Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Systems Architecture 52 (2006) 432–440
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

# Distributed computing using Java: A comparison of two server designs

Boris Roussev [a,*], Jie Wu [b]

[a] Information Systems Department, University of the Virgin Islands, Box 10,000, Kingshill, VI 00850, US Virgin Islands
[b] Computer Science and Engineering Department, Florida Atlantic University, Boca Raton, Fl 33431, USA

Received 27 January 2003; received in revised form 15 February 2006; accepted 16 February 2006
Available online 31 March 2006

**Abstract**

This paper proposes a new concurrent data structure, called parallel hash table, for synchronizing the access of multiple threads to resources stored in a shared buffer. We prove theoretically the complexity of the operations and the upper limit on the thread conflict probability of the parallel hash table. To empirically evaluate the proposed concurrent data structure, we compare the performance of a TCP multi-threaded parallel hash table-based server to a conventional TCP multi-threaded shared buffer-based server implemented in Java. The experimental results on a network of 36 workstations running Windows NT, demonstrate that the parallel hash table-based server outperforms the conventional multi-threaded server.


*Keywords:* Networking; Distributed computing; Client–server; Concurrent programming; Java

## 1. Introduction

With the growing demand for e-business applications, a net-centric information infrastructure has to stretch a range of scalability to accommodate growth in number of users, number of connections, and complexity of business processes [6,25]. Historically, scalability is achieved through various forms of parallel computing such as symmetric multiprocessing and massively parallel processors. Recently, due to the proliferation of low-cost hardware and the on-going technological convergence of LANs and massively parallel computers, scalability through clustering and networks of workstations (NOWs) [2,5] has been proposed and gained popularity. In NOWs and clusters, the workload is distributed across autonomous computers networked together.

A typical e-business system is architecturally divided into two parts: front-end, which presents data to clients, and back-end, where persistent data is stored and business logic is executed. To scale up such a system in a Windows NT environment, the front-end scales the number of simultaneous users by replicating the web services coupled with a stateless load balancing system to spread the load across the available clones; the back-end scales the complexity of business logic by partitioning data and

* Corresponding author.
*E-mail addresses:* brousse@uvi.edu (B. Roussev), jie@cse.fau.edu (J. Wu).

1383-7621/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.02.001

services across multiple specialized servers [24]. In both scenarios, scalability is achieved through a loose confederation of computers, providing fault-tolerant redundancy, load balancing, and computing power [32].

Since many computationally intensive applications exchange a large amount of messages, network communication latencies are often a bottleneck in such servers as HTTP request dispatchers, load balance brokers, name servers hosting centralized JNDI trees, dedicated state servers and centralized databases for HTTPSession failover in J2EE clusters [16], as well as traditional applications like Linda based frameworks for parallel computing [11]. In principal, clusters and NOWs bring down the hardware price/performance ratio, but in practice, the large network latencies involved in communicating among workstations make them low-performance parallel computers, suitable for running mainly coarse-grain processes concealing the network latency [9]. Therefore, the key to NOWs is the improvement of inter-process and inter-thread data communication efficiency.

This paper deals with the issue of concurrent access to a shared buffer implemented in Java and with high-performance networking. In particular, we propose a new concurrent data structure, called *parallel hash table* (PHT), for solving the synchronization problem in the classical producer/consumer model in a way that increases the performance of Java networking significantly. The PHT allows concurrent access of multiple consumers and a single producer to a shared buffer. It is a variant of a parallel dictionary [3] without the assumption that insert, delete and search instructions are presented to the dictionary in batches; and that each batch contains only one kind of instructions. Based on the PHT, we propose a new client–server design alternative. The new server design is implement in Java. We prove theoretically the upper limit of the conflict probability in the PHT and test the new server design on a network of 35 workstations running Windows NT. The presented performance results demonstrate the benefits from the proposed concurrent data structure. We also present results showing that for server designs, better performance is achieved by using a greater number of worker threads, in the range of 50–70 rather than in the range of 15 as routinely recommended [33].

Our work relates to existing work on concurrent access in the field of parallel computing. The literature separates concurrent access algorithms into three disjoint groups [19]. Traditional algorithms [1,21,29] are locking: a process must obtain a mutually exclusive lock to enter a critical section, thereby preventing other processes from entering concurrently. When such locks are used, a process stalled inside a critical section can delay all others for an arbitrary amount of time, a behavior termed blocking. Non-blocking algorithms [13,23,31] guarantee that some process makes progress in a finite amount of time, which implies that they do not enforce mutual exclusion. Non-blocking algorithms eliminate the need for data buffer sharing between threads and reduce thread synchronization. In the Leader/Followers pattern [31], one thread, the leader, waits for a client request to arrive on a handle set, while all other threads, called followers, are queueing up waiting their turn to become the leader. Upon detecting a new client request, the current leader promotes a follower to become the new leader thread, processes the request, and then reverts to a follower thread, waiting to become the leader thread again. Generally, non-blocking algorithms may increase the dispatching latency for simultaneous requests. It is also harder to implement request buffering and thread borrowing when priority thread pool segmentation is used because there is no explicit queue, which can lead to poor scalability [26]. Lock-free algorithms, the third group of algorithms, do not use locks, but can still result in blocking behavior [4,17,19,34].

Each type of algorithms has its own advantages and disadvantages. Lumetta and Culler [19] studied performance over a range of access contention tests from the message-passing literature. Stevens [33] discussed a number of client–server design alternatives implemented in C using POSIX threads [27] and contrasts their performance within a common experimental setup. Stevens' experimental results show that the overall TCP server response time depends entirely on the protocol used to synchronize the work of the main server thread accepting the connected sockets (the producer) and the worker threads processing the client requests (the consumers). This motivates our quest for a synchronization protocol that will decrease the overhead incurred by the worker threads when receiving a new client request from the main server thread. A non-blocking or a blocking concurrent queue algorithm reported in [22] can be used as the cornerstone of a pre-threaded server design. A large body of conservative check-and-act concurrent design patterns in Java can be found in [18,30]. The proposed

algorithm is most similar to preemption-safe locking solutions such as POSIX [27].

The remainder of the paper is structured as follows. Section 2 reviews Java network and concurrent programming and multithreading. Section 3 discusses a pre-threaded server design, in which references to connected sockets are passed through a shared buffer. Section 4 describes the PHT structure and operation, and presents a pre-threaded server design, in which references to connected sockets are passed through a PHT. Section 5 analyzes the complexity of the PHT operations and the PHT conflict probability. Section 6 reports performance results for the two server designs, and the last section concludes.

## 2. Java network and concurrent programming

The popularity of Java is generally due to its platform independence, pure object-orientation, and execution safety [12]. Recently, there has been a great upsurge in interest in the use of Java as a language for high performance computing on NOWs and clusters [10]. Java is believed to have the potential to supersede Fortran 90 and C/C++ for science and engineering as well as for systems implementation. Java enormously simplifies network programming by providing elegant TCP/IP API, object serialization, network class loading, i.e., code mobility, RMI, Servlets, JSP, Java Naming and Directory Interface, Java Transaction API, and built-in concurrent constructs.

As net-centric computing is a critically important foundation of the digital economy [20], a distributed information system typically has significant networking and data communication requirements [11,33]. To be successful in the distributed computing domain, Java should be instrumental in providing not only high computational performance, but also high-performance networking and concurrent constructs. Today, many new technologies, like Java Virtual Machine (JVM), multiprocessor, and multithreaded kernels, have matured. In JVM, for example, the time required to spawn a new thread or to obtain an object's lock in most implementations is little, yet not negligible. The use of multiprocessors is a norm. Mapping of threads to processors has been optimized tremendously. In the JDK 1.4 release, the new I/O packages, java.nio.*, finally address Java's long-standing shortcomings in the aspect of high-performance, scalable I/O [35]. These packages introduce several key abstractions, namely Buffer, Channel, and Selector, which work together with non-blocking reads toward solving the problems in traditional Java I/O. As a result, designs that have been impossible until recently become viable propositions.

In this paper we use the TCP protocol, which is a connection-oriented protocol that provides a reliable and full-duplex byte stream for a user process or thread [28]. TCP can use either IPv4 or IPv6 [8]. The Java class libraries relevant to network programming are java.net, java.io and java.nio.*. At the core of Java's TCP sockets support are the Socket, ServerSocket, InetAddress, InetSocketAddress, SocketChannel, ServerSocketChannel and Selector classes. To send and receive data over a TCP connection, Java provides a rich collection of stream classes, subclassing the abstract InputStream and OutputStream, providing methods for character-based data exchange.

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 [18]. The java.lang.Thread class contains methods for creating, controlling, and synchronizing Java threads. Java monitors are a relatively simple and expressive model that allows threads to implicitly serialize their execution and to coordinate their activities via explicit wait, notify, and notifyAll operations. However, Java monitors lack true conditional variables, and as a result, all threads are forced to use a single built-in Boolean variable. This forces developers to use the notifyAll( ) method when there are multiple threads waiting on different conditions within the same monitor, instead of calling the less expensive, single notification method notify( ). Writing portable multithreaded Java applications remains problematic for many reasons: (1) non-standard scheduling semantics—cooperative versus preemptive; (2) haphazard thread notification; (3) priority inversion problem caused by the haphazard notification semantics of notify; (4) differences in the number of priority levels in major operating systems, e.g., Windows NT and Solaris, and the availability of priority boosting in NT practically preclude the use of thread priorities as a control mechanism [14,15].

## 3. Server design using a shared buffer

We run multiple instances of the same client against each server: java TCPClient IP port nchildren nrequest [millisec | filename].

We specify the IP address of the server, its port number, the number of child threads for the client to spawn, the number of requests each child thread should send to the server, and the number of milliseconds the server should delay its response or the file name to request the server to return each time. The time interval is used to simulate the processing of a client request. It is important to note that the client closes the connection after receiving the server's response, so TCP's TIME_WAIT state occurs on the client rather than on the server. The experimental setup is similar to those of dedicated state servers used in J2EE clusters and HTTP servers.

The two types of servers extend a common abstract class, and implement its abstract method handleRequest( ), which processes a single client request. In addition, each server extends the super class constructor to kick off the required worker threads, monitors, locks, barriers and/or buffers used for synchronization and communication.

The concurrent TCP server uses pre-threading. Instead of spawning a new thread per client request, the server pre-spawns a pool of worker threads, as is illustrated in Fig. 1. The server thread accepts client requests and passes the connected sockets to the worker threads through a bounded buffer. The buffer synchronizes the work of the main server thread and the worker threads. It is implemented as a fixed array of references to sockets along with two indices that circularly traverse the array, keeping track of the next position to put and take, respectively. This is a classical producer/consumer solution where a monitor is used to synchronize the access to the shared buffer. The disadvantage of this approach is the so-called *thunder herd* problem. All worker threads are awakened even though only one can obtain the lock guarding the shared buffer. The single-entry point lock quickly degrades performance under a great number of contentions, especially on multiprocessors.

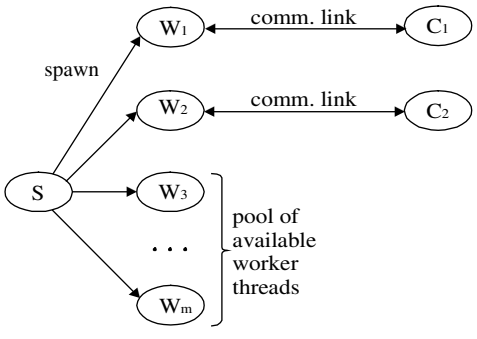


Fig. 1. The server thread $S$ pre-threads worker threads $W_i$ to serve clients $C_j$.

## 4. PHT-based server design

A dictionary is a dynamic set that supports the operations insert, search, and delete. A hash table is a data structure for implementing dictionaries [7]. The expected time to search for an element in a hash table is O(1). With hashing, an element $x$ with key $k$ is stored in slot $h(k)$, where $h$ is a hash function used to compute the slot index from the key $k$.

A PHT is a hash table whose slots can be accessed concurrently by multiple threads. We assume that at most one thread at a time, called *producer*, can execute insert and multiple threads, called *consumers*, can execute search and delete. PHT collisions are resolved using open addressing [7]. Each element in a PHT is of type SynchronizedRef [18]. SynchronizedRef is a class maintaining a single reference variable accessed and updated under synchronization. Using SynchronizedRef objects as PHT elements allows the system to loosen the synchronization among consumer threads, and between the producer thread and the consumer threads by employing an optimistic control strategy. In the optimistic approach, concurrent threads are not serialized by a global lock, as it is in the conventional server design, presented in the previous section. Instead, each thread can access any slot without acquiring a buffer-wide synchronization lock as a precondition, thus, creating favorable conditions for enhanced performance. There are two different cases to consider:

- The insert method, called by the producer, uses the hashCode( ) method of the Object class to compute the element key and double hashing [7] to compute the PHT slot.
- Consumer threads retrieve elements from the PHT using method search( ). They pass as an argument to search( ) a pseudo-randomly generated number, used to calculate the first slot to be checked for an element. If the slot is not null, the element is retrieved, and the slot is set to null. Otherwise, the next slot in the probe sequence is calculated and checked.

If a conflict occurs, i.e., two or more consumers access the same slot, because the PHT slots are objects of type SynchronizedRef, only one of the consumer threads will retrieve the element, and set

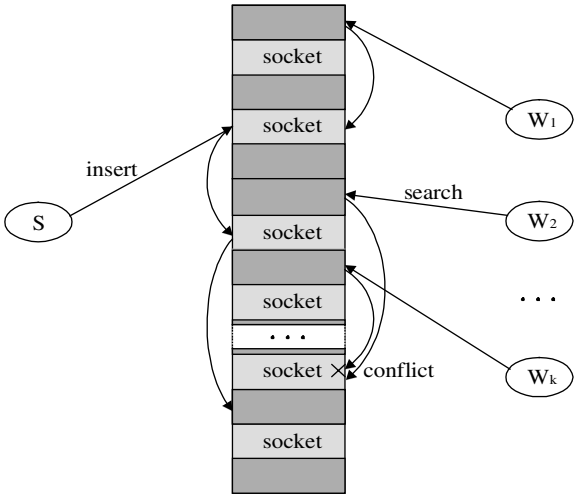


Fig. 2. Operation and structure of the PHT-based server design.

the slot to null. The rest will see the null value and try to retrieve an element from other slots.

Next, we introduce a PHT-based server design using pre-threading. In this design, the references to the connected sockets are passed through a PHT. The structure and operation of the PHT server are shown in Fig. 2. The main server thread, $S$, accepts client requests and inserts the connected sockets in the PHT using the table's insert( ) method. The worker threads, $W_i$, retrieve connected sockets from the PHT using the table's search( ) method.

## 5. Theoretical analysis of parallel hashing

The load factor $\alpha$ for a PHT $T$ with $m$ slots and $n$ elements is $n/m$, and it cannot exceed 1. In our analysis, we make the assumption of *uniform hashing*, i.e., each key is equally likely to have any of the $m!$ permutations of $\{0,1,\ldots,m-1\}$ as its probe sequence. In the PHT implementation we use double hashing [7], which is a suitable approximation for uniform hashing.

**Theorem 1.** *Inserting an element into a PHT with load factor $\alpha < 1$ requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.*

The proof is analogous to that of the corresponding theorem for standard hashing presented in [7]. If $\alpha$ is a constant, Theorem 1 predicts that inserting an element runs in O(1) time.

**Theorem 2.** *In a PHT with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $1/\alpha$.*

**Proof.** Threads retrieve elements from the PHT not by key but by using a pseudo-random probe sequence. In a successful search, every probe but the last accesses an empty slot and the last slot probed is occupied. Let us define $p_i = \Pr\{\text{exactly } i \text{ probes access empty slots}\}$ for $i = 0,1,2,\ldots$ For $i > (m-n)$, we have $p_i = 0$, since we can find at most $(m-n)$ empty slots. Thus the expected number of probes is

$$1 + \sum_{i=0}^{\infty} i p_i. \tag{1}$$

To evaluate (1) we define $q_i = \Pr\{\text{at least } i \text{ probes access empty slots}\}$ for $i = 0,1,2,\ldots$ Since $i$ takes on values from the natural numbers

$$\sum_{i=0}^{\infty} i p_i = \sum_{i=1}^{\infty} q_i.$$

The probability that the first probe accesses an occupied slot is $q_1 = (m-n)/m$. A second probe, if necessary, is to one of the remaining $m-1$ unprobed slots, $n$ of which are occupied. We make a second probe only if the first probe accesses an empty slot. Thus,

$$q_2 = \left(\frac{m-n}{m}\right)\left(\frac{(m-n)-1}{m-1}\right).$$

The $i$th probe is made only if the first $i-1$ probes access empty slots. Thus,

$$q_i = \left(\frac{m-n}{m}\right)\left(\frac{(m-n)-1}{m-1}\right)\cdots\left(\frac{(m-n)-(i-1)}{m-(i-1)}\right) \leqslant \left(\frac{m-n}{m}\right)^i.$$

After $(m-n)$ probes, all $(m-n)$ empty slots have been seen and will not be probed again. Thus $q_i = 0$ for $i > (m-n)$. Now, we can evaluate (2). Given the assumption that $\alpha < 1$, the average number of probes in a successful search is

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\leqslant 1 + \frac{m-n}{m} + \left(\frac{m-n}{m}\right)^2 + \left(\frac{m-n}{m}\right)^3 + \cdots \\ &= \frac{1}{\alpha}. \end{aligned}$$

□

If $\alpha$ is a constant, Theorem 2 predicts that searching an element runs in O(1) time.

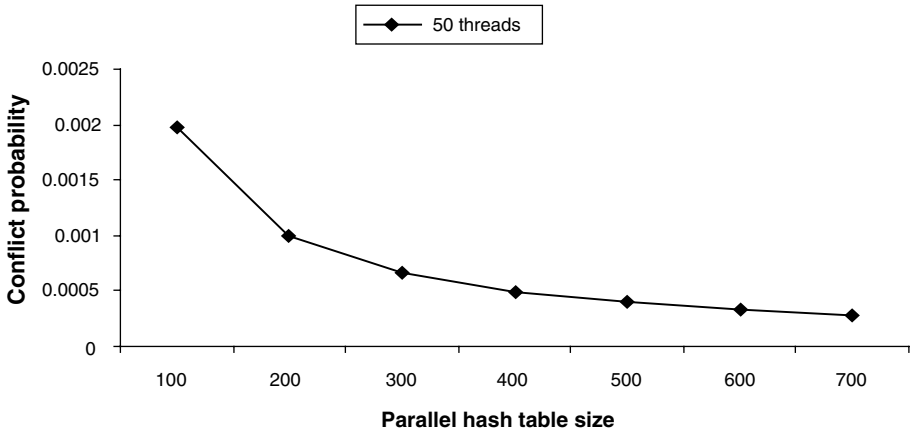


Fig. 3. Conflict probability in a PHT with 50 worker threads.

Next, we consider the net effect of multiple threads accessing the shared buffer and the number of conflicts that can occur in the PHT.

**Theorem 3.** *Given a PHT with m slots and k worker threads, the probability P of having two or more threads in conflict, i.e., accessing the same PHT slot, is given by the formulae*

$$\sum_{x=1}^{n}\binom{n}{x}\left(\frac{1}{m}\right)^{x}\left(1-\frac{1}{m}\right)^{n-x},$$

*where* $n$, $n = k/(t_r/t_a)$, *is the average number of threads accessing the PHT during the time interval* $t_a$ *necessary for retrieving one element from the PHT.* $t_r$ *is the time need by a worker thread to process one request.*

**Proof.** The average number of threads accessing the PHT during the time span of one request retrieval is $n$. Let us assume that a worker thread is retrieving a request from slot $i$, $1 \leqslant i \leqslant m$. Then, the probability of a second thread accessing slot $i$ is $1/m$; the probability of a third thread accessing slot $i$ is $1/m^2$; ...; the probability of a $j$th thread accessing the same slot is $1/m^{(j-1)}$. This can be modeled with the Binomial probability distribution with $n$ number of trials and probability of success $1/m$, which remains constant from trial to trial

$$p(x) = \binom{n}{x}\left(\frac{1}{m}\right)^{x}\left(1-\frac{1}{m}\right)^{n-x}.$$

We are interested in the cumulative probability $P$ $(x \leqslant 1)$ of two or more threads accessing slot $i$. The cumulative probability $P$ is given by the formulae

$$\sum_{x=1}^{n}\binom{n}{x}\left(\frac{1}{m}\right)^{x}\left(1-\frac{1}{m}\right)^{n-x} = \sum_{x-1}^{n}\left(\frac{n!}{x!(n-x)!}\right)\left(\frac{1}{m}\right)^{x}\left(1-\frac{1}{m}\right)^{n-x}. \quad \square$$

Fig. 3 shows the correlation between the PHT size and the conflict probability for the case of 50 worker threads.

## 6. Experimental results

To compare the performance of the two server designs, using a shared buffer and a PHT, respectively, we conducted a simulation experiment on a network of 36 single-processor workstations. Each workstation was equipped with a 1-GHz CPU and 256 MB of memory, and ran J2SE 1.4.0 Client VM with enabled JIT compiler over Windows NT. For each server design, we ran multiple experiments using different numbers of worker threads and buffer capacities to find out the optimal configuration. Each experiment was repeated 10 times.

In all tests, we ran multiple instances of the same client against each server and measured the time required to process a fixed number of requests. All server CPU timings were obtained by running the same client on 35 different hosts located on the same subnet as the server. In all the tests, each client spawned 4 child clients to create 4 simultaneous connections to the server, for a maximum of 140 (35 machines $\times$ 4) simultaneous connections at the server at any time. Each client makes 20 connections to the server, amounting to 2800 connections altogether. Each client sends 1 byte to the server, which

| # | Server description | Avg. CPU time | Avg. CPU time |
|---|---|---|---|
| 1 | 50 worker threads and 101-slot buffer | 16562 | 28960 |
| 2 | 50 worker threads and 101-slot PHT | 14562 | 24265 |

Fig. 4. Performance comparison between the two server designs.

in turn responds with 4000 bytes after waiting for a predefined interval of time specified in milliseconds. According to Theorem 3, the conflict probability for the experimental setup with 500 ms processing request time $t_r$ is 0.0019703951, given that the time to retrieve a request from the PHT $t_a$ is 2 ms.

The experiment results are summarized in the table in Fig. 4. Both servers are configured to have 50 worker threads and 101-slot buffer capacity. The CPU time recorded in columns 3 and 4 corresponds, respectively, to 250 and 500 ms server delay before sending back the response. It simulates the time, $t_r$, needed by a worker thread to process a request. The buffer capacity is measured by the number of its slots. Since each slot can hold at most one reference to a connected socket, at any moment, the number of client sockets accepted by the main server thread and not yet processed can be at most equal to the buffer capacity.

The results in Fig. 4 demonstrate that the proposed PHT-based design can improve the communication efficiency by 12% and 16% respectively for 250 and 500 ms server delays, when compared with the traditional shared buffer technique. The significant improvement gained justifies the small implementation complexity added by the proposed method. The implementation of the PHT operations and the SynchronizedRef class amounts to 60 lines of Java code.

The superior performance of the PHT server is due to the fine-grained serialization level used by the PHT. In the PHT server, the global lock guarding the shared buffer has been replaced by locks, guarding the individual elements. As a result, multiple threads can access different parts of the PHT without blocking each other, thus creating conditions for increased network performance. In this context, the server performance depends on two factors: the number of probes in each operation and the number of thread conflicts. Both theoretical analysis and experimental results indicate that the numbers of probes and conflicts and their associated delays do not offset the performance edge gained by reducing the isolation level in the PHT. Theorem 2 predicts that when the server is overloaded, e.g., a table with $\alpha = 0.95$, the average search will take 1.05 probes, meaning that a worker thread can quickly retrieve a socket and attend to a pending request.

The number of worker threads and the buffer capacities recorded in the table in Fig. 4 were found experimentally to give the best results for both servers. We have also found that, for both server designs, better performance can be achieved by using a greater number of worker threads, in the range of 50–70, rather than in the range of 15 as routinely recommended. These results are presented in the table in Fig. 5.

A few limitations regarding our experiment design must be acknowledged. First, due to limited resource availability, we have used only single-processor workstations for the experiment. On a multi-processor computer, we expect better experimental results for the PHT-based server. Second, we have used server CPU time as the primary measure of performance when comparing the two server designs. We realize that the efficiency of the server processing is only one aspect of the performance of a distributed client–server system. Another important determinant is the number of conflicts, which reflects how efficiently the server handles client requests. Even though we theoretically established the upper limit of the conflict probability, the real case scenario will be understandably better than the theoretical limit.

| # threads | 10 | 30 | 40 | 50 | 60 | 70 |
|---|---|---|---|---|---|---|
| Shared buffer CPU time | 74118 | 26506 | 22347 | 16562 | 18649 | 19077 |
| PHT CPU time | 71535 | 24325 | 17561 | 14562 | 17328 | 18240 |

Fig. 5. Effect of threads number on the performance the two servers.

## 7. Conclusion

In this work, we introduced a new concurrent data structure, called PHT, as an extension to the standard hash table. We presented a new server design based on the PHT. The experimental results show that on average the PHT-based server outperforms the conventional shared buffer server by 14%. We proved theoretically the complexity of the PHT operations, as well as the thread conflict probability. We considered the number of worker threads a server should spawn in order to get maximum performance. We found out that better performance is achieved by using a greater number of worker threads, in the range of 50–70, rather than in the range of 15 threads, as it is routinely recommended.

## References


[1] T.E. Anderson, The performance of a spin lock alternative for shared-memory multiprocessors, IEEE Trans. Parallel Distribut. Syst. 1 (1) (1990).

[2] T.E. Anderson, D.E. Culler, D.A. Pattersonthe NOW Team, A case for NOW, IEEE Micro 15 (1) (1995).

[3] H. Bast, M. Dietzfelbinger, T. Hagerup, A perfect parallel dictionary, in: Proc. of the 17th Int'l Symposium on Mathematical Foundations of Computer Science, Prague, 1992.

[4] E. Brewer, F. Chong, L. Liu, S.Sharma, J. Kubiatowicz, Remote queues: exposing message queues for optimization and atomicity, in: Symp. on Parallel Algorithms and Architectures, 1995.

[5] B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, D. Wu, Javelin: Internet-based parallel computing using Java, Concurrency: Practice and Experience 9 (11) (1997) 1139–1160.

[6] D. Clark, D. Estrin, P. Green, J. Kurose, B. Leiner, L. Masinter, J. Pasquale, D. Sincoskie, K. Sollins, Strategic directions in networks and telecommunications, ACM Comput. Survey 28 (4) (1996) 679–690.

[7] T. Cormen, C. Leiserson, R. Riveset, Introduction to Algorithms, The MIT Press, 1994.

[8] S. Deering, R. Hinden, Internet Protocol, version 6 (IPv6) specification, RFC 2460, 1998.

[9] T. von Eicken, A. Basu, V. Buch, Low-latency communication over ATM networks using active messages, IEEE Micro 15 (1) (1995).

[10] G. Fox, W. Furmanski, Java for parallel computing and as a general language for scientific and engineering simulation and modeling, Concurrency: Practice and Experience 9 (6) (1997) 415–425.

[11] D. Gelernter, Generative communication in Linda, ACM Trans. Progr. Languages Syst. 7 (1) (1985) 80–112.

[12] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Sun Microsystem, Inc., Palo Alto, CA, 1996.

[13] M. Greenwald, D. Cheriton, The synergy between non-blocking synchronization and operating system structure, Operating Systems Design and Implementation, 1996.

[14] A. Holub, Programming Java threads in the real World, JavaWorld, September 1998.

[15] P. Jain, D.C. Schmidt, Experiences converting a C++ communication software framework to Java, http://www.cs.wustl.edu/.

[16] A. Kang, J2EE clustering, JavaWorld, February 2001.

[17] V. Karamcheti, A. Chien, A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D, in: Int'l Symp. on Computer Architecture, June 1995, pp. 298–307.

[18] D. Lea, Concurrent Programming in Java, second ed., Addison-Wesley, 1999.

[19] S. Lumetta, D. Culler, Managing concurrent access for shared memory active messages, in: Proc. of IPPS/SPDP'98, Orlando, FL, 1998.

[20] S. March, A. Hevner, S. Ram, Research commentary: an agenda for information technology research in heterogeneous and distributed environments, Inform. Syst. Res. 11 (4) (2000).

[21] J. Mellor-Crummey, M. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Syst. 9 (1) (1991).

[22] M. Michael, M. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in: Proc. of the 15th ACM Symp. on Principles of Distributed Computing, PA, 1996, pp. 267–276.

[23] M. Michael, M. Scott, Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors, in: International Parallel Processing Symp., 1997.

[24] Microsoft Corporation, A Blueprint for Building Web Sites Using the Microsoft Windows DNA Platform, White Paper, 2000.

[25] NSF, Exploratory Research on Scalable Enterprise System, NSF 99–149, 1999.

[26] I. Pyarali, M. Spivak, R. Cytron, D.C. Schmidt, Evaluating and optimizing thread pool strategies for real-time CORBA, in: Proc. of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Embedded Systems, 2001, pp. 214–222.

[27] Portable Operating Systems Interface (POSIX), http://standards.ieee.org.

[28] J. Postel (Ed.), Transmission Control Protocol, RFC 793, 1981.

[29] D. Reed, R. Kanodia, Synchronization with eventcounts and sequencers, Commun. ACM 22 (2) (1979).

[30] B. Roussev, J. Wu, Client–Server design alternatives: back to pipes but with threads, in: Proc. of IEEE Int'l Conference on Networking, France, 2001.

[31] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2, Wiley & Sons, NewYork, 2000.

[32] M.A. Sportack, Windows NT Clustering Blueprints, SAMS Publishing, 1997.

[33] W.R. Stevens, Unix Network Programming, vol. 1, second ed., Prentice-Hall, 1998.

[34] J. Valois, Implementing Lock-Free Queues, in: Int'l. Conf. on Parallel and Distributed Computing Systems, 1994.

[35] J. Zukowski, New I/O functionality for J2SE 1.4. Available from: <http://java.sun.com>.

**Dr. Roussev** is an Associate Professor of Information Systems at the University of the Virgin Islands. His diverse background includes teaching and research experience in Europe, South Africa, and the US. Boris Roussev's interests are in the areas of object-oriented and economic-driven software development, requirements engineering and project management. He conducts research on causes of project failures, working from the presumption that value-neutral principles and practices in software development are unable to deal with the sources of project failures, and that in order to manage effectively the computer technology, one has to consider the social context in which it is deployed and produced. In addition, Dr. Roussev has experience in software risk management and software process quality factor analysis. All of the above is combined with industry experience in software methods such as domain engineering, product lines, MDA with xUML, generative programming, and aspect-oriented programming.

**Dr. Jie Wu** is a Professor at the Department of Computer Science and Engineering, Florida Atlantic University. He has published over 300 papers in various journal and conference proceedings. His research interests are in the area of mobile computing, routing protocols, fault-tolerant computing, and interconnection networks. Dr. Wu served as a program vice chair for 2000 International Conference on Parallel Processing (ICPP) and a program vice chair for 2001 IEEE International Conference on Distributed Computing Systems (ICDCS). He is a program co-chair for the IEEE 1st International Conference on Mobile Ad-hoc and Sensor Systems (MASS'04). He was a co-guest-editor of a special issue in IEEE Computer on "Ad Hoc Networks". He also editored several special issues in Journal of Parallel and Distributing Computing (JPDC) and IEEE Transactions on Parallel and Distributed Systems (TPDS). He is the author of the text "Distributed System Design" and is the editor of the text "Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks". Currently, Dr. Wu serves as an Associated Editor in IEEE Transactions on Parallel and Distributed Systems and several other international journals. Dr. Wu is a recipient of the 1996–1997 and 2001–2002 Researcher of the Year Award at Florida Atlantic University. He served as an IEEE Computer Society Distinguished Visitor and is the Chairman of IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a Member of ACM and a Senior Member of IEEE.