Conf-9409278--3

PNL-SA-25031

OBJECT-ORIENTED PARALLEL POLYGON RENDERING

R. W. Heiland

September 1994

Presented at the
GVIS '94 Graphics and Visualization
Conference
September 8, 1994
Richland, Washington

Prepared for
the U.S. Department of Energy
under Contract DE-AC06-76RLO 1830

Pacific Northwest Laboratory
Richland, Washington 99352

Object-Oriented Parallel Polygon Rendering

Randy Heiland
Computing & Information Sciences,
Pacific Northwest Laboratory,[1]
PO Box 999, MS K1-87
Richland, WA 99352
rw_heiland@pnl.gov

Abstract

Since many scientific datasets can be visualized using some polygonal representation, a polygon renderer has broad use for scientific visualization. With today's high performance computing applications producing very large datasets, a parallel polygon renderer is a necessary tool for keeping the compute-visualize cycle at a minimum. This paper presents a polygon renderer that combines the shared-memory and message-passing models of parallel programming. It uses the Global Arrays library, a shared-memory programming toolkit for distributed memory machines. The experience of using an object-oriented approach for software design and development is also discussed.

Introduction

The use of today's massively parallel or distributed networked multicomputers allows a scientist to expand the computational domain of a given application. This in turn results in larger static datasets that scientists want to visualize in a dynamic fashion -- rotating and zooming -- just as they have been acccustomed to for smaller datasets. Similarly, a computation may be temporal or iterative and may demand visual monitoring of large datasets at near real-time rates. Graphics algorithms running on a serial processor may simply be too slow for these type of applications. Hence, there is a need to develop parallel visualization algorithms.

The work presented here explores the possiblity of writing portable parallel graphics applications while avoiding the programming complexity normally present in parallel applications. It makes use of a shared-memory programming toolkit called Global Arrays.

This work focuses on one particular graphics algorithm, polygon rendering, because of its broad applicability to scientific datasets. A digital computer produces a discretized set of data that oftentimes lends itself quite naturally to some polygonal visualization. A common technique for visualizing a volume of scalar data is to display isosurfaces (surfaces of constant value, $f(x,y,z)=C$) within the volume. This technique works best when the range of data is smoothly varying throughout the three-dimensional (3-D) domain. One of the more popular algorithms for extracting a polygonal representation of an isosurface from a rectilinear 3-D grid of scalar data is known as 'marching cubes' [13,15]. This type of algorithm has also been parallelized [10]. If the data is defined over a non-rectilinear domain, other algorithms can be applied. For example, one can fit Delaunay tetrahedra to the data then perform a tetrahedral interpolation to obtain a polygonal isosurface. Specific applications often lead to their own specialized polygonal visualization techniques [9,12].

[1]The Pacific Northwest Laboratory is operated for the U.S. Department of Energy by Battelle Memorial Institute under Contract DE-AC06-76RLO 1830.

Polygonal representations are not only useful for visualizing functional datasets, but also for representing geometric primitives. Spheres and cylinders, commonly used in molecular graphics, are easily polygonalized. By taking advantage of symmetries inherent in these primitives, algorithms can generate and render them more efficiently.

Parallel Rendering

The rendering pipeline consists of mapping data between different coordinate spaces and eventually obtaining a raster image in pixel (screen) space [7]. When rendering a set of polygons, a hidden-surface algorithm and lighting model must be applied at some point in the pipeline. A standard polygon rendering pipeline consists of the following steps:

1. traverse the polygonal dataset
2. apply modeling transformations
3. trivially reject data outside the world window
4. apply viewing transformation
5. clip data outside viewport
6. rasterize data, performing hidden surface and illumination
7. display raster image (or write to disk)

Parallelizing this pipeline leads to a variety of algorithms. Whitman [18] offers a very good description and classification of parallel rendering methods. The most common high-level classification distinguishes object-based and image-based parallelism. Object-based parallelism simply refers to the partitioning of object space among all processors. Each node then performs the necessary geometry processing of steps 2-5 in the rendering pipeline. Image-based parallelism partitions the raster image plane among available nodes that perform the rasterizing, hidden-surface, and illumination in step 6.

Molnar et al. [14] discuss parallel rendering when these two approaches are combined, producing a system they call *fully parallel*, depicted schematically in Figure 1. They present a classification scheme which offers a clearer understanding and approach to solving the parallel rendering problem. The classification criterion is based on where the sort and redistribution of data occurs when mapping primitives in object space to pixels in screen space. To summarize their classification scheme, parallel rendering algorithms fall into one of three classes: sort-first, sort-middle, or sort-last.

In sort-first algorithms, each processor does just enough geometry processing to determine the region of the raster image that a primitive will belong. The primitive is then sent to the appropriate processor that was pre-assigned to perform both the geometry processing and rasterization for that region of the raster image. This type of algorithm is the least used of the three, primarily because of the possiblity for extreme load imbalancing.

Sort-middle algorithms are those that perform a sort and data redistribution at the obvious place -- between geometry processing and rasterization. In a sort-middle algorithm, each geometry processing node computes screen coordinates for each primitive that it has been assigned. It then determines the rasterizing node to which it will send a primitive's scanline information, based on the raster image's partition among nodes. The two sets of nodes responsible for geometry processing and rasterization can be disjoint or work in time-share fashion. One problem with sort-middle algorithms is that they are susceptible to load imbalancing of rasterizing nodes due to nonuniformly distributed primitives. However, it continues to be a very popular method. A good reference for implementing this type of algorithm is Crockett and Orloff [4].

The sort-last algorithms defer sorting until the very last stage. Primitives are arbitrarily distributed to available nodes.

Each node then performs both the geometry processing and rasterization of its assigned primitives, regardless of where they fall within the raster image. The final stage is to composite (sort and merge) the raster images from all nodes to form the final image. The sort-last classification is further subdivided into two types of implementations. Sort-last-sparse attempts to minimize communication by having nodes send only those pixels which were generated during rasterization. Sort-last-full, on the other hand, has each node send an entire raster image.

Global Arrays

Parallel programming applications are commonly divided into two models: message-passing and shared-memory. Applications that fall under the message-passing model typically require a significant amount of programming overhead and expertise. One consolation is that at least there exist MIMD programming libraries (Parasoft Express, MPI, PICL, PVM, TCGMSG) [6,11] that provide a great deal of portability among hardware platforms. Shared-memory applications, on the other hand, are easier to program; however, it is generally agreed that their implementations are not very portable.

A new programming model called Global Arrays (GA) [16] offers a partial solution to this dichotomy. GA is a suite of libraries, currently being developed at Pacific Northwest Laboratory (PNL), that provide a shared memory programming toolkit for distributed memory machines. The key concept of GA is that it provides a portable interface through which each process in a MIMD application can asynchronously access blocks of physically distributed data without the need for explicit cooperation with other processes. This significantly simplifies the programming complexity for parallel applications.

GA provides an application programmer with both SIMD and MIMD functionality. Currently, the application data must be defined as two-dimensional arrays. When these global arrays are created, they are distributed as disjoint blocks of data across processors. Each processor can then perform fast operations on its local subarray and, if necessary, still access data on other processors.

The GA library has been ported to a number of hardware platforms, including the Intel Delta and Paragon, the IBM SP-1, the Kendall Square KSR-2, and networks of Unix workstations.

Object-Oriented Software

Software engineering (SE), a term coined in 1968, is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software." The successes and failures of software engineering are described for the interested lay reader in [8]. Within the domain of engineering sciences, SE is an unsettled teenager among older, much more mature adults. This is both good and bad. SE is certainly an exciting, dynamic profession, ripe with a profusion of ideas and information, much of which is easily accessible via the Internet. However, it is a very frustrating experience to see one's own software become obsolete within a few years. This might happen for a number of reasons. Writing an application which relies heavily on a commercial software library that is replaced by another, significantly different library is a common example. Structuring software such that this type of disaster is minimized is just one goal of SE.

The genesis of programming languages and philosophies has brought us Fortran in 1956; structured programming in the 1970s; Smalltalk, an object-oriented language, in 1972; C in 1978; and C++ in the early 1980s. The programming philosophy enjoying much popularity in

the 1990s is the object-oriented philosophy. C++ is the object-oriented language of choice for many programmers. Blinn [1] has noted, somewhat humorously, that even mass market bookstores now carry a wide selection of books on C++ (and none on Fortran).

The key concept in C++ is that of a *class*. Classes provide a programmer with data-hiding, dynamic-typing, user-controlled memory management, and operator overloading [17]. The paradigm shift needed for programming in C++ is to think of an application as a set of interacting concepts (objects), rather than just data structures being modified by procedures. Defining an object such that it contains all relevant data and all relevant (member) functions enforces the notion of modularity. This in turn permits easier code maintenance.

The notion of parallel object-oriented languages is an intriguing idea. Assuming C++ is here to stay, parallel C++ languages will inevitably appear [2,3].

Implementation and Results

A scanline z-buffer triangle renderer with Gouraud shading has been implemented using Global Arrays. A triangle renderer was chosen because 1) any polygon can be tessellated into a set of triangles, and 2) triangle data structures are easier to manage than generic polygons. Following an earlier implementation in TCGMSG of a sort-middle rendering algorithm, the latest implementation is a sort-last algorithm. Currently, only the sort-last-full version is implemented.

The algorithm proceeds as follows:

1) Read in triangles and fill a (global array) buffer.
2) Create a global array, partitioning it (uniformly) among all nodes. Loop back to reading until the entire dataset resides in global arrays.
3) Each node performs geometry processing and rasterization on its local triangles (in the global arrays), resulting in a (local) raster image and accompanying z-buffer.
4) Composite the raster images from all nodes using the z-buffer information.

For the final compositing stage, a pairwise processor sort was implemented. This pairwise composition is depicted schematically in Figure 2. Such a scheme avoids load imbalancing of a more simplistic master-slave scheme whereby one master node would receive and composite the raster images from all other nodes. The current pairwise composition implementation is also rather simplistic in that a node is paired with its neighbor node. A better scheme might be to dynamically pair two nodes as they complete their rasterization step.

Although no timing results will be presented here (because the code is not yet optimized), speedup results for runs on the KSR-2, a shared-memory massively parallel computer, are shown in Figure 3. This chart plots (n,T(1)/T(n)) where n is the number of processors and T(1)/T(n) is the ratio between the rendering time using one processor, T(1), and the rendering time using n processors, T(n). For this plot, n=1,2,4,6,8, and 16. The number of triangles rendered on each run was about 500,000. Note that as the number of processors increases, the communication costs are greater for the compositing stage, thereby causing the algorithm to fall away from a linear scale.

Figure 4 shows the results of the sort-last-full algorithm on each of four processors and the resulting composite image. The data for this figure was taken from a TCGMSG parallel molecular dynamics simulation. Each molecule is represented as a triangulated sphere (randomly colored).

The previous sort-middle implementation in TCGMSG was written in C++ and ran on a network of Silicon Graphics workstations. The C++ classes included

the matrix and vector classes from Graphics Gems IV [5]. Additionally, classes were defined for certain geometric primitives. Because of C++'s operator overloading feature, one could then transform data as follows:

```
hydrogen = unit_sphere *
      H_scale + H_translate;
```

where 'hydrogen' and 'unit_sphere' are instances of C++ objects containing a list of triangles and 'H_scale' and 'H_translate' are scalar values.

During the second phase of this work, TCGMSG was replaced by the GA programming model and the KSR-2 was targeted as the machine of choice. These two choices led to the temporary dismissal of C++ as the implementation language. Because of GA's rather restrictive data types (2-D arrays), it does not fit into an object-oriented framework very naturally. However, the need to dismiss C++ for the second phase was realized when the C++ compiler stopped working on the KSR-2 for a period of time. Hence, the GA implementation of the sort-last renderer is currently in the C language.

Summary

The Global Arrays libraries should be a welcome addition for any programmer writing parallel applications. Using GA for this particular graphics application was relatively straightforward. Of course one always wants more features. Allowing for N-dimensional arrays, or better yet, structures of abstract data types, is just one request.

The fact that the C++ compiler was not working on the KSR-2 for some time suggests that this language has not matured as much as one would like. Nevertheless, it seems that C++ is here to stay for the forseeable future. Some of the features of C++ this novice object-oriented programmer finds most appealing are strong type-checking, data-hiding, and operator overloading. The first is desirable for minimizing programming errors and the latter two are desirable for programming elegance.

Planned future work is to:

1) optimize the current GA implementation (including testing sort-last-sparse)
2) perform timing tests on the KSR-2 as well as a network of Unix workstations
3) incorporate the parallel renderer into the Extensible Computational Chemistry Environment (ECCE) being developed at PNL
4) begin developing a parallel ray-casting volume renderer.

Acknowledgements

This work was supported by funds from the Environmental and Molecular Sciences Laboratory Construction Project at Pacific Northwest Laboratory (PNL) and from a Laboratory Directed Research and Development (LDRD) project through the Molecular Science Research Center at PNL. Pacific Northwest Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830. The author would like to thank Jarek Nieplocha for patiently answering many questions about Global Arrays and for providing feedback on this paper. Additional thanks go to Robert Harrison for answering TCGMSG questions, Tom Keller for answering C++ questions, and Don Jones for the encouragement to pursue this research.

References

[1] J.F. Blinn, replying to a letter to the editor, IEEE CG&A, July 1994, p. 4.

[2] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a Parallel C++ Runtime System for Scalable Parallel

Systems," Proceedings of Supercomputing '93, ACM SIGARCH and IEEE Computer Society, pp. 588-597.

[3] R. Chandra, A. Gupta, and J.L. Hennessy, "COOL: An Object-Based Language for Parallel Programming," Computer, August 1994, pp. 13-26.

[4] T.W. Crockett and T. Orloff, "A Parallel Rendering Algorithm for MIMD Architectures," Proc. Parallel Rendering Symposium, ACM Press, New York, 1993, pp.35-42.

[5] J.-F. Doue, "C++ Vector and Matrix Algebra Routines," *Graphics Gems IV*, P.S. Heckbert (ed.), Academic Press, 1994, pp. 534-557.

[6] C.C. Douglas, T.G. Mattson, and M.H. Schultz, "Parallel Programming Systems for Workstation Clusters," tech report tr975 on the Internet, ftp'd from casper.cs.yale.edu: pub/yale975.tar.Z.

[7] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990, pp. 229-281, 866-889.

[8] W.W. Gibbs, "Software's Chronic Crisis," Scientific American, September 1994, pp. 86-95.

[9] A. Gueziec and R. Hummel, "The Wrapper Algorithm: Surface Extraction and Simplification," IEEE Workshop on Biomedical Image Analysis 1994, pp. 204-213.

[10] C. Hansen and P. Hinker, "Massively parallel isosurface extraction," Proceedings of Visualization '92," pp. 77-83.

[11] R.J. Harrison, "Portable Tools and Applications for Parallel Computers," Int. J. Quant. Chem., 40, 1991, pp. 847-863.

[12] A. Koide, A. Doi, and K. Kajioka, "Polyhedral approximation approach to molecular orbital graphics," J. Molecular Graphics, 4, 1986, pp. 149-160.

[13] W.E. Lorensen and H.E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," ACM Siggraph, July 1987, pp. 163-169.

[14] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," IEEE CG&A, July 1994, pp. 23-32.

[15] G. Nielson and B. Hamann, "The asymptotic decider: Resolving the ambiguity in marching cubes," IEEE Conference on Visualization 1991, pp. 83-91.

[16] J. Nieplocha, R.J. Harrison, and R.J. Littlefield, "Global Arrays: A Portable 'Shared-Memory' Programming Model for Distributed Memory Computers," to appear in Proceedings of Supercomputing '94.

[17] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley. Reading, MA, 1991.

[19] S. Whitman, "Multiprocessor Methods for Computer Graphics Rendering," Jones and Bartlett, Boston, MA, 1992.

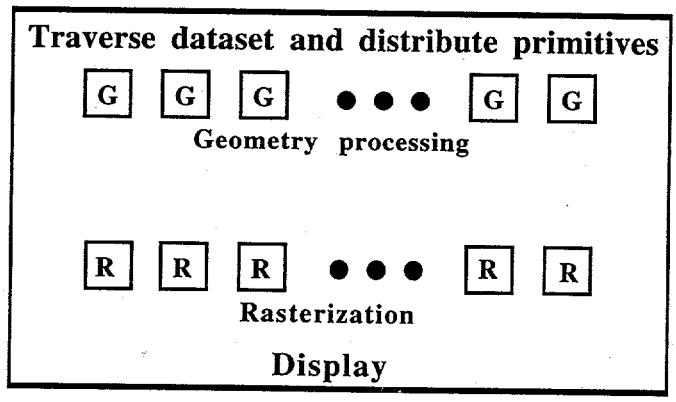

Figure 1. Fully parallel rendering.

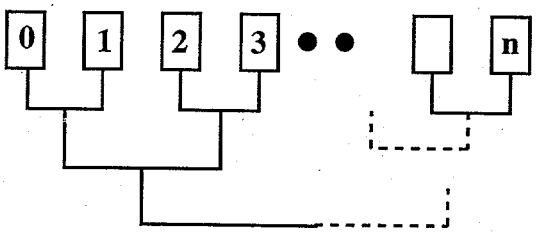

Figure 2. The pairwise composition scheme

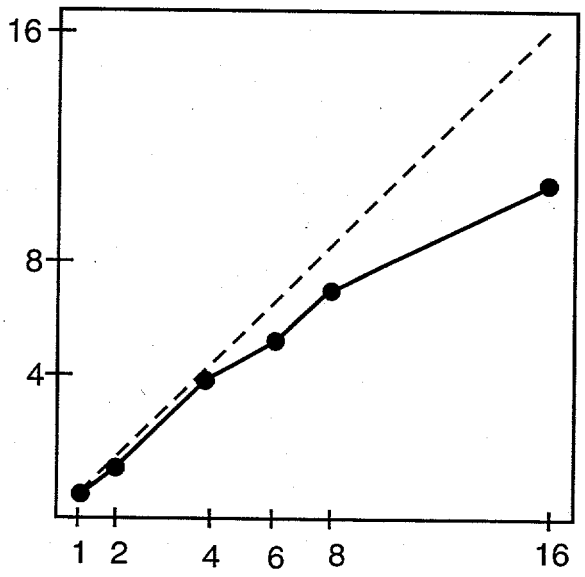

Figure 3. Speedup plot for the KSR-2.

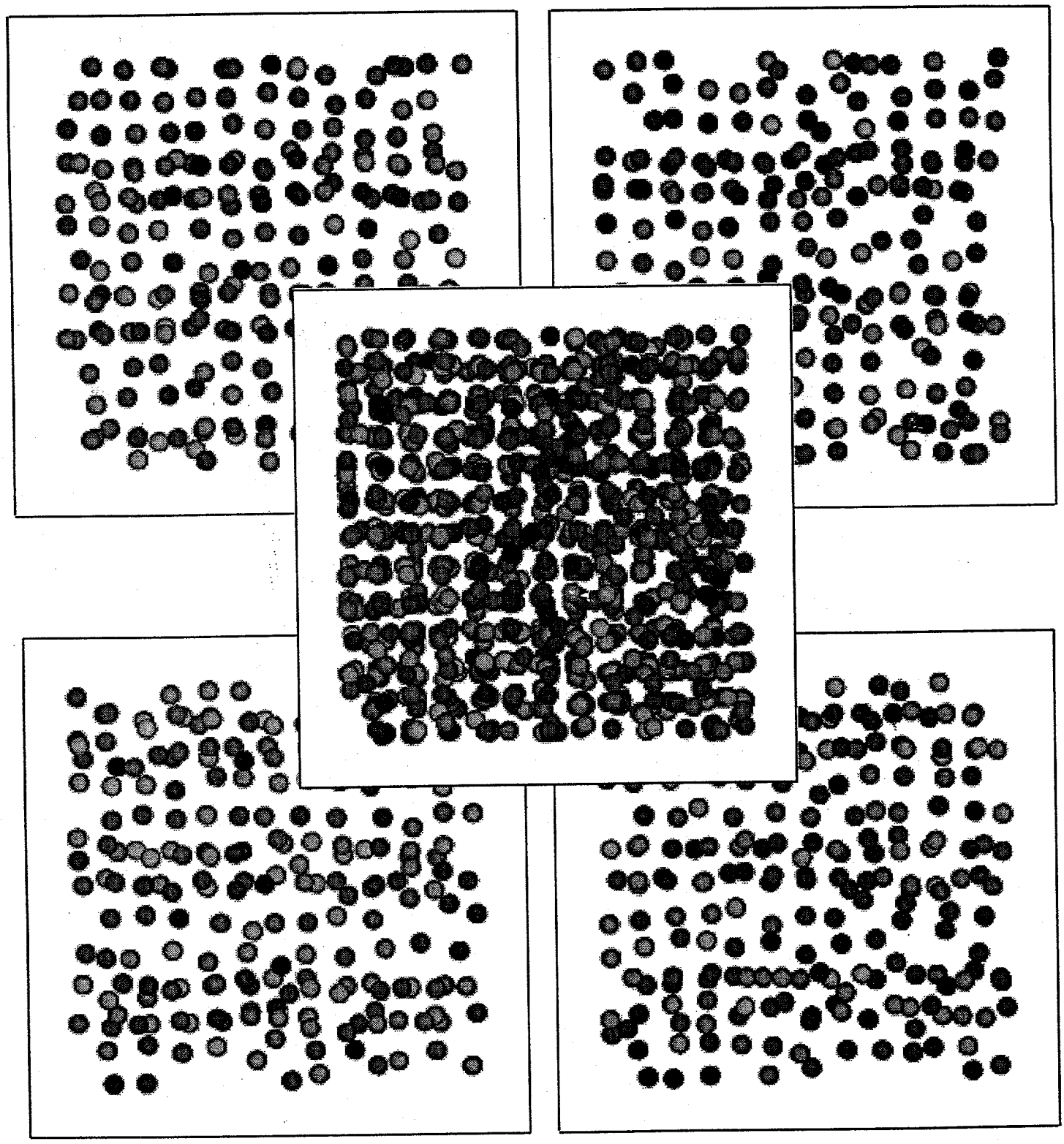

Figure 4. Sort-last-full images on each of four processors and the composite image.