNDARY MagneticField ....
    FORALL (P) IN ChargedParticle DO {
        compute the electric and magnetic fields on P
            by interpolating the fields at the grid points
            enclosing P ( CELL(P)[1..8] )
        update the position and velocity of P
    }
}
/*********************************************/
void ScatterRoutine() {
    FORALL (I,J,K) IN CurrentDensity INCLUDING
        BOUNDARY { ALL-DIR = 1 } DO
            CurrentDensity[I,J,K].X =
            CurrentDensity[I,J,K].Y =
            CurrentDensity[I,J,K].Z = 0.0;
    FORALL (P) IN ChargedParticle DO {
        for each grid point (I,J,K) in CELL(P) do:
            CurrentDensity[I,J,K] += function of (
                P−>coordX, ..... ,
                P−>velocityX, ..... )
    }
    WRITE-BOUNDARY CurrentDensity
        OVERLAP( { ALL-DIR = 1 },
                PERIODIC-BOUNDARY )
        OPERATION( DOUBLE-ADD );
}
```