Integrating Java and Matlab components into the same parallel and distributed application using JavaPorts

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

Electrical and Computer Engineering Department
Northeastern University, Boston, MA 02115

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

Abstract

As clusters of workstations become increasingly popular the need for component frameworks that facilitate the high level modeling and rapid prototyping of parallel and distributed applications on such systems is becoming pressing. Many scientists and engineers have image and signal processing applications that could benefit from cluster computing. However, these applications often exist as legacy code, such as serial Matlab functions, which are not easily parallelizable. The goal of the JavaPorts project is to provide a framework and a set of tools that make it easy to develop component-based parallel and distributed applications for networks of heterogeneous computing nodes. The latest version of the package supports the integration of Java and Matlab components into the same application and provides a mechanism for incorporating legacy Matlab functions into parallel processing applications. The design and salient features of the framework and associated tools are discussed here, and application examples are presented which highlight how JavaPorts can be used to model, develop, launch and restructure applications with any number of interacting Java and Matlab components.

1 Introduction

Motivation
Clusters of Workstations (COWs) have become increasingly available and provide a cost-effective alternative to traditional supercomputers for coarse grain parallel processing. Scientists and engineers routinely collect large amounts of sensor and image data that could be processed more efficiently in parallel. Unfortunately, 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 blocks to interacting modules and then stitch them together to create distributed applications for COWs.

Existing Tools
Although tools for cluster computing have matured considerably over the last decade, they are still not adequately supporting distributed application modeling at a high level of abstraction. There is also a need for optimized runtime systems that can manage tasks on distinct address spaces and APIs that free the application developer from the intricacies of low-level message passing.

Message passing libraries, such as MPI and PVM, may require that all nodes run the same operating system or that all tasks are implemented using the same programming language. Traditionally MPI implementations were provided only for well established languages, such as C and Fortran, and only recently there have been efforts to extend support to object-oriented languages such as C++ and Java. Furthermore, message passing programs are typically written in the Single Program Multiple Data (SPMD) style where the behavior of each process and the pattern of interprocess communication are tied to its “rank” (index variable).

What is the JavaPorts framework?
The JavaPorts components framework ([1, 2, 3]), along with its runtime system and development tools, supports the high level visual modeling of multi-component, parallel and distributed applications, as well as their implementation and deployment on a heterogeneous cluster of workstations. JavaPorts is written in Java and it can be utilized under any operating system that can execute a standard edition JVM. The JavaPorts project tries to extend the "write once run ev-

* 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)
erywhere” promise of Java to distributed and multithreaded applications. Although it is currently used for cluster computing, there are ongoing efforts in our group towards creating a version of JavaPorts that will support grid-scale distributed application development.

JavaPorts supports the integration of Java and Matlab components into the same application, a feature that, to the best of our knowledge, is not currently provided by any other cluster computing framework. The programmer does not need to include named sockets, any environment variables or any application naming inside the application code. Therefore the resulting components become truly location-transparent and reusable across applications. JavaPorts supports all four-phases of component development with well designed tools. Furthermore, it provides a “light” task-level distributed runtime that associates a PortManager object with each application task. Therefore each JavaPorts component interacts with its PortManager only and does not rely for run-time support on machine level daemon processes that need to be started on remote nodes before a distributed application can be launched.

Related Work summary
There are several on-going efforts aiming at exploiting or extending the Java programming language to facilitate coarse grain distributed application development. We mention here only a representative subset of those that primarily target cluster computing. For example, the Visper framework [4] is an object-oriented environment for building distributed Java applications. It provides a visual parallel programming model based on the Process Communication Graph (PCG) formalism and a comprehensive API for message exchanges between objects. Java Parallel (Javap) [5] is a Java library that uses reification and reflection to support transparent remote objects and seamless distributed computing in both shared and distributed memory multiprocessor systems. The JMPF project [6], a Message Passing Framework for cluster computing, provides portability of user applications over heterogeneous platforms while eliminating the need for complicated socket programming in the application code.

Efforts for parallelizing Matlab
Matlab [7] is a popular environment that integrates the rapid prototyping of numerical computations with high-level programming. However, because Matlab is an interpreted language its performance does not scale well with the problem size. In order to achieve performance gains, programmers often opt to parallelize their Matlab programs. There is recently a considerable amount of work in providing parallel processing support to Matlab applications [8]. Efforts in this area attempt to either provide MPI-like message passing to Matlab, build methods that will partition work among multiple sessions of Matlab, or convert Matlab scripts into parallel programs. For example, MultiMatlab [9] provides Matlab-style commands so that a programmer can launch MATLAB processes on remote nodes through an interactive session. Another package, the MATmarks [10], enables programmers to execute MATLAB functions in parallel. It extends the MATLAB instruction set in order to add synchronization mechanisms to the language and it supports a shared memory programming style.

To the best of our knowledge, there is currently no framework, other than JavaPorts [11], that can be used to build Matlab-based parallel and distributed applications by exploiting the capability of the latest Matlab versions to incorporate Java code in Matlab functions. Due to their complementary nature, combining both languages offers the potential for building interesting collaborative environments. Matlab tasks can utilize Java in order to interact in any desired way in a network computing environment or over the Internet. They can also take advantage of the many good features of Java including the rich set of exceptions, web-based programming, IDE and GUI tools, and threading. On the other hand, by interfacing with Matlab, Java tasks can access a large set of optimized and versatile toolboxes covering a variety of fields from image processing, to statistical analysis and bioinformatics. Therefore there exist exciting opportunities for developing tools that try to exploit the rapid prototyping capabilities of the one and the network computing capabilities of the other in order to achieve a painless transition of available serial legacy code to parallel implementations with acceptable performance.

Paper Organization
The rest of the paper is organized as follows: We provide a description of the basic elements of the JavaPorts framework in Section 2. We first explain how a Task Graph is used to model the multi-component distributed application. Then we describe the AMTP tree data structure used to store the task graph information. Subsequently, we demonstrate how the JavaPorts Application Configuration Language (JPACL) can be utilized to capture the task graph textually in a configuration file. In Section 3 we discuss the four phases of the JavaPorts application development cycle. We first introduce the JavaPorts Visual Application Composer (JPVAC) for graphically creating and editing task graphs, then the JavaPorts Application Compilation Tool (JPACT) for compiling the graphs to create code “templates” and scripts, and end this section with an outline of the runtime system and a unique distributed task termination mechanism it implements. In Section 4, we present in some detail the Port abstraction and the associated Application Programming Interface (API). We describe the analysis and design of the classes that implement port services. Next, in Section 5, we discuss how JavaPorts was used in a rapid prototyping image processing case study. We provide a summary and point to future directions in Section 6.
2 The JavaPorts Framework

The Application Task Graph
The task graph models the application as a collection of interacting components and allows the programmer to establish a high-level view of the overall computational strategy. Once the application is decomposed into interacting tasks, the user may start building the graph using the JavaPorts Visual Application Composer (JPVAC) tool [2, 12]. The JPVAC serves as the graphical front-end of the JavaPorts system and it allows incremental task graph construction using standard and advanced graph editing features. The JPVAC can save the task graph in persistent objects and in a configuration file. The programmer can then re-visit the application and complete its task graph at a later time.

A JavaPorts task represents a user process that will be used to execute a module (component) of the application. A newly defined task gets automatically translated by JavaPorts into a “template”, i.e. a program skeleton, written in either Java or Matlab. Each task is allocated to a compute node (machine), which the user has chosen for its execution. As it can be seen in Figure 1, several tasks (large solid line boxes) may be allocated to the same machine (dashed-line box). Each machine may be identified by its domain name or its Internet-Protocol (IP) address.

In the task graph, tasks are connected using edges. The endpoints of these edges, shown as small solid-line squares in Figure 1, are the Ports. Each task is connected to a peer task through a Port, an abstraction responsible for hiding the location (machine) of the destination task (and port) from the source task and for keeping the inter-task communication and coordination details transparent to the application layer. A pair of ports connected via an edge corresponds to a bi-directional logical link (channel) among directly interacting tasks. Logical links may connect tasks residing on the same machine or they may cross machine boundaries.

The AMTP Tree Data Structure
The Application-Machine-Task-Port (AMTP) tree is a data structure used to store the task graph information. The AMTP tree is saved as a binary file in the user’s JavaPorts directory space through the standard Java language ObjectOutputStream class. By doing so it becomes possible for JavaPorts to load the object in memory at runtime in order to acquire information about the executing application. As shown in Figure 2, there are four levels in an AMTP tree. The root (Application level node) stores the name of the application. The second level nodes carry information about the Machines on which the application will be deployed. The third level nodes store information about the Machines on which the application will be deployed. The fourth level nodes store information about ports and their connections to peer ports.

The JavaPorts Application Configuration Language
A task graph can be described textually using the JavaPorts Application Configuration Language (JPACL). The JPACL has a simple, self-explanatory syntax and can be used to create an application configuration file using any available text editor. If the JavaPorts Visual Application Composer (JPVAC) tool is used instead, the configuration file is created automatically every time a task graph is saved. The configuration file is the input that the JavaPorts Application Compilation Tool (JPACT) processes to create code templates for the components of a JavaPorts application and a script for launching it in the network. The configuration file corresponding to the Mandelbrot set application of Figure 1 is provided in Figure 3.

3 JavaPorts Application Development Tools

The Application Development Phases
The JavaPorts application development cycle follows a four-phase sequence inspired by sound software engineering principles. As it can be seen in Figure 4, the phases are event-ordered in a natural sequence, however, the user may backtrack to a previous phase if it is needed to re-engineer, re-design or correct certain problems. The four phases are described in more detail below:

- Phase 1: Application Modeling and Configuration
  During the first phase the programmer examines the application’s requirements and creates a task graph model for the application. If the JPVAC tool is used, the application configuration file is updated every time the model is saved.

- Phase 2: Generation of Templates and Scripts
  A configuration file is compiled using the JPACT tool. If there exist syntax or logical errors they should be fixed by the user. Code templates and scripts are generated upon successful compilation.

- Phase 3: Application Implementation
  The programmer adds code in the modifiable part of each newly created task template as needed to implement its application-specific functionality. Once each component is developed, the programmer can compile all Java code templates using the automatically generated cjtempl.sh script. Any syntax errors that surface should be corrected during this phase.

- Phase 4: Application launching
  The network application may now be launched from a user-designated (master) node using an automatically generated launching script. In order to address any runtime errors, the programmer should return to Phase 3, modify code templates as needed and re-compile the application before relaunching.

The JavaPorts Visual Application Composer (JPVAC)
The JPVAC [12] is a graphical user interface for visual configuration of distributed component-based applications.
Applications may be constructed by combining new components with existing components that are imported from other applications. For each application, the JPVC generates a configuration file that is processed by the JavaPorts Application Compilation Tool (JPACT) to create task templates and scripts. The JPVC includes many advanced features that enhance productivity. An Undo option allows the user to retrace a sequence of steps in case an error is made 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 JPVC also provides "off-the-shelf" commonly used task topologies that the user can simply import into an application; e.g. the user can specify the number of tasks in a star topology and the JPVC will automatically create it.

The JavaPorts Application Compilation Tool (JPACT)
The JPACT tool parses and compiles the application configuration file and, if that step is successful, it creates a “blank” template file (code skeleton of a JavaPorts component with no user code) for each newly defined task and a set of scripts that make it easy to compile and launch the application, or to cleanup in case of abnormal termination. In addition, the JPACT creates or updates the AMTP tree.

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 potentially heterogeneous cluster easy. Therefore JavaPorts provides the user application with light-weight distributed runtime support that decentralizes the management of application components. Every application task possesses a PortManager object, shown in Figure 5, which is responsible for creating, configuring and serving 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 asynchronously 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 transparently to the application layer.

The distributed task termination mechanism
The problem of distributed termination is to determine whether a computation has reached a state at which it is safe for its distributed components to terminate themselves. Although trivial in sequential computing, termination detection becomes difficult in distributed processing because a computation may not have come to an end when a participating component is locally completed. JavaPorts employs a simple and general enough distributed termination scheme summarized pictorially in Figure 6 that shows how the release protocol is carried out between two peer tasks.

The termination mechanism is handled by each task’s PortManager object. When a task is locally ready to terminate a call to the release() method of its PortManager blocks the execution of a task’s run() method until the PortManager determines that the task can safely terminate. First the PortManager ensures that all asynchronous write-request threads that have been issued on any of the task’s ports are successfully completed. Once all such threads have been joined, the PortManager contacts each one of the task’s peer ports and sets its Shutdown flag to true. After this is accomplished, the PortManager polls through its owner task’s ports and checks if their Shutdown flag has been set to true by their peers. The task is allowed to exit and terminate only after its ports have their Shutdown flag set to true by the PortManagers of their peers. To the best of our knowledge, no other component-based distributed computing framework provides a termination detection mechanism that is transparent to the application developer.

4 The Port Design

The Application Programming Interface (API)
The JavaPorts framework provides a simple Application Programming Interface (API) that allows both synchronous and asynchronous anonymous message passing operations on port objects. A message may contain primitive data types, complex data structures, or objects. The task sender does not require knowledge of the name or rank of the receiving task or its ports. The JavaPorts design employs buffering of incoming messages that are held in a linked list data structure (called Port List) at the receiver’s side. If a Port List becomes full a message FIFO queue is activated to buffer messages temporarily at the sender side until some space is freed in the receiver’s Port List. A write operation uses a message Key (integer) value that identifies the element in the Port List where the sent message will be buffered. A Read operation issued at a receiving port scans the Port List using the Key value and returns a handle to the message if it has already arrived. A detailed description of the semantics of the supported API methods is provided in [11].

The Port Model
The UML diagrams of the classes involved in the Port design are shown in Figure 8. A port object may contain multiple instances of the PortThread and PortException classes. The PortThread class extends the Thread class...
class and its instances are used to perform asynchronous writes on local ports. The PortException extends the Exception class in order to allow the implementation of the design to throw exceptions specific to possible malfunctions. A Port instance contains one PortDataList object for buffering incoming messages. It also contains one FIFOQueue object for temporarily buffering write requests on the sender side when the receiver’s Port List is full. Both the Port List and the FIFO Queue are managed by associated threads represented by one instance of the PortListManager and one instance of the FIFOQueueManager respectively. They both inherit from the Thread class and offer services such as expanding lists, dispatching pending elements and notifying the application of elements availability.

The UML class diagram of the PortManager class is shown in Figure 9. The PortInterface is necessary for the Port class to be remote since it advertises the methods that constitute the port API. The parent PortInterfaceSys includes the methods of the port that should not be visible to the application. The PortManager class utilizes these two classes, the InputObjectStream and the OutputObjectStream, in order to read the AMTP tree data structure from a file and extract information about the owner task and its ports. Every application task instantiates the PortManager class during runtime. The PortManager instance creates and registers the ports of its owner task. It uses the RegistrySetup class to manage the RMI registry on the host machine. If the RMI registry is not running then the RegistrySetup object will launch it and return a handle to it back to the PortManager object. The PortManager object launches the RMISecurityManager instance in order to allow trusted objects to be transferred across machine boundaries.

5 JavaPorts Application Examples

This section describes the development of a parallel application using the JPVAC tool for computing the Mandelbrot set [13] (a fractal image). This application is used quite often as a benchmark for master-slave computing and load balancing because it is easy to partition the problem into subproblems of varying complexities not known at compile time. We first describe the initial composition of the Mandelbrot distributed application. We demonstrate how the JPVAC can be used to build effortlessly different component configurations in order to achieve optimal performance. Then we demonstrate the power of JavaPorts in taking an existing serial Matlab function and turning it into a component of a parallel and distributed implementation. Finally, we demonstrate how a well engineered JavaPorts application can take full advantage of the availability of added network resources without requiring any code changes.

Component-based Application Development

The Mandelbrot application consists of three basic components (see Figure 10). The Display component is responsible for accepting user input and displaying the computed image. The Display component is connected to the Manager component, which may be either of type Static or Dynamic. The StaticManager component receives the image size parameters from the Display, partitions statically the image into as many row stripes as the number of available Matlab worker components (MMWorkers), and distributes a row stripe to each worker.

The number of repetitions necessary to produce a pixel is indicated by color. A red pixel indicates that only a single repetition 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. Therefore, due to the nature of the Mandelbrot set calculation, some regions of the image are more compute-intensive than others. Figure 12 shows the result of running the Mandelbrot application with a StaticManager component and four Matlab MMWorker components. Even though the StaticManager assigns the same number of rows to each worker, some MMWorkers may take longer than others.

The DynamicManager tries to alleviate this problem by using an adaptive work allocation strategy. It initially sends a small fraction of the total image to each available MMWorker to get it started. Subsequently, the amount of rows sent to each requesting MMWorker depends on how fast that component returned its previous results. In this way a form of dynamic load balancing is achieved, where the faster workers are asked to compute more pixels. Figure 10 shows how an existing DynamicManager component is imported easily into the application. This type of high-level application reconfiguration can be accomplished entirely within the JPVAC and requires no code changes.

Figure 13 shows the result of running the application again in the new configuration. In this case using the DynamicManager makes more efficient use of the available resources and reduces the overall parallel runtime from 17 seconds to 12 seconds in the same cluster. For a real-world application the amount of time that can be saved with load balancing can be significant, and the JPVAC allows the developer to test various component configurations rapidly to discover the best strategy with the least expenditure of time and effort.

Integrating Java and Matlab Components

The Display and Manager components have been implemented in Java, and the MMWorker components have been implemented in Matlab. A Manager component receives the total image dimensions from the Display component and distributes a portion of the problem to each MMWorker component.
component. As each MMWorker returns the results of its computation, the Manager relays this information back to the Display component.

Figure 7 shows the implementation of the MMWorker, a Matlab-based component which receives information about the portion of the image to be computed. The MMWorker component passes this information to the mSets function, which is a legacy serial Matlab code designed to compute the Mandelbrot set on a specified region of the complex plane. By partitioning the image among the available 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. Note that no changes are required to the original mSets function; it is encapsulated by the MMWorker component which handles all communication with other components. The code template for the MMWorker component is automatically generated by JavaPorts. The user has to add only the call to the mSets function and the read-data and write-results operations highlighted in Figure 7. Furthermore, many different configurations of the application can be built using the JPVAC and JPACT tools without requiring any changes to the code.

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 11). 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 in Figure 14. 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 linear “scaled speedup” is often a better indicator of the potential of a parallel algorithm-infrastructure combination to deliver useful computational throughput. Particularly in the signal and image processing field 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 [14].

6 Conclusions and future directions
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 modeling 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 JPVAC facilitates the development process by providing an intuitive graphical interface for visually building component-based parallel and distributed applications. This tool provides several productivity features that can help reduce the initial development time as well as enable rapid prototyping via the ability to reconfigure an application at the task graph level.

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. JavaPorts also provides transparent support for distributed termination, setting it apart from other tools for distributed component-based applications, including efforts to parallelize Matlab. All of these features combine to make JavaPorts an efficient and valuable tool set for parallel and distributed application development.

We are currently developing a Visual Task Modeler (JPVTM) tool that can be used in conjunction with the JPVAC, to construct visually a refined two-level structural/behavioral application model. Such models can be analyzed by the JavaPorts QoS system (under construction) to predict whether a desired QoS level can be delivered to
the application before any code is even developed. The latest trends in heterogeneous computing involve employing larger numbers of processors and a wider variety of platforms. The goal of Grid computing is to make accessing computational power as easy as accessing electrical power is today. Many mobile handheld devices now support Java and could also become part of this global computational resource. Future research may involve to grid-enable as well as produce JavaPorts versions that may run on power constrained handheld devices and embedded systems.

References


Figure 3. The JavaPorts application configuration file corresponding to the task graph of Figure 1. The JPACL reserved keywords are shown in uppercase.

Figure 5. The JavaPorts distributed Runtime System. There is one PortManager object for every task which creates and maintains the Port objects.

Figure 4. The JavaPorts component-based application development phases. The arrows between phases denote allowable transitions.

Figure 6. The JavaPorts distributed termination protocol. PortManager objects cooperate to ensure proper termination transparently to the application layer.
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 "~ equality operator"
        % call legacy Matlab function mSets to compute Mandelbrot set
        iterations = mSets(real min max, imag min max),
        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;
end

Figure 7. The MMWorker component receives a portion of the overall image and calls the serial Matlab function mSets to perform the Mandelbrot computation. Highlighted are the statements the Matlab programmer has to add.

---

Figure 8. UML class diagram showing the organization of the classes used to implement Port functionality. The classes arranged in the top row of the Figure are standard Java distribution classes.

---

Figure 9. UML class diagram depicting the static organization of the PortManager class. The classes arranged in the top row of the Figure are standard Java distribution classes.

---

Figure 10. JavaPorts allows the user to build a distributed application graphically by importing reusable software components from other applications or from a components library.
Figure 11. The JPVC allows the user to allocate a subset of components to any available machine without any code changes required.

Figure 12. The StaticManager sends the same number of rows to each Worker regardless of its measured computational performance (pixels/sec). This results in suboptimal processor utilization.

Figure 13. The DynamicManager sends more rows to the faster Workers and consequently the overall parallel runtime is reduced. Note: In a black and white image, red pixels (low iterations) appear grey.

Figure 14. Due to message passing overhead, the Mandelbrot algorithm does not scale well for a fixed problem size. However, when the problem size increases linearly with the number of processors, the algorithm exhibits close to linear speedup.