Performance engineering of coarse grain network computing applications based on task level behavioral modeling

Feras Al-Hawari a Elias Manolakos a,*

*Electrical and Computer Engineering Department, 442 Dana Research Building, Northeastern University, Boston, MA 02115

Abstract

We developed an interactive framework that automates the mapping of application tasks to Networks of Workstations (NOWs) and facilitates application adaptation at runtime. Part of this framework is a new tool used to construct graphically behavioral models for the application tasks and connect them structurally to build complete, two-level, hierarchical network computing application models. These models can be used for performance engineering and rapid prototyping of network computing applications before any code is developed. We also introduce a performance estimation scheme that uses hierarchical application models and static/dynamic resource state related data to predict accurately the expected total running time of a given distributed and multi-tasked application configuration. We conducted several experiments that demonstrate the accuracy of the obtained performance predictions in different kinds of scenarios.

Key words: Performance estimation, network computing, QoS management, parallel applications, performance engineering

1 Introduction

Networks of Workstations (NOWs) have become an increasingly attractive parallel processing platform for tackling compute intensive problems over the last decade. The wide availability of inexpensive PCs and the advent of off-the-shelf fast network technologies allow NOWs to offer a much better cost/performance ratio than supercomputers. Despite this reality, building distributed component-based applications for NOWs is
still a painstaking experience. Frameworks and tools that can help designers to model, simulate, build, and run efficiently coarse grain parallel applications will certainly contribute to the growth of NOW’s popularity and user base [1].

The resources of a NOW are usually heterogeneous and shared, making the system’s state quite dynamic. A network computing (NC) application that exhibits performance gains under light load conditions (relatively to a sequential program realization) may not enjoy any speedup in the same environment under heavy load conditions. If the objective is to deliver a targeted Quality of Service level (QoS) to an application (e.g. in terms of expected completion time or speedup), the system’s state should be taken used in finding the best acceptable mapping of the parallel application’s tasks to the available NOW resources before the application is launched (i.e. at startup time). Furthermore, the application should remain resource state aware and possibly adapt itself accordingly in order to keep meeting its QoS demands at runtime.

The task of monitoring the system’s state and providing information that can be exploited at startup as well as during the application’s runtime can be delegated to an application-level QoS management system. The main requirements for such a system are to be able to find automatically the most efficient tasks-to-machines mapping at startup time, and in addition facilitate application adaptation at runtime, while remaining transparent to the user. Our group has developed such a flexible QoS management system for coarse grain NC applications in the context of the ongoing JavaPorts framework project [1,3]. In this paper we present the design and main features only of the startup phase QoS management components.

The startup-phase QoS management system (see Figure 1) consists of a front- and a back-end subsystem. The front-end subsystem modules provide an interface between the user and back-end modules. They are user-friendly graphical tools that help a developer construct intuitive network computing application models, build an available resources database, launch the QoS management system in a NOW, specify QoS requirements for the distributed application, and send requests to (or get results from) the back-end modules. At the back-end, a QoS Manager module interacts with local and remote Resource Monitoring Agents and a Performance Estimator in order to accurately, quickly, and automatically find if there exist a tasks-to-machines allocation strategy that satisfies the QoS targets set by the user. The resources monitors are lightweight modules that measure and communicate to the QoS Manager dynamic resource state information (such as the current workload of each available machine, throughput of the network links etc). The Performance Estimator uses: (i) a structural top-level model describing how the tasks interact in the NC application, (ii) a behavioral model for each task involved, and (iii) NOW resource condition related data, and provides an estimate of the expected performance of a mapping under investigation.

In this paper, first we present the design of the JavaPorts Visual Task Composer (JPVTC), a set of algorithms integrated into a graphical tool which can be used, in conjunction with the JavaPorts Visual Application Composer (JPVAC) [2], to construct hierarchical, two-level structural-behavioral network computing application models. We then present methods for estimating the performance of different application task configurations based on the constructed application models i.e. before any coding is attempted. These methods are implemented in the Performance Estimator tool and can help the designer (i) understand the behavior of the interacting tasks of an application under different resource conditions, and (ii) detect potential deadlock situations. The whole QoS system is designed to facilitate the rapid prototyping and performance
2 Background and Related Work

2.1 The JavaPorts Distributed Components Framework

The JavaPorts (JP) project [1,3] has provided a components framework and a set of tools for the modeling and design of distributed multi-component applications for NOWs. The JP framework provides abstractions for developing and composing components (called tasks) and APIs for anonymous message passing between them while also hiding the inter-task communication and coordination details from the application designer. The JP tools facilitate the modeling, development, configuration, and deployment of coarse grain parallel and distributed JP applications. In addition, a unique feature of the latest JP version 2.5 is that it supports the co-existence and interaction of reusable Java and Matlab components in the same application.

A JP network application is composed by multiple, concurrent and possibly interacting tasks. Its structure is captured by using Application Task Graph (ATG) abstraction, where nodes correspond to tasks (com-
Fig. 2. (a) An Application Task Graph (ATG) example; (b) the corresponding JPCL textual description; (c) the corresponding AMTP data structure representation. See text for details. (section 2.1)

The ATG can be captured either textually, using the JP Configuration Language (JPCL), or graphically using the JP Visual Application Composer (JPVAC) tool [2]. The ATG is represented as a four-level Application-Machine-Task-Ports (AMTP) tree data structure. A snapshot of an ATG generated using the JPVAC tool is shown in Figure 2(a). The corresponding JPCL textual description and AMTP tree data structure are shown in Figures 2(b) and 2(c), respectively.

The JP Application Configuration Toolset (JPACT) is used to generate Java or Matlab JP code templates (executable code skeletons) for every task defined in the ATG, based on the parsed configuration file, as well as scripts  for compiling and launching the distributed application effortlessly from a user-specified machine in the network (called the "Master" machine). The user needs to add the task specific code to complete the automatically generated templates. The generated scripts can be used to compile the completed templates and to launch the distributed application from the Master (machine M1 in the example of Figure 2).

A JP task may communicate with a peer task using anonymous message passing. In anonymous communications the name (and port) of the destination task does not need to be provided explicitly in the message passing method call [1]. A list data structure is maintained for each JP port for buffering messages. Each port list consists of elements uniquely identified by keys. A message key is used to identify the port list element to be used for buffering when writing/reading a message to/from a port. The JP Ports communication API is very simple, consisting of only four methods:

\[\text{1 Currently Linux and Solaris clusters with/without NFS are supported.}\]
Fig. 3. (a) The ATG for a Manager-Worker example; (b) code snippet showing how the Manager and the Worker components may use the anonymous JP communications API to exchange messages.

1. `public Object AsyncRead(int MsgKey)`: Does not block the calling task. Returns a handle to a message if the message has already arrived at the port list element identified by the specified key, otherwise it returns null.

2. `public void AsyncWrite(Object msg, int MsgKey)`: Does not block the calling task. Spawns a new thread to transfer and store the message in the peer task's port list element with the specified key.

3. `public Object SyncRead(int MsgKey)`: Blocks the calling task until a message arrives at the port list element with the specified key.

4. `public void SyncWrite(Object msg, int MsgKey)`: Blocks the calling task until the sent message is written to the peer port's list element with the specified key.

In Figure 3(a), the ATG for a simple Manager-Workers application is shown. In Figure 3(b), two code snippets are provided to demonstrate how the anonymous message passing JP API can be used to exchange messages. The Manager first calls the non-blocking `AsyncWrite` method on its port[1]. This will result in writing a message, personalized by `key1`, to the respective list element of the peer port, i.e. port[0] of the Worker task to which port[1] of the Manager is connected. Subsequently, the Manager may be blocked (synchronous read) until a message it expects has arrived at its port[1] list, `key2` element. On the other side, the Worker task may be blocked (synchronous read) until the message sent by the Manager's `AsyncWrite` is deposited in its port[0] list, `key1` element. After receiving this message the Worker calls the blocking `SyncWrite` operation on its port[0], which results in adding a message in the `key2` element of the Manager's port[1] list. The potentially blocked Manager (at the `SyncRead`) and Worker (at the `SyncWrite`) are released (rendezvous point) when the Manager reads the message identified by `key2` from its respective port[1] list element.

### 2.2 Related Work

Several research groups, such as AppLeS [4], Condor [5], GrADS [6], and Prophesy [7], developed performance models for network computing applications. The existing models can be categorized into four types: equation based [8–14], graph based [15], Petri Nets based [16–19,18], and skeleton based [7,20,21]. Equation based
methods use cost functions that depend on application and resource information to estimate the performance of a distributed application. Such models are not easy to construct and are not generic enough to describe all application classes (e.g. some models are only suitable for pipeline [14] or master-slave [11] type applications). Moreover, most of them do not support multi-tasking i.e. more than one task allocated to the same machine, a very realistic situation often arising in large scale computing.

A graph based model, similar to our behavioral graphs, is introduced in [15]. However, this method does not support asynchronous communication operations and loops. Conversely, the Petri Nets based models, such as those in [16,18], can capture the basic code constructs in a task, including synchronous/asynchronous communication operations, but they can not model anonymous message passing operations and tend to over-estimate performance, because they assume that the tasks have exponentially distributed residence times.

Skeleton based models, such as those in [7,21], are short running, synthetically generated, programs that capture the execution behavior of an application. Thus, this approach requires executing the skeleton code when a performance estimate is needed, which can be inefficient due to the startup and contention delays that this short application may incur in a dynamic and heterogeneous NOW. Unlike previous performance estimation methods, our method can estimate the performance of distributed, multi-tasked and multithreaded applications based on structural and behavioral models, as well as static and dynamic resource data. It can also account for queuing and synchronization delays and queuing effects of other applications. Moreover, it can detect a deadlock, if encountered.

Several visual programming tools, such as those reported in [22–27], were introduced to generate automatically executable parallel code from a graphical representation of the application. These tools neither address the performance engineering activities during the application’s development cycle, nor do they focus on QoS related issues. They mostly support the graphical representation of the application’s structure. Few of them attempted to also capture the behavior of individual application tasks. For example the tool discussed in [28] focuses on performance estimation and attempts to model the behavior of the tasks using Petri Nets. However, it does not model anonymous message passing operations.

3 Task Behavioral Models

In this section we present an overview of the JavaPorts behavioral task modeling methodology. A (behavioral) task graph consists of nodes modeling basic code constructs and edges for defining the nodes execution ordering. It describes the general organization of a task as a sequence of computation and communication elements while abstracting low level details. Several task models can be composed to form a two-level hierarchical representation of a multi-component application that can be used: (a) to perform "what-if" performance analysis and QoS evaluation of different task configurations, and (b) to generate executable Java and Matlab code templates (skeletons) for each task. The JP Visual Task Composer (JPVTC) is a tool that we have recently added to the JavaPorts suite, to help application designers construct graphically task behavioral models.
Table 1
The basic JP task graph elements and their attributes.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Attribute(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>codeSegment</td>
<td>Average execution time</td>
<td>Sequential code block; in time units on a reference machine.</td>
</tr>
<tr>
<td>fork</td>
<td>-</td>
<td>Spawn a new thread.</td>
</tr>
<tr>
<td>beginIf</td>
<td>Probability of entering a block</td>
<td>Begin of a conditional block.</td>
</tr>
<tr>
<td>endif</td>
<td>-</td>
<td>End of a conditional block.</td>
</tr>
<tr>
<td>beginLoop</td>
<td>Number of iterations</td>
<td>Begin of a loop block.</td>
</tr>
<tr>
<td>endLoop</td>
<td>-</td>
<td>End of a loop block.</td>
</tr>
<tr>
<td>AsyncWrite</td>
<td>Data size (Kbytes), port, message key</td>
<td>Non-blocking write.</td>
</tr>
<tr>
<td>SyncWrite</td>
<td>Data size (Kbytes), port, message key</td>
<td>Blocking write.</td>
</tr>
<tr>
<td>AsyncRead</td>
<td>Port, message key</td>
<td>Non-blocking read.</td>
</tr>
<tr>
<td>SyncRead</td>
<td>Port, message key</td>
<td>Blocking read.</td>
</tr>
<tr>
<td>beginAsyncReadLoop</td>
<td>Port, message key</td>
<td>Begin of an AsyncRead loop block.</td>
</tr>
<tr>
<td>endAsyncReadLoop</td>
<td>-</td>
<td>End of an AsyncRead loop block.</td>
</tr>
</tbody>
</table>

Fig. 4. (a) The basic JP task modeling elements and their symbols. (b) Task graph with nested loops with AsyncWrites where the AW port and key depends on the loop indices. (c) Task graph modeling an AsyncRead loop.

As with any Java program, a JP task may include sequential code blocks, iteration constructs (loops), conditionals (switch and if statements). Moreover, it may spawn new threads and contain message passing port operations. Our modeling methodology and the JPVTC tool support the necessary basic elements for modeling JP code constructs. Elements are annotated with attributes used in performance estimation. The currently supported elements and their attributes are listed in Table 1. Each element has its own symbol within the JPVTC; a snapshot of these symbols is

Task models may also include loop bodies with message passing operations in which the ports and/or message keys are not fixed but depend on the loop indices (to be called parameterized ports and keys). An example is provided in Figure 4(b). In this example, the port on which the AsyncWrite method is called
and the message key value it uses vary in each iteration and depend on the indices of loops bloop1 and bloop2 respectively. The JPVTC tool user can specify these loops in the AsyncWrite element attributes form launched by right clicking on the element, as shown in Figure 4(b). This kind of loops structure is convenient e.g. for modeling a Manager-Workers interaction pattern, in which the load is partitioned among several Workers and the Manager needs to send (receive) several personalized messages to (from) each Worker. Supporting parameterized elements reduces the complexity of the constructed graphs, since a loop with a single communication operation is in this case equivalent to an one-to-many communication primitive (scatter). All supported communication elements can be parameterized.

Another useful structure is the AsyncRead loops modeled using the beginAsyncReadLoop and the endAsyncReadLoop elements as shown in Figure 4(c). Such a loop is exited when the AsyncRead method call to a specified port finds that a message has been deposited in the port’s list element matching the current value of the AsyncReadLoop’s key. This kind of polling loop operation is handy for modeling tasks that perform useful work while also checking periodically for a message arrival in any one of its ports. AsyncReadLoops can be nested within regular loops possibly including other parameterized communication elements, thus providing great flexibility in modeling non-trivial behaviors with task graphs of manageable size.

A behavioral task graph is implemented internally as a linked list of basic elements. An element is realized as a structure storing its attributes(type, ID, (x, y) coordinates, annotated performance data, etc) as well as pointers for easy access to parent and children elements in the list. The data structure representing the task graph of Figure 5(a) is shown in Figure 5(b). A task graph is considered invalid if it contains cycles i.e. there exist a path that starts and ends at the same element, or it is disconnected i.e. it consists of more than one group of connected elements that are separate.

A structural (application-level) and several behavioral (task-level) graph models should be linked in order to build a network computing application model. This can be easily accomplished using the JPVTC tool by choosing the Assign Graph item under the Tools menu. This action launches a JP application browser.
that allows selecting an application. The AMTP tree of the selected application is then loaded and a list of its tasks pops up. Upon choosing a task, the active behavioral graph is associated with it. In addition, the application AMTP tree becomes associated with the active behavioral graph. The AMTP tree will be used to extract the task’s ports list as needed for the annotation of a port used in communication operations. The same procedure is repeated until all tasks are associated with a behavioral graph.

Behavioral task graphs can be saved in an eXtensible Markup Language (XML) format defined in a graph Document Type Definition (DTD). The DTD consists of a list of elements including the behavioral graph elements and an arc element for capturing connectivity information. Every element has a unique identifier. All elements, except the arc element, have a location statement to specify their placement in the internal-frame. In addition, most elements have lists that capture their annotation and attributes. Figure 5(c) provides the XML description of the behavioral graph shown in Figure 4(b).

4 Performance Estimation and Deadlock Detection

The proposed Performance Estimation module (see Figure 1) takes as input: the AMTP tree that captures the multi-tasked application structure (generated by the JPVAC tool), the task behavioral graphs (constructed using the JPVTVC tool), and information related to the state of the NOW resources). As we will see in this section, for a given tasks-to-machines mapping, the proposed performance estimation method accounts for the execution, queuing, and synchronization delays of all tasks in the distributed application. Although the JP QoS system can find good candidate mappings automatically, discussing how this is accomplished is out of the scope of this paper and will be discussed elsewhere. For the rest of this paper we will assume that the candidate mapping is determined before performance estimation is initiated. The evaluation of another mapping requires only the modification of the AMTP tree.

A high level overview of the proposed performance estimation and deadlock detection scheme is provided by the flowchart of Figure 6(a). At initialization time, port list data structures are generated according to the point-to-point connections between peer ports as specified in the AMTP tree. Port lists are needed to model the JP anonymous message passing operations. Moreover, for each machine a ready queue is created, as needed to model the task element queuing delays. Data structures are also generated for each task graph. The first basic element in each task graph is inserted into the appropriate machine queue. If tasks are allocated to the same machine (multi-tasking) their elements are inserted into the same machine queue. The expected execution time of an element is recalculated, if necessary, taking into account the static and the latest dynamic information available for a machine, before it is inserted in its queue.

The algorithm loops over all machine ready queues. A queue is updated (i.e. the elements that completed execution are removed and the elements that depend on them (children) enter the queue) at the beginning of each iteration. The elapsed times (i.e. queuing and execution delays) of elements in the queue are calculated at the end of each iteration. The calculated elapsed time for the current iteration is added to the total running time of the corresponding task(s). Then another iteration over the same queue resumes. The algorithm exits the loop of a machine ready queue and moves to update the next machine queue when either the current
Initialization
begin for each machine
  Queue empty?
  Application complete?
end for each machine
Update machine queue
Calculate elapsed time of element(s) in queue
Resolve SYNC COMPLETE
YES
NO
YES
NO
Deadlock?
YES
NO
DEADLOCK
START
NEXT PASS OVER ALL MACHINES
NEXT ITERATION OVER QUEUE
SYNC and wait?
NO
(a)

AGGREGATE
DONE
SYNC
DONE or DEADLOCK
DONE
DEADLOCK
SyncRead or SyncWrite that sent its message visited
Synchronization resolved or Schedule parallel threads
Expected Message never arrived
Start
(b)

Fig. 6. (a) High level overview of the performance estimation and deadlock detection scheme; (b) the task states diagram.

queue becomes empty or a task in it is waiting for a synchronization event to occur. Upon exiting the loops of all machine queues (i.e. upon completion of the current pass), the algorithm tries to resolve any pending synchronization events, checks for deadlocks and exits if one is detected. If the application did not complete, another pass starts by looping over all machine queues once more. This process continues until the application completes or a deadlock is detected. At the end, the estimated application total running time corresponds to the completion time of the task with the longest delay.

A task can be in one of the following states: AGGREGATE, SYNC, DONE, or DEADLOCK, as shown in the task state diagram of Figure 6(b). A task is in the AGGREGATE state, if it is not waiting for a synchronization event. A task is in the SYNC state, if it has reached a SyncRead element that is waiting for a message arrival or a SyncWrite element that is blocked waiting for the sent message to be read. A task is in the DONE state, if all its model’s elements have been executed and removed from the queue. A task is in the DEADLOCK state, if a waited upon message never arrives at one of its ports. A task has to be at the DONE or SYNC states at the end of a pass. A task that never enters the SYNC state transitions to the DONE state in one pass (but possibly after many iterations over the machine ready queue). On the other hand, a task with synchronization elements needs more than one pass to complete.

The complexity of this algorithm is $O(P \times I \times M)$, where $P$ is the number of required passes, $I$ is the expected number of required iterations over a machine queue for a pass, and $M$ is the number of machines queues. The number of passes depends on the number and order of execution of the synchronization elements (i.e. when there are no synchronization elements in any of the behavioral graphs $P=1$). The number of iterations is proportional to the expected number of elements in the task behavioral graphs assigned to each machine. The number of queues is the same as the number of machines in the ATG. Usually the number of machines
4.1 Delay Modeling and Calculation

The total running time of a task includes execution, queuing, and synchronization delays. The execution part accounts for the actual service time of each one of the basic code constructs in the task's behavioral model. The queuing delay accounts for the time that a task spends waiting for service in the machine ready queue. The synchronization delay models the time that a task is blocked on message-passing operations, waiting for synchronization events to occur. To estimate accurately the total running time of a parallel and distributed application, all types of delays should be accounted for accurately. Furthermore, the distribution of the delays for each task should be available after the end of the performance evaluation.

4.1.1 Execution Delays

The calculation of the elements execution time takes into account the annotated performance data, the latest system’s state, and the tasks onto machines mapping. Certain elements are annotated using performance data
in a reference (benchmark) machine. These elements are divided into two groups: the first group includes the codeSegment, beginIf, and beginLoop elements; the second group includes the SyncWrite and AsyncWrite elements.

The codeSegment element is annotated with the expected execution time of the code block it represents in a reference (benchmark) machine (see Table 2). Since coarse grain code segments usually correspond to one or more subroutine calls, the execution time of a block can be measured on a reference machine (e.g. the fastest machine in a cluster) in non-shared mode. If codeSegment element is part of a task allocated to a slower machine, its execution time is automatically scaled down. Furthermore, in order to account for potential CPU contention effects of other applications, the scaled execution time is also multiplied by a dynamically updated load factor. The QoS Monitoring Agents are used to estimate the load average of used machines. The average ready queue length i.e. the average number of processes waiting in the ready queue plus the number of processes currently executing is considered as the load factor in this case.

The beginIf element is annotated with the probability of entering the conditional code block. The beginLoop is annotated with the number of expected loop iterations. These attributes are application and not mapping dependent. The average execution time of the SyncWrite and AsyncWrite elements depends on the communicated message size and the throughput of the link connecting the machines on which the tasks containing the communication elements are allocated. The throughput of the links is measured by the monitoring agents at runtime. The formulas used to estimate the execution times of all elements supported by the JPVTC based on benchmark data and runtime measurements are summarized in Table 2.

4.1.2 Queuing Delays

Queuing delay (due to multi-tasking) estimation is based on time sliced scheduling. We assume that the time slot is much smaller than the average execution time of the elements in the queue and that the task context switching overhead is negligible. Based on these assumptions, if there are two elements with the same execution time in a machine queue, the elapsed time (i.e. queuing plus execution delay) of an element is equal to double its execution time (i.e. half of the elapsed time accounts for the time spent waiting in the ready queue and the other half accounts for the actual service time). In order to account for the queuing delays, the elements of multiple threads or tasks that contend for the same CPU are inserted into the same machine queue. In each iteration the algorithm estimates the elapsed times of elements in each machine queue based on the pseudo code of Figure 7(a). In the example of Figure 7(b), we demonstrate how the algorithm is applied to a machine queue that contains three elements, a (codeSegment (cs1) and two AsyncWrites (aw1 and (aw2))). In this example, it is assumed that the three elements in the queue have no children, hence no other element will enter the queue.

4.1.3 Synchronization Delays

In order to account for the synchronization delays and model accurately the behavior of the supported message passing operations the performance estimation scheme mimics the way JP port lists operate. At initialization phase, it builds port lists for each task based on information available in the AMTP tree.
When an Async/Sync write operation is visited, it adds a message to the peer port list with the specified key. The message is stamped with the time the write operation completes its execution, modeling message transmission and determined using the associated formula in Table 2. The synchronization delay is the time a Sync element may have to block. For a SyncWrite, it is the greater of zero and the difference between the time when the sent message is read and the message arrival time (as indicated by its time stamp). A SyncRead removes the message from the port list with the specified key when it arrives. For a SyncRead the synchronization delay is the greater of zero and the difference between the message time stamp and the SyncRead visit time. On the other hand, an AsyncRead removes a message from the port list with the specified key only when its visit time is greater than or equal to the message time stamp and does not encounter any synchronization delay.

Let us consider the simple Master-Workers application whose ATG and AMTP tree are provided in Figures 2(a) and 2(c), to illustrate how the JP port lists and message passing operations are modeled. Figures 8(a) and 8(b) provide the pseudo code and corresponding behavioral graphs for the tasks involved. The Manager task asynchronously sends a message to each worker and then it may block while it waits to receive back a message from each Worker. On the other hand, each Worker waits until it receives a message from the Manager and then it sends back a message to the manager. Based on the AMTP tree, the algorithm initializes
public synchronized void run (){
    // Initialization code.
    port_[0].AsyncWrite(message, key1);
    port_[1].AsyncWrite(message, key1);
    // get message from Matlab worker
    message1 = (Message)port_[1].SyncRead(key2);
    // get message from Java worker
    message2 = (Message)port_[0].SyncRead(key2);
    // Release ports
}

public synchronized void run (){
    // Initialization code.
    message = (Message)port_[0].SyncRead(key1);
    // send message to manager
    port_[0].SyncWrite(message, key2);
    // Release ports
}

function Worker2(AppName,TaskVarName)
    - 
    Import classes
    - Initialization code
    % get message from manager
    message = port_(1).SyncRead(key1);
    % send message to manager
    port_(1).SyncWrite(message, key2);
    - Release ports
    quit;

Fig. 8. Port lists and message passing operations modeling: (a) Tasks pseudo code; (b) tasks behavioral graphs; (c) initial port lists; (d) port list elements upon visiting the *AsyncWrite* operations in the Manager graph; (e) port lists upon visiting the *SyncWrite* operations in the Workers graphs. See text for details.

two port lists for the Manager task T1, and one port list for each Worker, T2 and T3, as shown in Figure 8(c). When visiting the *AsyncWrite* operations in the Manager graph the *key1* elements are generated in the lists of ports T2.P[0] and T3.P[1], as shown in Figure 8(d).

Let us assume that a Manager *AsyncWrite* needs 2 time units (t.u.) to transmit a message. Since the two *AsyncWrites* are executed in the same machine concurrently each one will encounter a queuing delay (while the other one is serviced) so that they both complete after 4 t.u. This completion time is reflected in the time stamp stored along with the message in the *key1* element of the receiving port’s list, as indicated in Figure 8(d). The receiving Worker reads and removes the message from the list. The synchronization delay encountered by the Worker’s *SyncRead* operations is 4 t.u. (i.e. the message time stamp time [4 t.u.] minus the worker’s *SyncRead* visit time [assumed to be 0]). The algorithm dynamically initializes two *key2* elements at port lists T1.P[0] and T1.P[1] upon visiting the *SyncWrite* operations in the Worker graphs, as shown in Figure 8(e). Let us assume that the *SyncWrites* of tasks T3 and T2 take 3 and 2 t.u. to complete respectively. The *SyncWrite* of T3 completes sending the message at time 7 (the *SyncWrite* visit time [time 4] plus the write time [3 t.u.]) and does not need to block because the Manager is ready to read message1 as soon as it arrives. On the other hand, the *SyncWrite* of T2 completes sending the message at time 6 (the *SyncWrite* visit time [at time 4] plus the write time [2 t.u.]) but has to block for an additional t.u. until the Manager is ready to read message2, hence it encounters a 7-6=1 t.u. synchronization delay. This is because the Manager reads message2 after message1.
Using the previous example, let us discuss what happens if the Manager was using AsyncRead operations instead of SyncRead. In this case, the Manager will try to read message1 and message2 at time 4 but with no luck since the messages have not been delivered by the workers yet. Since the AsynchReads are non-blocking the Manager completes. Both workers will eventually enter the DEADLOCK state since the messages sent by their respective SyncWrite operations are never read by the Manager.

### 4.2 Updating Machine Queues

The algorithm determines whether to keep, remove, or add elements in a machine queue based on the type and Remaining Execution Time (RET) of elements already queued. The algorithm skips a fork element and immediately queues its children when it is visited. All fork's children are queued at the same time because children threads execute concurrently with their parent thread in the same machine. The non-blocking behavior of an AsyncWrite element is modeled by queuing it when visited along with its child, if any. Hence, an AsyncWrite is treated as a concurrent thread that shares the same machine resources with its parent thread. The SyncRead and SyncWrite elements remain in the queue until they are allowed to unblock to account for synchronization delays. Other elements are removed from the queue when their RET becomes zero.

Let us use the behavioral task graph of Figure 9 to show how elements are added and removed to a machine queue. For simplicity and without loss of generality we will assume that all graph elements, except fork, have the same execution time. Element cs1 is queued at iteration 1 of the algorithm. At iteration 2, cs1 has been completed, its child fork1 is skipped (because its execution time is zero), fork1’s children cs3 and aw1 are queued, and cs2 is also queued because aw1 is non-blocking. At iteration 3, cs2, aw1, and cs3 have been completed and its child cs4 is queued. Finally, at iteration 4 cs4 has been completed and the queue is now empty (the task graph shown has completed execution). In this example, the task enters the DONE state in one pass (with 4 iterations) because it was never blocked (SYNC state).

---

We assume that new threads launch time is negligible.
4.3 Synchronization Events and Deadlock Detection

A task enters the SYNC state if it should be blocked when a SyncRead or SyncWrite of its element is visited. As a result, all current pass operations in the queue of the machine executing the task should be halted. Halting queue processing is needed to determine if the children of synchronization elements contribute to the queuing delay of other elements, something that depends on when the synchronization event is resolved. Let us call iterationElapsedTime the queue’s expected elapsed time at the end of an iteration, i.e. queue.iterationElapsedTime = queue.CurrentTime + numberOfNonSYNCElements * minRemainingExecTime of the non SYNC elements in queue. The children of sync elements contribute to the queuing delays of other elements if the synchronization event for their parent occurs at time less than queue.iterationElapsedTime.

Let us use the graph of Figure 10(a) to illustrate this situation. Initially, the fork1 element is skipped and its children sr1 and cs1 are inserted in the queue. The queue is updated for the current iteration and the task enters the SYNC state because a SyncRead just entered the queue and we assume that no message is available to be read (task blocked). The operations on this machine queue will be halted until a message arrives at the specified port. Let us assume that this happens after 1 t.u., which is less than queue.iterationElapsedTime = 2 t.u. In this case, the algorithm removes the SyncRead element from the queue, advances the elapsed time of cs1 by 1 t.u. (i.e. cs1 ET = 1, and RET = 1), and inserts cs2 (with ET = 0, and RET = 2) into the queue. This scenario demonstrates the need to halt the operations on the queue until the synchronization event is resolved (see Figure 10(b), left side). If the queue was not halted in this case, the effects of cs2 would not be taken into account in the elapsed time calculation of cs1 thus resulting in its under estimation (2 instead of 3 t.u.). On the other hand, if we assume that the message arrives at 3 t.u., which is larger than queue.iterationElapsedTime, then sr1 and cs1 are completed and removed from the queue before cs2 is inserted. Therefore, in this case the elapsed time of cs1 is not affected at all by cs2 (see Figure 10(b), right side).
At the end of each pass, a task can be either in the DONE or in the SYNC state. The algorithm uses the following techniques to try to resolve synchronization events. First, it forces synchronization elements in the queues to read messages written in port lists during the previous pass. If this fails to change the task's state, or no messages were written, the algorithm will schedule the concurrent non-sync elements in the queue with minimum iterationElapsedTime to move one step in the next pass. This is repeated until all concurrent elements have finished execution or a task has changed state. This action allows any halted writes to deliver their messages, which in turn may result in resolving other pending synchronization events. The application will enter the DEADLOCK state, if the algorithm has scheduled all concurrent elements and some synchronization events are still unresolved.

Let us use the example of Figure 11 to illustrate this mechanism. In this example two tasks T1 (graph1) and T2 (graph2) allocated to two different machines send asynchronously a messages to each other and then read synchronously each other’s message. Let us assume that the AsyncWrite in T1 (T2) takes 1 (2) t.u. to deliver a message respectively. During pass 1, both machine queues enter the SYNC state when visiting their SyncRead operations, which results in halting them thus preventing the AsyncWrites from delivering their messages. The algorithm sets the scheduleConcurrentThreads flag of the queue with the minimum iterationElapsedTime (i.e. queue[1] of T1) to TRUE in order to allow the concurrent element(s) of that
Fig. 12. (a) A conditional block and the state of the probability stack after visiting the second `beginIf`; (b) nested loops and the state of the iterations stack after visiting the second `beginLoop`; (c) a loop within a conditional block and the state of the probability and iterations stacks after visiting the `beginIf` and `beginLoop` respectively; (d) a `SyncWrite` element within a loop block and the state of the iterations stack after visiting the `beginLoop`.

queue to move one step ahead in the next pass, which may result in resolving the synchronization events. This action allows the `AsyncWrite` of T1 to deliver its message in pass 2. At the end of pass 2, the algorithm detects that the `AsyncWrite` of T1 delivered its message and will force the `SyncReads` to try to read the message in the next pass. This results in resolving the `SyncRead` of T2 in pass 3, which in turn allows its child `cs1` to proceed concurrently with the `AsyncWrite`. Note that if the queues were not halted in pass 1, the effects of `cs1` would not be taken into account in the elapsed time calculation of the `AsyncWrite` in T2, thus resulting in the underestimation of the synchronization delay of the `SyncRead` in T1. In pass 4, the `SyncRead` of T1 reads the arrived message and this application segment completes.

In the same example, let us assume that the `AsyncWrite` of T2 uses a different message key, not matching the key of the `SyncRead` element in T1. The sequence of queue operations will remain the same as in Figure 11 up until the end of pass 3. However, in pass 4 the `SyncRead` element of T1 does not find received a message with a key matching its own key. As a result, at the end of pass 4 task T1 enters the DEADLOCK state since the algorithm has already scheduled all concurrent elements in both queues but a synchronization event still remains unresolved.
4.4 Loops and Conditionals

The algorithm maintains two special stacks to handle conditionals and loops. Visiting a \textbf{beginIf} element results in pushing its annotated probability value into a probability stack. This value is popped out of the stack when the matching \textbf{endif} element is encountered. Similarly the annotated expected number of iterations will be pushed into the iterations stack when a \textbf{beginLoop} is visited and popped out when an \textbf{endLoop} is encountered. The values pushed in both stacks are used to calculate a factor (returned by the function \texttt{getFactor}()), that is multiplied with the nominal expected execution time of any \texttt{codeSegment} found within a conditional and/or loop block. Examples on how the stacks are used are provided in Figures 12(a)-(c). Loops with no communication elements are flattened to speed up the algorithm’s execution. However, when a loop block contains message passing operations, we need to iterate over the elements inside the loop block as many times as the number of iterations annotated in the \textbf{beginLoop} element, to account for dependencies between elements and associated queuing and synchronization delays. This is facilitated by also storing in the iterations stack a pointer to the first element in the loop body. Then, visiting an \textbf{endLoop} results in decrementing the loop count and returning control to this element if the count is greater than zero. An example is shown in Figure 12(d).

5 Experimental Validation

To validate our approach we used a parallel algorithm that calculates the time domain currents entering an \(N\)-port circuit model (black box) characterized by a given matrix of \(Y\)-Parameters (admittances). Let us assume that the time domain voltage stimulus at each circuit port is known. The frequency domain voltages can be obtained by using the Discrete Fourier Transform (DFT) of the time domain voltages. The frequency domain currents entering each port can then be calculated using the equations:

\[
I_1(j\omega) = Y_{11} \, V_1(j\omega) + Y_{12} \, V_2(j\omega) + \cdots + Y_{1N} \, V_N(j\omega)
\]
\[ I_2(j\omega) = Y_{21} * V_1(j\omega) + Y_{22} * V_2(j\omega) + \cdots + Y_{2N} * V_N(j\omega) \]

\[ \vdots \]

\[ I_N(j\omega) = Y_{N1} * V_1(j\omega) + Y_{N2} * V_2(j\omega) + \cdots + Y_{NN} * V_N(j\omega) \]

where \( N \) is the number of ports, \( I_n(j\omega) \) and \( V_n(j\omega) \) the frequency domain current and voltage at port \( n \) respectively, \( 1 \leq n \leq N \), and \( Y_{ij} \) the \( ij \)-th element of the admittances matrix. The Inverse DFT (IDFT) is finally employed to transform the calculated frequency domain currents back to the time domain.

5.1 Application Setup

We have developed a parameterized Manager-Workers style, parallel and distributed application to calculate the time domain current waveform at each circuit port. A Manager task sends the required data to the available \( W \) workers, collects the results returned from them, and displays the final result. The work is partitioned evenly among the \( W \) workers i.e. each worker calculates \( L = N/W \) port currents (using equations (1))\(^3\). The Manager first distributes \( L \) sets of voltage pulse parameters (delay, rise time, fall time, duration, etc) and \( L \) rows of the \( Y \)-Parameters matrix, one set to each worker. The workers uses the received parameters to generate \( L \) time domain voltage pulses, calculate their DFT and send the resulting frequency domain voltage vectors back to the Manager. The Manager aggregates all collected \( V(j\omega) \) vectors and distributes them back to all the workers. At this point each worker has all the data it needs to calculate the \( L \) frequency domain currents it is responsible for and their IDFT. Finally, the Manager collects the resulting \( L \) time domain current vectors sent by each worker and stores them to a file for display.

We used JPVAC to construct the application’s task graph (ATG) and generate the corresponding AMTP tree and configuration file. The ATG for a Manager with \( W = 4 \) workers configuration is shown in Figure 13(b). JPVTC was then used to develop and assign behavioral models to each application task. The tasks behavioral graphs created using the JPVTC are shown in Figure 14(b) and (d) for the Manager and Worker tasks respectively. JP was used to generate automatically Java code templates for the Manager and Worker task components defined in the ATG. After adding application specific code to the Manager and Worker component templates we came up with an implementation that follows closely the structure of their behavioral graphs; the corresponding pseudo-code is provided in Figures 14(a) and (c).

We used a homogeneous cluster consisting of 333MHz Sun Sparc/Solaris workstations running with a Network File System (NFS). Using a reference machine, we conducted benchmark experiments to estimate the execution time for the basic code blocks shown in the behavioral graph of Figure 14 (DFT, IDFT, etc) as needed to properly annotate them.

Furthermore, we used a simple JP ping-pong experiment to measure the application-level port-to-peer-port message transfer delays over the 100Mbps Ethernet network links connecting the workstations, using messages ranging from 1K to 10K elements. (An element is a complex number consisting of two 64-bit

\(^3\) For simplicity and without loss of generality we assume that \( L \) is an integer.
public synchronized void run ()
{
  // Initialization code // cs1
  - set N // number of rows (i.e. equations)
  - set W // number of workers in mapping
  - L = N/W // each worker calculates L equations

  // Set parameters
  for( i = 0; i < N; i++ ) {
    // Get the pulse params and the V-Parameter row of data[i]
    // bloop1
    data = (MyData)port_[0].SyncRead(k); // sr1
    // eloop1
  }

  // Send pulse params and y rows to workers
  for( p = 0; p < W; p++ ) {
    // bloop2
    port_[p].AsyncWrite(data[k], k); // aw1
    // eloop2
  }

  // Get each V_jw vectors from workers
  for( p = 0; p < W; p++ ) {
    // bloop3
    V_jw_all[k] = (CmplxVect)port_[p].SyncRead(k); // sr1
    // eloop3
  }

  // Send V_jw_all to workers
  for( p = 0; p < W; p++ ) {
    // bloop4
    port_[p].SyncWrite(V_jw_all, key1); // sw1
    // eloop4
  }

  // Get I(t) from workers
  for( p = 0; p < W; p++ ) {
    // bloop5
    I_t[k] = (DblVect) port_[p].SyncRead(k); // sr2
    // eloop5
  }

  - Write I_t to file for display // cs2
  // Release ports
}

public synchronized void run ()
{
  // Initialization code // cs1
  - L = N/W // load of this worker

  for( k = 0; k < L; k++) {
    // bloop1
    // Get pulse parameters and y row for current row
    data = (MyData)port_[0].SyncRead(k); // sr1
    // Generate time domain pulse using the given parameters // cs2
    - Calculate DFT of pulse to get V(jw) // cs3
    // Send V(jw) to manager
    port_[0].AsyncWrite(V_jw, k); // aw1
    // eloop1
  }

  // Get the other V(jw) from manager
  V_jw_all = (CmplxVect[],) port_[0].SyncRead(key1); // sr2

  for( k = 0; k < L; k++) {
    // bloop2
    // Calculate I(jw) using V_jw_all and y row // cs4
    - Calculate IDFT of I(jw) to get I(t) // cs5
    // Send I(t) to manager
    port_[0].SyncWrite(I_t[k], k); // sw1
    // eloop2
  }

  - Release ports
}

Fig. 14. (a) Manager component pseudo code;(b) behavioral model graph of Manager task; (c) Workers component pseudo code; (d) behavioral model graph for Worker tasks

doubles (real and imaginary part)). The performance estimation algorithm uses the expected message size (annotated) and the measured transfer time to estimate the message transmission time of a write operation.
5.2 Experiments

Using the application described above, we have conducted several experiments in order to assess the performance prediction capabilities of the proposed method in a variety of different scenarios. The task behavioral models were developed, annotated using the reference machine data, and loaded using the JPVTTC tool. The performance estimation algorithm (also built into the JPVTTC) was used to estimated the execution time of a given application configuration (corresponding to the ATG generated by the JPVAC tool), based on the annotated tasks behavioral models and system related static/dynamic information.

In the first experiment (Exp1) we compared the estimated and measured overall application times for a distributed configuration with one Manager and $W = 4$ workers. Each task is assigned to a different machine (no-multitasking). In the second experiment (Exp2) we used the same number of workers but only $M = 3$ machines with multitasking; the Manager is assigned to machine M1, workers W1 and W2 are assigned to machine M2, and similarly workers W3 and W4 to machine M3. In both experiments we used only Java tasks and lightly loaded machines and varied the message size (voltage vector size) between 1K-5K elements. Moreover, $N$ is equal $W$ in Exp1-Exp4. As it can be seen from Figures 15(a) and (b), the predicted (by the performance estimator using the models) and actual performance (measured running the completed code templates) were very close.

In the third experiment (Exp3) we investigated several configurations in which an application instance with one manager and $W = 7$ workers is mapped onto different sets of machines. We used heterogeneous workers i.e. six Java and one Matlab worker in all cases. Our objective was to find the most efficient configuration that maximizes performance while utilizing the smallest possible number of machines. After fixing the message size to 1K elements, we considered several different tasks-to-machines mappings (see Figure 15(d)). All tasks are assigned to lightly loaded machines in this experiment. As the estimated results suggest (Figure 15(c)) performance saturates when using 4 machines and the predictions are again very accurate in all cases. Therefore the most efficient solution is to use 4 machines (4-Map configuration) which leads to optimal performance and at the same time frees 4 machines.

In experiment four (Exp4), we considered an application instance consisting of 8 Java tasks (1 Manager and 7 Workers) mapped onto $M = 8$ machines of variable load. Specifically, all machines were lightly loaded except machines M7 and M8 that were overloaded. We have evaluated several configurations differing on how tasks are allocated to the six lightly loaded machines (see Figure 15(f)) trying to find a mapping with less than $M = 8$ machines that achieves better performance than the fully distributed case. As we have explained in section 4.1.1 if a task is allocated to an overloaded machine, the performance estimation algorithm uses the dynamically measured load average to adjust the execution times of elements in the task's behavioral graph. Both the estimated and measured performance results (see Figure 15(e)) suggest that configuration 6M-map2 meets our objective. We conclude that by mapping the 8 tasks onto 6 lightly loaded machines (multi-tasking) we can achieve the same overall application execution time as with 8 machines when two of them are overloaded. M6-map2 reserves fewer machines and does not add load to the already overloaded machines.

In the last experiment (Exp5), we varied both $L$ (from 1 to 512) and $W$ (from 1 to 8) in order to assess better
The distribution of the error (relative difference between measured and predicted performance). We used only Java workers, no multi-tasking and a fixed message size of 1K words. The 3D plots for the measured and estimated execution times as a function of both $L$ and $W$ are presented in Figures 16(a) and (b) respectively. Their relative difference never exceeded 8% as shown in Figures 16(c) and (d).

Furthermore, we measured the time to estimate the application elapsed time (i.e. the overhead of the performance estimation algorithm discussed in section 4) for each point in Exp5 (see Figures 17(a)-(b)). Based on these results, the performance estimation time is asymptotically proportional to $WL^2$. In this experiment, the number of passes ($P$) is proportional to $L$, which is due to the synchronization elements that exist in most of the Manager/Worker loops. In addition, the number of iterations ($I$) is proportional to $L$, which is proportional to the expected number of elements after flattening the loops in the task behavioral graphs assigned to each machine. Moreover, the number of machines ($M$) is equal $W + 1$.

Note that, the performance estimator provides the user with a log report that contains a breakdown of the estimated overall time of each application task (i.e. computation, communication, and idle times), as well as the tasks with the (min, max) computation, communication, idle, and total times. This information is very
Fig. 16. Exp5: (a) Measured, and (b) Estimated execution time as \( W, L \) increase; (c) the relative error distribution, (d) the relative error does not exceed 8%.

Fig. 17. Exp5: (a) Simulation time as \( W, L \) increase, (b) the simulation time is proportional to \( WL^2 \).

useful to understand and identify the bottlenecks in the application in order to try to avoid them by finding a better application configuration. A snapshot of the simulation report for the \( W = 4 \) and \( L = 128 \) in Exp5 is shown in Figure 18.

6 Summary and Future Work

We have introduced a task modeling methodology that uses a small number of basic elements which can be combined in an easy and flexible manner to construct quite complex yet compact task behavioral descriptions. Task behavioral graphs can be composed graphically using constructs such as conditionals, loops of different kinds, forks etc. Constructs can be nested and may include in their bodies code blocks and message passing...
elements with ports and keys parameterized by the loop indices. Graphical task modeling can be performed easily using a new tool, the JP Visual Task Composer (JPVTC) developed for this purpose. At a higher level, completed behavioral task models can be connected structurally (via the task ports) to build complete, two-level, hierarchical network computing application models. The hierarchical application models can be used for performance engineering and rapid prototyping of network computing applications before any code is developed.

We have also introduced a performance estimation scheme that uses hierarchical application models, static and dynamic system state related data and can predict accurately the expected total running time of a given distributed and multi-tasked application configuration. The performance estimation algorithm (implemented as a module within the JPVTC) can be used to perform what-if performance evaluations under varying conditions and before any coding is attempted. This unique performance based engineering capability of the JavaPorts framework helps application designers understand how the interacting tasks are expected to behave and make adjustments as needed to meet their objectives and constraints. Our algorithm, unlike other existing performance estimation methods for parallel computing applications, can account not only for the execution, but also for the queuing and synchronization delays of the tasks. It can estimate the performance of nested loops containing message passing operations with ports and keys depending on the loop indices. Moreover, it can detect the potential for application deadlock. We have conducted several experiments demonstrating the accuracy of the obtained performance predictions in different kinds of scenarios (homogeneous vs. heterogeneous tasks, lightly vs. heavily loaded machines, with vs. without multi-tasking etc).

Behavioral task modeling and accurate performance estimation are two important parts of a complete interactive application-level QoS management system that we have developed as part of the JavaPorts project and which also supports: (i) Finding automatically the best application mapping using the hierarchical application models and the recent past resource conditions. "Best" mapping in this context is considered one that satisfies user-specified QoS metrics at application startup time. (ii) Allowing each task to assess dynamically and programmatically (through an API) the state of the underlying resources in its neighborhood so that it can adapt its behavior at runtime according to the varying resource characteristics. These elements
of the system have not been discussed in this paper and will appear elsewhere. The ultimate goal of our research has been to develop an interactive, user-friendly, application-level QoS management system that can help developers model, develop, configure, evaluate, and if needed reconfigure or adapt component-based multi-tasked network computing applications so that they meet their QoS targets throughout their life cycle.

References


