Authors' address:
Jie Li / Elise de Doncker
Department of Computer Science
Western Michigan University
Kalamazoo MI-49008, U.S.A.

Presenting/corresponding author:
Elise de Doncker
elise@cs.wmich.edu
tel.: (616) 387-5645, (616) 387-5656
fax: (616) 387-3999

Note:
Paper contains color figures. Please print on color printer if possible.

II

Dynamic Visualization of Computations on the Internet

Jie Li and Elise de Doncker *

Western Michigan University, Kalamazoo MI 49008, USA,
{jli,elise}@cs.wmich.edu

Abstract. We present the design of a visualization tool which can be incorporated in parallel/ distributed code to track computational parameters with respect to convergence and the distribution of the work remaining for the parallel processes. We describe its functionality, algorithms and implementation in Java, as it is applied to a package for multivariate integration. Implementation aspects include message passing and control flow in its interaction with the application code. The approach is mainly asynchronous and based on multi-threading.

1 Introduction

Software implementing scientific algorithms is often complex especially for users from different disciplines. For a user it is thus helpful to view the behavior of the algorithm for the problem at hand, to verify that the problem is posed appropriately (and the inputs supplied accordingly), for help in debugging and for assessing the algorithm's efficiency (or its lack of it). This is particularly useful when a software package supplies what seems like more than one viable method to accomplish the same task. Furthermore, the user's faith in a method may be really taunted if the problem is solved on parallel processors.

On the other side, a tool for visualization of the computations helps the developers with debugging and with the cycle of performance evaluation and improvements. In this paper we will discuss the development and functionality of a tool for the visualization of computations on a distributed memory system. The framework is generally that of adaptive algorithms where tasks are selected and partitioned, their subtasks evaluated and inserted into a task pool. Our application is to a package (ParInt) which handles the computation of multivariate integrals by adaptive methods [1]. Other applications that fit a task pool model are, for example, some radiosity algorithms for image rendering, and some branch-and-bound methods.

The visualization component is invoked from a Graphical User Interface (GUI) implemented as a Java applet. The latter allows running the computational software over the Internet by a user at a remote site. In this paper we will describe the design of the visualization tool, its incorporation into the

* Research supported in part by the National Science Foundation under grant CCR-9405377.

GUI applet, its interaction with the Java server and its functions called by the back-end software for collecting/ sampling of the data.

2 Computational characteristics

The terms **automatic**, **adaptive** and **distributed** characterize the algorithms supported by this type of visualization tool. The term *automatic* refers to the fact that the results are required to within a user-specified accuracy or error tolerance ϵ. The latter may measure an absolute or a relative error or a mixture of both. The algorithm performs the computation of its approximation A to the (unknown) solution S, along with an error estimate E pertaining to A, until it finds that $E \leq \epsilon$. The estimated error E is intended to bound the actual absolute error $||S - A||$.

The term *adaptive* indicates that the course of the computation depends on the problem instance at hand. This may be achieved by partitioning the domain of the problem. A fixed formula is used over the original domain; if the error estimate is not satisfactory the domain is subdivided and the local formula applied over each part. From then on the region (task) with the largest error estimate is selected for further subdivision at each step. As a result, successive subdivisions are concentrated in areas where the estimated error is high, i.e., where the problem is difficult. The priority queue is implemented by a heap on the subregions, keyed with the subregion estimated error.

The automatic adaptive algorithm is *distributed* by partitioning the original domain among p available (worker) processes. A distributed global heap version may consider the next p regions of highest estimated error at each step. Alternatively, each worker process keeps a local heap [2]. Since the local heaps at different processes generally differ in size depending on the behavior of the problem on the originally assigned regions, a load balancing mechanism is in place to send work from busy to less busy processes.

The visualization tool displays dynamic information on: 1) the regions with error estimates larger than a given threshold, for each worker process; 2) the newly generated error estimates (over a set time window) for each worker process; and 3) the distribution of the regions over the worker processes at a given time. Type 1) gives information on the load of each process in terms of the number of tasks of which the error estimate exceeds the given threshold; 2) on the recent changes of load for each process; and 3) on the distribution of error and load over the processes.

3 System architecture and design

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 graphs, histograms ...);
3. communication among the different parts: the back-end program, the Java applet (GUI) and the Java server.

The architecture of the system as applied to ParInt is depicted in Figure 1. Initially, the user starts the Java applet from a web browser and selects the

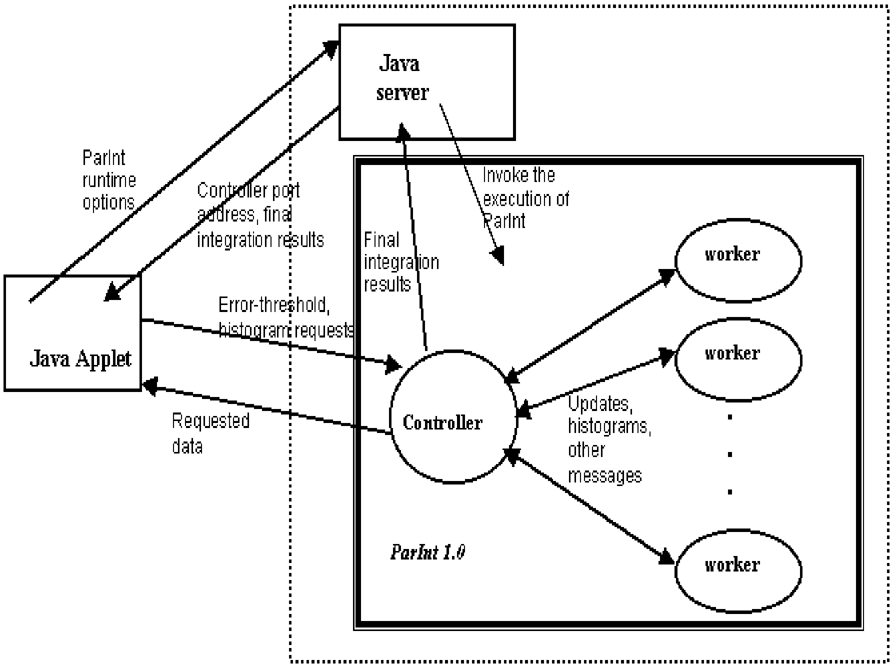

Fig. 1. Architecture

desired (ParInt) runtime options. Visualization can be turned on or off. A connection is established with the Java server at the ParInt site. The Java server then invokes the ParInt program according to the given runtime options. ParInt 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 Java server. The Java server sends the address (socket port number) of the controller to the applet and the applet connects with the controller. During the computation, the applet sends visualization requests to the controller; the worker processes send the visualization data to the controller and the controller sends the necessary data to the applet. The applet displays the data in the requested form. At the end of the computation, the Java server collects the final results from the controller and sends them to the applet, where they are displayed for the user.

Regarding the interface with the back-end program (ParInt), the main design goals were 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.

Regarding the types of display in the Java part, we chose to represent the error distribution by a histogram of the *number of regions (tasks)* vs. *error level*. A time-series diagram is used to view the behavior of the estimated error and the load (number of error estimates above a given threshold) for each process. The relative independence of the visualization code from the rest of the applet has been an important design issue, especially since the applet is still under development.

The underlying communication interface between the three parts was realized using sockets, since they provide an easy way to connect C and Java programs.

4 Implementation

In this section we outline the incorporation of the visualization tool into the ParInt code (for multivariate integration) written in C and layered over MPI. We describe the message passing required for the visualization functions in the controller and worker algorithms of the computational code. We also give the the data retrieving algorithms for the time series diagrams representing processor loads (number of tasks) and errors vs. time, and for the histograms representing the number of tasks vs. error level. The organization of the visualization code is also outlined.

Message passing and control flow. If visualization is turned on, the controller will first create a socket and listen on the provided port number, until there is a connection request from the applet. During the computation, the controller probes to receive new error thresholds or histogram requests from the Java applet, in order to forward them to the workers. The controller also probes for messages from the workers with error estimates, loads or histogram data in each iteration of the computation loop; upon arrival these are sent to the applet.

Within each iteration of their major computation loop, the workers probe for new error thresholds or histogram requests. If an error threshold is received, then the subsequent counting of the regions will be based on this value, until a newer one arrives. The error estimates and corresponding number of regions are sent to the controller, embedded in an update message. If a histogram request is received, the worker sends the data to the controller including the worker id, the histogram array, the minimum and maximum error level (for scaling along the horizontal axis) and a time stamp.

In order not to slow down the computation significantly, the communications are performed asynchronously, apart from those of the initial error thresholds.

Data retrieving algorithms. The local error estimates required for display are provided by the back-end code as they are used to key the local heap of their process. Thus, additional work is only required to count the regions with error above the specified threshold (constituting the load) and to calculate the histogram of the number of regions per error level. These data are obtained by walking through the heap. The heap traversal is performed iteratively (as opposed to recursively) to minimize computational overhead (cf. Figure 2 and Figure 3 with pseudocode algorithm representations for the load and for the histogram, respectively). The histogram is scaled and divided into error levels of

```
Count = 0; // number of regions
Bool ParentOK; // parent does/ does not exceed threshold

P points to root of heap;
While(true) {
        If P's region-error exceeds threshold {
                ParentOK = true;
                Count++;
        }
        If ParentOK and has left child then
                P points to left child;
        Else if ParentOK and has right child then
                P points to right child;
        Else { // no children or Parent's region-error too small
                Do {
                        Walk up through heap;
                } until unvisited right sibling or root is reached;
                If root was reached then
                        Return Count;
                P points to unvisited right sibling;
        }
}
```

Fig. 2. Algorithm for load

equal width. The algorithm counts how many regions of the heap fall into each error level.

Code Organization. To minimize the changes to the back-end code, all major visualization functions are confined to separate files (visual-ctrl.c for the controller and visual-wrkr.c for the worker; header file visual.h was created to contain the new prototypes). The main changes in the ParInt controller and worker functions are done by calls to the visualization functions only; which is

```
P points to root of heap;
Divide error levels evenly between low_exp and high_exp;
While(true) { // associate region pointed at by P with appropriate error level
        If P has left child then
                P points to left child;
        Else if P has right child then
                P points to right child
        Else { // no children or Parent's region-error too small
                Do {
                        Walk up through heap;
                } until unvisited right sibling or root is reached;
                If root was reached then // finished traversal
                        Return;
                P points to unvisited right sibling;
        }
}
```

Fig. 3. Algorithm for histogram

made possible by the modular design. For maximum modularity, file scope static variables and functions are used in visual-ctrl.c and visual-wrkr.c.

4.1 The Java applet

Visualization parameters. The user may specify parameters for the three types of diagrams drawn for this application.

- The local estimated error and load vs. time for each process shows the N most recent samples and is updated dynamically. The user may select the time interval to refresh the diagram and the number of samples N. Sample diagrams depicting local estimated errors and loads of four worker processes are given in Figure 4. For the sample problem at hand, the graphs show that worker 0 has most of the work initially, but the other workers receive work subsequently through load balancing.
- The global estimated error and number of regions diagram shows the current values vs. time and is updated dynamically. The user specifies the time interval for updating.
- The histogram of the number of regions per error level can be queried by the user at any time during the computation for any worker. The user can also "zoom in" on parts of the histogram by specifying the new error range. A sample is given in Figure 5.

  In summary, the following options can be set via the applet to control the visualization functions: to turn visualization on/off; whether or not to display grids on the diagrams; to set the number of samples displayed in the time series diagrams; to specify the time interval for updating the dynamic

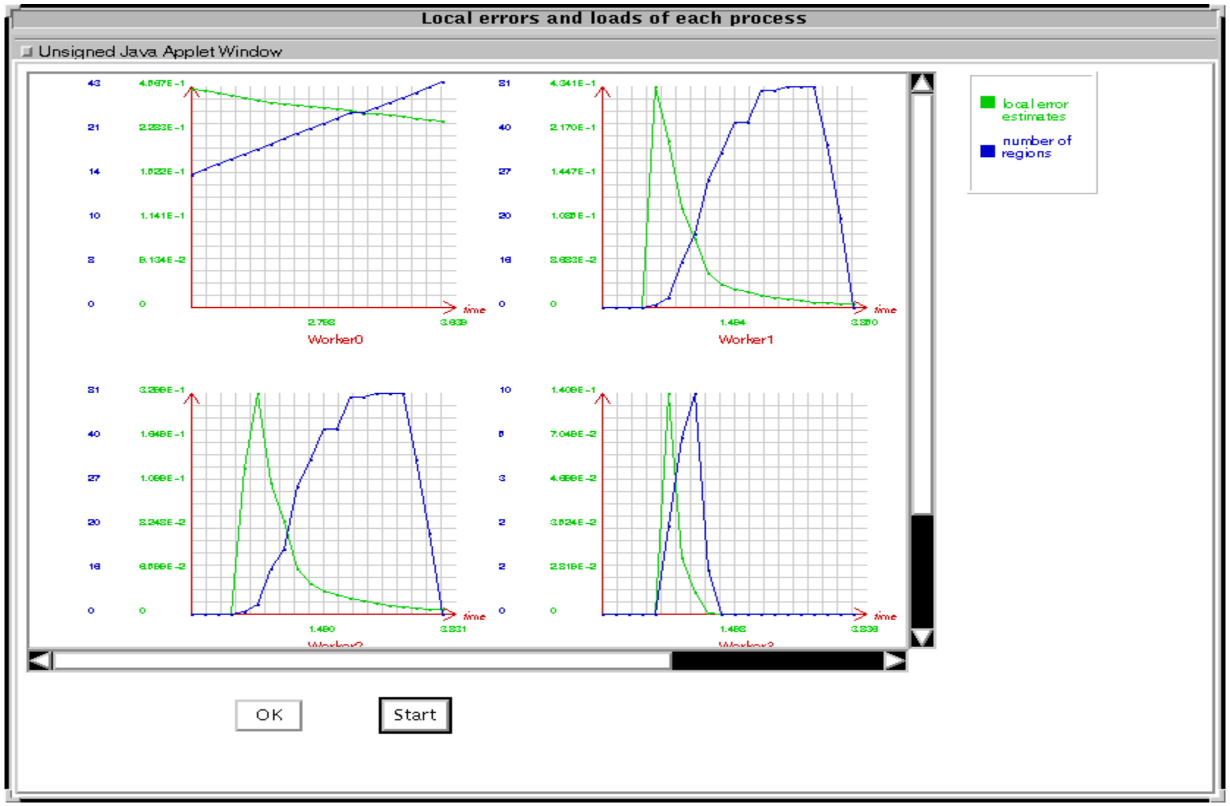

Fig. 4. Local estimated errors and loads

diagrams; and to set the error threshold. To this end, six windows can be opened: the visual options dialog; the local error estimates and load window; the global error estimates and number of regions window; the process id dialog; the histogram window; and the ZoomIn dialog, to focus on a histogram slice by specifying the new error range.

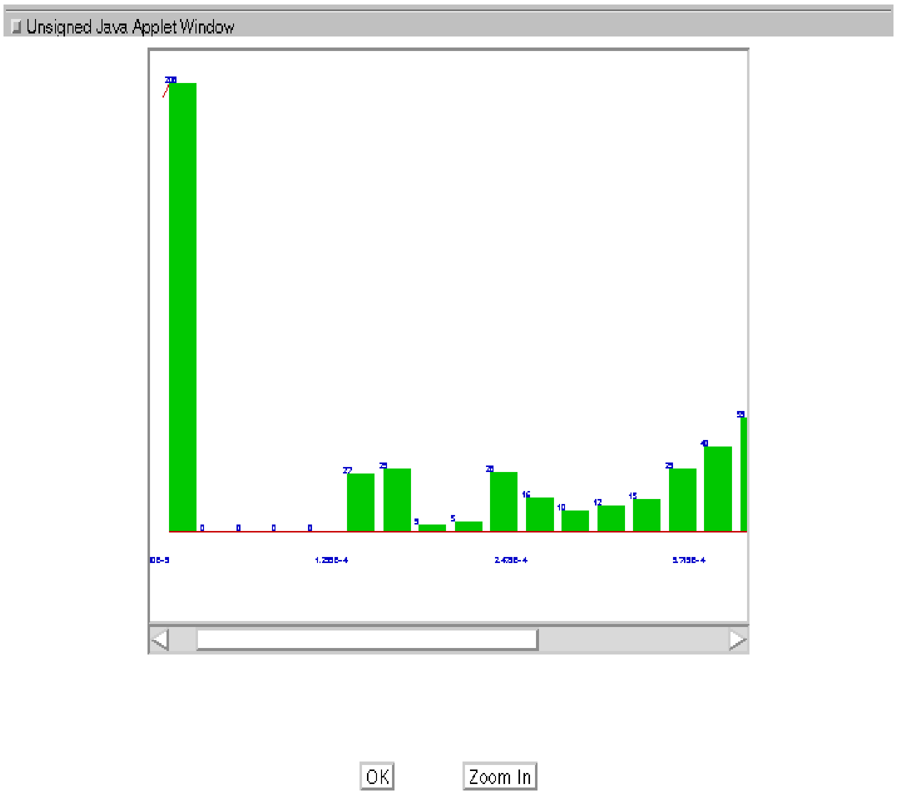

Fig. 5. Histogram

Synchronization and mutual exclusion. In Java, the Socket read and write primitives are all blocked primitives, which means that the thread will block on read or write if it is not ready to receive or send the data. We use multiple threads to resolve this problem since we need to receive data continuously in order to display the newest data dynamically.

When the connection with the ParInt controller is established, a thread listener is started which is dedicated to receiving data from the ParInt controller. A second thread is used to await the final results from the Java server. After the results arrive they are displayed. These threads run concurrently with the applet to alleviate blocking. There are two cases which require a synchronization of threads. 1) When the user requests a histogram, the query is sent to the controller immediately. Since all the data are received by the listener thread, a problem exists with respect to determining the arrival of the histogram. Therefore, the communication of the histogram needs to be synchronized. 2) The listener thread updates the data each time it receives an update from the controller. On the other hand, the drawing threads read the data at fixed time intervals. Updating the data while a diagram is being read and drawn would lead to inconsistent results; the updating and reading operations are mutually exclusive.

A semaphore class is used to resolve these problems [3]. In the first case, a P() at the GUI action handling thread operation is used to wait for the result; a V() operation is used by the listener to acknowledge the arrival of expected data and release the blocking thread. In the second case, pairs of P() and V() are used at the reading and writing ends to implement mutual exclusion.

5 Conclusion

We presented a dynamic visualization tool which can be incorporated into parallel/ distributed computational code to view information regarding the convergence of the computation and the partitioning of the work remaining for the parallel processes. For the algorithm or software designer the visualization tool may serve to assess the efficacy of the computational methods such as a load balancing strategy, or to tweak implementation parameters. For the user it provides an indication that the problem is solved efficiently or, otherwise, that it may need to be posed differently. In the context of multivariate integration, for example, a singularity of the integrand function interior to a subregion will escalate the number of subdivisions around the singularity and the load of the affected process. In that case, the user would generally benefit from pre-partitioning the domain so that the singularity is on or along an edge if possible. The interface of the visualization system with the application back-end code is relatively simple in view of the modularity of the design. Our use of Java for the implementation enhances its portability.

By way of comparison, some visualization systems of this nature are very application oriented. For example, the CAST project focused on the visualization of ocean data and delivered the ENVIS ENvironment VISualization system (http://www.cast.msstate.edu/CompletedProj/ENVIS.html). The goal of the SMART project is to design a Parallel Volume Visualization toolkit (PVV), with applications such as the modeling of pollution dispersion and the visualization of medical data (www.lsi.usp.br/smart). Others are targeted to certain parallel architectures or systems. The ParVis project targets massively parallel computer architectures (www.rigi.csc.uvic.ca/anu/parvis). The Distributed Array Query and Visualization (DAQV) system enables external clients to access and interact with distributed arrays. It is oriented toward Fortran applications implemented as SPMD processes which may communicate via a message passing system (www.cs.uoregon.edu/ hacks/research/daqv). Yet another class of visualization systems allow for process event tracking, for example to visualize sends and receives (as supported by XPVM/ PGPVM; and MPE). Exploratory Visualization (swarm.cs.wustl.edu/projects/explore) enables dynamic and animated process event monitoring.

Acknowledgment. We thank Srinivas Hasti and Rodger Zanny for discussions regarding the Java server and the computational back-end code of ParInt.

References

1. E. de Doncker, A. Gupta, J. Ball, P. Ealy, and A. Genz, *Parint: A software package for parallel integration*, in 10^{th} ACM International Conference on Supercomputing, Kluwer Academic Publishers, 1996, pp. 149–156.
2. E. de Doncker, A. Gupta, and P. Ealy, *Two methods for load balanced distributed integration*, in Lecture Notes in Computer Science, vol. 1067, Springer-Verlag, 1996, pp. 562–570.
3. E. W. Dijkstra, *Co-operating sequential processes*, in Programming Languages, F. Genuys, ed., Academic Press, New York, 1965, pp. 43–112.