A New Paradigm for Scientific Visualization

Elise deDoncker Karlis Kaugars*

Abstract

We outline the design of a scientific visualization system based on a data card display invariant to provide information on the creation and processing of tasks in distributed task pool algorithms. Information aggregates over the tasks and processes are incorporated via non-classical histogram representations showing the distribution of the data within individual bins and by user-controllable tree displays.

Keywords: Visualization; Interactive Visual Analysis; Distributed Task Pool; Algorithm Convergence

1 Introduction

We present a new paradigm for scientific visualization which can be incorporated in parallel/ distributed code to track computational characteristics with respect to algorithm convergence and load distribution. It is our aim to provide parallel program visualization not only to the expert but also to the user who is not necessarily a parallel program expert.

To this end we are re-defining the preliminary visualization tool embedded in ParInt [7], which allows for the dynamic presentation of graphs of estimated error and load on the processors as a function of time, as well as of the global error and load distribution. Our design is based on the data card, a metaphorical digital instantiation of the index card, to store the basic information related to each task. Digital cards can be scaled or stretched to provide variant displays of information and can support both semantic zooming and non-linear scaling. Since the data cards are scalable from single pixel to full display window size, representing large amounts of data becomes tractable. Apart from the individual data cards, which provide the user with a consistent view of the details of a computation, support is also needed for dynamic information aggregates. The display invariant and information aggregates are discussed in Section 2. Section 3 outlines the architecture of the system and Conclusions are given in Section 4.

There are many well-known systems for visualizing and debugging distributed parallel program performance - among them ParaGraph, SIEVE, MINGraph, Monsoon, the TAU environment and Ariane. Several parallel programming systems such as XPVM, AIMS include programmer-oriented visualizations, and several collections of papers have been published [4, 9, 11] to provide users and practitioners an overview of current research.

Following the approach of interactive visual exploration [5], we focus on enabling the potential users of parallel programs to gain an intuitive understanding of program behavior and result validity without actually examining the code. In contrast to interactive visual exploration, we would like to extend this intuitive understanding to individuals who are not necessarily domain experts.

Although not exclusively, we target algorithms that are automatic, adaptive and distributed. The term *automatic* means that the computations terminate if the user-specified accuracy or error tolerance is attained. In an *adaptive* algorithm, the course of the computations depends on the problem instance at hand, which may be achieved by partitioning the domain of the problem. At each algorithm iteration, the most "important" task is selected for further processing at each step. As a result, successive selections are driven by the importance of the tasks, resulting in a concentration of effort where the problem is difficult. The associated priority queue can be implemented by a heap on the subregions, keyed with the importance measure. These problems thus fit a task pool model.

For example, most algorithms in adaptive numerical integration are governed by the meta-algorithm on the left of Figure 1, where the estimated error associated with each region is used as the importance measure. As another example, progressive refinement radiosity methods compute the global illumination of a scene. The scene is subdivided in patches and the patch with the most energy is selected next to distribute ("shoot") its energy into the environment. The operations of the right part of Figure 1 are iterated [2].

*Department of Computer Science, Western Michigan University, Kalamazoo MI 49008; {elise,kkaugars}@cs.wmich.edu; Presenter: Elise deDoncker; Tel. 616 387-5645; Fax 616 387-3999; For PDPTA'99

adaptive integration algorithm

```
initialize;
while(acceptance criteria are not satisfied)
  /* select next region */
  get_regions(region_list);
  /* subdivide region and integrate over subregions */
  partition_region();
  /* add new regions to the pool */
  put_regions(region_list);
  /* update results and flag abnormal conditions */
  update_results();
  update_globals();
```

task pool radiosity algorithm

```
Select (source) surface with largest unshot radiosity;
For all target surfaces (≠ source) {
    Subdivide target surface;
    Compute form-factors between
        source and obtained subsurfaces;
    Obtain radiosities and unshot radiosities
        of target surface and its subsurfaces;
    If radiosity gradients between subsurfaces
    are large, then
        replace target surface by its
        subsurfaces in the pool;
}
```

Figure 1: Adaptive integration and task pool radiosity algorithms.

Even in discrete optimization, best-first search branch-and-bound provides an example which fits the task pool model [8], where the solution space is subdivided and the task selection is according to least-lower-bound-next.

These algorithms can be *distributed* by partitioning the task pool among p available (worker) processes. This may be implemented by keeping a local heap on the tasks in each worker [1] and using a type of load balancing to provide near-idle processors with new work. Our visualization technique is intended to allow the tracking of computational characteristics in target applications, with respect to the algorithm convergence and the distribution of the remaining work.

2 Display Invariant and Information Aggregates

We utilize data cards to represent the fundamental data associated with each task. For example, a task may consist of a hyper-rectangular region determined by the coordinates of its center and the lengths of its half axes in each coordinate direction, with the approximations obtained for the integral and error over the region [1]. Accordingly, a datacard representing a region includes display areas for the region information and for the estimated error over the region.

In contrast to the physical instantiation of index cards, digital cards can be scaled or stretched to provide variant displays of information and can support both semantic zooming [10] and non-linear scaling [3]. The display is determined by user preference and available display space. If a card has very little space for display (as little as a single pixel) our initial design calls for a focus on the estimated error (or importance measure in a more general context), which can be represented by a color-scaled area. At intermediate size, the card can display this value as a colored bar proportional to the task's relative contribution to the pool. As more space becomes available, we can add graphical and/or numeric representations. The left part of Figure 2 shows two simple graphical forms of regions: the top card represents a two-dimensional region and the (4) components of its error (vector); the bottom card depicts a five-dimensional region representation.

Information aggregates can be broadly classified as either state or history aggregates. State aggregates must show a global view of the computation at an instant of time and generally must aggregate over either the processes or tasks of the computation. History aggregates need to summarize the state of computation over time. The current state of computation can be viewed at multiple levels of detail over processes, importance levels or task hierarchies. The interface design supports all three types of aggregates without discarding the data card focus, through the design and development of a process data card. The user can be presented with multiple process views by dividing the available display area into a grid of cells, one per process. Each cell contains the data card representing the process and via semantic zooming and user preferences may also contain graphical representations of the tasks maintained by this process.

Depending on the computational model chosen, processing may terminate when we have achieved a small error distributed across many tasks or the absence of active tasks. In this case, the user can gain significant information about the progress of computation by examining a histogram display of error (importance) values, where each bin of the histogram contains not just a numeric count of the values in the bin but rather a set of cards sorted by this value. Selecting one of the bins of the histogram leads to splitting the bin into separate bins. For example, the right side of Figure 2 displays a histogram of random data with a zoom-in on bin 6.

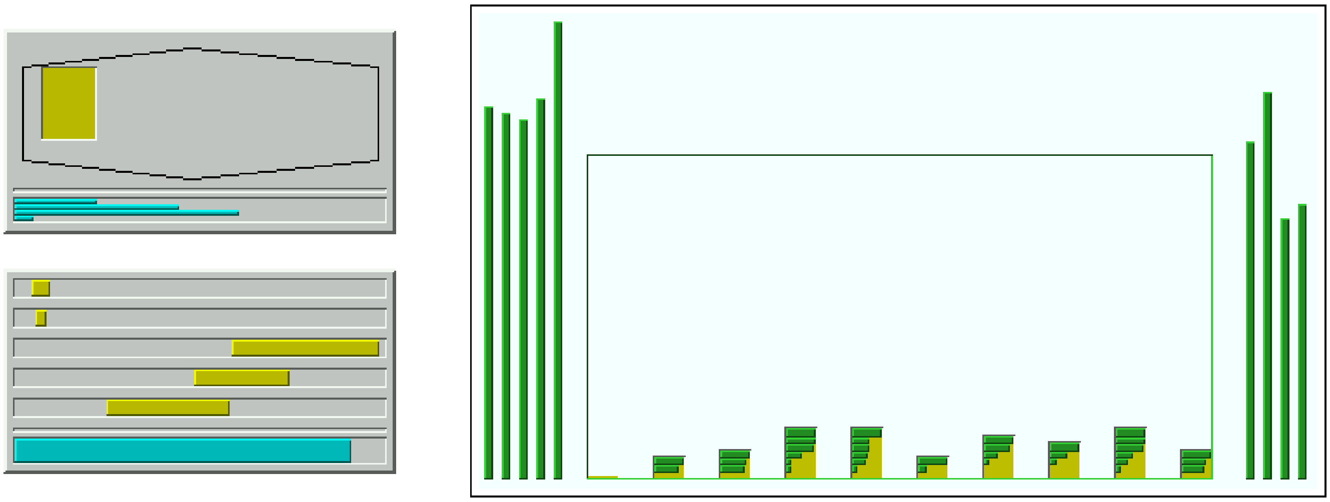

Figure 2: Graphical representation of regions and an error agregate.

Computational are commonly generated by partitioning tasks into subtasks in order to improve the results of the computation. The original task therefore forms the root of the tree and the final results of the computation are based on the leaf nodes. Thus another important indicator of the state of computation is the current task tree, which can be displayed using OTree [6] to support data card labels of nodes. The OTree display manages the exponential growth of tree structures through a combination of scaling and omission to provide a user-controllable tree display. The display allows the user to interactively select subtrees to examine in detail and these selected subtrees are shown to the user at full scale. The rest of the tree is shown at reduced scale and subtrees which do not fit on the display are pruned. This allows the user to easily traverse even very large trees and compare the task generation in various parts of the computation; or to compare trees generated by different algorithms.

Aggregate information over time is at least as important as computational state displays. Although the tree display implicitly encodes history information in the depth of a tree, it does not display linear time but rather the history of a task and the creation of subtasks. The visualization system of [7] includes a set of visualization functions to display the state of computation over time. These can be adapted to use the data card invariant.

3 System Architecture

Following the approach of [7], the major tasks of the visualization system are partitioned into three parts: 1) Data sampling and collecting: done by the back-end computational code, which incorporates the function calls to generate the information needed for visualization; 2) Data visualization: to convert the raw data for display (into easily understood representations such as datacards); 3) Communication among the different parts: the back-end program, the visualization system and the server. The architecture of the system is shown in Figure 3.

Initially, the user starts the visualization application and specifies the computation server. A connection is established with the server at the remote site. The server reports the available computation types and run-time parameters to the visualization client which then displays these options in pull-down menus and dialog boxes. After the user selects the desired options, these options are transmitted to the server process.

The server then invokes the remote program according to the given runtime options. The program starts its parallel execution by invoking the requested number of processes, one of which acts as a controller. If the visualization option is turned on, the controller acts as a stream server listening on a specific port number assigned by the server. The server sends the address (socket port number) of the controller to the visualization client and the client connects with the controller.

During the computation, the visualization client sends information requests to the controller; the worker processes send the visualization data to the controller and the controller sends the necessary data to the client. If it becomes necessary for the controller to notify the client of any extraordinary events (such as errors, system shutdown, etc...) the controller may also push unrequested data to the client. The visualization client displays the data dynamically in

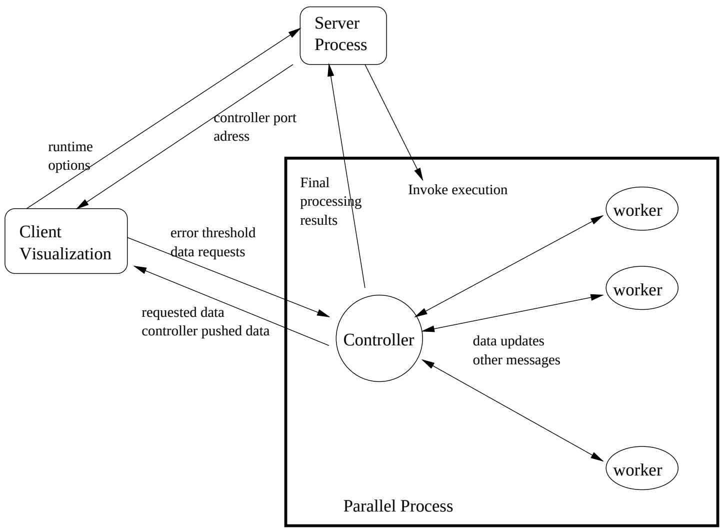

Figure 3: Architecture.

the form requested by the user. At the end of the computation, the server collects the final results from the controller and sends them to the client, where they are displayed for the user.

Regarding the interface with the back-end program, the main design goals are to minimize the modification of the original back-end code and the extra communication and computation overhead involved in the visualization. The latter involves the additional information passed between the controller and workers as well as the computation required to process the visualization data. Note that the total amount of data can be restricted to any level by displaying only the data pertaining to tasks above a specified importance level threshold. The underlying communication interface is realized using sockets, since they provide an easy way to connect C, C++ and Java programs.

4 Conclusion

Our research effort contributes to the fields of parallel computation and visualization by generating a new paradigm for visual presentation of computational processes. Through the deployment of the interface, the acceptance and usability of parallel systems for disciplines such as chemistry, electrical engineering, physics, statistics and psychology will be enhanced.

Specific contributions are in the development of interface metaphors and visualizations of computation which are targeted towards the end users of parallel systems, and the design contributes significantly to the field of Human-Computer interaction through the development of new paradigms for presenting parallel computation.

The development will initially utilize ParInt [1] as a test bed. Testing and evaluation will involve inter-disciplinary groups including casual users as well as parallel processing experts and will be based on participatory design, user evaluation and survey based feedback methods.

References

[1] de Doncker, E., Gupta, A., Ball, J., Ealy, P., and Genz, A. Parint: A software package for parallel integration. In 10^{th} *ACM International Conference on Supercomputing* (1996), Kluwer Academic Publishers, pp. 149–156.

[2] de Doncker, E., Kapenga, J.and Oliver, C., and Mittal, A. Parallel algorithms for radiosity. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* (1995), R. F. Sincovic, D. E. Keyes, L. M. R., L. R. Petzold, and D. A. Reed, Eds., pp. 1019–1024.

[3] Furnas, G. W. Generalized fisheye views. In *Human Factors in Computing Systems: CHI'86 Conference Proceedings* (1986), pp. 16–23.

[4] Haring, G., and Kotsis, G., Eds. *Performance Measurement and Visualization of Parallel Systems.* Elsevier Science Publishers, 1993.

[5] Hart, D., Kraemer, E., and Roman, G.-C. Interactive visual exploration of distributed computations. In *Proceedings of 11th International Parallel Processing Symposium* (Apr 1997), pp. 121–127.

[6] Kaugars, K. Otree:a tree visualization using scaling and omission. Under Review.

[7] Li, J., and de Doncker, E. Dynamic visualization of computations on the internet. In *Lecture Notes in Computer Science* (1999), Springer-Verlag. To appear.

[8] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization - Algorithms and Complexity.* Prentice-Hall, Inc., 1982.

[9] Simmons, M. L., Hayes, A. H., Brown, J. S., and Reed, D. A. *Debugging and Performance Tuning for Parallel Computing Systems.* IEEE Computer Society Press, 1996.

[10] Stasko, J. T., and Muthukumarasamy, J. Visualizing program executions on large data sets. In *Proceedings of the 1996 IEEE Symposium on Visual Languages* (Sept. 1996), pp. 166–173.

[11] Tomas, G., and Ueberhuber, C. W. *Visualization of Scientific Parallel Programs.* Springer-Verlag, 1994.