Visualizing Distributed Data Structures *

Suresh Srinivas, Dennis Gannon
Department of Computer Science, Lindley Hall 215,
Indiana University, Bloomington, IN 47405.
{ssriniva,gannon}@cs.indiana.edu

Abstract

A new programming style for large-scale parallel programs centered around distributed data structures has emerged. The current parallel program visualization tools were intended for the old style and do not deal with distributed data structures. We show, with several examples of visualizations and animations developed for large scale pC++ programs, that visualizing and animating distributed data structures is an important part of debugging and performance tuning for the new style parallel programs. Our approach is based on a new methodology for recording execution behavior that uses I/O abstractions and compile time source analysis and instrumentation. We also discuss a new framework for investigating the execution behavior of large-scale parallel programs and show where visualization and animation fit in

1 Introduction

Large-scale parallel programs are characterized by long computation times, thousands of lines of code, and complex data structures. They are increasingly being applied to solve scientific problems in diverse areas such as cosmology, physics, and biology.

In recent times, there is increased activity in designing high-level parallel programming languages for developing these large-scale parallel programs. We are seeing the emergence of a *new style* for writing applications. In this style, the data is partitioned across the nodes of the parallel machine but is accessible as a single *distributed data structure* in the parallel program. In essence this means that parallel programs are parallel algorithms operating on distributed data structures. It is then the responsibility of the compiler to handle the message passing in sharp contrast to the *old style* where the user had to specify all of the message passing explicitly. A number of parallel programming languages that support this style are now available. They include pC++[7],HPF [5] and Split-C [2].

Developers of these large-scale parallel programs need tools for debugging and performance tuning. But unfortunately tools have not kept up with the changes in style. In particular tools that answer questions like the following do not exist today:

- How does a particular distributed data structure evolve over the course of a parallel program?

*This research is supported in part by ARPA under contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616.

- How are the data structures distributed and how does program performance depend on the distributions?
- How does the program performance relate back to the parallel algorithm and the distributed data structures?

We will address these questions in this paper using visualization and animation to illustrate the distribution, evolution, and communication patterns of distributed data structures [1]. And we will show that visualization and animation of distributed data structures are essential part of tools for the development of these new style large-scale parallel programs.

The new programming style has also rendered the conventional methods for recording execution behavior ineffective. We propose a solution based on I/O abstractions for distributed data structures and compile-time source analysis and instrumentation. We also describe a new *framework* for developing extensible and customizable tools for investigating such large-scale parallel programs and show where visualization and animation fit into the framework.

2 Background

Consensus is emerging in two aspects of parallel computing: the organizational characteristics of the current generation of parallel machines and the way we program them.

The organizational characteristics of a number of parallel machines are very similar. They are a collection of workstation-like computers, each consisting of an off-the-shelf microprocessor, sizable memory and connected to an interconnection network. They differ in the structure of the interconnection network (fat-tree on Thinking Machines CM-5, Mesh on Intel Paragon and 3D torus on the Cray T3D) and the processor used.

These parallel machines are increasingly being programmed in high-level parallel languages such as pC++, HPF, Split-C *etc.* - that support distributed data structures and thus a shared name space for the data. The application programmer writes programs in these languages without thinking about the details of message-passing between nodes. The compiler automatically generates calls to the message-passing routines when non-local pieces of the distributed data structure are referenced. The main difference between these languages is in the underlying language they extend and the paradigm they allow for parallel programming. pC++ is an extension of C++ meant for writing portable parallel object-oriented programs. HPF is an extension of Fortran90 with distributed arrays and several fast intrinsics to operate upon them. Split-C is an extension of C with global pointers that can be used to build distributed data structures.

We will discuss pC++ in detail since the visualization and animation examples are for parallel programs written in that language. However the ideas apply in the context of other languages as well.

2.1 pC++ overview

pC++ is an extension of the C++ language and is designed to allow the programmer to create a *Collection* of objects distributed across the nodes of a parallel machine. The objects are constructed from a base *element* class. Member functions of this element class can be applied to the entire collection. The programmer provides specifications for the distribution of the objects.

The pC++ collection library provides a base collection, called the `SuperKernel`. It is designed to be used as the base collection for all other collections. It builds arrays of element objects and provides a global name space

[1] The proceedings do not have color plates and to view the full version of the paper in color please look at `http://www.cica.indiana.edu/sage/docs.html`

for the element objects. The declaration: `SuperKernel<T> MyCollection(&D, &A);` creates a collection called `MyCollection` which is a set of objects of type `T`. The objects `D` and `A` specify both the size and the distribution pattern for `MyCollection` and their definitions are omitted for brevity. Assume `T` has a member function `foo`; then the invocation `MyCollection.foo();` is a parallel application of `foo` to the entire collection.

3 Investigating large-scale parallel programs

The development of efficient parallel programs is more difficult than their sequential counterparts. Parallel programs need two kinds of debugging in contrast to sequential programs: correctness debugging and performance debugging. Therefore tools that implement such debugging, in the context of large-scale parallel programs, are extremely important.

These tools work by investigating the execution behavior of the parallel program either interactively or post-mortem. The kind of questions an application programmer might expect these tools to answer include:

- Is the data structure evolving correctly?
- Do the communication access patterns in the program look right?
- Does the program have good load balance?
- Are there redundant communication access patterns?

3.1 Recording execution behavior

To investigate execution behavior after the program execution, a history of the parallel program execution has to be recorded. Depending on the level of detail, this could include the states of distributed data structures, the inter-processor communications, and even lower level information like memory references. This is true for investigating both the old style parallel programs as well as the new style parallel programs.

The following method is used for recording execution behavior in the old style parallel programs:

- The application programmers code their own file output for recording the states of data structures. In the absence of distributed data structures this is accomplished by collecting the local data structures at a single node and performing all the file output from that node or by recording the local data structures independently on each node.
- The message-passing libraries are instrumented to record the inter-processor communication. The parallel program is then linked with these instrumented libraries instead of the regular message-passing ones. This instrumented program on execution produces a history of the inter-processor communication ([6], [9], [13]).

The above method is largely inadequate and cumbersome for the new style parallel programs for the following reasons:

- The programmer worries about the lower level machine specific I/O which may not be uniform across architectures. Also collecting all the local data structures and recording them as a single logical data structure involves additional code, which is application specific.

- Instrumention in the message passing libraries provides no information about the reference, of the distributed data structure, that caused the communication. [2]

We propose Input/Output(I/O) abstraction and compile-time source analysis and instrumentation as solutions to these problems.

I/O abstraction for distributed data structures is a simple and portable way of storing and retrieving whole distributed data structures. It is implemented by special syntax in the programming language or library calls to special I/O routines [8]. With I/O abstraction the programmer does not have to worry about the lower level machine specific I/O. Also the I/O abstractions can be implemented to efficiently use the parallel I/O capabilities of the parallel machine.

Source analysis and instrumentation for the new style parallel programs involves identification of distributed data structure references in the program and adding additional code to the source to record them during execution. This modified source (called the instrumented program) produces a history of the inter- and intra-distributed data structure references on execution.

3.2 Framework for tools

In the past, several architecture specific, language specific or application specific tools were built for debugging parallel programs. It is increasingly becoming clear, due to a plethora of tools which are not extensible, that we need to concentrate on building flexible tools. Small size, high extensibility and customizability are their desired characteristics. With the advent of high quality tools for symbolic manipulation (Mathematica), visualization (AVS or Explorer), and analysis (SAS or Splus) it becomes important for parallel programming tools to interact with them to avoid reimplementing functionality.

We sketch an underlying framework for building tools with the above characteristics. The treatment is brief and the main intent here is to show where visualization and animation fits into such a framework. The details can be found in [15].

There are two aspects to the framework. A programming language, the *metaprogramming language*, for making tool extensible and customizable. And the *support* provided by parallel programming languages for tool builders.

The important design issues for a metaprogramming language are to keep the language small, simple, and provide hooks to extend it with new types and primitives. Also it should allow top-level definitions for functions and have input/output capabilites.

For example, a metaprogramming language for tools to investigate large-scale parallel programs should provide:

- Primitives for visualization and animation.
- Primitives for operating on the parallel program source.
- Primitives for extraction of relevant information from execution history.
- The ability to build modules that produce scripts for other tools [3].

The support that a parallel programming language implementation provides for tools largely determines its success. In the previous section we identified two requirements for recording execution behavior of parallel

[2] The reason being, the compiler generates calls to the message passing library routines and these library routines do not know where they were called from.

[3] For example instead of reimplementing the plotting abilities of `gnuplot` the metaprograming language should allow the construction of a module to produces scripts for gnuplot and invoke it.

programs. They are I/O abstractions for recording distributed data structures and program source analysis and instrumentation for recording distributed data structure references. The parallel programming language should allow it's internal representation to be accessible for implementing the above requirements. For example, we have implemented the I/O abstraction and source analysis/instrumentation for pC++ using the powerful C++ library [3] which allows source analysis and transformation.

4 Visualization and animation in investigations

Recorded execution behavior by itself does not provide much insight. Visualization and animation have been successfully used in the past to *view* the execution behavior ([9], [13], [14], *etc.*). The new style of writing parallel programs with distributed data structures has made the visualizations too low level and ineffective. We solve this problem by basing our visualizations and animations on distributed data structures.

Parallel program execution with distributed data structures have three basic properties that are interesting for visualization and animation. They are:

Distribution which describes the partitioning of the distributed data structure across the nodes of the parallel machine.

Evolution which describes the state changes of the distributed data structure during the course of a subroutine or a program.

Communication which describes the inter and intra distributed data structure references and communication that result from those references.

We show with examples the visualizations and animations that result from the above three properties. All of our visualizations and animations are based on the idea of providing multiple perceptive cues to the user using color and 3D graphics. They are also high level *i.e.,* closer to the program source and architecture independent. Many of the visualization and animations in this paper were programmed in IRIS Inventor^{TM}. It is an object-oriented tool kit for developing interactive, three dimensional graphics applications on the Silicon Graphics machines.

4.1 Visualizing the distribution of data structures

In languages like HPF and pC++ the user specifies the partitioning of the distributed data structure through directives. The process of partitioning is based on a two-level mapping of the distributed data structures onto the processors. These distribution directives when used correctly can give high performance.

With a growing number of applications written in these languages there is a need for tools that help understand how these directives affect performance. Visualization can be used to show:

- How the elements of the distributed data structure are partitioned across a given number of processors.
- How different partitioning of the distributed data structures perform for a given reference pattern of inter and intra distributed data structures.

Consider a simple example to illustrate the point. Note, that the visualization[4] we present here does not scale up to large number of processors.

[4]The visualizations were generated by a tool `dist-gaze` developed under the framework of the previous section

Given 4 processors and a 2 dimensional collection `A` of size 8 by 8. We are interested in invoking a method `foo` over the diagonal of the collection `A`. We would like to know which of the following distributions would give good performance:

1. `BLOCK` distribution in the 1st dimension and `BLOCK` distribution in the 2nd dimension.
2. `BLOCK` distribution in the 1st dimension and `CYCLIC` distribution in the 2nd dimension.
3. `CYCLIC` distribution in the 1st dimension and `BLOCK` distribution in the 2nd dimension.
4. `CYCLIC` distribution in the 1st dimension and `CYCLIC` distribution in the 2nd dimension.

Figure 2 shows the visualizations resulting from the distributions and points out that the `CYCLIC, BLOCK` and `BLOCK, CYCLIC` give the best performance for the above problem.

4.2 Observing the evolution of distributed data structures through animation

Even for simple parallel programs observing the state changes of distributed data structures during the course of the program can lead to insights into program behavior. Large-scale parallel programs sometimes have dynamic distributed data structures and irregular computation over them. Watching the evolution of these dynamic data structures is often times a necessity during their development.

We illustrate the importance of evolution with animations of distributed data structures in an adaptive mesh refinement program and a particle mesh program written in pC++.[5]

4.2.1 Animation of adaptive data structures

Adaptive mesh refinement algorithms allow higher resolution computation to be performed over selected subdomains of the computational domain. For grid based computation this is achieved by a hierarchical tree of grids.

We have developed such an adaptive mesh refinement program in pC++ and have made an animation of the evolution of the adaptive data structure. The main data structure in the program is an adaptive data structure, which consists of five levels of increasingly finer grids. A simple gaussian function over the grid identifies the areas where higher resolution computation is needed.

A number of different views were programmed to identify the ones that maximize the visual cues to the human eye. The 3D perspective views and a 2D mesh view turn out to be the most effective. *Figure 2* shows the 3D perspective view. This animation helped us identify several bugs in the adaptive mesh refinement program. An important bug was in the copying stage where data is copied from the coarser grid to the finer grid. This bug revealed itself as small peaks in the finer grid where there was no refinement.

4.2.2 Animation of particle mesh densities

We have programmed an N-body code, the Particle Mesh (PM) code in pC++. The PM[6] code computes long-range gravitational forces in a galaxy or galaxy cluster system by solving the gravitational potential on a mesh. For details about the pC++ implementation refer to [4].

[5] These were generated by a tool **peewee** for astrophysics applications developed in the framework from Section 3.0

[6] The original PM code is from the Grand Challenge Cosmology Consortium (GC^3)

There are two distributed data structures: a one dimensional particle list collection and two dimensional mesh collection. The two dimensional mesh collection represents the three-dimensional space. Each element of the two dimensional mesh collection contains an array of mesh points that have the same x and y coordinates but different z coordinates.

The following is a brief sketch of the mesh collection distributed data structure:

```
class MeshElement {
 public:
  double mass[nz], position[nz];
  double density[nz];
};
Collection Mesh:SuperKernel {
  public:
   Mesh(double lx, double ly, double lz,
        Distribution *T, Align *A);
   void computePotential();
   ...
};
// Defn of DP and AP are omitted
Mesh<MeshElement> mesh(&DP, AP);
```

As the particles move due to gravitational interaction, the densities of the particles change over the mesh. Animation of the scalar field represented by the densities of particles over the mesh reveals how the forces of gravitation work and the evolution of the particles during the course of the simulation. *Figure 3* and *Figure 4* illustrates snapshots from the animation of an N-body code using an adaptive grid data structure as well as one using a regular grid data structure.

4.3 Animation of communication patterns

To achieve good performance, the designers of parallel programs should take into consideration the communication behavior in the program. Communication occurs through distributed data structure references. Inter-processor or non-local distributed data structure references are several orders slower than local references.

There are two kinds of distributed data structure references. *Intra-data structure references* which occur completely within the distributed data structure. And *inter-data structure references* which occur across two or more distributed data structures. They both can be either local or non-local references.

We believe that animation of the inter- and intra-distributed data structure references can help identify redundant non-local references, and the interaction between distributions and references. Eliminating redundant references or changing distributions to minimize non-local references leads to improved performance (refer to [2] for a good discussion about redundant references in a distributed graph and the performance improvement attained by eliminating them).

Now we give examples of intra distributed data structure references in the conjugate gradient and bitonic sort programs written in pC++.

Conjugate gradient is one of the programs in the pC++ test suite. It solves the equation $A * u = f$ for u, given f by a conjugate gradient method. The details of the pC++ implementation are described elsewhere[10].

The main distributed data structure is the `Grid` collection that represents the sites in a two dimensional grid. The individual elements at each site of the Grid collection are small 2D array's. *Figure 5* shows the distributed data structure references.

We record the distributed data structure references that occur in this program by analyzing the pC++ program and instrumenting the references and running the program. We use an execution driven simulation technique for executing the pC++ program. Details of the technique are beyond the scope of this paper but can be found in [16].

In our N-body program (the PM code), the particles in the particle list distributed data structure are sorted periodically to preserve their physical locality i.e particles that are neighbors in the sorted list are closest to the same mesh point. A parallel bitonic sort is used for the sorting. This sorting involves several intra distributed data structure references (see *Figure 6*).

5 Related work

Much of the visualization and animation work in parallel computing has concentrated on performance. Sieve from Indiana, Pablo from Illinois, Paragraph from Oakridge and AIMS from NASA are some of the tools for performance visualization.

They are all event based, have 2D views and concentrate on details like message traffic between processors. They are not very extensible or customizable although Sieve does provide a macro language. They have worked well in the past but are largely inadequate for the new style of parallel programming for reasons that we detailed earlier (mainly their lack of handling distributed data structures and relating performance back to the user-level source).

The University of Oregon is developing performance analysis tools for pC++, called Tau [12], and have similar goals of extensibility and customization that we have described in the framework. May and Berman stress the importance of extensibility and creation of new views in the context of a parallel debugger Panorama[11]. Both Tau and Panorama have Tcl as their extension language. Our extension language, described in [17, 15] is based on Scheme.

6 Conclusion

New style tools have to be built for the emerging new style of parallel programming. The main contribution of this paper has been to show visualization and animation of distributed data structures will be an important part of such tools. The other contributions are in the new methodology for recording execution behavior and the framework for building extensible and customizable new style tools.

To summarize the experiments described in this paper show that:

- Visualization and animation of distributed data structures is extremely useful. They have revealed several (both correctness and performance) bugs in parallel programs. Two examples are the bug in the copying stage of the adaptive mesh refinement and the bug in the distribution portion of the runtime system.

- Three dimensional views of distributed data structures are essential for large-scale parallel programs. Much of the earlier visualization work for parallel programs has been 2D.
- There are a plethora of tools and so we need a programming language in which we can write modules that can treat other tools as targets and produce code for them instead of reimplementing functionality.
- Tools to study the new style large-scale parallel programs need support from the compiler and the language.

Acknowledgements

Thanks the members of the Sage project[7] for the excellent software infrastructure. We would also like to thank: Shelby Yang, Sekhar Sarukkai, Sudhir Rao, Peter Shirley, Jacob Gotwals and the anonymous referees for improving the paper.

References

[1] T. Casavant, editor. *Journal of Parallel and distributed computing*, volume 18. June 1993. Special issue on tools and methods for visualization of parallel systems and computation.

[2] D. Culler and et. al. Parallel programming in Split-C. In *Proceedings of Supercomputing 93*, November 1993. Available by anonymous ftp from ftp.cs.ucb.berkeley:pub/CASTLE.

[3] D.Gannon, P.Beckman, F.Bodin, J.Gotwals, S.Narayana, S.Srinivas, and B.Winnika. Sage++: An object oriented toolkit for program transformations. In *Proceedings of Oonski 94*, April 1994. Available by anonymous ftp from moose.cs.indiana.edu:pub/sage/oonski94.ps.

[4] D.Gannon, S.Yang, S.Srinivas, V.Menkov, and P.Bode. Object-oriented methods for parallel execution of astrophysics simulations. In *Proceedings of Mardigras94*, February 1994. Available from gannon@cs.indiana.edu.

[5] D.Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1:25–42, 1993. Available by anonymous ftp from titan.cs.rice.edu:pub/HPFF.

[6] C. Fineman, P. Hontalas, S. Linstgarten, and J. Yan. A users guide to AIMS-the Ames InstruMentation System. Technical report, NASA Ames Technical Report, 1992. Contact yan@ptolemy.arc.nasa.gov for details.

[7] D. Gannon, S. X. Yang, and P. Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Computer Science Department, Indiana University, available from moose.cs.indiana.edu:/pub/sage by ftp, 1994.

[8] J. Gotwals and S. Srinivas. I/O abstractions for pC++. Working notes.

[9] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 6(5):29–39, September 1991.

[7] Accessible through the World Wide Web from `http://www.cica.indiana.edu/sage/home-page.html`

[10] J. K. Lee and D. Gannon. Object oriented parallel programming: Experiments and results. In *Proceedings of Supercomputing 91 (Albuquerque, Nov.)*, pages 273–282. IEEE Computer Society and ACM, 1991.

[11] J. May and F. Berman. Creating views for debugging parallel programs. In *Proceedings of Scalable High-Performance Computing Conference, SHPCC94*, pages 833–840, May 1994.

[12] B. Mohr, D. Brown, and A. Malony. Tau: A portable parallel program analysis environment for pc++. Technical report, University of Oregon, 1994. Available as conpar94.ftp.ps from moose.cs.indiana.edu:pub/sage.

[13] D. A. Reed and et. al. The Pablo performance analysis environment. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, October 1992. Accessible from WWW as http://bugle.cs.uiuc.edu/.

[14] S. R. Sarukkai and D. Gannon. Sieve: A performance debugging environment for parallel programs. *Journal of Parallel and distributed computing*, 18(2):147–168, June 1993.

[15] S. Srinivas. Towards a framework for tools used in investigation of large-scale parallel programs. Working notes.

[16] S. Srinivas and D. Gannon. Executing object-oriented parallel programs on high performance simulators. In *Submitted to International Conference on Simulation 1995.*

[17] S. Srinivas and D. Gannon. Interactive visualization and animation of parallel programs. In *Proceedings of Workshop on Parallel and Distributed Debugging*, May 1993. Available from ssriniva@cs.indiana.edu.

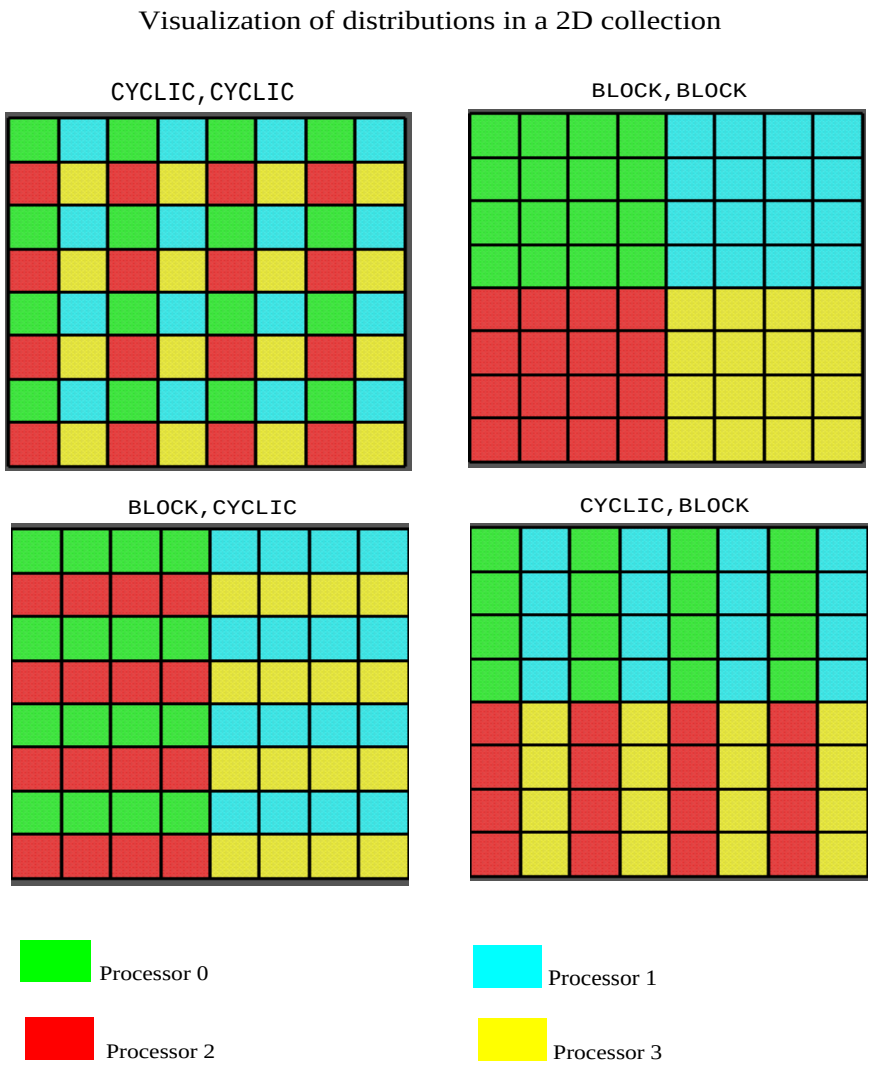

Figure 1: **VISUALIZING THE PARTITIONING OF A DISTRIBUTED COLLECTION: We first assign colors to the various processors and then visualize the two dimensional collection with its elements colored by the various processors. Looking at the diagonal elements in the four cases we see that the elements have been allocated to all the four processors in cases 2) and 3) but only to two processors in cases 1) and 4) and so we see that using the distribution of either case 2) or 3) would give the best performance.**

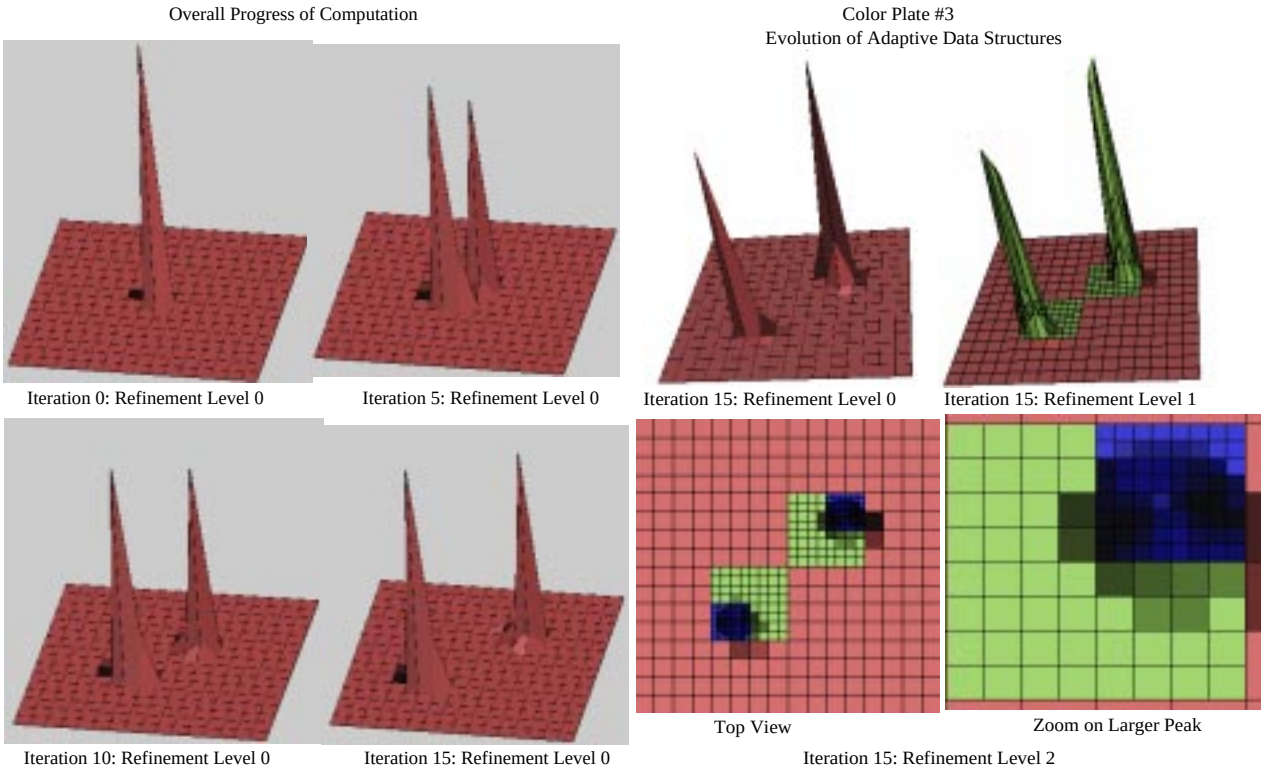

Figure 2: **SNAPSHOTS FROM THE ANIMATION SHOWING ADAPTIVE DATA STRUCTURE's logical time based evolution. Colors and transparency are used to show the finer resolution computation. The finer resolution occurs in the area of interest (identified by the peaks). The figure to the left shows the overall progress of computation in the program. The figure to the right shows the adaptiveness of the computation.**

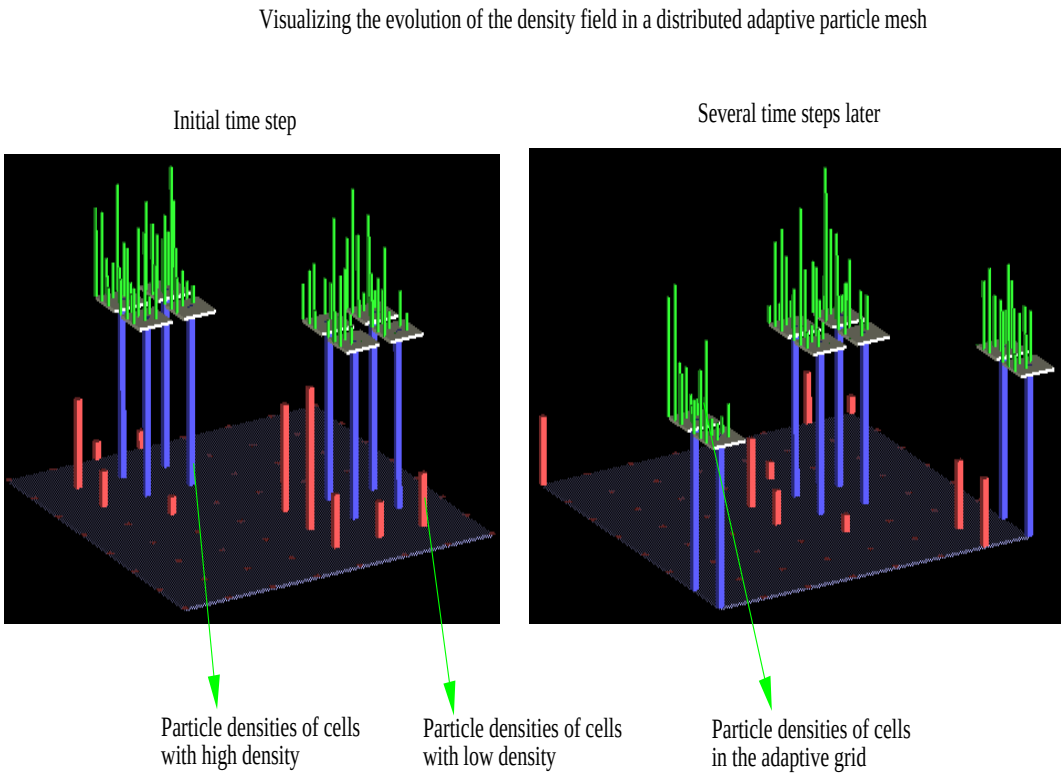

Figure 3: **SNAPSHOTS FROM THE ANIMATION OF PARTICLE DENSITIES IN AN ADAPTIVE N-body code. The particle densities over the grid are visualized as a 3D histogram and the histogram is also colored to show the areas of low and high densities as well as the densities in the refined region. The multiple visual cues of histogram height and histogram color make the animations more effective.**

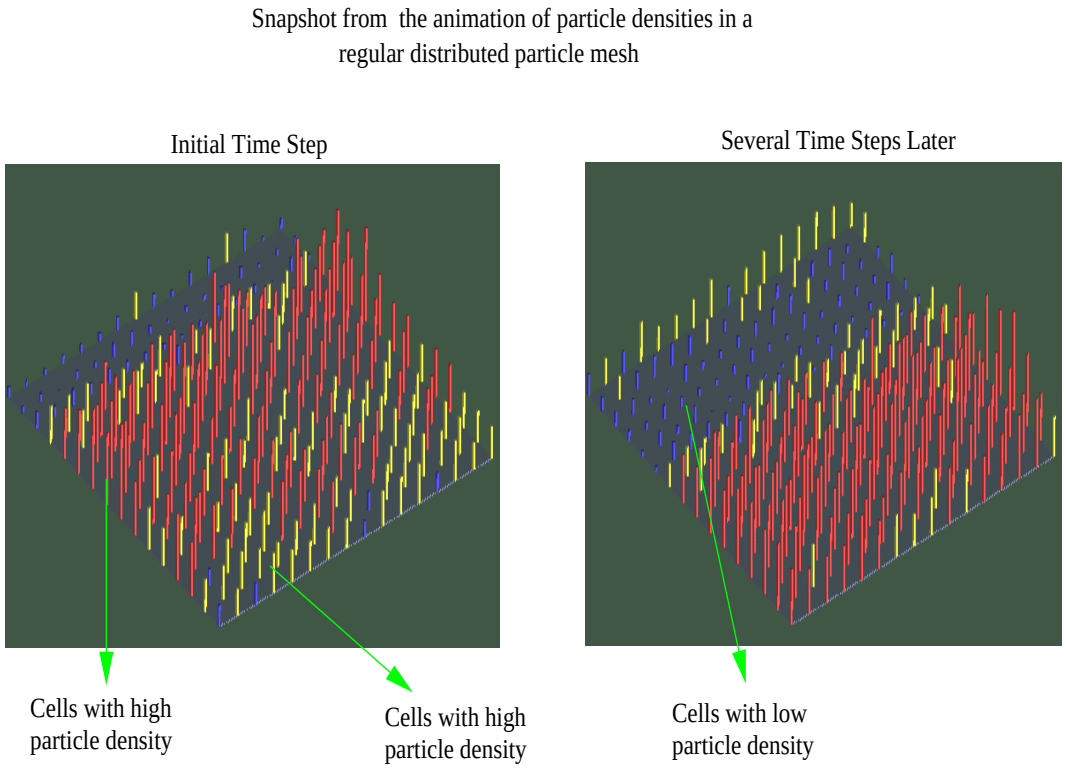

Figure 4: **SNAPSHOTS FROM THE ANIMATION OF PARTICLE DENSITIES IN A REGULAR N-body code. The particle densities over the grid are visualized as a 3D histogram and the histogram is also colored to show the areas of low, medium and high densities.**

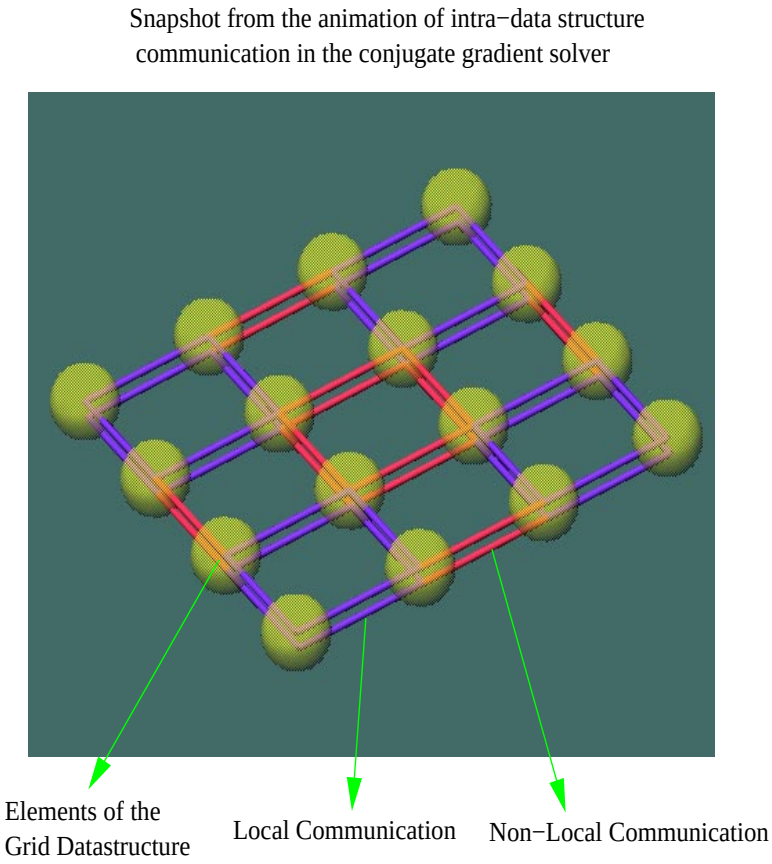

Figure 5: **SNAPSHOTS FROM THE INTRA DISTRIBUTED DATA STRUCTURE PATTERN ANIMATION showing the distributed data structure references that occur in the `Grid` distributed data structure which is distributed over 4 processors. The references have a NEWS structure i.e the sites in the grid reference all it's neighbors (in the North, East, West, and South directions) The elements are represented by spheres and the references are represented by lines. The lines are colored to illustrate local (blue) and non-local (red) references. The animation occurs in logical time and the references are faded away over time. The fading can be varied to see multiple time steps.**

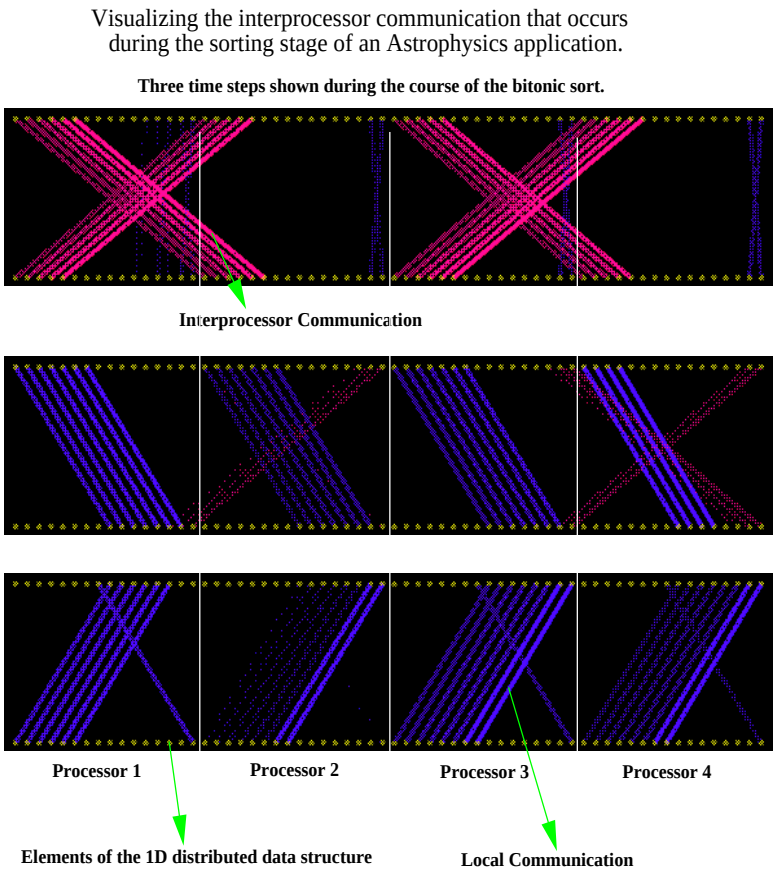

Figure 6: **SNAPSHOTS FROM THE REFERENCE PATTERN ANIMATION which shows the distributed data structure references that occur in the `ParticleList` distributed data structure during the parallel bitonic sort. The elements are represented by spheres and the references are represented by lines. The lines are colored to illustrate local (blue) and non-local (red) references. The animation occurs in logical time and the references are faded away over time. The fading can be varied to see multiple time steps.**