Developing parallel applications using the JavaPorts environment

Demetris G. Galatopoullos Elias S. Manolakos *

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

Abstract. The JavaPorts system is an environment that facilitates the rapid development of modular, reusable, Java-based parallel and distributed applications for networked machines with heterogeneous properties. The main goals of the JavaPorts system are to provide developers with: (i) the capability to quickly generate reusable software components (code templates) for the concurrent tasks of an application; (ii) a Java interface allowing anonymous message passing among concurrent tasks, while keeping the details of the coordination code hidden; (iii) tools that make it easy to define, assemble and reconfigure concurrent applications on clusters using pre-existing and/or new software components. In this paper we provide an overview of the current state of the system placing more emphasis on the tools that support parallel applications development and deployment.

1 Introduction

The foremost purpose of the JavaPorts system [1] is to provide a user-friendly environment for targeting concurrent applications on clusters of workstations. In recent years, the *cluster* topology has brought to surface a powerful alternative to expensive large-scale multiprocessor machines for developing coarse-grain parallel applications. With the latest advances in interconnection networks, the spare CPU power of such clusters achieves performance comparable to those obtained by supercomputers for compute-intensive problems.

With the introduction of the Java programming language [2], the multiplatform integration of applications running on heterogeneous workstations became a reality. Our system takes advantage of the object-oriented nature of Java in order to provide an environment where programmers may develop, map and easily configure parallel and distributed applications without having extensive knowledge of concurrent or object-oriented programming. This goal is accomplished through the separation of the details of processors coordination from the user-defined computational part and the ability to create and update software components for a variety of cluster architectures.

The developer builds software components for concurrent Java applications using the JavaPorts Application Configuration Tool (Section 3). The tool offers

* e-mail: demetris{elias}@cdsp.neu.edu

a complete software cycle process with incremental phases. A *Task Graph* is first specified using a configuration language and/or a graphical user interface. *Tasks* in this graph are assigned to compute *Nodes* (machines) and may communicate via *Ports*. A Port in the JavaPorts system context, is a software abstraction that serves as a mailbox to a task which may write to (or read from) it in order to exchange messages with another peer task [1]. Ports ensure the safe and reliable point-to-point connectivity between interconnected tasks.

The JavaPorts system will automatically create Java program *templates* for each one of the defined tasks using the information extracted from the Task Graph. The part of the code necessary for creating and registering the ports is automatically inserted by the system. The JavaPorts Application Configuration Tool may be called multiple times each time the configuration file is modified by the user without affecting the user-added code. This scheme allows for the correct incremental development of reusable software components.

There are several new and on-going research projects around the world aiming at exploiting or extending the services of Java to provide frameworks that may facilitate parallel applications development. It would be impossible to account for all of them in the given space, so we just mention here the Java// [3, 4] and Do! [5] projects that employ *reification* and *reflection* to support transparent remote objects and high level synchronization mechanisms. The JavaParty [6] is a package that supports transparent object communication by employing a run time manager responsible for contacting local managers and distributing objects declared as `remote`. (JavaParty minimally extends Java in this respect). The Javelin [7] and WedFlow [8] projects try to expand the limits of clusters beyond local subnets, thus targeting large scale heterogeneous distributed computing. For example, Javelin is composed of three major parts: the broker, the client and the host. Processes on any node may assume the role of a client or a host. A client may request resources and a host is the process which may have and be willing to offer them to the client. Both parties register their intentions with a broker, who takes care of any negotiations and performs the assignment of tasks. Message passing is a major bottleneck in Javelin because the messages destined for remote nodes traverse the slow TCP or UDP stacks utilized by the Internet. However, for compute-intensive applications, such as parallel raytracing, Javelin has demonstrated good performance.

Notionally, the major difference between our design and the above projects is that the programmer visualizes a parallel application as an assembly of concurrent tasks with well-defined boundaries and interface mechanisms. This view is inspired by the way complex digital systems are designed with the aid of modern hardware description languages, such as VHDL [9]. The work distribution of a parallel application is described through a simple configuration language while allowing the user to have absolute control over "what-goes-where". After the application configuration file is completed and compiled, the JavaPorts system will automatically create reusable software task templates. Tasks will be able to exchange information with their peer tasks through ports using bi-directional asynchronous and anonymous point-to-point communication. The JavaPorts

system is able to handle all possible types of data and therefore the message scheme is general and user-defined.

The rest of the paper is organized as follows. In the next section we briefly describe the main elements of the JavaPorts system that include the Task Graph and its allocation, the structure of the Java code templates generated by the system and the interface used for transparent inter-task communications. A more detailed description of the JavaPorts system is offered in [1]. Then in section 3 we outline the application design flow. In section 4 we present some preliminary experimental results using the system. Our findings are summarized in section 5 where we also point to work in progress.

2 The JavaPorts system

Each task in the JavaPorts system is an Ideal Worker which simply carries out a certain job without being concerned about how the input data arrives at its ports or where the output data goes to after it leaves its ports. Moreover, each task is unaware of the identity of its peer tasks (anonymous communications) [10, 11]. The synchronization among concerned ports is handled by the JavaPorts system classes and it is transparent to the programmer. However, the definition of the connectivity among computing entities is user-defined. Once this definition is complete, the JavaPorts system will return to the user a Java code template for each defined task. These templates will be used by the programmer in order to complete the implementation of the specific application.

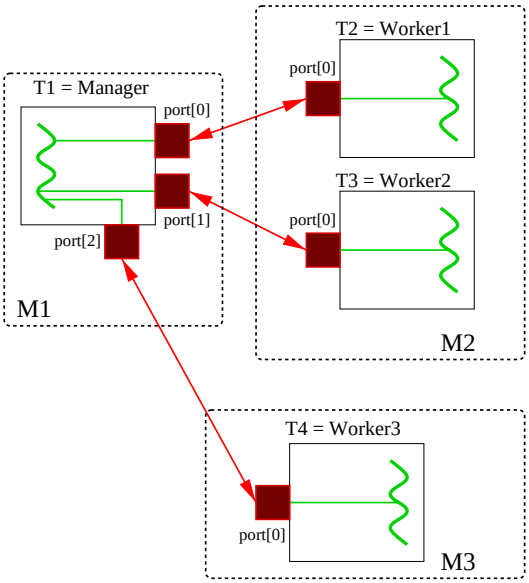
2.1 Task Graph

The programmer may visualize the mapping of the work partitions onto the nodes of the cluster by using the Task Graph. The topology of this graph is captured into a configuration file created and edited by the programmer. The JavaPorts system will make use of the parameters set in the configuration file in order to generate or update the task templates.

A sample Task Graph is shown in Figure 1, where boxes represent tasks and dotted boxes machines (cluster nodes) respectively. This Task Graph depicts a star configuration with machine M1 assuming the role of the central node. There are four user-defined tasks, one allocated on machine M1 (task T1), two on machine M2 (tasks T2 and T3) and one on M3 (task T4). The task which is local to M1, utilizes three ports in order to communicate and the same is true for tasks T2, T3 and T4 which are remote to T1.

We have defined a set of commands available to the user which are sufficient for a complete description of the mapping of the application onto the cluster:

• define machine `machine_var = ''machine_dns_name'' machine_type_flag`
With this command the user will be able to define a machine in the cluster. The `machine_var` is a user-defined label. The `machine_dns_name` is the machine IP name as identified by the Internet domain or the cluster's local domain server.




```
begin configuration
  begin definitions
    define    machine    M1= "corea.cdsp.neu.edu" master
    define    machine    M2= "walker.cdsp.neu.edu"
    define    machine    M3= "hawk.cdsp.neu.edu"
    define    task    T1= "Manager"    numofports=3
    define    task    T2= "Worker1"    numofports=1
    define    task    T3= "Worker2"    numofports=1
    define    task    T4= "Worker3"    numofports=1
  end definitions
  begin allocations
    allocate    T1    M1
    allocate    T2    M2
    allocate    T3    M2
    allocate    T4    M3
  end allocations
  begin connections
    connect    M1.T1.P[0]    M2.T2.P[0]
    connect    M1.T1.P[1]    M2.T3.P[0]
    connect    M1.T1.P[2]    M3.T4.P[0]
  end connections
end configuration
```

Fig. 1. Task Graph and the corresponding configuration file.

The machine_type_flag is used to specify which node will act as the application "master" i.e. used to launch the distributed application on the JavaPorts system.

• define task task_var = ''task_name'' numofports = number_of_ports
The user may define a task with this command. The task_var is a user defined label for each task. The task_name is a mnemonic string that will be used to name the corresponding Java template file generated by the JavaPorts system. The numofports parameter is an integer denoting the number of defined ports for the task.

• allocate task_variable machine_variable
Provided that a certain task and machine have already been defined, this command will assign the task to the specified machine for execution.

• connect machine_var.task_var.P[port_index]
machine_var.task_var.P[port_index]
The connect command is used to specify port-to-port connections between two tasks. All port indices are zero-based. For example, if a task has three (3) ports defined, it can read/write to local ports with names P[0], P[1], P[2] .

• begin/end section_name
These commands are required in order to create readable and easy to debug configuration files. The boundaries of each section and of the whole configuration file should be identified by a begin and an end statement.

In Figure 1 we provide a task graph and its corresponding description. The user may alter the allocation of tasks to nodes, without having to re-write any code he or she added in an existing template, and have the parallel algorithm executed on a different cluster setup. The JavaPorts system will perform all the code template updates needed to run the concurrent tasks in the new cluster setup, even if all tasks are assigned to the same node (machine). We are currently developing a graphical tool for defining and modifying task graphs that will simplify even further the application development process.

2.2 User Program Templates

For illustration purposes we depict the templates generated for tasks T1 (Manager) and T3 (Worker2) in Figures 2 and 3 respectively. A more detailed discussion is provided in [1]. The automatically generated JavaPorts system code is denoted with bold letters. The rest of the code is user-added and remains un-altered after possible application configuration modifications. In this part, the Manager task first forms a data message that is sent to Worker2 and then continues with some local computation. The worker task waits until it receives the data message and then produces some results that are sent back to the Manager.

After the compilation of the configuration file, the JavaPorts system provides the user with pre-defined templates in order to complete the specific appli-

cation implementation. Specialized scripts will automatically dispatch the user executable files to their correct destination nodes.

```
public class Manager

   // User-added variables
   Object ResultHandle;

   run()
   {
       Init InitObject = new Init();
       InitObject.configure(port);

       // User-added algorithm implementation starts here

       // Form the data message and send it out
       Message msg = new Message(UserData);
       port[1].AsyncWrite(msg,key2);
                              ⋮
       // Code for any local processing may go here
                              ⋮
       // Get references to expected results
       ResultHandle = port[1].AsyncRead(key2);
                              ⋮
       //Code to retrieve results may go here
   }

public static void main()
{
   Runnable ManagerThread = new Manager();
   Thread manager = new Thread(ManagerThread);
   manager.start();
}
```

Fig. 2. Template for the Manager task T1.

Port objects communicate among them using the Remote Method Invocation (RMI) [12] package. Each port is registered locally and its counterpartner is looked up remotely. Such operations have proven to be time consuming and therefore they are only performed during the initialization phase of each task.

The JavaPorts system does not constrain the user from exploiting additional concurrency in an attempt to improve the performance of an application. In fact, concurrency is encouraged since multiple tasks may be executed on the

```
public class Worker2

   // User-added variables
   Object ListHandle, DataHandle;

   run()
   {
       Init InitObject = new Init()
       InitObject.configure(port)

       // Get a reference to the object where the incoming data message is stored
       ListHandle = port[0].AsyncRead(key2);

       // Use it to wait, if needed, until the data has arrived
       DataHandle = ListHandle.PortDataReady();
                                  ⋮
       // The algorithm that uses the incoming data and produces results goes here
                                  ⋮
       // Prepare a message with the results
       Message msg = new Message(ResultData)
                                  ⋮
       // Write back the results
       port[0].AsyncWrite(msg,key2)
   }

public static void main()
{
   Runnable Worker2Thread = new Worker2();
   Thread worker2 = new Thread(Worker2Thread);
   worker2.start();
}
```

Fig. 3. Template for the Worker2 task T3.

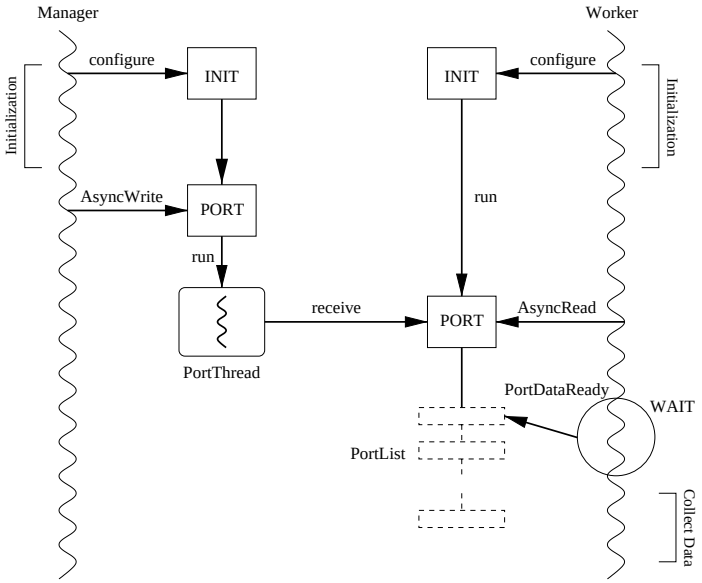


Fig. 4. A message communication using JavaPorts.

same node by treating the local destination ports in a similar fashion as the remote ports. If the appropriate hardware resources become available concurrent tasks may be executed in parallel, otherwise they will be time-sharing the same CPU. Such capability eliminates the considerable development costs for re-engineering the application code each time the allocation of software components to machines is altered for some reason.

2.3 The Ports interface

Using the interface concept of the Java language in order to deal with multiple inheritance, we have introduced an allowable set of operations on a port [1]. These operations may be properly applied on a port object in order to realize the messages exchange with remote ports. We briefly outline these operations.

- `AsyncWrite(Msg, MsgKey)`: It will write the Msg message object to a port along with a key for personalizing the message.
- `AsyncRead(MsgKey)`: It signifies the willingness of the calling task to access a certain message object expected at this port. It returns a reference to a port list object where the expected message will be deposited upon arrival.
- `PortDataReady()`: It is an accessor method to the objects of the port list. When a message has arrived at a port this method returns a reference to the object where the incoming data resides; otherwise it waits until the message arrives.

The communication protocol between two JavaPorts task threads residing on different address spaces is outlined in moderate detail below. We only describe the communication operation in one direction, from the Manager (sending thread) to the Worker (receiving thread), since the operation in the reverse direction is symmetrically identical. A snapshot of such operation is illustrated in Figure 4.

The Manager task thread issues an `AsyncWrite` to its associated port with a user-defined key. This key will identify the data on the remote side. The `AsyncWrite` method spawns a secondary thread, the `PortThread`, and returns. This thread will be responsible for transporting the data to the remote port asynchronously and without blocking the Manager task thread. The `PortThread` calls the `receive` method of the remote port object. This method makes use of the RMI serialization mechanism in order to transfer the message to the remote node. The incoming data will be inserted in the appropriate object element of the remote port list.

On the receiving node, the Worker task thread issues an `AsyncRead` to its associated port using the corresponding key of the expected incoming data. The method will return a reference to the specific object in the port list where the incoming message will be deposited upon arrival. At the point in time when the Worker thread needs this data, it calls the method `PortDataReady` on the specified list object and then enters the wait state. The `receive` method, which the Manager `PortThread` calls during the transportation of data, inserts the incoming data to the corresponding list object and then notifies all the waiting threads in the address space of the receiving node. Once the Worker thread is notified, it exits the wait state and checks whether the incoming data has the correct key. If not it re-enters the wait state, otherwise it retrieves the data.

3 The JavaPorts Application Development Environment

3.1 Application Design Flow

In this section we describe a procedure that facilitates incremental development of Java-based parallel applications in the JavaPorts environment. Each phase of the procedure corresponds to a step in the application development cycle. Upon completion of each step the user may checkpoint the correctness of the current portion of the application before advancing to the subsequent step.

The procedure makes use of the JavaPorts Application Configuration Tool (JACT). JACT translates the user's task graph specifications (captured in a configuration file) into Java code template files for the specified tasks. At a later stage in the design, these files will be completed by adding the needed user code that implements the specific targeted algorithm. All phases of the application design flow are depicted in Figure 5 and described below:

• Phase 1: Define a Task Graph.
This is the initial stage of the development cycle. The user may use any text

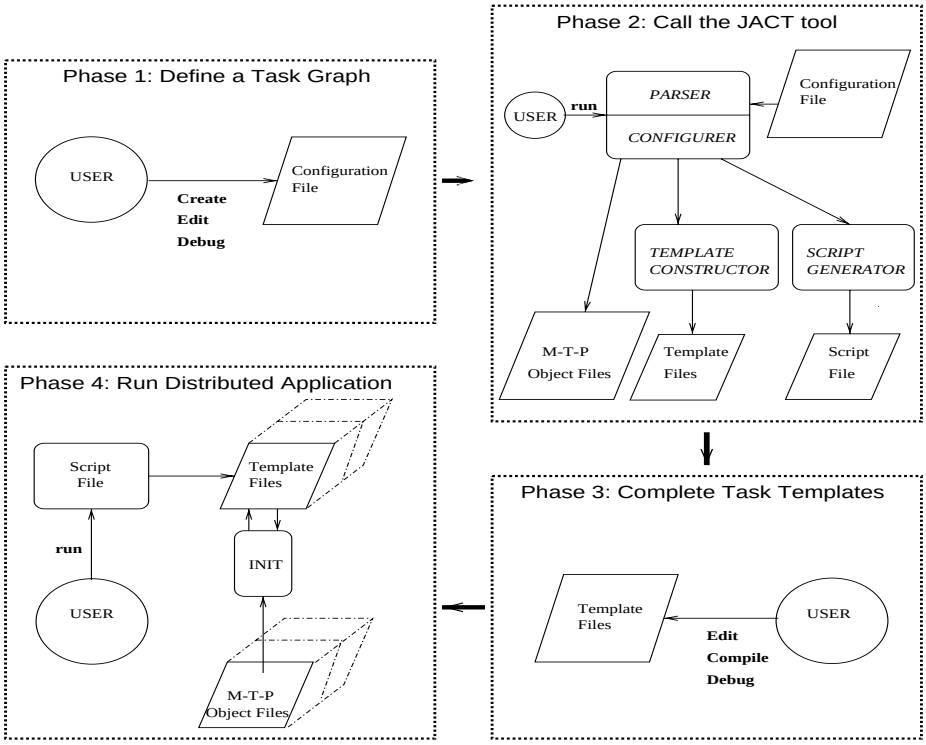


Fig. 5. Application Design Flow.

editor in order to create and save the configuration file. The name of the file is user-defined and should be provided as an input to JACT in Phase 2.

• Phase 2: Call the JavaPorts Application Configuration Tool. During this phase, the user will make calls to the JACT, in order to compile the configuration file and subsequently create the necessary templates for each defined task. The JACT is primarily composed of a Configurer, a Parser, a Template Constructor and a Script Generator.

The Configurer is the central module of the JACT and controls the operation of the rest of the modules in the tool. First the Configurer calls the Parser which compiles the configuration file and alerts the user for syntax or configuration errors. The syntax errors are due to incorrect usage of the configuration language commands. JACT will return informative messages about such errors to guide the user towards fixing them. Configuration errors occur when the user violates certain rules, for example when there are missing machine or task definitions, or the M-T-P property of the system does not hold and thus an incorrect scenario is specified (see for more details section 3.2). Once the user configuration file has been successfully parsed, the Configurer will automatically store the information needed for each machine into a separate object file. These files will be later used

by the JavaPorts system in order to create and register the necessary ports per task.

At this point, the Configurer will call the Template Constructor module which is responsible for generating or updating the templates files. Each template file corresponds to a task with the same name, specified by the user in the configuration file.

As a last step, the Configurer will call the Script Generator to create a startup script that can be used to launch the distributed application from a single (master) node. As an example, the simple script generated automatically for the star topology described by Figure 1 is given below:

```
setenv CLASSPATH $HOME/public-html/codebase
rsh walker setenv CLASSPATH $HOME/public-html/codebase
rsh hawk setenv CLASSPATH $HOME/public-html/codebase
rsh walker.cdsp.neu.edu java JACT.Worker1 walker
rsh walker.cdsp.neu.edu java JACT.Worker2 walker
rsh hawk.cdsp.neu.edu java JACT.Worker3 hawk
java JACT.Manager corea
```

• Phase 3: Complete the Task Template Programs.

During this stage the user will complete the template programs by adding the necessary Java code into each template file in order to implement the desired algorithm. These templates will be compiled using the Java standard compilers. The Java Virtual Machine running on the current workstation will inform the user about any syntax errors. The user will not be aware of potential runtime errors in the algorithm's implementation until the next stage of the design.

• Phase 4: Running the Distributed Application.

This is the final stage of the design flow. The user will reach this stage once the completed template files have been successfully compiled. The user may start the JavaPorts system and run the current application by calling the startup script which was created in Phase 2. In a UNIX environment the script will be already pre-compiled using the `source` system command. The user may simply type the name of the script at the shell prompt in order to run it. The script file will initiate each task on its corresponding machine. Subsequently each task will instantiate an object of class `Init` that will read the necessary configuration information from the corresponding M-T-P Object file and will update the ports configuration and their connections. At this stage the user is likely to be faced with runtime errors due to network communication failures among machines, null objects, deadlocks etc. The JavaPorts system is designed to utilize the rich set of exceptions that the Java programming language supports and inform the user of such problems when they are detected.

3.2 The M-T-P Tree files

The Configurer of JACT stores the information acquired from the configuration file into Tree Structures Object files. This information will be used to

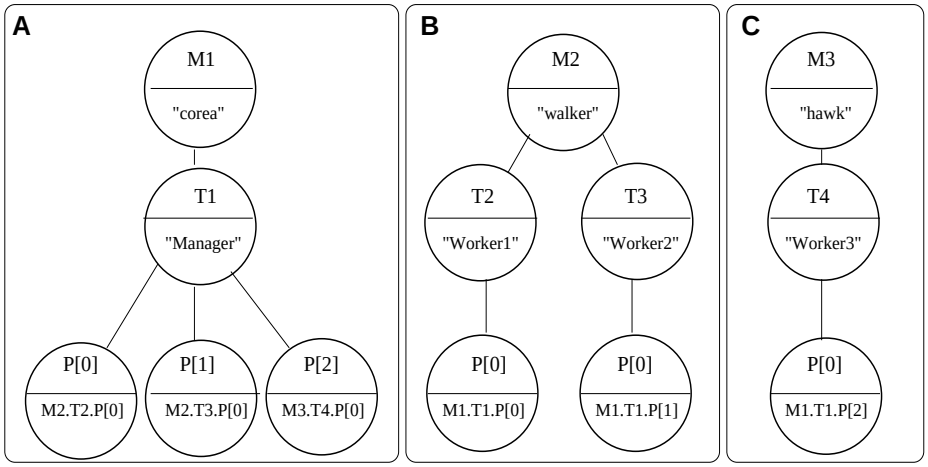


Fig. 6. M-T-P Object Files.

achieve port configuration in a manner transparent to the user. The Machine-Task-Port triplets are organized in tree data structures. The root of a tree is always a machine; the first level nodes are tasks and the leaf nodes are ports. (Hence the name M-T-P tree).

Each node at any level of the tree carries certain properties. A machine root node includes the `machine_variable` assigned to a specific machine by the user at configuration time as well as the `machine_dns_name`. A task node stores the `task_var` and `task_name` assigned to it and a port node stores the connection data comprised of the M-T-P triplet of the matching port. Figure 6 displays the three Object files which the JACT will create for the Task Graph depicted in Figure 1. The Object File A corresponds to machine M1, Object File B to machine M2 and C to machine M3 respectively. The task nodes represent the tasks defined in the corresponding configuration file and the leaves the ports which connect tasks. The connection data for each port as well as the information data for the machines and tasks are shown inside each node (circle).

JACT performs all necessary checking during parsing so that the data stored onto each tree represents legal and correct configurations. The M-T-P policy requires that each component of the tree be added through the configuration file. Furthermore, the root of a tree should always be a machine node, the tasks always reside at the first level nodes and all ports will always be at leaf nodes. If no tasks are defined for a machine, a tree file will be created for it including only the root node carrying the machine information. Every time a configuration file is altered JACT should be called to parse it and update the tree object files, as needed.

The JACT tool makes use of the object serialization and de-serialization procedures provided by the Java language. During the serialization process, all data types and graphs of the M-T-P objects are flattened and written to an output

stream which in turn is stored persistently into a file. During the de-serialization process, the `INIT` object of each task process will open the corresponding file, read the object from the input stream and reconstitute it to its original form. The `INIT` object will extract all the necessary information from the tree which belongs to the current task in order to create, register with RMI and connect the corresponding ports.

4 Experimental Results

In this section we outline some experiments conducted using the JavaPorts system and present only few indicative performance results. A more detailed evaluation is outside the scope of this paper and will appear elsewhere [13]. As a first illustrative example we have chosen the matrix-vector multiplication problem applied to a variety of two-node configurations, since the same scenario was also chosen by other researchers (see [3] and [14]). The multiplicand is assumed to be a 1000-by-1000 square matrix and the multiplier an 1000-by-1 column vector. The multiplication is conducted by two concurrent tasks (a manager and a worker) allocated on two nodes (Sparc-4 workstations by Sun Microsystems running Solaris version 2.5.1 and belonging to a local 10Mb/s Ethernet subnet). The part of the multiplicand matrix needed by the worker task resided on the remote node before the commencement of the experiment. The communication overhead, i.e. the time it takes for the multiplier vector to be sent from the manager to the remote worker task as well as the time it takes for the partial product vector to be returned back from the worker to the manager has been measured and taken into consideration in the timing analysis.

In the two-node scenario, the number of rows of the multiplicand matrix allocated on each one of the two nodes was varied from 0 to 1000. The manager task starts first, sends the multiplier vector to the remote worker, then performs its local computation and finally collects back the partial result returned by the worker and assembles the overall product vector. In the left panel of Figure 7, the solid line curve shows the total time needed by the manager task to complete as a function of the number of rows allocated to the remote worker task. The dashed line curves show the times needed for the computation of the local partial product in the manager and the time the manager should wait subsequently for the remote partial product vector to arrive from the worker task. In essence, the decomposition of the total manager time to its major sequential components verified that as the work allocated to the remote worker increases the manager local computation time decreases and the waiting time (for worker results) increases. From the analysis of the results we observe that the minimum overall time is achieved close to the point where the rows of the matrix are split evenly among the two machines. This minimum time corresponds to 62% of the time needed to solve the problem (1000 rows) using a single thread Java program in one machine. Therefore, the achieved speedup by using two concurrent tasks in two nodes is about 1.6, a very good result considering that the network is a standard 10Mb/s shared Ethernet and the problem (message) size not too large.

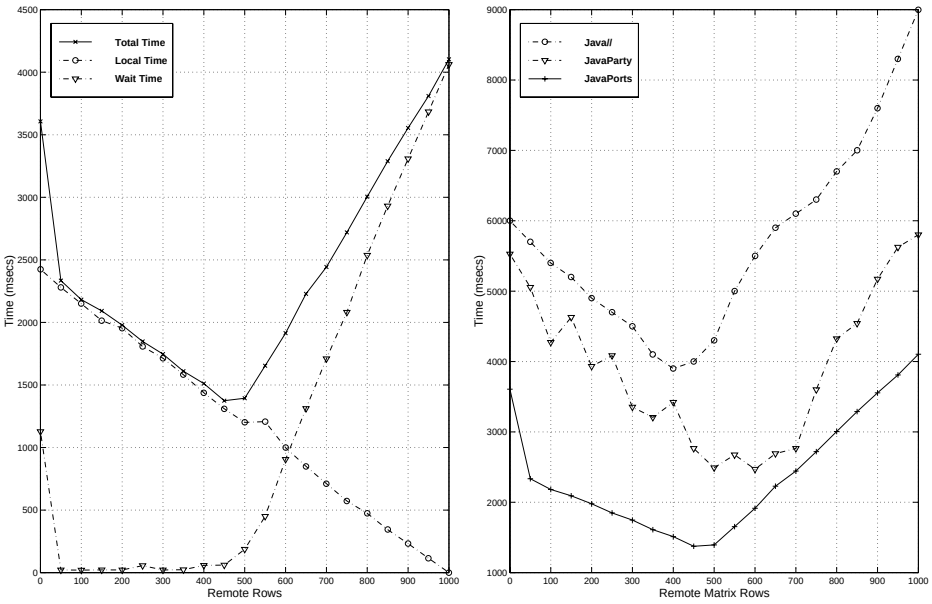


Fig. 7. *Left panel:* Time of the manager vs. rows allocated in the remote worker in a two-node scenario; *Right panel:* Comparison of three approaches using the same matrix-vector multiplication example.

For the two-node scenario, the JavaPorts performance was also compared to that of two other major designs, the Java// [3] and the JavaParty [6]. Similar to our system, the JavaParty application was ran on two Sparc-4 nodes whereas the Java// experiment made use of two UltraSparc stations according to [3]. To be able to obtain the JavaParty times, a single thread Java program was written. Following the guidelines of the JavaParty group, the worker task was declared as remote, thus allowing the system to migrate the worker task object to the remote node. As it can be seen from the plot in the right panel of Figure 7, the minimum time was achieved at approximately the same number of remote rows for all three designs. Among them, the JavaPorts design exhibited the best performance across the full range of rows allocated to the remote worker node.

5 Conclusions

We have provided an overview of the JavaPorts project, an environment for flexible and modular concurrent programming on cluster architectures. The design encourages reusability by allowing the developer to build parallel applications in a natural and modular manner. The components generated automatically by the system may be manipulated and (without modifying any part of the user code) they can be utilized to run the same concurrent application in a new cluster setup. The user-defined message object may be altered to customize

the needs of transferring any type of data across the network. By giving the developer such capabilities, the JavaPorts environment may be customized for specific client-server applications where a large variety of services may be available. We managed to keep the task of tracking remote objects and marshalling of data between different address spaces transparent to the developer. The user is not aware of how the data is communicated among nodes in the cluster.

Currently there is a considerable amount of on-going work. The completion of the graphical tool for allowing the developer to enter and modify a task graph in the JavaPorts system is our top priority. This tool will provide the user with a visual sense of how the application is configured to run on the cluster. This is an important aid in parallel computing where partitioning data and functionality among nodes is much easier to understand when it is depicted visually. In addition we are improving the configuration language which provides a text based approach to the allocation of tasks to nodes. Experienced users may find this entry point more convenient for the development and customization of their application.

We are also targeting large scale applications using a variety of commonly employed task graph patterns (star, pipeline, mesh, cube). In addition to compute-intensive computations we are considering other application domains, such as distributed network management, distributed simulation and networked embedded systems that may take advantage of the easy configuration capabilities of the JavaPorts system. Some of these applications are selected to allow us to fully test and optimize specific aspects of the JavaPorts design before it is released for general use.

References

1. D. Galatopoullos and E. S. Manolakos. Parallel Processing in heterogeneous cluster architectures using Javaports. *ACM Crossroads*, (1999, to appear).
2. K. Arnold and J. Gosling. *The Java Programming Language*, (1996) Addison Wesley Longman, Inc.
3. D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, **10** (1998) (11–13):1043–1061.
4. Proactive PDC - Java//. http://www.inria.fr/sloop/javall/index.html, (1998).
5. P. Launay and J. Pazat. Generation of distributed parallel Java programs. Technical Report 1171, IRISA, France, (1998).
6. M. Philippsen and M. Zenger. Javaparty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, **9** (1997) (11):1225–1242.
7. B.O. Christiansen et al. Javelin:Internet Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, **9** (1997) (11):1139–1160.
8. Dimple Bhatia et al. WebFlow–A Visual Programming Paradigm for Web/Java Based Coarse Grain Distributed Computing. *Concurrency: Practice and Experience*, **9** (1997) (6):555–577.
9. Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems.* McGraw Hill, (1998).
10. F. Arbab. The IWIM model for coordination of concurrent activities. *Coordination '96, Lecture Notes on Computer Science*, **1061**, (1996).

11. F. Arbab, C.L. Blom, F.J.Burger, and C.T.H.Everaas. Reusability of Coordination Programs. Technical Report CS-TR9621, Centrum voor Wiskunde en Informatica, The Netherlands, (1996).
12. Remote Method Invocation Specification. Sun Microsystems, Inc., (1996).
13. D. Galatopoullos and E. S. Manolakos. JavaPorts: An environment to facilitate parallel computing on a heterogeneous cluster of workstations. *Informatica*, (1998, submitted).
14. R.R. Raje, J.I. William, and M. Boyles. An Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, **9** (1997) (11): 1207-1211.