A Comparison of Two Java Runtime Systems for Parallel Execution of ultithreaded Java Applications on Networks of Workstations

Borislav Roussev* Jie Wu†

Abstract

This paper assesses the performance of two Java frameworks for high performance computing (HPC) on networks of workstations (NOWs). The lottery-based work stealing algorithm is intrinsically distributed, and consequetly scalable to an extremely large number of participant workstations. Although proved to be near optimal for the distribution of well-structured multithreaded computations across large number of workstations, no claim has been made regarding the performance of the algorithm for a limited number of workstations or for relatively small computations. In this paper the performance of the lottery-based work stealing algorithm has been compared to that of a centralized work scheduling algorithm to determine whether the lottery-based algorithm is efficient across the board. It was found that, the centralized work scheduling algorithm is more efficient for small number of workstations. This conclusion suggests that HPC will benefit greatly by adopting a composite work scheduling algorithm. This composite algorithm could use the lottery-based work stealing algorithm to distribute computations across large networks, and a centralized algorithm to distribute within smaller networks.

Key words: High Performance Computing, Java, Networks of Workstations.

1 Introduction

For the past 20 years parallel computing has been used successfully in many applications such as weather forecasting, molecular modeling, tax return, etc [1]. Despite some success, 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 [2], 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 NOWs over parallel systems. We can identify the following motivating factors for using NOWs for 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

*Information Systems Department, Susquehanna University, Selinsgrove, Pa 17870, USA, roussev@roo.susqu.edu
†Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Fl 33431, USA, jie@cse.fau.edu

16th IMACS World Congress (© 2000 IMACS)

	Decomposition	Mapping	Communication	Synchronization	Languages
1.	implicit	implicit	implicit	implicit	Haskel
2.	explicit	implicit	implicit	implicit	Concurrent Prolog, Multilisp
3.	explicit	explicit	implicit	implicit	BSP, LogP
4.	explicit	explicit	explicit	implicit	Emerald, Concurrent Smalltalk
5.	explicit	explicit	explicit	explicit	Java; PVM, MPI; Ada

Table 1: Models for parallel computations

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 [3], see Table 1, based on the ways decomposition, mapping, communication, and synchronization are done. Table 1 also shows some representative language/libraries for each model. Decomposition of a program into threads (column 1 of Table 1) and mapping of threads to processors (column 2 of Table 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. Java [4], an object-oriented language, has become popular because of its platform independence and safety. 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. However, Java is firmly fixed at the lowest level of the parallel computing model hierarchy. Several projects use Java as a language for HPC on NOWs and clusters [5, 6, 7, 8, 9, 10]. Invariably their aim is to hide one or more of the characteristics of the language, see Table 1, that make it ill-suited for parallel programming.

The aims of this research work are to develop a Java runtime system for efficient scheduling of multithreaded Java applications on NOWs and to improve the lottery-based work stealing algorithm introduced in [10] for small number of participant workstations. This is achieved by implementing both the lottery-based work stealing algorithm and a centralized work scheduling algorithm. A centralized algorithm is chosen because it exibits far less communication overhead during scheduling than the lottery-based work stealing algorithm does. The centralized algorithm directs participant workstations to a single controller workstation to collect work. The lottery-based work stealing algorithm requires participant workstations to query each other to solicit work. This practice can often result in inefficiencies where a workstation wanting work contacts another workstation without available work, requiring the process to be repeated until work is located.

The remainder of the paper is structured as follows. In Section 2 we review the lottery-based work stealing model [10] and outline a centralized scheduler. In Section 3 we describe the architecture and the implementation of the Java runtime system used to carry out the experiments. Then, in Section 4 we present experimental results about the performance of the runtime system employing the work work scheduling algorithms described in Section 2. In the final section we conclude.

2 Work scheduling models

In our work we use the programming model developed by Robert Blumofe at MIT [11]. 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 the distributed or the centralized scheduler; communication is done implicitly through shared variables; and synchronization is achieved through continuation passing style [15]. We use Java as an implementation language. 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. 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. In Section 2.1 we describe the lottery-based work stealing programming model as well as its victim selection algorithm. In Section 2.2 we present a centralized thread scheduler.

2.1 The Lottery-based work stealing model

In the lottery-based work stealing 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 threads 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 two purposes: First, lowers 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, biases the search to obtain favourable results while at the same time avoid system bottleneck.

2.2 The centralized work scheduling model

In the centralized work scheduling model a single workstation acts as a task distributor. All participant workstations know the transport address of this centralized node and when they become idle they direct their job requests to it. The centralized node maintains a bag of tasks sorted by age and originating from the overloaded workstations.

3 Architecture and implementation of the Java runtime system

A parallel Java program 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 [15]. 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, and assigned pool. 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 the lottery work stealing framework, a pop on an empty ready pool triggers a steal request being sent to a victim worker. The victim worker is chosen using the algorith described in Section 2.1. 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. The thief may then begin work on the stolen closure. If the victim has no ready closures, it informs the thief who then tries to steal from another processor until a ready closure is found or program execution completes.

In the centralized work scheduling model, the request is sent to the centralized node. Then the centralized node responds with the oldest computation in its bag of tasks.

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

# of processors	Lottery-based work stealing	Centralized work scheduling	Improvement in %
1	—	37.2	—
2	—	40.5	—
3	—	29	—
4	42.25	32.75	−22.48
5	35.7	28.5	−20.28
6	47.5	48.6	+2.32
7	44	53.3	+17.44
8	63	71	+11.27
9	69.25	95.3	+27.33

Table 2: Comparison between the performance of the lottery-based work stealing algorithm and the centralized work scheduling algorithm (Nqueen problem)

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. In the lottery-based framework, every 2 seconds the registry multicasts a list of the network addresses and ages of all registered workers.

4 Performance evaluation

One of the assumptions of this research work is that there is a great number of idle CPU cycles. 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. 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.

Given the two scheduling algorithms, it was possible to compare their performances for problem instances of different computational size and number of participant workstations. The performance of the two runtime systems was evaluated using `fibonacci` (double recursive implementation) and `nqueens`. These applications generate a workload suitable for evaluating the performance of the two scheduling algorithms. `fibonacci` is not computationally intensive but spawns a large number of threads (in billions) which makes it appropriate for evaluating the synchronization of the runtime systems. `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 scheduler incurs by wrapping threads in closures and reflecting upon closures to find out threads' constructors.

Table 2 compares the performance of the lottery-based work stealing algorithm to the performance of the centralized work stealing algorithm. Columns 2 and 3 display the wall clock time in seconds for the lottery-based work stealing algorithm and the centralized algorithm, respectively, for different number of processors involved.

Table 2 shows that the centralized algorithm outperforms the lottery-based algorithm for up to five workers. The improvements in running times are not insignificant and for five processors it is approximately 20

5 Conclusion

We have compared the performance of two Java runtime systems for parallel execution of multithreaded Java applications on NOWs. The first runtime system incorporates a distributed lottery-based work scheduler, while the second uses a simple centralized work scheduler. It was found that for a limited number of workstations the centralized work scheduling algorithm is more efficient than the lottery-based work stealing algorithm.

The results show that it would be beneficial to current research activity to adopt a composite work scheduling algorithm. This composite algorithm could use the lottery-based work stealing algorithm to distribute computations across large networks and a centralized algorithm to distribute computations within smaller networks. The motivation for an interest in a limited number of workstations is prompted by two factors. Firstly, today's internetworks from latency problems when internetworks become rather large. This points out that it may be more efficient to restrict

computations from different organizations to their respective LANs rather than scatter the computations to faraway workstations. Secondly, individual organizations with small networks will make considerable use of the distributed processing paradigm before enough confidence exists to extend the paradigm to larger internetworks like the Internet. The major concerns here are: the individual workstation's security, the threat from eavesdropping, the confidence in the results received from an untrusted distant workstation, and the reliability and the fault-tolerance of the distributed work scheduling algorithms.

References

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

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

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

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

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

[6] L. Sarmenta, S. Hirano, and S. Ward, "Towards Bayanihan: Building and extensible framework for volunteer computing using Java," MIT Laboratory for Computer Science, Cambridge, MA, 1998.

[7] P. Gray and V. Sunderman, "IceT: Distributed computing using Java," Emory University, Atlanta, GA, 1997.

[8] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima, "Ninflet: A migratable parallel objects framework using Java," http://ninf.etl.go.jp, 1998.

[9] D. Caromel and J. Vayssiere, "A Java framework for seamless sequential, multithreaded, and distributed programming," INRIA Sophia Antipolis, France, http://www.inria.fr/sloop/javall, 1998.

[10] B. Roussev and J. Wu, "A Java runtime system for parallel execution of mulithreaded Java applications on networks of workstations" In *IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge Massachusetts, USA, November 3-5, 1999.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.