Towards Efficient Shared Memory Communications in MPJ Express

Aamir Shafi
aamir.shafi@seecs.edu.pk
School of Electrical Engineering and Computer Science
National University of Sciences and Technology
Pakistan

Jawad Manzoor
jawad.manzoor@seecs.edu.pk
School of Electrical Engineering and Computer Science
National University of Sciences and Technology
Pakistan

Abstract

The need to increase performance while conserving energy lead to the emergence of multi-core processors. These processors provide a feasible option to improve performance of software applications by increasing the number of cores, instead of relying on increased clock speed of a single core. The uptake of multi-core processors by hardware vendors present variety of challenges to the software community. In this context, it is important that messaging libraries based on the Message Passing Interface (MPI) standard support efficient inter-core communication. Typically processing cores of today's commercial multi-core processors share the main memory. As a result, it is vital to develop devices to exploit this. MPJ Express is our implementation of the MPI-like Java bindings. The software has mainly supported communication with two devices; the first is based on Java New I/O (NIO) and the second is based on Myrinet. In this paper, we present two shared memory implementations meant for providing efficient communication of multi-core and SMP clusters. The first implementation is pure Java and uses Java threads to exploit multiple cores. Each Java thread represents an MPI level OS process and communication between these threads is achieved using shared data structures. The second implementation is based on the System V (SysV) IPC API. Our goal is to achieve better communication performance than already existing devices based on Transmission Control Protocol (TCP) and Myrinet on SMP and multi-core platforms. Another design goal is that existing parallel applications must not be modified for this purpose, thus relieving application developers from extra efforts of porting their applications to such modern clusters. We have benchmarked our implementations and report that threads-based device performs the best on an Intel quad-core Xeon cluster.

Key Words: Multi-core Programming, Shared Memory Communications, Java HPC, MPJ Express

1 Introduction

These are exciting times for the computer software and hardware industry. The evolution of multi-core processors is no less than a revolution. Processors vendors like Intel and AMD have shifted their business model from increasing clock speed of a single power-hungry core to increasing energy-efficient processing cores—this was observed in an influential article by Sutter et al [10]. Scaling performance by increasing the clock speed of a single processor is inefficient since the power consumed is proportional to (at least) the square of the clock rate. At some point, it is not practical to increase the clock rate further, as the power consumption and cooling requirements would be excessive.

Multi-core processors promise to provide improved performance for applications by supporting multiple lightweight processing elements or "cores" per processor chip. But an application must exploit thread-level parallelism to fully utilize the potential of multi-core processors. This sea-change in processor architecture will have tremendous impact on computer software—Dongarra et al [5] discuss this in the context of computational science software.

In the early 1990s, the Message Passing Interface (MPI) [8] emerged as a standard for writing parallel applications on distributed memory networked computers like clusters. The MPI standard provides bindings for C, C++, and For-

tran programming languages. There have been efforts to produce bindings for other popular languages like Java. The Java Grande Forum—established due of interest of HPC community in using Java—has produced two major bindings versions; the mpiJava 1.2 API and the MPJ API. There are several Java MPI-like libraries available, which include MPJ Express, mpiJava, and MPJ/Ibis.

The MPJ Express Java messaging system is our open-source and free implementation of the mpiJava 1.2 API. This library currently provides two communication devices. The first one is based on the Java New I/O (NIO) package and is meant to use on interconnects like Fast and Gigabit Ethernet. The second communication device is based on the Myrinet eXpress (MX) [9] library. Other unique features of the software include the explicit buffer management layer and portable runtime system.

Currently there is no efficient mechanism for programming multi-core High Performance Computing (HPC) systems with the MPJ Express software. The main reason is lack of support for optimized communication on such processors. Users of the MPJ Express software typically rely on Java NIO or MX device for communication between processing cores within a node. These devices have certain degree of overhead for this kind of communication.

This paper presents the design and implementation of shared memory devices for the MPJ Express software. Here we implement two devices; the first one uses plain Java threads and the second one uses the UNIX System V (SysV) IPC library. The main aim is to achieve best possible performance for intra-node communication. We also evaluate performance of our implementations on a Linux cluster with quad-core processing nodes.

In this paper we discuss different approaches for the design and implementation of shared memory communication device for MPJ Express. These two approaches include Java threads and UNIX SysV IPC libraries and are highlighted in Figure 1. Further, we discuss the overhead during intra-node communication in the present design of MPJ Express and explain how this overhead can be avoided to boost the performance by implementing shared memory communication between processors on the same node.

Rest of the paper is organized as follows. Section 2 presents the related work. This is followed by a discussion of our two approaches to implementing shared memory devices in the Section 3. Section 4 evaluates performance of these communication devices. We conclude and discuss future work in Section 5.

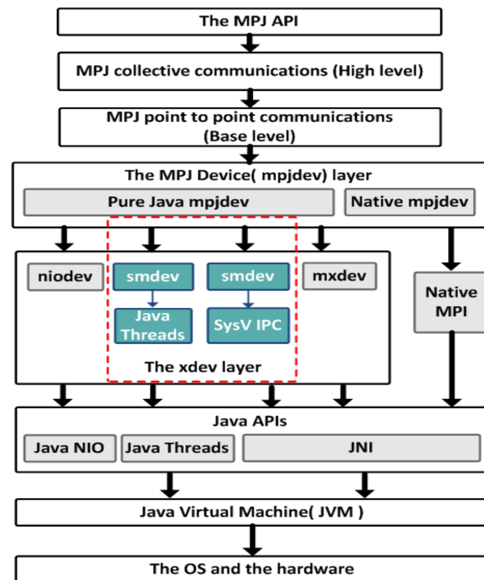


Figure 1. MPJ Express Architecture

2 Related Work

In the last decade many Java messaging systems have emerged. This clearly shows HPC community's interest in using Java for large scale applications. Mainly there have been three approaches in building such libraries; use pure Java, rely on Java Native Interface (JNI), or use some higher messaging API like Remote Method Invocation (RMI).

mpiJava [1] provides a fully functional and efficient object-oriented Java interface to MPI. A unique feature of this messaging library is that it can work with any native C MPI library and thus can indirectly benefit from shared memory devices in these native libraries. MPJ/Ibis [2] is an implementation of the MPJ API specification on top of Ibis. The communication devices used by MPJ/Ibis are not thread-safe. MPJ/Ibis has communication devices for `java.io` package, Java NIO package, and Myrinet. This software currently does not provide any shared memory communication device.

To the best of our knowledge, none of the existing Java messaging system implement a shared memory communication device. mpiJava is unique in this context because it can utilize the shared memory communication device of the native C MPI library. On the other hand, popular C MPI libraries including Open MPI and MPICH2 provide high performance shared memory communication devices on various platforms.

Open MPI [6] is an open source production quality implementation of both MPI-1 and MPI-2. Open MPI is a merger of LAM/MPI, LA-MPI, and FT-MPI. It fully supports concurrent and multi-threaded application. To efficiently support a wide range of parallel machines, high performance drivers for TCP/IP, shared memory, Myrinet, Quadrics, and Infiniband are being developed. Open MPI includes the features of thread safety and concurrency, dynamic process spawning, high performance on all plat-

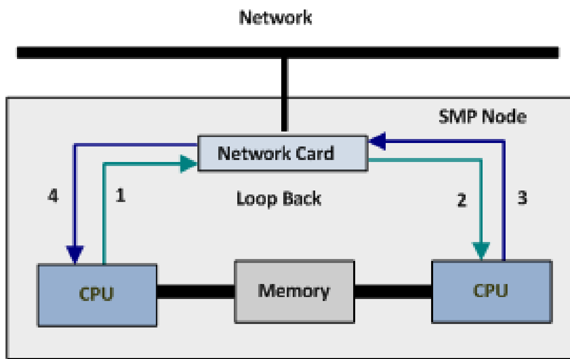


Figure 2. Existing Intra-node Communication Design

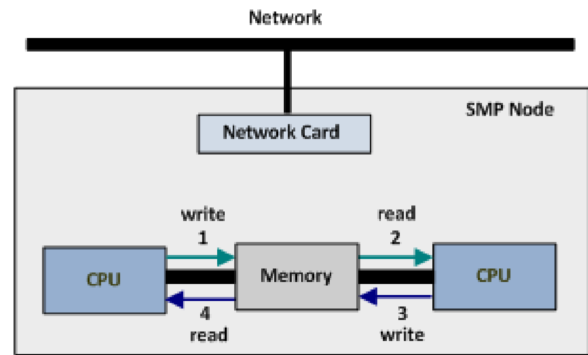


Figure 3. Optimized intra-node communication design

forms, portable and maintainable, component-based design and documented APIs.

MPICH2 [7] is another implementation of MPI. MPICH2 is designed for implementation of MPI on SMPs, clusters, and massively parallel processors. The MPICH2 library provides a variety of shared memory communication libraries, which include `ch3:shm`, `ch3:ssm`, and `ch3:sshm`. In addition, MPICH2 has a communication subsystem called Nemesis [3]. A unique aspect is the use of lock-free queues in this subsystem, which increases scalability as number of processors or cores increase.

3 Shared Memory Communication Devices for MPJ Express

Currently the MPJ Express software does not provide shared memory communication device. Users of the software rely of Java NIO and Myrinet device for intra-node processor communication. This kind of communication typically involves a loop back from the network card— this is depicted in Figure 2.

Ideally communication between processors or cores within a single node must be done via the memory bus instead of the network card. This is depicted in Figure 3 and forms the basis of our two implementations discussed in this section.

Both shared memory communication devices are implementations of the xdev device layer as shown in Figure 1. The xdev device layer only defines blocking and non-blocking point-to-point communication methods. Advanced MPI features including the collective communication, derived datatypes, and virtual topologies are implemented at the higher levels of the MPJ Express software.

In this section, we first cover the threads-based shared memory communication device. This is followed by a discussion of SysV IPC library implementation of the xdev layer.

3.1 Threads-based Device

An innovative way to build a shared memory device for a Java messaging system like MPJ Express is to rely on Java threads. This idea originally comes from the shared memory implementation of the Adlib communication library for HPJava [4]. Using this approach, each MPJ process is essentially represented by a Java thread and data is communicated using shared data structures. An obvious advantage of this approach—especially in the context of Java—is that an application does not compromise portability. Other shared memory devices rely on the JNI API and some underlying native implementation, which obviously varies for different OS platforms. Another advantage of this approach is better performance since we can avoid JNI and additional copying overheads. We demonstrate performance benefits in Section 4.

This device is an implementation of the xdev API. For this purpose, upper layers of the software can transparently access this particular communication driver.

Figure 4 shows the initialization routine for the native device. The MPJ Express runtime passes some meta-data like the total number of processors involved in computation. Also the device keeps track of registered threads— this is indicated by the `numRegisteredThreads` variable. Each time a thread calls the initialization procedure, `numRegisteredThreads` is incremented. When the value of this variable equals `nprocs`, then all threads are notified to continue execution. This implies that all threads have called the initialization routine and can begin their computational tasks.

Figure 5 and 6 shows implementation sketch of non-blocking send and receive methods. Our device extensively uses `sendQueue` and `recvQueue` for non-blocking communication functionality. We first focus on

```
1 public class SMPDeviceImpl {
2
3   ...
4   int numRegisteredThreads = 0;
5
6   ProcessID id = new ProcessID(
7                    UUID.randomUUID());
8   int size ;
9   Thread [] threads ;
10   HashMap ids ;
11   xdev.ProcessID id = null;
12   xdev.ProcessID[]  pids = null;
13
14   SMPDeviceImpl WORLD =
15                   new SMPDeviceImpl();
16   ...
17
18   ProcessID[] init(String file,
19                             int rank) {
20
21     Thread currentThread =
22            Thread.currentThread() ;
23     nprocs is the total number of procs
24
25     if (numRegisteredThreads == 0) {
26
27       WORLD.size = nprocs ;
28       WORLD.pids =
29         new ProcessID [WORLD.size];
30       WORLD.threads =
31            new Thread [WORLD.size];
32       WORLD.ids = new HashMap() ;
33
34       .. assign a context for the
35       xdev-level MPI communicator
36       representative ..
37
38     }
39
40     if(currentThread is not
41               already registered) {
42
43       WORLD.id =
44         new ProcessID(UUID.randomUUID()) ;
45       WORLD.pids[rank] = WORLD.id;
46       WORLD.threads [rank] = thread ;
47       WORLD.ids.put(thread, WORLD.id) ;
48
49       numRegisteredThreads++ ;
50
51       if(numRegisteredThreads
52                         == WORLD.size) {
53         initialized = true ;
54         notify all waiting threads
55       }
56       else {
57         currentThread waits
58       }
59
60     }
61
62     return WORLD.pids ;
63
64   }
65 }
```

Figure 4. Pseudocode for `init` method

```
1 ...
2 RecvQueue recvQueue = new RecvQueue() ;
3 SendQueue sendQueue = new SendQueue() ;
4
5 public Request isend(mpjbuf.Buffer buf,
6                      ProcessID destID,
7                      int tag, int context)
8                      throws XDevException {
9
10   initialize sendRequest
11
12   acquire class-level lock {
13
14     find and remove matchingRecvRequest
15                            from recvQueue
16
17     if(matchingRecvRequest is found) {
18       copy message from sender buffer
19                        to receiver buffer
20       set pending flag to false in
21        sendRequest and matchingRecvRequest
22
23       notify the receiver thread
24     }
25     else {
26       add sendRequest object to sendQueue
27     }
28   }
29
30   return sendRequest
31
32 }
33 ...
```

Figure 5. Pseudocode for `isend` method

```
1 ...
2 RecvQueue recvQueue = new RecvQueue() ;
3 SendQueue sendQueue = new SendQueue() ;
4
5 public Request irecv(mpjbuf.Buffer buf,
6                      ProcessID srcID,
7                      int tag, int context)
8                      throws XDevException {
9
10   initialize recvRequest ;
11
12   access class-level lock {
13
14     find and remove matchingSendRequest
15                          from sendQueue
16
17     if(matchingSendRequest is found) {
18       copy message from sender buffer
19                      to receiver buffer
20       set pending flag to false in
21       recvRequest and matchingSendRequest
22
23       notify the sender thread
24     }
25     else {
26       add recvRequest object to recvQueue
27     }
28   }
29
30   return recvRequest ;
31
32 }
33 ...
```

Figure 6. Pseudocode for `irecv` method

the functionality of non-blocking send method. Here a `sendRequest` is initialized, which stores the sending buffer reference, `destID`, `tag`, and context information. Later this `sendRequest` is used to find and later remove a `matchingRecvRequest` from `recvQueue`. A matcing receive request object will only be found if the non-blocking receive method has already been called by the receiver. If it exists, then message is directly copied to buffer location specified by the receiver. Otherwise, the `sendRequest` is simply stored in the `sendQueue`—physical message transfer occurs when the non-blocking receive method is called.

Similarly Figure 6 shows the non-blocking receive functionality. Here a `recvRequest` is initialized and used to find a matching send request from the `sendQueue`. A match will only exist if the send method has already been called by the sender thread. If no match exists, then the `recvRequest` is added to the `recvQueue`.

3.2 SysV IPC based device

The SysV IPC library is a C shared memory communication API for UNIX based systems. Our second shared memory communication device for the MPJ Express library is an xdev implementation that uses the SysV IPC library. The JNI API is used to invoke C routines from the Java code. These routines include the `shmget()`, `shmat()`, `shmdt()`, and `shmctl()`.

The first operation supported by any communication device is the initialization phase. In this operation, all processes exchange their process identifiers and create a local index table. The xdev device layer forces all implementations to use an object of `java.util.UUID` object as process identifier—this is a 128 bit number divided into 64 bit Least Significant Bits (LSBs) and 64 bit Most Significant Bits (MSBs). During the initialization step, each process also creates shared memory segments, which are used to communicate data with other processes. A process uses its own rank, an integer number, to acquire a reference to its reserved shared memory segment. The total number of shared memory segments at each process are equal to the total number of processes—one segment is reserved for communication with a particular process.

Other important operations implemented by this device include sending and receiving of messages. To send a message, the sender process first acquires the shared memory segment using its own rank and the destination process rank. Once a reference to the reserved segment has been obtained, the sender process copies data on it that is later read by the receiver process. Figure 7 shows a shared memory region, which is divided into four subsections—one for each destination process. The figure shows that the sender process is writing messages on the first memory subsection.

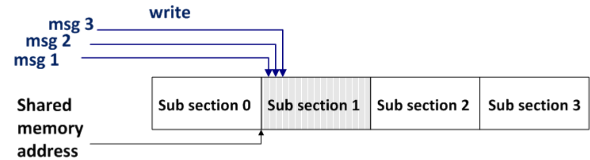


Figure 7. The Sender Process Writing Messages on the Shared Memory

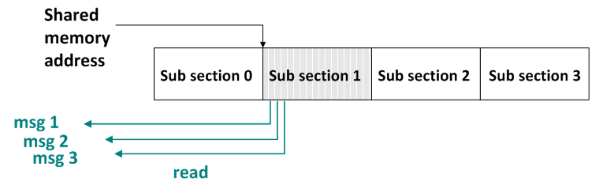


Figure 8. The Receiver Process Reading Messages on the Shared Memory

Similarly, the receiver process acquires the memory address of reserved shared memory segment using source process rank. Once the reference to the shared memory segment has been acquired, the receiver process reads data from this region. This is depicted in the Figure 8.

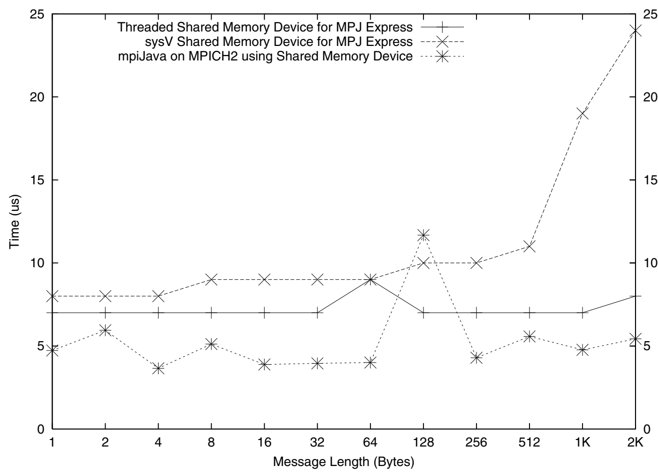
Towards the end, parallel applications typically execute the finalize method, which is supported by this communication device. At this stage, the device detaches and deallocates all shared memory segments.

4 Preliminary Experimental Evaluation

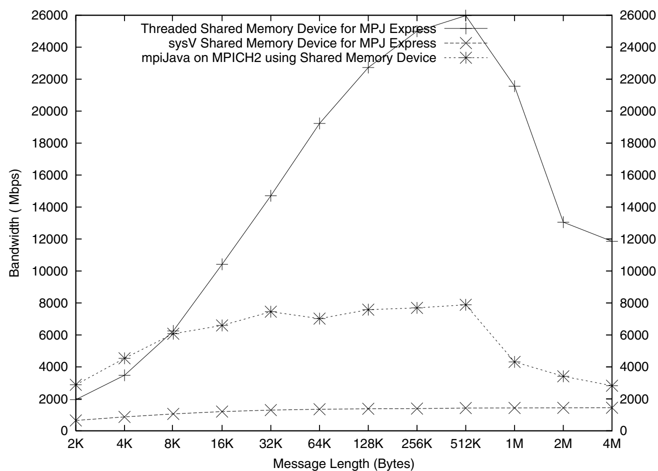
We begin by describing our test environment, which consisted of a 32 processing core Linux cluster at our institute. The cluster consists of eight compute nodes. Each node contains a quad-core Intel Xeon processor. The nodes are connected via Myrinet and Gigabit Ethernet. The compute nodes run the SuSE Linux Enterprise Server (SLES) 10 Operating System and GNU C Compiler (GCC) version 4.1.0. Each compute node has 2 Gigabytes of main memory. We used Open MPI version 1.2.4 as the C MPI library. For the parallel Java version, we used the latest development version of MPJ Express with the Sun Java Development Kit (JDK) 1.6 Update 4.

We compared the performance of the two different approaches of shared memory communication for MPJ Express. We also benchmarked them against the shared memory device of mpiJava.

We wrote a simple ping pong program in which sender and receiver repeatedly pass a message back and forth. We started from message size of 1 byte and went up to 4 Megabytes. In the start, a warm up loop of 20K iterations



(a) Transfer Time Comparison



(b) Throughput Comparison

Figure 9. Performance Comparison of Shared Memory Devices

was carried out and then the average time was calculated for 10K iterations.

Figure 9(a) shows the transfer time graph for message sizes from 1 Byte to 2 Kilobytes. Similarly, Figure 9(b) shows throughput graph for message sizes from 4 Kilobytes to 4 Megabytes. The latency comparison depicted in the Figure 9(a) shows that the mpiJava library performs the best, followed by our threads-based device. The reason is the efficient communication performance of the MPICH2 library—mpiJava uses JNI to interface with this native C library. For smaller messages, JNI copying overhead is minimal and thus mpiJava library performs the best. On the other hand, the throughput comparison shown in Figure 9(b) graphs show that our threads-based device performs the best followed by mpiJava's shared memory device. This shows that inter-thread communication is more efficient than interprocess communication.

5 Conclusions and Future Work

MPJ Express is our implementation of MPI-like bindings for the Java language. There is a growing community of researchers using this software to develop their portable parallel applications. In the context of emerging multi-core processors, it is vital for any MPI library to support efficient shared memory communication between cores on a single processor. This paper presented two implementations of this kind of shared memory devices. The first device is based on Java threads where a single thread represents a MPI process. The second device is based on the SysV IPC API. We also conducted a preliminary evaluation and compared our results with mpiJava's shared memory communication device. We found out that our threads-based approach performs better for large messages. On the other hand, mpiJava performs the best for small messages. A free copy of the MPJ Express software can be obtained from `http://mpj-express.org`.

In the future, we plan to link our shared memory communication devices with the top layers of the MPJ Express software. Once this is done we plan to benchmark point-to-point and collective communications of the MPJ Express software. In our previous work we introduced "nested parallelism" by mixing threading and messaging at the application level. In future we will compare nested parallelism approach with the shared memory communication approach.

Acknowledgments

The authors would like to thank Dr Bryan Carpenter, University of Portsmouth for his contribution and collaboration on the development of the threads-based device. We would also like to thank Mr Kamran Hameed, NUST for configuring and benchmarking the mpiJava software on our local cluster.

References

[1] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. An Object-Oriented Java interface to MPI. In *Proceedings of the International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[2] Markus Bornemann, Rob van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting*,

Lecture Notes in Computer Science, pages 217–224. Springer, 2005.

[3] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[4] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. HPJava: Data Parallel Extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[5] Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1), February 2007.

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 97–104, Budapest, Hungary, September 2004. Springer.

[7] William Gropp. MPICH2: A New Start for MPI Implementations. In Dieter Kranzlmüller, Péter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Proceedings of the 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, page 7. Springer, October 2002.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, 1995. www.mcs.anl.gov/mpi.

[9] Myricom. http://www.myri.com.

[10] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.