# Divisive Parallel Clustering for Multiresolution Analysis

Bjoern Heckel[1], Bernd Hamann[2]

[1] 445 Francisco #204, San Francisco, CA 94133, USA
`bjoern_heckel@hotmail.com`
[2] Center for Image Processing and Integrated Computing (CIPIC)/
Department of Computer Science
University of California, Davis
2063 Engineering II
Davis, CA 95616-8562, USA
`hamann@cs.ucdavis.edu`
`http://graphics.cs.ucdavis.edu/`

**Abstract.** Clustering is a classical data analysis technique that is applied to a wide range of applications in the sciences and engineering. For very large data sets, the performance of a clustering algorithm becomes critical. Although clustering has been thoroughly studied over the last decades, little has been done on utilizing modern multi-processor machines to accelerate the analysis process. We propose a scalable clustering technique that benefits from existing parallel computers and networks of workstations. It supports the creation of multiresolution representations for very large geometric data sets. The output of the clustering process can be used for interactive data exploration, useful for view-dependent rendering, user-guided refinement, and progressive transmission.

## 1 Introduction

Data sets consisting of giga- or even terabytes of information have become increasingly common. The increasing capabilities of high-precision and high-level-of-detail engineering applications (e.g., computational fluid dynamics simulations) were made possible by advances in computer systems and computing methods. As a consequence, we have to deal with data sets of ever increasing sizes. Massive data sets also reside in corporate data warehouses, storing information about business and production processes. In digital libraries, millions of documents are accumulated and available to individuals. The complexity of such massive data collections has far surpassed our cognitive abilities to fully understand them as single entities. To gain some form of higher-level insight into massive data it is crucial to develop technology that supports interactive data exploration at different levels of resolution and abstraction.

Clustering is a classical data analysis technique that has been studied thoroughly during the last decades [5]. It has also been adopted as a standard technique in the emerging field of data mining [4]. Clustering is used in a variety of applications, ranging from fields like earthquake prediction, whale monitoring, marketing, psychology, biophysics, criminology, information retrieval, image processing, to phonetic taxonomy [2]. Its goal is to establish a set of groups such that objects assigned to the same group have certain similarities while they differ from objects in other groups. These groups are not known a priori and must be determined by examining the characteristics of the given objects. Often, one is interested in data partitions providing different levels of granularity – so-called *multiresolution levels*.

Applied to massive data sets, hierarchical clustering can be used for feature extraction, data summary, or creation of categories that allow interactive exploration. For example, applied to a database of customer records, it can provide insight into sales patterns that can be used for a focused marketing campaign. Used in the context of digital libraries, it produces a hierarchical index that assists in finding related documents and supports browsing by step-wise refinement. For very large data sets, however, creating a cluster hierarchy can require significant time. Particularly, with dynamic data sets, i.e., data sets whose object characteristics change over time or data sets where objects are inserted and deleted, the performance of the clustering process is pivotal. In addition, the output of an analysis process depends on various configuration parameters that describe how to interpret the data. When the clustering process is part of the analysis or exploration loop, a fast algorithm is necessary to guarantee an acceptable response time.

In the field of scientific visualization, clustering has been introduced in various form by various researchers, for multi-resolution analysis solving a variety of problems, ranging from feature extraction [9], and surface reconstruction [10],[11] to vector field compression [12],[15],[16] and mesh simplification [14]. Weber et al. use the multiresolution representation generated by a clustering process for a procedural grid generation method for scattered data approximation and visualization [13]. Since massive data sets have become increasingly common in scientific visualization applications, it is important to create an environment that supports processing and exploring such large data sets. Creating multiresolution representations from large data sets — as part of data preprocessing — enables interactive data exploration by supporting operations like view-dependent rendering, user-guided refinement, and progressive transmission.

We investigate the design and implementation of a parallel clustering approach (PaC) that is based on a divisive hierarchical paradigm. For a given data set and *dissimilarity measure* (or *distance function*) defined on the domain space, PaC creates either a set of multiresolution levels or a multiresolution hierarchy. We believe that our approach is helpful in applications such as parallel volume rendering [19] [21] [22] [23], parallel polygon rendering [24], and remote visualization [20].

In the following section, we discuss a sequential clustering approach. The parallelization of this approach is described in section three. In the fourth section, we explore the characteristics and performance of PaC.

## 2 Sequential Hierarchical Clustering

The input for our clustering process is a sequence of $n$-dimensional vectors $\{v_i\}$. Each vector is describing an *object* $o_i$. The elements of these vectors $a_j$ are considered *attributes* characterizing the represented object $o_i$. For the clustering process, a dissimilarity measure $D$ is used to measure the 'distance' between pairs of objects, and between objects and groups of objects, so-called *clusters*, and between pairs of clusters. Each cluster is characterized by an attribute vector, called *cluster center* or *cluster centroid*, that is most similar to the subset of all objects. The *cluster error* is the sum of the distance between the cluster center and all objects the cluster represents. The *global error* is the sum of all cluster errors. The goal of the clustering process is to find a set of partitions of different sizes such that for each partition the global error is minimized. When it is desired to create a multiresolution hierarchy an additional constraint requires the set of partitions to be hierarchically nested.

Our method for creating multiresolution representation is based on an incremental and divisive paradigm. Initially, all data objects are placed in one cluster $c_1$ that is recursively split until a termination criterion is met. This could be, when a certain number of clusters has been created, a certain global error criterion is met, or no cluster exceeds a certain *cluster error bound*. In each step of the algorithm, we replace the cluster with the highest internal error by new clusters. For two new clusters and $n$ objects that are assigned to the cluster to be split, there exist $2^{n-1}$ ways of dividing the cluster into two homogeneous child clusters. Even when the clustering criterion is guaranteed to produce convex clusters only, it is still computationally infeasible to examine all possible divisions. Therefore, a heuristic approach has to be used to calculate the centers of the child clusters. Performing reclassification only locally results in substantial performance gains in comparison to utilizing a global reclassification scheme, which is a NP-hard problem [5]. Effective approaches are, for example, successively removing objects from one cluster to build up the second cluster, selecting the most dissimilar pair of objects in the split cluster as 'seed points' for the child clusters, or – as we do – applying a local k-means algorithm. When it is not desired to create a multiresolution hierarchy, the global error can often be further reduced by reassigning objects in the neighborhood where a cluster has been replaced.

To determine the *local neighborhood* $N_i$ of a cluster $C_i$, a neighborhood graph $G$ (e.g., the Delaunay or Gabriel's graph, see [7]) is constructed incrementally using the cluster centers as vertices. A cluster $C_1$ is considered a *neighbor cluster* of a given cluster $C_2$, when the distance in the neighborhood graph does not exceed a *neighborhood* *threshold* $t_N$. The distance $Dis(C_1, C_2)$ in the neighborhood graph corresponds to the minimum number of edges to be traversed to reach $C_1$ from $C_2$. Alternatively, the neighborhood $N$ of a cluster can be defined as the $k$ closest neighbors, see [10].

After reclassification, clusters that have been affected by object reassignments are updated (i.e., the cluster center and cluster error are computed). The neighborhood graph is updated locally replacing a region R of G that is affected by the reclassification by an updated subgraph R'. Algorithm 1 summarizes our sequential clustering method:

```
While termination criterion is not met {
    Determine cluster C to be split; (1)
    Create w new clusters $C_i$ by splitting cluster C; (2)
    Determine Neighborhood $N_i$ of cluster $c_i$; (3)
    Reclassify data of clusters in $N_i$; (4)
    Update changed clusters; (5)
    Update neighborhood $N_i$; (6)
    Update priority queue; (7)
}
```

This approach yields multiple partitions of a data set, called *multiresolution levels*, for a varying number of clusters. Since generating a multiresolution hierarchy is a special case of computing multiresolution levels (the *neighborhood threshold* $t_N$ is zero), we will in the following only consider the more general approach.

Using a priority queue sorted by internal cluster error, the cluster to be split can be determined in constant time. We assume that new clusters can be created and initialized in constant time. The average number of neighbors of a cluster is expected to be constant for a fixed dimensionality of the data domain using, for example, a Delaunay graph or Gabriels's graph to define the local neighborhood. Therefore, the neighborhood of a cluster can be determined in constant time for an incrementally built graph. Since the expected number of neighbor clusters is expected to be constant, the complexity of reclassification as described above is expected to be proportional to the number of objects $l$ to be reclassified[1]. For a reclassification tree of depth $d_t$, the number of objects $l$ to be reclassified is expected to be proportional to

$$l \approx \frac{1}{2^{d_t}}. \tag{1}$$

Since the number of neighbors is bound by O(1), steps (5) and (6) require constant time. Updating the priority queue is done in O(log $c$) time, the number of clusters at the current iteration being $c$. The reclassification is by far the most time-consuming step, especially when clusters are large. Therefore, when designing a parallel clustering algorithm one should focus on exploiting parallelism during the reclassification stage.

## 4 Parallel Clustering - PaC

The parallelization of the algorithm described above is implemented in C++ utilizing MPI [17] as a low-level communication extension. Multiple instances of the parallel program are created on a set of processing nodes. Each instance, subsequently referred to as a *task*, has a unique identifier in the parallel application, called *rank*. Each task is assigned to a single processor on a network of workstations and/or a multiprocessor. The association of tasks and processors remains constant during the clustering process. The parallel application uses a master-worker model with a *master task* controlling the clustering process performed by several *worker tasks*, see [18]. The parallel clustering process is divided into three phases: In the first phase, the data is distributed over the tasks. In the second phase, all tasks work on computing one iteration of the clustering process (*cooperative parallel clustering*). In the third phase, each task independently computes one iteration (*concurrent parallel clustering*).

### 4.1 Data Distribution

During the first phase (initialization phase) the data set is read, and each vector is randomly assigned to exactly one of the worker tasks. Two different data distribution mechanisms have been implemented:

1. Data distribution using network transfer:
The data is read from an input file by the master process and sent to the worker processes. This approach allows using a network of workstations that do not share a file system. To reduce the communication overhead during the data distribution, the mas-

[1] Heckel et al. also split clusters in convex regions using principal component analysis (PCA), which also exhibits a linear time complexity with respect to the number of objects to be reclassified [12]. For more complex local optimization strategies – using, for example, simulated annealing – the complexity for the reclassification is expected to be higher.

ter task is caching a certain number of vectors in buckets that are transmitted to a worker task, when an overflow occurs.

2. Data distribution using a shared file system:
All worker tasks read the input file at the same time. Each read vector is used as a key for a pseudo-random hash function. This function determines which worker task should store a datum. Since all tasks use the same hash function and the same key a consistent assignment is guaranteed.

After distributing the data, each vector is stored in the associated task only. This association remains constant during the second phase. In the third phase, vectors might be reassigned and transferred to other tasks. However, each vector is assigned to exactly one task at all times of the clustering process.

The relative performance of the processing nodes can be different, e.g., when using a heterogeneous network of workstations as run-time environment. To minimize idle time of tasks on fast processors at synchronization points during the cooperative clustering phase and to increase overall performance, the number of vectors assigned to a task is chosen to be proportional to the relative performance of the processor a task is running on. The relative performance of a processing node is determined by using the sequential clustering program as a benchmark.

## 4.2 Cooperative Parallel Clustering

During the *cooperative parallel clustering phase* all worker tasks are helping to compute one iteration of the clustering process. The objects assigned to each cluster are distributed over all tasks. During each iteration, all worker tasks simultaneously split the same cluster with the highest priority and insert a new cluster. The steps (1) to (3) of the iteration – that are identical to the sequential clustering – are carried out simultaneously by all tasks. The following local reclassification (step 4) is by far the most time-consuming and is conducted cooperatively. Each task generates a list of vectors in the region that is affected by reclassification. Since the data is randomly distributed over all tasks, each task stores a subset of the region. Then, each task assigns each locally stored vector to its closest cluster. Based on the resulting assignment of the local data, each task is recalculating the cluster centers of the reclassified region. Each task broadcasts a summary of the partial reclassification to the other tasks (step a in Algorithm 2). After receiving this information from all worker tasks (step b in Algorithm 2) each task updates the clusters in the reclassified region (step 5).[2] Finally, all tasks simultaneously update the priority queue (step 6) and the neighborhood graph (step 7).

[2] The local reclassification scheme can be applied iteratively in a distributed k-means fashion.

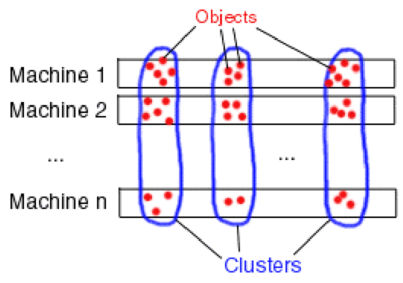


**Fig. 1.** Data distribution during phase 2, cooperative parallel clustering

While the classification process progresses, the average cluster size decreases. As a result, the amount of work per iteration decreases, and the ratio of communication overhead and execution time increases. When the average cluster size does no longer exceed some threshold limit $\tau_{\text{Switch}}$, the classification process enters the third phase, which utilizes a more efficient strategy for large sets of small clusters.

```
For all tasks: While termination criterion is not met {
    Determine cluster C to be split; (1)
    Create w new clusters $C_i$ by splitting cluster C; (2)
    Determine neighborhood $N_i$ of cluster $c_I$; (3)
    Locally reclassify data of clusters in $N_i$; (4)
    Broadcast local cluster information; (a)
    Receive cluster information from other tasks; (b)
    Update changed clusters; (5)
    Update neighborhood $N_i$; (6)
    Update priority queue; (7)
}
```

### 4.3 Concurrent Parallel Clustering

In the third phase, each task is concurrently computing one iteration of the clustering loop. Each cluster with its associated objects is assigned to exactly one task. At the end of the second phase, the domain space is partitioned in different regions, each of them being assigned to a particular worker task. In the third phase, all clusters in one region are initially hosted by the corresponding task. Since at the end of phase 2 clusters are distributed over all tasks, cluster fragments have to be transferred to a single task whenever it must be considered for reclassification by that task.

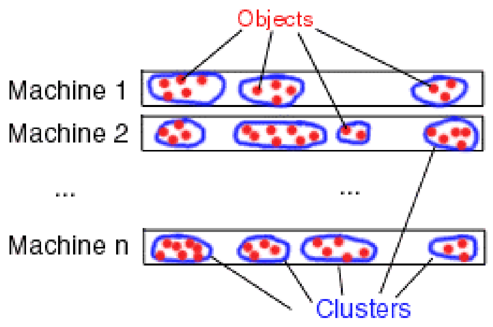


**Fig. 2.** Data distribution during phase 3, concurrent parallel clustering

To avoid inconsistencies and ensure a deterministic behavior, a task can only split a cluster provided it has exclusive access to its neighbor clusters. When clusters in the local neighborhood are not present in the worker task, it sends out a request to the master task. The master tasks determines the host of the requested clusters and forwards the requests to the corresponding worker tasks, which transfer the ordered clusters directly to the requesting task. For a sufficiently large value of $\tau_{Switch}$, the number of *request collisions* can be expected to be low. Therefore, the tasks can work relatively independent with very low communication cost. After reclassification, a worker task serves the outstanding cluster requests and transmits these clusters to the requesting task. By checking for requests directly before determining a split candidate, the interference of task is decreased. However, it is possible that a split candidate *x* is sent from task *l* to a remote task *r* when it is member of the split candidate's neighborhood in *r*. Since *x*'s neighbor's might change after reclassification of *r*, the list of requested clusters in *l* has to change as well. Clearly, *l* cannot proceed unless *x* has been transferred back to *l*. When task *l* receives the previously determined split candidate *x*, it updates its request list to *x*'s neighbors. For consistency reasons, cluster *x* is not permitted to be split by task *r*, or any other task different from *l*. After it has been identified as a split candidate by *i*, its 'dirty flag' is set. Dirty clusters are simply ignored during the determination of split candidates. Once *x* is split by *l* its dirty flag is cleared, and it might be split by a remote task after it has been transferred.

```
For all worker tasks: While termination criterion is
not met {
    Determine cluster C to be split; (1)
    Create w new clusters $C_i$ by splitting cluster C; (2)
    Determine neighborhood $N_i$ of cluster $c_I$; (3)
    **Send requests to master task, if not locally present; (c)**
    **While waiting to receive clusters:**
        **Check for requests and transfer clusters; (d)**
    Reclassify data of clusters in $N_i$; (4)
    Update changed clusters; (5)
    Update neighborhood $N_i$; (6)
    Update priority queue; (7)
    **Check for requests and transfer clusters; (f)**
}
```

## 5 Application

To test the performance of our parallel clustering algorithm we have applied it to a set of unorganized points in 3D space. In this case, clustering is used to reconstruct surfaces at different level of resolution, each one described by a triangular mesh, see Figure 3. A detailed discussion of this technique and the application domain is provided in [10].

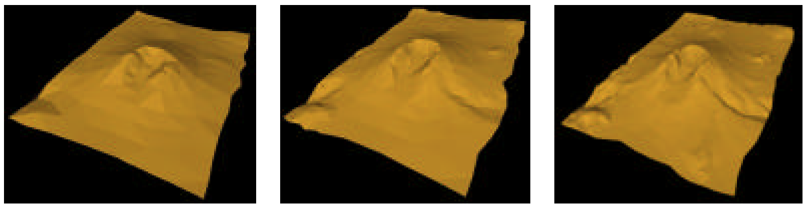


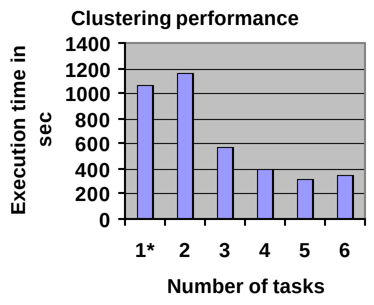
**Fig. 3.** Three resolution levels of Mt St. Helens data set

Data distribution costs:
Using the network transfer via the master task, the timing for data distribution is linear with respect to the size of the input. For very large data sets, this distribution mechanism requires substantial time and is about twenty times slower than the initialization in the sequential clustering program for our system configuration. This distribution mechanism is therefore a performance bottleneck that highly impacts the performance of the overall parallel clustering tool. When all work tasks concurrently read from the input file and assign the data using a pseudo-random hash function as described above, the data distribution shows a performance similar to the sequential clustering algorithm.

Message size:
Using network transfer via the master task for the data distribution, the message length during the first phase depends on the chosen size of the buckets and the dimensionality of the domain. In the second phase, the message length depends on the dimensionality of the domain and the number of neighbors, which is a function of dimensionality. Our experiments have shown that the message size in phase two varies between about 200 and 400 bytes, with an average of about 260 bytes, for a 3D real space. In the third phase, the message length depends on the size of the cluster transferred between tasks, which is bounded by $\tau_{Switch}$ and is decreasing while the classification is progressing.

Performance:
We have applied PaC to a topographic data set that consists of roughly 2 million 3D points. Each point describes the height z at a certain location (x,y). Our program is used to reconstruct the relief of the topography at different levels of detail. The size of the input file is 47.3 MB, and 100 clusters are computed. PaC was executed on an SGI Onyx with 4 processors and 512 MB main memory. The clustering time is measured for the parallel application and the sequential version of the clustering program (*1**), see . The speed-up factor is computed by dividing the execution time of the sequential algorithm by the execution time of PaC. With two tasks and only one worker task, PaC exhibits a performance that is 8% slower than sequential program. Utilizing more processors yields an almost linear speed-up.

With four tasks, three processors are fully utilized, while one processor, running the master task, has a low utilization. Adding one task yields a higher performance, but increases the overhead for context switching, since more tasks than processors are used. Increasing the number of task results in a performance loss for an increased overhead. The state diagram in Figure 5 shows that randomly distributing the data yields a good workload balance for the multiprocessor machine.



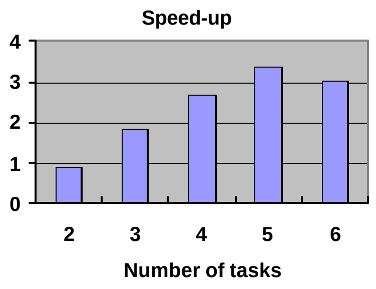


**Fig. 4.** Performance and speed-up in dependency of number of tasks

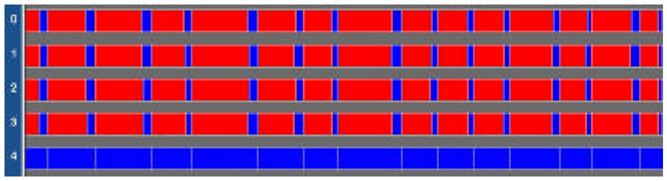
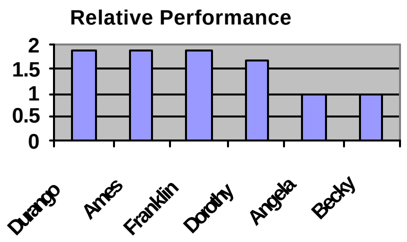


**Fig. 5.** States of 5 tasks on SGI Onyx 2 - red: splt and reclassify, blue: communicate and update clusters

<u>Workload balancing:</u>
When using a network of workstations instead of a single multiprocessor machine, the relative performance of the used machines may vary. In this case, using a random distribution may degrade the performance of the existing hardware. In our experiment, we use six machines with varying relative performance, see Figure 6. The relative performance is measured by running the sequential clustering algorithm on the participat-



ing machines as a benchmark.

**Fig. 6.** Relative performance of the six used workstations

The network of workstations determines 100 clusters for a data set that consists of 100,000 3D points. The execution time is about 36 seconds. The state diagram in Figure

7 reveals that the first three worker tasks (tasks 0 through 3) spend much time on the update phase waiting for task number 4, which runs on the slowest machine (Angela) to reach the synchronization point.

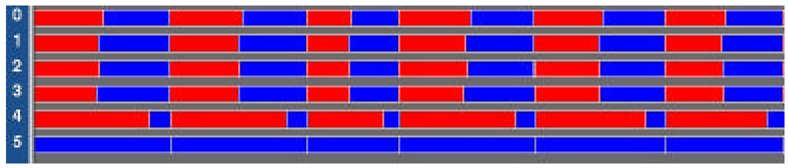
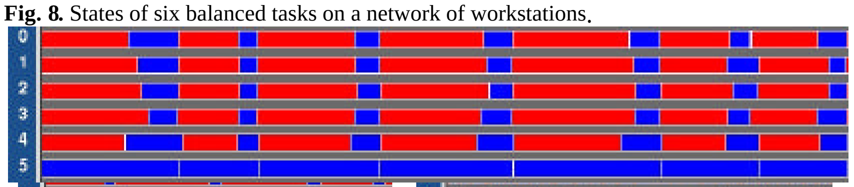


**Fig. 7.** States of six unbalanced tasks on a network of workstations.

Weighting the workload by the relative performance reduces the execution time to 26 seconds. The state diagram shown in Figure 8 documents that considering the relative performance results in a better processor utilization. The clustering time for the sequential program is 110 seconds on Durango, which is the fastest of the used machines. Considering that only five workers are used, this speed-up is almost ideal. The low utilization on the master task's machine also indicates that many more tasks could be used to cluster a sufficiently large data set using the proposed master-worker architecture.

**Fig. 8.** States of six balanced tasks on a network of workstations.



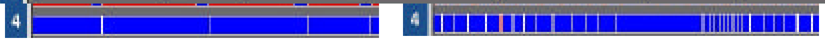


**Fig. 9.** States of five tasks at the beginning and the end of the clustering process

**Granularity:**

While the clustering in phase two progresses, the average cluster size decreases. This results in an increasing overhead/computation ratio in phase two, see Figure 9. In contrast, during phase three the overhead/computation ratio decreases over time, since the request collision and cluster transfer frequency decrease with an increasing number of clusters. To get the best of both approaches we start with cooperative clustering and switch to concurrent clustering after a sufficiently large number of clusters have been generated. The overhead for transferring the partial cluster fragments is spread out over time, since cluster fragments are only transferred on demand. The optimal value for the parameter $\tau_{Switch}$ must be determined empirically and is dependent on the system configuration (e.g., number of processing nodes, network, type of machines), the characteristics of the data set, and the error metric. For our system configuration, we have determined that the data set and $\tau_{Switch}$ have to be fairly large to gain considerable speed-ups compared to using phase two only.

## 6 Conclusion

We have presented a new method that enables to create cluster hierarchies utilizing modern multiprocessors and network of workstations. Our evaluation has demonstrated that our method is scalable and can be used to analyze very large data sets benefiting from existing multiprocessor machines. Our approach should be extremely beneficial for many applications, where the evaluation of multiresolution representations of very large data sets is required.

## 7 Acknowledgements

This work was supported by the IBM Almaden Research Center, San Jose, California; by the National Science Foundation under contract ACI 9624034 (CAREER Award); the Office of Naval Research under contract N00014-97-1-0222; the Army Research Office under contract ARO 36598-MA-RIP; the NASA Ames Research Center through an NRA award under contract NAG2-1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the North Atlantic Treaty Organization under contract CRG.971628 awarded to the University of California, Davis. We also acknowledge the support of ALSTOM Schilling Robotics, Davis, California and Silicon Graphics, Inc. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

## 8 References

1. A. D. Gordan, “Hierarchical Classification”, in: P. Arabie, L. J. Hubert, G. De Soete, eds., *Clustering and Classification*, World Scientific Publ., River Edge, 1996.
2. P. Arabie, Lawrence J. Hubbert, “An Overview of Combinatorial Data Analysis”, in: P. Arabie, L. J. Hubert, G. De Soete, eds., *Clustering and Classification*, World Scientific Publ., River Edge, 1996.
3. R. Franke, G. M. Nielson, “Scattered Data Interpolation and Applications: A Tutorial and Survey”, in: Hagen, H. and Roller D., eds., *Geometric Modeling*, Springer-Verlag, New York, 1991.
4. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, “The KDD Process for Extracting Useful Knowledge from Volumes of Data”, in: *Communications of the ACM*, 39(11), pp. 27-34, November 1996.

5. P. Arabie, L. J. Hubert, G. De Soete, "Clustering and Classification", World Scientific Publ., River Edge, 1996.
6. M. de Berg, M. van Kreveld, O. Overmars, O. Schwarzkopf, “Computational Geometry - Algorithms and Applications”, Springer-Verlag, Berlin, 1997.
7. A. Okabe, B. Boots, K. Sugihara, “Spatial Tessellatio ns”, John Wiley & Sons, Chic hester, 1992.
8. R. Cypher, A. Ho, S. Konstantinidou, P. Messina, "A Quantitative Study of Parallel Scientific Applications with Explicit Communications", Journal of Supercomputing, 10(1):5-24, March 1996.
9. B. Heckel, and B. Hamann. Visualization of cluster hierarchies. In: R. F. Erbacher and A. Pang, eds., Visual Data Exploration and Analysis V, Proc. SPIE Vol. 3298, SPIE – The International Society for Optical Engineering, Bellingham, Washin g-ton, pp. 162-171, January 1998.
10. B. Heckel, A. Uva and B. Hamann. Highly efficient generation of hierarchical surface models. In: C. M. Wittenbrink and A. Varshney, eds., Proceedings of IEEE Visualization ’98 – Hot Topics, IEEE Computer Society Press, Los Alamitos, CA, pp. 50-55, October 1998.
11. B. Heckel, A. Uva, B. Hamann and Joy. Surface Reconstruction using adaptive clustering methods. In: G. Brunnett, H. Bieri and G. Farin, eds., “Geometric Modelling: Dagstuhl 1999,” Computing Suppl. 14, Springer Verlag, pp. 199-218, 2001.
12. B. Heckel, G. Weber, K. Joy, and B. Hamann. Multiresolution analysis of vector fields. In: D. S. Ebert, M. Gross, B. Hamann, eds., Proceedings of IEEE Visualization ’99, IEEE Computer Society Press, Los Alamitos, CA, pp. 19-25, October 1999.
13. G. Weber, B. Heckel, K. Joy, and B. Hamann. Procedural generation of triangulation-based visualizations. To appear in: Proceedings of Visualization ’99 (Hot Topics), A. Varshney, C. M. Wittenbrink, H. Hagen, Eds., IEEE Computer Society Press, Los Alamitos, CA, Oct 1999.
14. B. Heckel. Clustering-based Multiresolution analysis for Scientific Visualization. Ph.D. thesis, University of California, Davis, 2000.
15. H. Garcke, T. Preuer, M. Rumpf, A. Telea, U. Weikard, J. van Wijk. Continuous Clustering Method for Vector Fields. In: C. Hansen, C. Johnson, S. Bryson, eds., Proceedings of IEEE Visualization 2000, IEEE Computer Society Press, Los Alamitos, CA, pp. 351-358, October 2000.
16. A. C. Telea and J. J. van Wijk. Simplified representation of vector fields. In: D. S. Ebert, M. Gross, B. Hamann, eds., Proceedings of IEEE Visualization ’99, IEEE Computer Society Press, Los Alamitos, CA, pp. 34-42, October 1999.
17. W. Gropp, Ewing Lusk, and Anthony Skjellum. Using Mpi : Portable Parallel Programming With the Message-Passing Interface. MIT Press, Scientific and Engineering Computation Series, 1994.
18. Ian T. Foster. Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering. Addison-Wesley, 1994.
19. P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. In: Cox, M., Uselton, S. P. and Wittenbrink, C. M., eds., Proc. 1995 Parallel Rendering Symposium, Atlanta, GA, October 30-31, 1995, pp. 15–22.

20. P. P. Li, W. H. Duquette, D. W. Curkendall. Remote interactive visualization and analysis (RIVA) using parallel supercomputers. In: M. Cox, S. P. Uselton, and C. M. Wittenbrink, eds., Proc. 1995 Parallel Rendering Symposium, Atlanta, GA, October 30-31, 1995, pp. 71–78.
21. K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In: Cox, M., Uselton, S. P. and Wittenbrink, C. M., eds., Proc. 1995 Parallel Rendering Symposium, Atlanta, GA, October 30-31, 1995, pp. 23--30.
22. K.-L. Ma, J. S. Painter, C. D. Hansen, M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In: Crockett, T., Hansen, C. and Whitman, S., eds., Proc. 1993 Parallel Rendering Symposium, San Jose, CA, October 25-26, 1993, pp. 15–22.
23. U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multiprocessors. In: Crockett, T., Hansen, C. and Whitman, S., eds., Proc. 1993 Parallel Rendering Symposium, San Jose, CA, October 25-26, 1993, pp. 97--104.
24. S. Whitman. A load-balanced SIMD polygon renderer. In: M. Cox, S. P. Uselton, and C. M. Wittenbrink, eds., Proc. 1995 Parallel Rendering Symposium, Atlanta, GA, October 30-31, 1995, pp. 63--69.