DISTRIBUTED MATLAB BASED SIGNAL AND IMAGE PROCESSING USING JAVAPORTS

Elias S. Manolakos *Demetris G. Galatopoullos* *Andrew P. Funk*

Communications and Digital Signal Processing Center for Research and Graduate Studies
Electrical and Computer Engineering Department, Northeastern University, Boston, MA 02115

{elias, demetris, afunk }@ece.neu.edu

ABSTRACT

Many scientists and engineers have signal and image processing applications that involve large data sets and could benefit from parallel processing on readily-available clusters of workstations (COWs). Unfortunately these applications often exist as legacy code, such as Matlab functions, which are not easily parallelizable. The goal of the JavaPorts project is to provide a framework for flexible development of parallel and distributed component-based applications for heterogeneous COWs. The latest version of the package supports the integration of Java and Matlab components into the same application. The main features of the JavaPorts framework and its tools are discussed and application examples are presented which highlight its ability to support the rapid prototyping of parallel and distributed computing strategies with communicating Java and Matlab components.

1. INTRODUCTION

Clusters of Workstations have become increasingly available and provide a cost-effective alternative to traditional supercomputers for coarse grain parallel computing. Signal processing practitioners routinely collect large amounts of sensor and image data that could be processed more effectively in parallel. However, they often lack the expertise, or time, required to parallelize their algorithms. Furthermore, many scientific algorithms currently exist as modular Matlab functions (or other legacy code) that is not readily parallelized. Rather than taking the time to become expert parallel programmers themselves, most researchers would prefer to have at their disposal a development environment that would allow them to easily translate legacy code to software modules (components) that they could then connect together visually to create distributed applications for COWs.

Matlab [1] is a popular computational environment that integrates numerical computations with high-level programming. It offers rapid prototyping of matrix computations which makes it ideal for exploring new algorithm ideas. However, the performance of Matlab code does not scale well with the problem size. Due to the increasing popularity of the language there is a resurging interest in providing parallel processing support to Matlab applications [2]. Efforts in this area attempt to either provide MPI-like message passing to Matlab, or build in Matlab methods that will partition work among multiple sessions, or convert automatically Matlab scripts into parallel programs.

This work has been supported in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the ERC Program of the National Science Foundation (Award Number EEC-9986821).

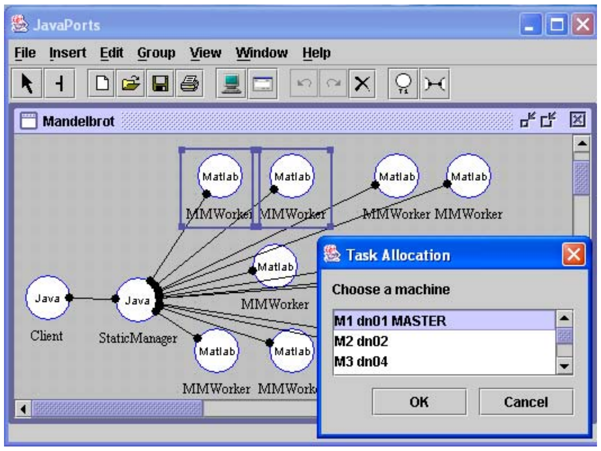

Fig. 1. The JPVAC tool allows the user to build a distributed application graphically by importing software components from other applications or from a components library and to allocate a subset of components to any available machine. Re-allocating components to different set of machines does not require any code changes.

To the best of our knowledge, JavaPorts [3] is the first framework that provides message passing support to Matlab components by exploiting their capability (in Version R13 and beyond) to incorporate Java code. Using both Matlab and Java software components in the same distributed application offers the potential for building broad and flexible collaborative environments. Matlab components can utilize Java in order to interact in a network computing application. They can also take advantage of the many good features of Java such as the rich set of exceptions, web-based programming, IDE and GUI tools etc. On the other hand, by interfacing with Matlab, Java applications can access a large set of optimized and versatile function toolboxes for signal and image processing, statistics, bioinformatics etc. We believe that the complementary nature of the two languages offers a lot of exciting possibilities for making painless the transition of serial legacy code to better performing scalable parallel implementations, by exploiting the rapid prototyping capabilities of the one with the network computing capabilities of the other.

In the next section we summarize the main elements of JavaPorts environment and the capabilities of the associated development tools. Then in section 3 we provide an example of a Java/Matlab distributed image processing application, discuss its characteristics and provide an indication of the expected performance of similar applications.

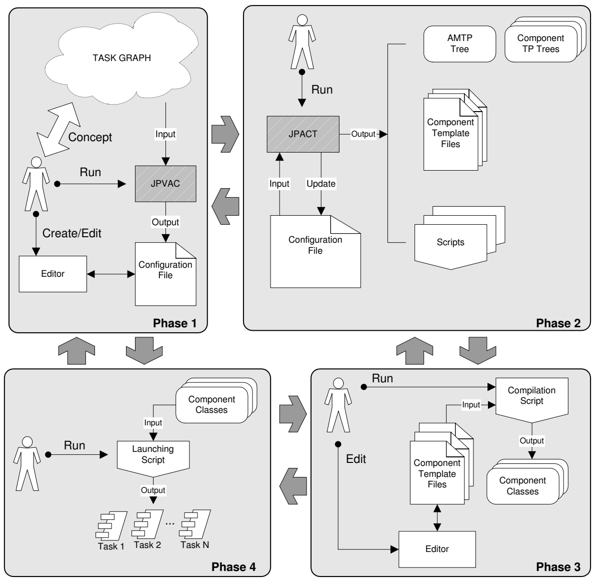

Fig. 2. The JavaPorts component-based application development phases. The tools JPACT and JPVAC are shown using a darker shade. The arrows between phases denote allowable transitions.

2. THE JAVAPORTS ENVIRONMENT

2.1. Application Modeling and Development tools

A task graph models a multithreaded and distributed application as a collection of interacting components and allows the software engineer to establish a high-level view of the overall computational scheme before any code is developed . The Visual Application Composer (JPVAC) tool [4, 5] serves as the graphical front-end of the JavaPorts system and allows incremental task graph construction using standard and advanced graph editing features. The JPVAC can save the task graph in a persistent object (called the AMTP[1] tree) and in a configuration text file. The saved models can be re-visited and modified as desired at a later time.

A JavaPorts *task* represents a user process employed to execute a module (component) of the application. When the graph is compiled JavaPorts creates a "template" i.e. a program skeleton written in Java or Matlab, for every newly defined task. Each task is allocated to a compute node (machine) for execution. As it can be seen in Figure 1, several tasks may be allocated to the same machine

Interacting tasks are connected with edges. The edge endpoints, shown as dark dots in Figure 1, are the *Ports*. A Port here is a JavaPorts abstraction introduced to support *anonymous* message passing between connected tasks. A task does not need to be aware of the "rank" of the destination task to send it a message. It can simply delegate the communication responsibility to one of its local ports. This port will have to buffer and transmit the message to its peer port in the destination task transparently to the application layer. A pair of connected ports correspond to a bi-directional logical link (channel) among tasks. Channels may connect tasks allocated to the same machine, or they may cross machine boundaries.

[1]Application-Machines-Tasks- Ports.

The JPVAC tool includes a powerful Undo option that can be used to nullify a sequence of steps during task graph development. Another powerful feature is the ability to concurrently display and edit multiple applications in a single session. This feature, combined with the ability to hierarchically group tasks, makes cross-application development and evaluation straightforward. Certain patterns of tasks used in one application can be viewed and reused in another, thus enhancing application development productivity. The JPVAC also provides "off-the-shelf" commonly used task topologies (such as stars, rings, meshes etc) that the user can simply import into an application.

The JavaPorts Visual Task Modeler (JPVTM) tool, may be used to construct, in conjunction with the JPVAC, a more refined two-level structural/behavioral application model. Using the JPVTM tool the behavior of each individual task can be modeled as a graph of interconnected elements. Such elements include serial code blocks, fork, join, loop, branch as well as synchronous and asynchronous communication operations. Expected performance data can be used to annotate different elements. The hierarchical multithreaded application models, along with available benchmarks, machine and network load data, are processed by the JavaPorts Quality of Service (QoS) system in order to estimate the expected performance of different mappings of tasks to network nodes. The QoS system may predict whether specified application-level QoS requirements can be met under certain conditions before the application code is developed.

The JavaPorts Application Compilation Tool (JPACT) is responsible for parsing and compiling the application configuration file. If that step is successful, it creates a "blank" template file for each defined task (Java or Matlab), as well as a set of O/S specific scripts that make it easy to compile and launch the application from a single node in the network, or to cleanup in case of abnormal termination. The latest version (v. 2.5) of JavaPorts supports distributed applications running under the Linux or Solaris operating system, in clusters with or without NFS support. JavaPorts application development follows a four-phase sequence inspired by sound software engineering principles: Application modeling, automatic generation of code templates and scripts, components code completion by the user, and application launching from a single node in the network. The tools used in each phase are mentioned in Figure 2. The phases are event-ordered in a natural manner, however, it is possible to backtrack to a previous phase to re-engineer, re-design or correct certain problems as needed.

2.2. The JavaPorts Distributed Runtime System

The JavaPorts runtime system was designed with the requirement to make the remapping of tasks to the nodes of a heterogeneous cluster easy. Therefore JavaPorts provides the user application with light-weight distributed runtime support that decentralizes the management of its components. Every application task is supported at runtime by a `PortManager` object, shown in Figure 3, which is responsible for creating, configuring, serving, and terminating its ports. The `PortManager` performs remote lookups and once it discovers the task's peer ports, it connects them to the task's ports. It returns an array of handles to the local ports that the task's implementation can utilize to exchange messages anonymously with its peer tasks (local or remote). The `PortManager` is also responsible for disconnecting and disposing of the task's ports and for performing distributed termination. All `PortManager` operations are performed transparently to the application layer. To the best of our knowledge, JavaPorts is the only

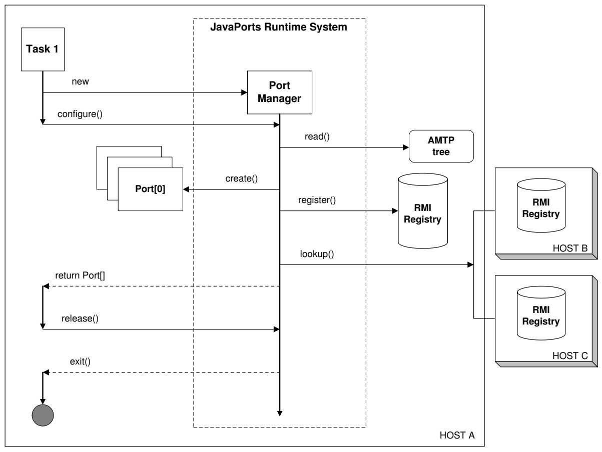

Fig. 3. The JavaPorts Distributed Runtime System. There is one `PortManager` object for every task which creates, maintains and terminates the Port objects for its owner task.

distributed components framework that relieves the user from the responsibility of implementing a distributed termination protocol in the application code.

3. JAVAPORTS APPLICATION EXAMPLES

This section describes the development of a parallel and distributed application for computing a fractal image, known as the Mandelbrot set. The number of repetitions necessary to produce a pixel is indicated by color. A red pixel indicates that only a single iteration caused the algorithm to jump outside the specified range, while at the other extreme a black pixel indicates that the algorithm terminated only after the limit of repetitions was reached. Due to the nature of the Mandelbrot set calculation, some regions of the image are more compute-intensive than others.

3.1. Connecting Java and Matlab Components

The Mandelbrot application consists of three basic components (see Figure 1). The `Client` Java component is responsible for accepting user input and displaying the computed image. It is connected to a `StaticManager` Java component which receives the image size parameters from and partitions statically the image into as many row stripes as the number of available Matlab worker components (`MMWorkers`).

Figure 4 shows the implementation of the Matlab `MMWorker`. This component receives from the Manager a portion of the image to be computed and passes this information to a `mSets` Matlab code designed to compute the Mandelbrot set on a specified region of the complex plane. By dividing the total image among all the `MMWorkers`, and calling a separate instance of the `mSets` function to perform the Mandelbrot computation on each subregion of the image, the original serial Matlab code is transformed into a module of the parallel implementation. No changes are required to the original `mSets` function; it is encapsulated by the `MMWorker` JavaPorts component which handles all communication with other components. The template for the `MMWorker` component is automatically generated by JavaPorts. The user has to add the call to the `mSets` function and the JavaPorts read and write operations to a local port highlighted in Figure 4. The developer of the

```
function  MMWorker(AppName, TaskVarName)

  % register ports
  portmanager = PortManager;
  port = portmanager.configure(AppName, TaskVarName);

  % read first message
  data = port(1).SyncRead(readkey);

  % loop until EXIT message received
  while (data.status ~= Message.EXIT)

    % call legacy Matlab function mSets to compute Mandelbrot set
    iterations = mSets[real˙min real˙max imag˙min imag˙max],
                                        maxiter, [xrange yrange])


    result = Message(data.x1, data.x2, data.y1, data.y2,
                                    iterations, data.maxiter, time,
                                    TaskVarName+'' (Matlab)'');

    % send result
    port(1).SyncWrite(result, writekey);

    % read next message
    data = port(1).SyncRead(readkey);

  end

  % distributed termination
  portmanager.release;

quit;
```

Fig. 4. The Matlab `MMWorker` component receives a portion of the overall image and calls Matlab function `mSets` to perform the Mandelbrot computation. Highlighted are the statements the Matlab programmer has to add to the JavaPorts created code template.

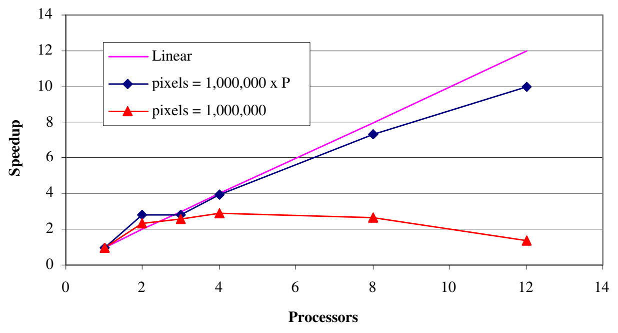

Fig. 5. Due to message passing overhead, speedup saturates for a fixed problem size. However, when the problem size increases linearly with the number of available processors, the algorithm exhibits close to linear speedup.

`MMWorker` J component does not need to know the name of the task or port to which the local port is connected, or worry about distributed termination.

3.2. Rapid Prototyping

The JPVAC also provides the ability to port a parallel application to a new computing platform in the event that new hardware becomes available. In this example, the Mandelbrot application is originally developed on a cluster of single-processor workstations. In this configuration a single `MMWorker` component is allocated to each computing node. The application is then reconfigured to take advantage of a newer cluster of dual-processor workstations. In this configuration, two `MMWorker` components are allocated to each computing node to take advantage of the dual-processors (see Figure 1). As with the previous example, this reconfiguration can be performed within the JPVAC and requires no code changes.

The performance of the dual-processor configuration is shown

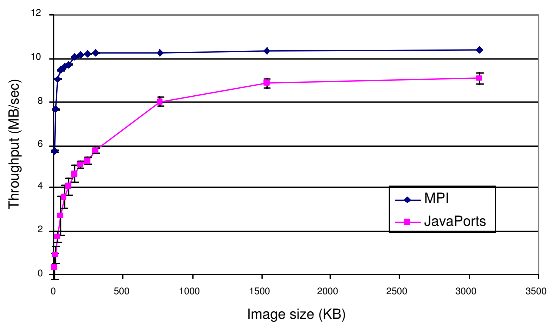

Fig. 6. Comparison of C/MPI to JavaPorts achieved bandwidth in image ping-pong experiments.

in Figure 5. The Mandelbrot application was first run with a fixed problem size of one million pixels (1,000 x 1,000 grid). The application achieves some measure of speedup as the number of processors increases from one to four. As the number of processors increases beyond four, however, the speedup decreases. This speedup saturation is expected as processors get added while the problem size remains fixed. In contrast, when the problem size is increased linearly with the number of processors, the speedup continues to increase. This measure of "scaled speedup" is often a better indicator of the potential of a parallel algorithm-infrastructure combination to deliver useful computational throughput. Particularly in signal and image processing there are many applications which require processing large amounts of data in parallel. These results demonstrate that a combination of Matlab code and JavaPorts infrastructure can provide a flexible environment for rapid prototyping with the ability to process large data sets efficiently.

3.3. Communication Performance Benchmarking

To assess the communication throughput that can be achieved when using JavaPorts in image processing applications, we conducted a ping-pong experiment in a 32 node Linux cluster of PCs connected via fast (100 Mbit/sec) Ethernet. The test data were scaled 8-bit color bitmap images ranging in size from 4 KB up to 3 MB. The test application was a simple ping-pong algorithm involving two nodes that has been coded using both JavaPorts and C/MPI. In each case, the Manager task begins by reading an image file from disk into memory and sends the image data as a byte array to the Worker task. The Worker task receives the data and simply returns it back to the Manager. This procedure is repeated 100 times for each image size to obtain the median round trip time (*rtt*).

The *rtt* values were used to calculate the throughput as a function of image size for JavaPorts and C/MPI. As shown in Figure 6, the calculated throughput for C/MPI rises sharply with increasing image size and levels off at approximately 10 MB/s, that is close to the 100 Mb/s capacity of the underlying network. The JavaPorts achieved throughput increases more gradually but is also leveling off close to the capacity limit, at approximately 9 MB/sec. The throughput difference is likely attributed to the overhead of using Java/RMI[2]. However, these results show that for reasonably large messages (greater than 1 MB in size), this overhead is small, relatively to the overall message transfer time, and the performance of JavaPorts becomes comparable to that of C/MPI. It should also be noted that a great deal of time and effort has been spent in optimizing MPI implementations. It is reasonable to expect that as Java/RMI technology matures its performance will improve and this will make the difference between between the C/MPI and JavaPorts achieved bandwidth even less significant.

[2]Remote Method invocation.

4. CONCLUSIONS

The recent trend toward computing on networks of heterogeneous nodes has created a need for new development tools to meet the unique requirements of these platforms. Such tools should allow the application to be configured independently of its implementation, so that when some aspect of the platform changes, the application may be reconfigured at a high level, without requiring any implementation changes. Ideally the tool should also provide the productivity features that are commonplace among non-parallel application development environments.

JavaPorts provides a high-level task graph abstraction for building a distributed application, along with a Ports API that supports anonymous message passing and a run time system that supports task location transparency and distributed task termination. Built on top of Java's Remote Method Invocation (RMI) technology, it is inherently platform independent. These features allow the developer to test different software and hardware configurations quickly, without having to modify the source code.

The latest version of JavaPorts provides for the seamless integration of Java and Matlab components in the same distributed application. To the best of our knowledge JavaPorts is the only framework that allows realizing any desirable task graph of Matlab and Java components. This enables existing serial Matlab code to be converted easily into components that may interact in any desired way to solve large size problems in parallel.

Using the JavaPorts environment and communication infrastructure we are currently building a software prototype of a virtual distributed sensing and imaging laboratory. This laboratory includes Matlab components for acquiring and filtering sensor data (running in machines associated with the instruments that produce the data) as well as Matlab components for fusing and processing selected images (running on distributed compute servers). Users of the laboratory can graphically combine available "producer" and "consumer" components to build image processing pipelines that can be launched in the network. We are also investigating the design of a JavaPorts version for low power handheld devices and embedded systems.

5. REFERENCES

[1] The MathWorks, Inc. MATLAB and Simulink. http://www.mathworks.com, 1994–2003.

[2] R. Choy. Parallel Matlab Survey. http://theory.lcs.mit.edu/~cly/survey.html, 2003.

[3] D. Galatopoullos. *JavaPorts: A framework to facilitate heterogeneous cluster computing* Electrical Engineer degree Thesis, Dept. of Electrical and Computer Eng., Northeastern University, Nov. 2003.

[4] E.S. Manolakos, D.G. Galatopoullos, and A. Funk. Component-Based Peer-to-Peer Distributed Processing in Heterogeneous Networks using JavaPorts. In Proc. of IEEE Int'l Symp. on Network Computing and Applications, pp. 234–238, 2001.

[5] A. Funk. *A Tool for the Visual Composition of Distributed Java/Matlab Applications.* Master's Thesis, Dept. of Electrical and Computer Eng., Northeastern University, May 2003.