Contents

1 Taks Scheduling on NOWs using Lottery-Based Workstealing **1**

1.1 Introduction 2

1.2 The Cilk Programming Model and Work Stealing Scheduler 6

1.2.1 Java programming language and the Cilk programming model 6

1.2.2 Lottery victim selection algorithm 8

1.3 Architecture and Implementation of the Java Runtime System 10

1.3.1 Architecture of the Java runtime system 10

1.3.2 Implementation of the Java runtime system 13

1.4 Performance evaluation 13

1.4.1 Applications 13

1.4.2 Results and discussion 16

1.5 Conclusions 18

1.6 Bibliography 19

Chapter 1

Taks Scheduling on NOWs using Lottery-Based Workstealing

BORIS ROUSSEV AND JIE WU*

Department of Information Systems
Susquehanna University
514 University Avenue, Selinsgrove, PA 17870, USA
roussev@roo.susqu.edu

*Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Fl 33431, USA
jie@reality.cse.fau.edu

Abstract

This paper presents a Java framework for high performance computing (HPC) on networks of workstations (NOWs). We introduce a new lottery-based work stealing algorithm for efficient scheduling of large-scale multithreaded computations on NOWs. In the proposed algorithm, idle workstations actively search out work to do rather than wait for work to be assigned. In the lottery game, each workstation is equipped with a set of tickets and the number of tickets is proportional to the age of the oldest thread in the ready pool of the workstation. A winning ticket is drawn at random and the workstation with the winning ticket becomes the victim from which the idle workstation steals work. The proposed selection procedure serves for two purposes. First, we try to lower communication costs by stealing large amounts of work, with the logic behind being that old-aged computations are likely to spawn more work than relatively young computations. Second, we would like to bias the search to obtain favourable results while at the same time avoid system bottleneck. Our approach has been implemented and tested on NOWs under the Solaris OS. Several examples have been used to demonstrate the potential performance gain.

Keywords: *High Performance Computing (HPC), Java, Network of Workstations (NOWs)*

1.1 Introduction

For the past twenty years parallel computing has been used successfully in many applications such as weather forecasting, molecular modeling, airflow modeling, tax return, etc [20]. Despite some success and he fact that parallel processing have been conjectured to be the most promising solution to the computing requirements of many problem domains [26], parallel computing is not widely accepted in industry. Parallel computers conjure images of sophisticated and expensive multiprocessor architectures, running obscure operating systems, and executing programs written in non-portable special-purpose languages. The on-going technological convergence of local area networks (LANs) and massively parallel computers augments the effect of the reverse computing food chain law [1], where in contrast to biology, the smallest fish, personal computers, is eating the market of workstations, which has consumed the market for minicomputers and now is eating away the market for larger mainframes and supercomputers. The driving force behind this "law" is the better price/performance ratio of networks of workstations (NOWs) over parallel systems. We increasingly find NOWs making inroads into domains once monopolized by supercomputers [11].

We can identify the following motivating factors for using NOWs for high performance computing (HPC): (1) Surveys show that the utilization of CPU cycles of desktop workstations is typically less than 10%. (2) Performance of workstations and PCs is rapidly improving. (3) As performance grows, percent utilization will decrease even further. (4) Organizations are reluctant to buy large supercomputers, due to the large expense and short life span. (5) The communication bandwidth between workstations increases as new networking technologies and protocols are implemented in LANs and wide area networks (WANs). (6) NOWs are easier to integrate into existing networks than special parallel computers. (7) The development tools for workstations are more mature than the contrasting proprietary solutions for parallel computers - mainly due to the non-standard nature of many parallel systems. (8) NOWs are cheap and really available alternative to specialized HPC platforms. (9) Use of NOWs as a distributed computing resource is very cost effective (incremental system growth). Therefore, one could expect HPC on NOWs to become more and more attractive as time goes on. This gives a new impulse to the field of parallel computing.

A model of parallel computation is an abstract machine, providing a set of primitives to the programming level above. It is designed to separate software development concerns from effective parallel execution concerns. According to the abstraction they provide, models for parallel computing can be classified in five categories [25], see Table 1.1, based on the ways decomposition, mapping, communication, and synchronization are done. Table 1.1 also shows some representative language/libraries for each model.

Decomposition of a program into threads (column 1 of Table 1.1) and mapping of threads to processors (column 2 of Table 1.1) are known to be computationally expensive. Communication requires placing two ends of the communication in the

correct threads and at the correct place. Synchronization requires the understanding of the global state of the computation, which is immense for practical purposes.

Given the aforementioned reasons for using NOWs for HPC, we might expect NOWs to have rapidly moved into the mainstream of computing. This is clearly not the case. We could identify the following problems and difficulties with using NOWs, which explain why its advantages have not (yet) led to its widespread use: (1) In principal, NOWs have much unused compute power to be exploited. In practice, the large latencies involved in communicating among workstations make them low-performance parallel computers. Typically larger-grain processes are used to help conceal the latency. The increasing use of optical interconnection and ATM for connecting workstation changes the situation. (2) Models for NOWs include systems such as MPI [22] and PVM [26], which belong to the lowest level of the model hierarchy. Most importantly, these models do not hide much of the decomposition and communication. The developer must specify in thorough detail the implementation, which makes building software extremely difficult. (3) The models used must address the heterogeneity of the processors (architecture, OS, GUI) on typical NOWs. (4) Workstations must trust the programs being executed on their machines. (5) The system must secure the application from spying by participating workstations. (6) The system must be able to mask the intentional (or unintentional) data loss and data corruption caused by the inherent partial failures. (7) The system must reward (economic incentive) the participation of workstations in a distributed computing infrastructure. (8) The theory required for parallel computations is immature [25]. Skillicorn and Talia argue that our knowledge about abstract representation of parallel computations and reasoning about them is insufficient and rudimentary.

Java [21], an object-oriented language, has become popular because of its platform independence and safety. It has greatly simplified network programming by providing elegant TCP/IP API, object serialization, network class loading (code mobility), RMI (remote method invocation) [16], Servlets, JSP (Java server pages) [18] and built-in concurrent constructs. Java is a shared memory thread-based language with built-in monitors and binary semaphores as a means of synchronization at the object and class level. Though Java is firmly fixed at the lowest level of the parallel computing model hierarchy, it addresses concerns (3) and (4) above. This along with its phenomenal growing popularity entails a rapidly expanding body of projects that use Java as a language for HPC on NOWs and clusters [3], [4], [11], [12], [13], [14], [17], [20], [23], [24]. Invariably, their aim is to hide one or more of the characteristics of the language, see Table 1.1, that make it ill-suited for parallel programming.

Next, we discuss the existing Java-based systems according to the criteria outlined in Table 1.1. JPVM [12], IceT [14] and MPIJ [23] are systems based on message passing using *sends* and *receives* to specify the message to be exchange, process identifiers and address. They implement models from the bottom-level of the model abstraction hierarchy.

JavaSpaces [17] stands one level higher than message passing systems. It is

a new realization of Linda "tuple spaces" [10]. In essence, JavaSpaces simplifies process communication by using a large pool into which data values are placed by processes, and from which they are retrieved associatevely. Charlotte [4] is one of the first systems to use Java for parallel computing. Charlotte programs are written for a virtual parallel machine where the runtime system automates the mapping of threads, called routines, to processors and communication is done through shared variables allocated from a distributed shared memory. The system implements the fork-join model of parallel programming and introduces fault-tolerance through "eager scheduling." With eager scheduling the manager assigns a job repeatedly until it is executed to completion by at least one worker. The authors introduce a new memory management technique, called "two-phase idempotent execution strategy," to ensure the correct execution of of shared memory programs under eager scheduling. What we see as a disadvantage is the centralized association of workers and computations.

Next, in the model hierarchy comes Atlas [3], a Java realization of the Cilk [5] programming model, which is best suited for tree-like computations, see below. The system automates the placement of computations and communication and achieves near-optimal load balancing. Bayanihan [24] implements a generic set of components that support master-worker programming style similar to Charlotte through a form of barrier synchronization and eager evaluation. In addition, the generic objects of the runtime system can be changed for performance optimization of different distributed algorithms and even for implementation of new programming paradigms. Javelin [11] is a seminal infrastructure for global computing based on Java-enabled Web technology (applets, Web servers, and HTTP). It achieves load balancing through a distributed task queue (scheduler) using work stealing. The developer is abstracted both from mapping of threads to processors and from inter-thread communication. In Javelin communication layer, communication between applets is routed through their associated servers, which further increases the network latency, making the system suitable for running mainly coarse-grain parallel applications.

Further, the projects using Java could be divided into applet-based and standalone. This classification is orthogonal to the classification based on the model abstraction. Both approaches have advantages and disadvantages. The former is severely restricted by the applet security model. Load balancing is difficult to achieve since there is always the bottleneck created by virtue of the centralized node (the Web server). These systems target the Internet and its unlimited resources. Standalone implementations target both NOWs and the Internet. They require that either the clients (users) have access privileges to the participating machines or the workstation owners download and install the runtime system. The chief problem to overcome in the latter approach is having to prove to and convince the owners that their privacy and security would not be violated by executing foreign computations. For a comprehensive discussion of some other considerations to be addressed when choosing between applet-based and standalone implementations refer to [19].

The aims of this research work are to develop a Java runtime system for efficient

	Decomp.	Mapping	Comm.	Sync.	Languages
1.	implicit	implicit	implicit	implicit	Haskel
2.	explicit	implicit	implicit	implicit	Concurr. Prolog, Multilisp, Cilk
3.	explicit	explicit	implicit	implicit	BSP, LogP, Linda
4.	explicit	explicit	explicit	implicit	Emerald, Concurrent Smalltalk
5.	explicit	explicit	explicit	explicit	Java, PVM, MPI, Ada

Table 1.1. Models for parallel computations.

scheduling of multithreaded Java applications on NOWs and to improve the random work stealing algorithm used in Cilk-NOW [5], [6], [7], [8]. Cilk-NOW is a runtime system based on a C thread package designed for multiprocessor architectures. The runtime system can be ported to and used on a NOW. However, it suffers from the inherent limitations of the C programming language: nonportability, lack of reflection API, lack of serialization, and lack of network loading. The former restricts the portability of the runtime system as well as its deployment to homogeneous environments. The lack of network loading decreases the scalability of the system, e.g., the participating workstations should share a common file system in order to load the code of the applications.

In our work we use the programming model developed by Robert Blumofe at MIT [5]. This model requires that decisions about the breaking up of available work into threads be made explicit while relieving the software developer of the ramifications of such decisions: mapping of threads to processors is done automatically and efficiently by the distributed scheduler that implements a random work stealing algorithm; communication is done implicitly through shared variables; and synchronization is achieved through continuation passing style [2]. In other words, in writing a parallel application in Java, a programmer expresses parallelism by coding instructions in a partial execution order by structuring the code into totally ordered sequences called *threads*. The programmer need not specify the processor in the system that executes a particular thread nor exactly when each thread should be executed. These scheduling decisions are made by the run-time systems scheduler. In our work we use Java as an implementation language. The proposed model allows Java to enjoy the benefits of being a member of the family of languages in the second category in Table 1.1. Making the Java programming model more abstract could reap tremendous spin-offs. Parallel applications are easier to design, verify, and debug while efficient implementation is still possible.

The remainder of the paper is structured as follows. In Section 2 we review the Cilk language and work stealing scheduler [6], [7], [8] adapted to our needs and introduce a new work stealing algorithm. In Section 3 we describe the architecture and the implementation of the proposed Java runtime system. Then, in Section 4 we present experimental results about the performance of the runtime system employing the work stealing algorithm described in Section 2. In the final section we outline plans for future work and conclude.

1.2 The Cilk Programming Model and Work Stealing Scheduler

NOWs offer a tremendous processing capacity. However, in order to realize this computing capacity we need a good programming model and an efficient distributed scheduler for redistributing the load among the workstations. In Section 2.1 we describe the Cilk programming model as well as its random work stealing algorithm. In Section 2.2 we present a new distributed scheduler based on a victim selection algorithm through lottery.

1.2.1 Java programming language and the Cilk programming model

The Cilk programming model contains a graph of instructions and a tree of threads that unfold dynamically during program execution. A multithreaded computation is composed of a set of threads, each of which is a sequential order of instructions.

During the course of execution, a thread may create, or spawn, other threads. The spawning thread can operate concurrently with the spawned one. The spawned threads are considered to be children of the thread that did the spawning, and a thread may spawn as many children as it desires. In this way the threads are organized into a *spawn tree*.

In addition to spawning threads, a multithreaded computation may also contain dependency between the threads. As an example of a data dependency, consider an instruction in one thread that produces a data value consumed by an instruction in another thread. Dependencies allow threads to synchronize.

An execution schedule for a multithreaded computation determines the processor in the system that executes a given instruction at each step. An execution schedule must obey the spawning dependencies in that no processor may execute an instruction in a spawned child thread until after the spawning instruction in the parent thread has been executed. It must also obey the data and control dependencies among the threads in order to achieve proper thread synchronization.

In a strict multithreaded computation, every dependency goes from a thread to one of its ancestor threads. In a fully strict multithreaded computation, every dependency goes from a thread to its parent. Fully strict computations are "well-structured" in that all dependencies from a subcomputation emanate from the subcomputations root thread. A distinctive feature of strict computations is that once a thread has been spawned, a single processor can complete the execution of the entire subcomputation rooted at this thread even if no other progress is made on the other parts of the computation.

A program in the Cilk programming model consists of one or more classes and objects with one or more threads of control. Threads are nonsuspendable. The runtime system manipulates and schedules the threads. A Java program generates parallelism at runtime by instantiating a runnable object or a subclass of class `Thread` and executing its run method. After this the parent and the child may execute concurrently (asynchronous method invocation). After spawning one or more children threads, the parent thread does not wait for its children to return.

Instead, the parent thread additionally spawns a successor thread to wait for the results from the children. Thus, a thread may wait to begin executing, but once it begins execution there is no suspending it [2]. Sending a result to a suspended thread is done via the `sendArg` method. The Java runtime system implements these primitives using two types of classes: *closures* and *continuations*.

Closures are classes employed by the runtime system to keep track of and schedule the execution of spawned threads. The runtime system associates one closure object with each spawned thread. The absence of templates in Java does not allow to hide the existence of closures from the software developer without an additional preprocessing step. A closure consists of the class name of a runnable object, a slot for each of the specified arguments in the object's constructor, and a *join counter* indicating the number of missing arguments that need to be supplied before the object is ready to be instantiated and its run method executed in a separate thread. If the closure has received all of its arguments, then it is ready; otherwise, it is waiting. To run a ready closure, the runtime system uses reflection API to find out the object constructor having the same number and type of arguments as specified in the closure and then invokes it. When the run method of the instantiated object dies, the closure is deleted (freed).

A `Continuation` is a reference to an empty argument slot of a closure. An executing thread sends a value to a waiting thread by placing the value into an argument slot of the waiting thread's (runnable object's) closure. The executing thread uses the `sendArg` method of a `Continuation` object for this purpose. The empty slot of the waiting closure is specified by the argument passed as a parameter to the constructor of the `Continuation` object.

At runtime, each processor maintains four pools of closures: ready pool , waiting pool, assigned pool, and the pool of stolen closures. The ready pool is a *deque* (double-ended queue) which contains all of the ready closures. Whenever a closure is created, if its join counter is 0, then it is placed on the head of the ready deque; otherwise, it is added to the waiting pool. Whenever a `sendArg` is invoked, the join counter is decremented, and if the join counter reaches 0, then the closure is removed from the waiting pool and placed at the head of the ready deque. When a thread finishes, the next closure is chosen from the head of the ready deque and instantiated (its thread executed.)

In Figure 1.1, a worker pushes spawned tasks on its local ready deque and pops the task from its head when it finishes the current task. A pop on an empty ready pool triggers a steal request being sent to a victim worker. When the steal request arrives at the victim worker, if its ready deque is not empty, the task at the tail of the deque is removed and sent to the requesting worker.

If no closures are available in the ready pool, a processor becomes a *thief*. In Cilk-NOW [5], to steal a work, a processor chooses another processor, called *victim*, at random and requests a closure to be sent back (see Figure 1.1). If the victim processor has any closures in its ready deque, one is removed from the tail of its ready deque and sent across the network to the thief whom will add this closure to its own ready deque. The thief may then begin work on the stolen closure.

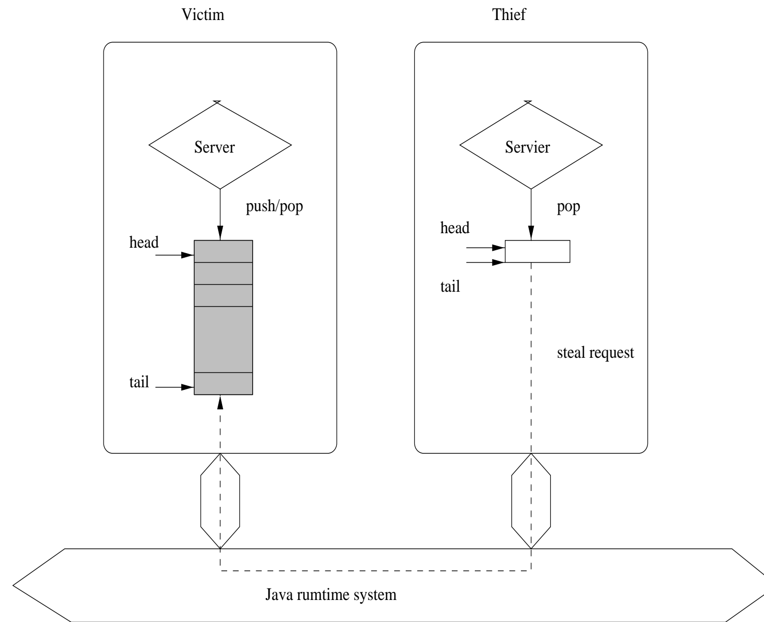

Figure 1.1. A description of thief and victim algorithm

If the victim has no ready closures, it informs the thief who then tries to steal from another randomly chosen processor until a ready closure is found or program execution completes.

Our runtime system consists of several processes, executing Java Virtual Machines (JVM), running on several different workstations. One process, called *registry*, runs a Java program responsible for keeping track of all the other processors that cooperate on a given job. These other processes are called *workers*. Each worker registers with the registry by sending it a message containing its own transport address. The registry responds by assigning each worker a unique name. Workers periodically check in with the registry. Every 2 seconds each worker sends a message to the registry containing the level of the closure at the tail of its ready deque. The level of a closure is equal to the height of the root of the multithreaded spawn tree minus the height of the node of the closure in concern. Every 2 seconds the registry multicasts a list of the network addresses and ages of all registered workers.

1.2.2 Lottery victim selection algorithm

In order to execute multithreaded programs on NOWs efficiently we need to construct execution schedules dynamically. We introduce below a new distributed scheduler based on work stealing that builds execution schedules at run time as the

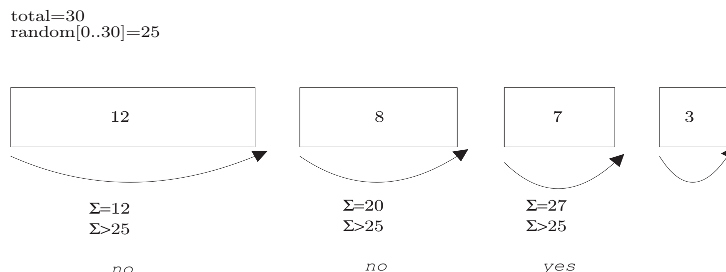

Figure 1.2. Description of the lottery victim selection algorithm.

computation unfolds.

As distinguished from Cilk-NOW, what comes across as novel in our runtime system is that when a worker becomes a thief it does not chose a victim uniformly at random. Instead, it incorporates a *lottery* scheduler [27] making use of the information about the level of the closure (thread) at the tail in each processor's ready deque.

Lottery scheduling has been used successfully to provide efficient and responsive control over the relative execution rates of computations running on a uniprocessor. It has been shown efficient and fair even in systems that require rapid, dynamic control over scheduling at a time scale of milliseconds in seconds. Lottery scheduling implements proportional-share resource management where the resource consumption rates of active computations are proportional to the relative shares they are allocated.

In the proposed randomized victim selection algorithm, each processor is associated with a set of tickets and the number of tickets associated with each processor is proportional to the level of the tail thread of its ready pool. For every thief processor, the victim processor is determined by holding a lottery. The victim is the processor with the winning ticket. For example, if the registry has multicasted a list of four processors with levels of their ready deque tail threads 12, 8, 7, and 3 respectively, there is a total of $12 + 8 + 7 + 3 = 30$ tickets in the system, see Figure 1.2. Next, assume that a new processor has just joined the computation and has received the multicast message from the registry. Initially, this new processor has an empty ready pool so it becomes a thief immediately. In order to select a victim the new processor holds a lottery based on the information in the multicast message. Assume that the 25th ticket is (randomly) selected. The list of processors is searched for the winner. For every processor the partial sum of tickets from the beginning of the list is computed. If the partial sum is greater than the number of the winning ticket than the current processor is the winner and the search is aborted; otherwise, the search continues with the next processor in the list. For our four-processor example, the winner is the third processor. Therefore, the new processor will try to steal work from the third processor in the list multicasted by the registry. Further, let us assume that another new processor joins the compu-

tation. It will also hold a lottery based on the information in the same multicast message. It is likely that the winner will be the first or the second processor because of the great number of tickets representing them. In this way the selection algorithm probabilistically avoids congestions at the busiest nodes in the system while at the same time it allows work stealing from them.

1.3 Architecture and Implementation of the Java Runtime System

This section presents our prototype runtime system and describes its core components and their interactions.

1.3.1 Architecture of the Java runtime system

At the highest level, the runtime system implements the following functions:

Threads scheduling
The scheduler distributes tasks from a distributed task queue and manages load balancing through random work stealing, see Section 2.

Adaptive parallelism
The system makes use of idle processors which are idle when the parallel application starts or become idle during the duration of the job. When a given workstation is not being used, it joins in the system. When the owner returns to work, that processor automatically leaves the computation. Thus, the set of workers shrinks and expands dynamically throughout the execution of a job.

Macroscheduling
A background daemon process runs on every processor in the network. It monitors the processor state to determine when the processor is idle so that it could start a worker on that machine.

The three main components of the runtime system are the registry, the workers, and the node managers. The registry is a super server providing the following services, each of which is implemented in a separate server:

- registering/deregistering of workers,
- updating the information about the workers currently involved in the computation, and
- multicasting the list of network addresses and ages of the workers.

Each worker consists of the following components:

- `Master` object synchronizing the access to the four pools of closures through guarded suspension and execution state variables.
- `Compute` server fetching jobs from the ready deque and executing them. If the ready deque is empty, the worker becomes a thief and triggers the `Thief` thread.
- `Thief` runnable object executed in a separate thread. This object implements the victim selection algorithm and the actual work stealing. A shortcoming of most distributed schedulers is the need for the workstations to share a common file system, such as NFS. The `Thief` incorporates a network classloader that allows the downloading of executable code on demand. The latter overcomes the requirement for the workstations to have a common file system and improves the scalability of the proposed runtime system.
- `Victim` server object. This server is contacted by the `Thief` clients of other workers in the course of their work hunt.
- `Result` server object. Results from stolen threads are returned to this server which updates the corresponding closure in the waiting pool.
- `Register` client responsible for registering to and periodic updates with the registry.
- `Listener` which listens continually for the datagrams multicasted by the registry. It writes the information received in a 1-bounded buffer. The information is read from the buffer and used by the victim selection algorithm which is invoked by the `Thief` thread.
- `VictimSelection` object implementing the victim selection algorithm. We use the library class `java.util.Random` to generate a stream of pseudo-random numbers. Each worker uses as a seed its unique ID assigned by the registry. A victim worker is selected by holding a lottery. First, a winning ticket is selected at random. Then, the list of workers is searched to locate the victim worker holding that ticket. This requires a random number generation and $O(n)$ operations to traverse a worker list of length n, accumulating a running ticket sum until it reaches the winning value.

In [27] various optimizations are suggested to reduce the average number of elements of the worker list to be examined. For example, ordering the workers by decreasing level can substantially reduce the average search length. Since those processors with the largest number of tickets will be selected more frequently, a simple "move to the front" heuristic can be very effective. For large n, a more efficient implementation is to use a tree of partial sums, with clients at the leaves. To locate a client holding a winning ticket, the tree is traversed starting at the root node, and ending with the winning ticket leaf node, requiring only $O(\lg n)$ operations.

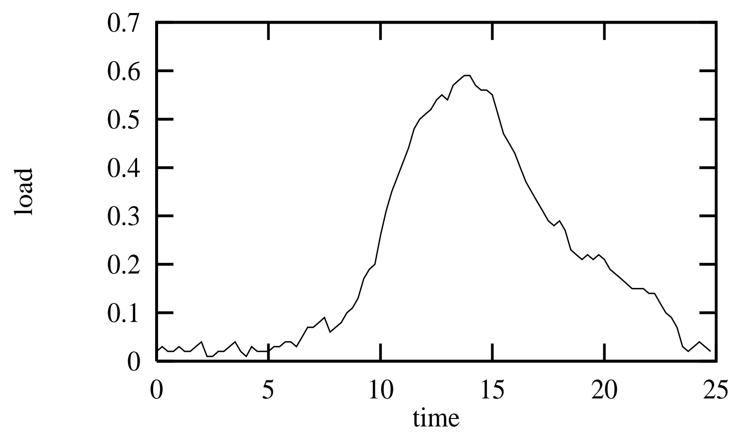

Figure 1.3. Load distribution on a NOWs running Solaris OS

Scheduling by lottery is probabilistically fair. The expected allocations of victims to thieves is proportional to the number of tickets the victims hold. Since the scheduling algorithm is randomized, the actual allocated proportions are not guaranteed to match the expected proportions exactly. However, the disparity between them decreases as the number of allocations increases.

1.3.2 Implementation of the Java runtime system

For efficiency, all communication protocols, except the initial registering of the workers, are implemented over UDP/IP. Some of the protocols add reliability to UDP by incorporating sequence numbers, timeouts, adaptive algorithms for evaluating the next retransmission timeout, and retransmissions [15]. The application protocol used to register new workers to the registry is developed over TCP/IP because of the needed reliability during the connection establishment and connection termination.

One of the assumptions of this research work is that there is a great number of idle CPU cycles. Figure 1.3 plots the average number of jobs in the ready queue of the machines comprising our network [1]. A script was run for two weeks collecting the average load across the workstations at 15 minute intervals. The results were combined to produce an average load during a day. As can be seen from this plot, though more machines are idle at night, a significant number of idle CPU cycles exists at various time slots throughout the day. The results confirm that a network of workstations does indeed provide a valid environment for HPC. It is also possible to calculate the average daily load from Figure 1.3. By rough approximation, the average load of the workstations is around 0.25, indicating that about 75% of the CPU time of each workstation is wasted every day.

1.4 Performance evaluation

In this section we present experimental results about the performance of our prototype runtime system for scheduling of multithreaded Java applications on networks of workstations. All experiments are done and measurements taken down on a network of 15 workstations running Solaris OS. Subsection 4.1 shows the implementation of a sample application and Subsection 4.2 presents some experimental results and interpretations of these results.

1.4.1 Applications

Consider the following example taken from [5] and rewritten in Java. The Fibonacci function $fib(n)$ for, $n \geq 0$, is defined as

$$fib(n) = \begin{cases} n & \text{if } n < 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

[1] This network is in the departmental lab of Computer Science and Engineering, Florida Atlantic University.

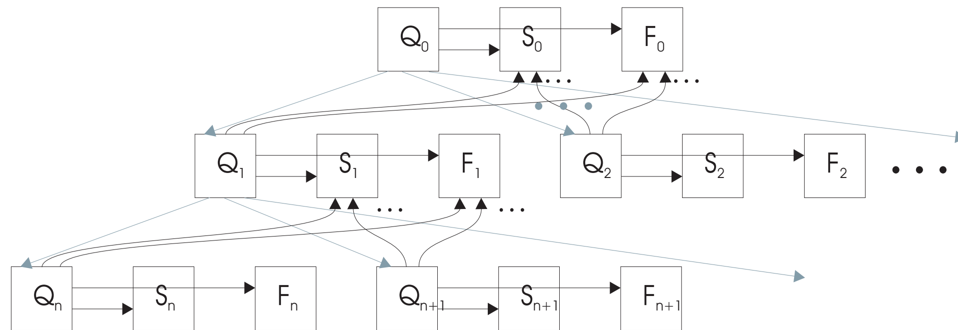

Figure 1.4. The tree grown by the execution of the `nqueens` program. Q_i stands for an `NQueen` object which is executed in a separate thread. S_i (F_i) stands for successor thread of type `Successor` (`Failure`). The edges creating successor threads are horizontal. Spawn edges are straight, shaded, and point downward. The edges created by `sendArgument()` are curved and point upward.

Another example that we consider is `nqueens`. `nqueens` application is a classical example of searching using backtracking. The objective is to find a configuration of n queens on an $n \times n$ chess board such that no queens can capture each other.

This following shows the way this function is written as a Java program. The double recursive implementation of the Fibonacci function is a fully strict computation. The Java code is given below which is structured in the run methods of the two runnable objects.

```
class Fib implements Runnable {
   Continuation dest;
   int n ;
   public Fib( Continuation k, int n ) {
      ...
   }
   public void run () {
      if ( n < 2 )
         dest.sendArg( n ) ;
      else {
         Continuation x = new Continuation () ;
         Continuation y = new Continuation () ;
         ClosureSum s = new ClosureSum( dest, x, y ) ;
         ClosureFib fib1 = new ClosureFib( x, n-1 ) ;
         ClosureFib fib2 = new ClosureFib( y, n-2 ) ;
      }
      return ;
   }
}

class Sum implements Runnable {
   Continuation dest ;
   int x,y ;
   public Sum( Continuation k, int x, int y ) {
      ...
   }
   public void run () {
      dest.sendArg( x+y );
   }
}
```

The Java code for `nqueens` is given in the Appendix. The `nqueens` problem is formulated as a tree search problem [9] and the solution is obtained by exploring this tree. The nodes of the tree are generated starting from the root, which is the empty vector corresponding to zero queens placed on the chess board. The code is structured in the run method of the classes `NQueens`, `Success`, and `Failure`. On each iteration, a new configuration is constructed, called `config` in the code, as an extension of a previous safe configuration, thus spawning new parallel work. A configuration is safe if no queen threatens any other queen on the chess board. The

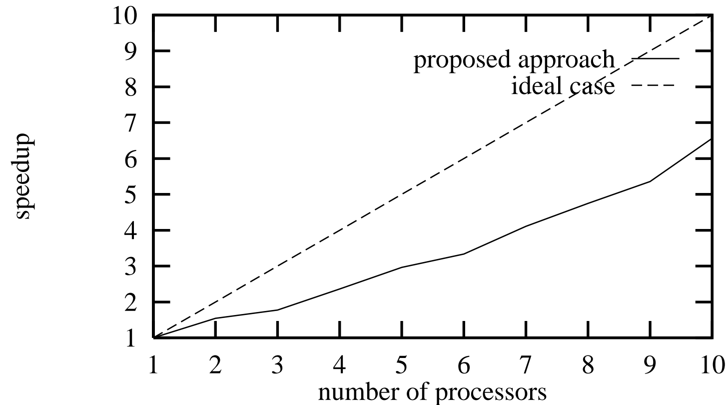

Figure 1.5. Parallel speedup

algorithm uses depth-first search to traverse the generated tree. On termination of the for loop of `NQueens.run()` method, the variable `count` contains the number of `Nqueens` closures pushed in the ready pool. This information is used to set the number of missing arguments of the `Failure` runnable object that is used in the backtracking stage if a dead end is reached.

Since we use continuation-passing style for thread synchronization, after spawning one or more children, the parent thread cannot then wait for its children to return. Rather, as illustrated in Figure 1.4, the parent thread (Q) additionally spawns two successor threads, namely `Failure` (F) and `Success` (S), to wait for the values returned from the children. The communication between the child threads and the parent thread's successors is done through `Continuation` objects. We use two different successor threads because failure and success have different semantics. In order for a thread to return failure, all of its child threads should report failure, while to return success, it suffices only one of its child threads to report success. It is important to note that `nqueens` spawns off parallel work which it later might find unnecessary. This "speculative work" can be aborted in our runtime system using the `abort` method of the `Master` object which synchronizes the access to the four pools of closures of each worker. Subsequently, the `abort` message is propagated to all workers involved currently in the computation. The latter allows `nqueens` program to terminate as soon as one of its threads finds a solution.

1.4.2 Results and discussion

The performance of the runtime system was evaluated using `fibonacci` and `nqueens` applications. Even though both of the applications are not real life, they generate a workload suitable for evaluating the performance of our system. `fibonacci` is not computationally intensive but spawns a large number of threads (in millions) which makes it appropriate for evaluating the synchronization of the runtime system.

# of proc.	Random	Lottery-based	Impro. in %
5	43.4	35.7	17.74
6	49.25	47.5	3.55
7	63.6	44	30.81
8	82.67	63	23.79
9	83.25	69.25	16.81

Table 1.2. Comparison between the performance of the random work stealing algorithm and the lottery work stealing algorithm (Fibonacci numbers)

`nqueens` features behaviour typical of most search algorithms employing backtracking.

First, we present the *serial slowdown* incurred by the parallel scheduling overhead. The serial slowdown of an application is measured as the ratio of the single-processor execution of the parallel code to the execution time of the best serial implementation of the same algorithm. The serial slowdown stems from the extra overhead that the distributed scheduler incurs by wrapping threads in closures, reflecting upon closures to find out threads' constructors, and work stealing.

Serial slowdown data for `fibonacci` and `nqueens` are 6.1 and 1.15, respectively. As expected `fibonacci` incurs substantial slowdown because of its tiny grain size. The slowdown of `nqueens` is insignificant.

Figure 1.5 shows the parallel speed up of the `fibonacci` application. In all experiments all workstations have been started up at the same time and therefore have taken a fare share of the load. The speedup is measured as the ratio of the execution time of the parallel implementation running with one participant to the average execution time of the parallel implementation running with m participants, where m is the number of workstations involved.

Tables 1.2 and 1.3 compare the performance of the classical work stealing algorithm where victims are chosen uniformly at random to the performance of the proposed work stealing algorithm which makes use of the information about the levels of the tail closures in the ready pools of the workers.

Tables 1.2 and 1.3 show that the lottery-based work stealing algorithm consistently outperforms the random work stealing algorithm for `fibonacci` and `nqueens` applications, respectively. However, we need to run more experiments with applications spawning a range of different subcomputations in order to provide stronger evidence in support of that statement. In Table 1.2 and 1.3, Columns 2 and 3 display the wall clock time in seconds for the classical work stealing algorithm and the lottery-based work stealing algorithm, respectively, for different number of processors involved.

# of proc.	Random	Lottery-based	Impro. in %
5	419	291	30.54
6	398	230	42.21
7	324	197	39.2
8	304	190	37.5
9	249	154	38.15

Table 1.3. Comparison between the performance of the random work stealing algorithm and the lottery work stealing algorithm (Nqueens problem)

1.5 Conclusions

We have devised and implemented a new victim selection algorithm. In the proposed victim selection algorithm, each processor is given a set of tickets whose number is proportional to the age of the oldest subcomputation in the ready pool of the processor. The victim processor is determined by holding a lottery, where the victim is the processor with the winning ticket. The experimental results have shown that the proposed work stealing algorithm outperforms the classical work stealing algorithm where the victims are selected uniformly at random. We have also designed and implemented a Java runtime system for parallel execution of strict multithreaded Java applications on networks of workstations employing the proposed lottery-based victim selection algorithm. The runtime system features:

- Distributed thread scheduler that manages efficiently load balancing through a variant of work stealing.
- Adaptive parallelism which allows the utilization of idle CPU cycles without violating the automicity of the workstations' users.
- Network class loader which lifts up the restriction requiring that all workstations share a common file system and improves the scalability of the runtime system.

Our future plans involve adding fault-tolerance to the runtime system through distributed checkpointing so that the system could survive machine crashes. The challenge of this enterprise stems from the absence of a common file system shared by all workstations.

For the Internet-based version of our runtime system we plan to incorporate in the work-stealing algorithm information about the communication delays among the processors in the system. On a LAN communication delays cannot have dramatic impact on the performance of the system since they are more or less uniform. However, on a WAN or internetwork they have to be taken into account in order to achieve efficient scheduling of the subcomputations. For the estimation of the communication delays between the processors of the network we are going to design

and implement a distributed algorithm, where each processor in the system obtains its partial view of the delays in the system through its communications with the rest of the processors. We also plan to justify theoretically the performance of the proposed work stealing algorithm based on lottery victim selection.

1.6 Bibliography

[1] T. Anderson, D. Culler, and D. Patterson, "A case for NOW (networks of workstations)," *IEEE Micro*, 15(1), 1994.

[2] A. Appel, *Compiling with Continuations.*, Cambridge University Press, New York, 1992.

[3] J.E. Baldeschwieler, R.D. Blumofe, and E.A. Brewer, "ATLAS: An infrastructure for global computing," In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web," In *Proceedings of the 9th International Conference on Parallel and Distributed Computing*, 1996.

[5] R. Blumofe, *Executing Multithreaded Programs Efficiently.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[6] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, Santa Fe, New Mexico, November 1994.

[7] R. Blumofe and D. Park, "Scheduling large scale parallel computations on networks of workstations," In *Proceedings of the Third International Symposium on High Performance Distributed Computing (HPDC)*, pp. 96-105, San Francisco, California, August 1994.

[8] R. Blumofe and P. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," In *Proceedings of the USENIX 1997 Annual Technical Conference on Unix and Advanced Computing Systems*, Anaheim, California, January 6-10, 1997.

[9] G. Brassard and P. Bratley, *Fundamentals of Algorithmics.* Prentice-Hall, 1996.

[10] N. Carriero and D Gelernter, "The S/Net's Linda kernel," *ACM Transaction on Computer Systems*, 4(2), pp.110-129, 1986.

[11] B. Christiansen, P. Ionescu, M. Neary, K. Schauser, and D. Wu, "Javelin: Internet-based parallel computing using Java," *Concurrency Theory and Practice*, 1997.

[12] A. J. Ferrari, "JPVM: Network parallel computing in Java," *Technical Report CS-97-29*, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA, http://www.cs.virginia.edu/jpvm/doc/jpvm-97-29.ps.gz

[13] "Globus metacomputing infrastructure," http://www.mcs.anl.gov/globus

[14] P. Gray and V. Sunderman, "IceT: Distributed computing using Java," In emphProceedings of ACM 1997 Workshop on Java for Science and Engineering, 1997.

[15] V. Jacobson, "Congestion avoidance and control," *Computer Communication Review*, 18(4), pp. 341-329, August 1988.

[16] JavaSoft Team, *RMI: Java Remote Method Invocation–Distributed computing for Java*. Sun Microsystems, Inc., Palo Alto, CA, 1998.

[17] JavaSoft Team, *The JavaSpaces specification*. Sun Microsystems, Inc., Palo Alto, CA, 1999.

[18] JavaSoft Team, *The JavaServer Pages 1.0 specification*. Sun Microsystems, Inc., Palo Alto, CA, 1999.

[19] L.F. Lau, A.L. Ananda, G. Tan, and W.F. Wong, "JAVM: Internet-based parallel computing using Java," submitted for publication, 2000.

[20] P. Launay and J. Pazat, "A framework for parallel programming in Java," *IRISA Internal Publications*, (1154), 1997.

[21] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1998.

[22] Message Passing Interface Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pp.878-883, IEEE Computer Society, 1993.

[23] MPIJ 1.1. http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html

[24] L. Sarmenta, S. Hirano, and S. Ward, "Towards Bayanihan: Building and extensible framework for volunteer computing using Java," In emphProceedings of ACM 1998 Workshop on Java for High Performance Network Computing, 1998.

[25] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *Computing Surveys*, June 1998.

[26] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, 2(4), pp.315-339, Dec. 1990.

[27] C. Waldspurger and W. Weihl, "Lottery scheduling: Flexible proportional-share resource management," In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Usenix Association, November 1994.

Appendix

```
public class NQueens implements Runnable {
   Continuation success, fa ;
   Continuation failure ;
   private int n; //the total number of queens
   private int i; //already placed queens
   private int[] config; //The current configuration of queens on the chessboard

   public NQueens( Continuation s, Continuation f, int[] a,
   Integer nQueens, Integer placedQueens ){
      ...
   }

   public void run() {
      int j = 0;
      if (i == n) {
         System.out.println("Done");
         for (j = 0; j < i; j++)
            System.out.print( "" + config[j] + " " );
         System.out.println( "" ) ;
         success.sendArgument( config ) ;
         return;
      }

      Continuation x = new Continuation () ; // success
      Continuation y = new Continuation () ; // failure
      ClosureSuccess cSuccess = new ClosureSuccess( this.success, x ) ;
      ClosureFailure cFailure = new ClosureFailure( this.failure, y ) ;

      short count = 0 ;
      for ( j = 0; j < n; j++ ) {
         int[] newConfig = (int[])config.clone();
         if ( safe( newConfig, i, j ) ) {
            count++ ;
            newConfig[i] = j ;
            ClosureNQueens q = new ClosureNQueens( x, y, newConfig, n, i+1 );
         }
      }
      if ( count == 0 ) {
         failure.sendArgument( new Integer( 0 ) ) ;
      }
      else
         cFailure.setJoinCount( count ) ;
      return ;
   }
```

```
   boolean safe(int config[], int n, int j) {
      int r = 0;
      int s = 0;
      for (r = 0; r < i; r++) {
         s = config[r];
         if (j == s || i - r == j - s || i -r == s - j) {
            return false;
         }
      }
      return true;
   }
}

public class Failure extends Task {
   Continuation destination ;
   Integer fail ;

   ...//constructor and helper methods

   public void run () {
  //send failure notification to the failure successor of the parent thread
      destination.sendArgument( fail ) ;
   }
}

public class Success extends Task {
   Continuation destination ;
   int[] config ;

   ... //constructor and helper methods

   public void run () {
      Master.getWorker().abortReadyClosures () ;
  //send configuration to the successor of the parent thread
      destination.sendArgument( config ) ;
   }
}
```