Proceedings of the Second International Conference on
"Emerging Research in Computing, Information, Communication and Applications"
ERCICA 2014
ISBN: 9789351072621

ELSEVIER

Overall Aspects of Java Native Threads on Win32 Platform

Bala Dhandayuthapani Veerasamy[1,*] and G. M. Nasira[2]

[1]*Research Scholar in Information Technology, Manonmaniam Sundaranar University, Tirunelveli, Tamil Nadu, India.*
[2]*Assistant Professor in Computer Science, Chikkanna Govt. Arts College, Tirupur, Tamil Nadu, India.*
e-mail: dhanssoft@gmail.com, nasiragm99@yahoo.com

Abstract. A parallel programming model is a set of software technologies to articulate parallel algorithms and match applications with the underlying parallel systems. It surroundings with applications, languages, libraries, compilers, communication systems, and parallel I/O. Programmer have to decide a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform. Multithreaded programming is written in many programming languages with usually increased performance. Thread libraries can be implicit or explicit. OpenMP, MPI, Intel Threading Building Blocks (TBB) are implicit thread libraries. Pthreads and Windows Threads are explicit thread libraries. Threads can be accessed by different programming interfaces. Many software libraries provide an interface for threads usually based on POSIX Threads, Windows threads, OpenMP, MPI and Threading Building Blocks frameworks. These frameworks provide a different level of abstraction from the underlying thread implementation of the operating system. The general parallelism is the execution of separate tasks in parallel. These multithreading libraries provide difference features. For example, Java support flexible and easy use of threads; yet, java does not have contained methods for thread affinity to the processors; because, green threads are scheduled by the virtual machine itself where as Windows or POSIX thread can fix thread affinity. The native threads are scheduled by the operating system that is hosting the virtual machine. This research finding carry overview aspects on how Java can facilitate Win32, POSIX, TBB threads through JNI, which enables Java threads, to add other threading library features in Java.

Keywords: Affinity mask, Intel TBB, JNA, JNI, Kernel, Multithread, NativePthread, Pthread, Win32 API.

1. Introduction

A multi-core is an architecture [1–8] design that places multiple processors on a single die (computer chip). Each processor is called a core. As chip capacity increased, placing multiple processors on a single chip became practical. These designs are known as Chip Multiprocessors (CMPs) because they allow for single chip multiprocessing. Multi-core is simply a popular name for CMP or single chip multiprocessors. The concept of single chip multiprocessing is not new, and chip manufacturers have been exploring the idea of multiple cores on a uniprocessor since the early 1990s. Recently, the CMP has become the preferred method of improving overall system performance. Manufacturers are increasingly adding more processor core is often a highly efficient way to buy more power at low incremental cost. Of course our main purpose in writing parallel programs is usually increased performance. Parallelism is a proven way to run programs fast. This is a departure from the approach of increasing the clock frequency or processor speed to achieve gains in overall system performance. Increasing the clock frequency has started to hit its limits in terms of cost - effectiveness.

CMPs [7] come in multiple flavors: two processors (dual core), four processors (quad core), and eight processors (octa core) configurations. Some configurations are multithreaded; some are not. There are several variations in how cache and memory are approached in the new CMPs. The approaches to processor to processor communication vary among different implementations. Several parallel computing platforms focused on particular multi-core platforms, offer a shared address space. A natural programming model for multi-core architectures is a thread model, in which all threads have access to shared variables. These shared variables are then used for information and data exchange.

*Corresponding author

To coordinate the access to shared variables, synchronization mechanisms have to be used to avoid race conditions in case of concurrent accesses.

Parallel programming environments [1–8] provide the basic tools, language features, and application programming interfaces (APIs) needed to construct a parallel program. A programming environment [1–8] implies a particular abstraction of the computer system called a programming model. Programming models too closely aligned to a particular parallel system lead to programs that are not portable between parallel computers. Because the effective lifespan of software is longer than that of hardware, many organizations have more than one type of parallel computer, and most programmers insist on programming environments that allow them to write portable parallel programs. Also, explicitly managing large numbers of resources in a parallel computer is difficult, suggesting that Higher-level abstractions of the parallel computer might be useful. Multithreaded program expected to exploit multi-core environments. The green threads are scheduled by the virtual machine itself. This is the original model for Java Virtual Machine mostly follows the idealized priority based scheduling. This kind of thread never uses operating system threads library. The native threads are scheduled by the operating system that is hosting the virtual machine. The operating system threads are logically divided into user level threads and system level threads. The operating system itself that is the kernel of the operating system lies at system level threads. In a user thread model, on the other hand, the application code implements the thread operations.

2. Java Threads Libraries

The Class java.lang.Thread [9–13]: The class Thread defines thread objects. When the start() method is called, an actual running thread is created which the Thread object can control. It is important to distinguish between the object (which is just memory and a set of methods) and the running thread (which executes code). All static thread methods apply to the current thread. There is a second method of creating a Java thread. In this method you write a class that implements the Runnable interface, defining a run()method on it. You then create a thread object with this Runnable as the argument and call start() on the thread object.

The java.util.concurrent API [9–13]: Java 7 introduced ExecutorService, manages a pool of threads and allows us to schedule tasks for execution by threads in its pool. It is, however, up to us to decide how many threads will be in the pool, and there is no distinction between tasks we schedule and subtasks these tasks create. Java 7 brings a specialization of ExecutorService with improved efficiency and performance the fork-join API [14]. The ForkJoinPool class dynamically manages threads based on the number of available processors and task demand. Fork-join employs work-stealing where threads pick up (steal) tasks created by other active tasks. This provides better performance and utilization of threads. Subtasks created by active tasks are scheduled using different methods than external tasks. We'd typically use one fork-join pool in an entire application to schedule tasks. Also, there's no need to shut down the pool since it employs daemon threads. To schedule tasks, we provide instances of ForkJoinTask (typically an instance of one of its subclasses) to methods of ForkJoinPool. ForkJoinTask allows us to fork tasks and then join upon completion of the task. ForkJoinTask has two subclasses: RecursiveAction and RecursiveTask. To schedule tasks that don't return any results, we use a subclass of RecursiveAction. For tasks that return results, we use a subclass of RecursiveTask. The fork-join API is geared toward tasks that are reasonably sized so the cost is amortized but not too large (or run indefinitely in loops) to realize reasonable throughput. The fork-join API expects tasks to have no side effects (don't change shared state) and no synchronized or blocking methods. The fork-join API is very useful for problems that can be broken down recursively until small enough to run sequentially. The multiple smaller parts are scheduled to run at the same time, utilizing the threads from the pool managed by ForkJoinPool.

ParallelArray [15]: Java 8 introduced ParallelArray, encapsulates a ForkJoinExecutor and an array in order to provide parallel aggregate operations. The main operations are to apply some procedure to each element, to map each element to a new element, to replace each element, to select a subset of elements based on matching a predicate or ranges of indices, and to reduce all elements into a single value such as a sum. A ParallelArray is not a List, but can be viewed as one, via method asList(), or created from one, by constructing from array returned by a list's toArray method. Arrays differ from lists in that they do not incrementally grow or shrink. Random accessibility across all elements permits efficient parallel operation. ParallelArrays also support element-by-element access (via methods get and/set), but are normally manipulated using aggregate operations on all or selected elements.

3. Java Native Threads Models

Using Java Native Interface (JNI) [16,17]: Java Native Interface (JNI) is a strong feature of the Java platform. An application that uses the JNI can incorporate native codes written in other programming languages such as C and C++. The JNI is a strong feature that permits us to take benefits of the Java platform, but still uses code written in other languages. As a part of the Java Virtual Machine implementation, the JNI is a two-way interface that permits Java applications to invoke native code and vice versa. The JNI is designed to unite Java applications with native code. As a two-way interface, the JNI can support two types of native code: native libraries and native applications.

JNI allows writing native methods that allow Java applications to call functions implemented in native libraries. Java applications call native methods in the same way that they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in other language and reside in native libraries. In order to write Java Native Interface application that calls a C or C++ function, consists of the following steps:

1. Declaring Native Methods in Java Class
2. Compiling Java Class and Creating Native Method Header
3. Implementing Native Method
4. Compiling the C++ Source and Creating Native Library
5. Testing Native Program

Using Java Native Access (JNA) [18]: It can download from https://jna.java.net. JNA makes Java programs easy access to native shared libraries without using the Java Native Interface. JNA's design aims to provide native access in a natural way with a minimum of effort. The JNA library uses a small native library called foreign function interface library (libffi) to dynamically invoke native code. The JNA library uses native functions allowing code to load a library by name and retrieve a pointer to a function within that library, and uses libffi library to invoke it, all without static bindings, header files, or any compile phase. The developer uses a Java interface to describe functions and structures in the target native library. This makes it quite easy to take advantage of native platform features without incurring the high development overhead of configuring and building JNI code. JNA is built and tested on Mac OS X, Microsoft Windows, FreeBSD/OpenBSD, Solaris, and Linux. It is also possible to tweak and recompile the native build configurations to make it work on other platforms. Java Native Access lets you access C libraries programmatically. In situations where Java does not provide the necessary APIs, it is sometimes necessary to use the Java Native Interface (JNI) to make platform-specific native libraries accessible to Java programs.

JNA approaches to integrate native libraries with Java programs. It shows how JNA enables Java code to call native functions without requiring glue code in another language. It is useful to know JNA, because the Java APIs with their architecture-neutral emphasis will never support platform specific functionality. Though Java itself is architecture neutral, JNA is perforce on platform-specific. The Java Native Access project is hosted on Java.net, where you can download the project's online Javadoc and the software itself. Although the download section identifies five JAR files, you only need to download jna.jar. The jna.jar file provides the essential JNA software and is required to run all of the examples you'll find here. This JAR file contains several packages of classes, along with JNI-friendly native libraries for the UNIX, Linux, Windows, and Mac OS X platforms. Each library is responsible for dispatching native method calls to native libraries. Here are a few things you have to take care of when starting a JNA project:

1. Download jna.jar from the JNA project site and add it to your project's build path. This file is the only JNA resource you need. Remember that jna.jar must also be included in the run-time classpath.
2. Find the names of the DLLs that your Java code will access. The DLL names are required to initialize JNA's linkage mechanisms.
3. Create Java interfaces to represent the DLLs such as kernel32.dll, user32.dll and etc on your application that will access.
4. Test linkage of your Java code to the native functions.

4. Implicit Threading

Implicit threading libraries [7] take care of much of the minutiae needed to create, manage and synchronize threads. There are algorithms written concurrently to take advantage of the limited scope of features within the implicit libraries. These threading libraries are compiler directives. There are three renowned implicit libraries focused here.

OpenMP [19]: It is a set of language extensions implemented as compiler directives. Implementations are currently available for FORTRAN, C, and C++. OpenMP is frequently used to incrementally add parallelism to sequential code.

It consists of compiler directives, library routines and environment variables that specify shared-memory concurrency in FORTRAN, C, and C++ programs. The rationale behind the development of OpenMP was to create a portable and unified standard of shared-memory parallelism. All major compilers support the OpenMP language. This includes the Microsoft Visual C/C++ .NET for Windows and the GNU GCC compiler for Linux. The Intel C/C++ compilers, for both Windows and Linux, also support OpenMP. OpenMP directives demarcate code that can be executed in parallel and control how code is assigned to threads. The threads in an OpenMP code operate under the fork-join model.

OpenMP for Java is JaMP [20], which can download from https://www2.cs.fau.de/EN/research/JavaOpenMP/index.html. JaMP is an implementation of the well-known OpenMP standard adapted for Java. JaMP allows one to program, for example, a parallel for loop or a barrier without resorting to low-level thread programming. JaMP currently supports all of OpenMP 2.0 with partial support for 3.0 features, e.g., the collapse clause. JaMP generates pure Java 1.5 code that runs on every JVM. It also translates parallel for loops to CUDA-enabled graphics cards for extra speed gains. If a particular loop is not CUDA-able, it is translated to a threaded version that uses the cores of a typical multi-core machine. JaMP also supports the use of multiple machines and compute accelerators to solve a single problem. This is achieved by means of two abstraction layers. The lower layer provides abstract compute devices that wrap around the actual CUDA GPUs, OpenCL GPUs, or multi-core CPUs, wherever they might be in a cluster. The upper layer provides partitioned and replicated arrays. A partitioned array automatically partitions itself over the abstract compute devices and takes the individual accelerator speeds into account to achieve an equitable distribution. The JaMP compiler applies code-analysis to decide which type of abstract array to use for a specific Java array in the user's program.

MPI [21]: it is a set of library routines that provide for process management, message passing, and some collective communication operations. The operations are involving all the processes involved in a program, such as barrier, broadcast, and reduction. MPI programs can be difficult to write because the programmer is responsible for data distribution and explicit inter-process communication using messages. Because the programming model assumes distributed memory, MPI is a good choice for MPPs and other distributed memory machines.

MPI for Java is Open MPI [22], which can download from http://www.open-mpi.org. The Java interface is provided in Open MPI on a provisional basis. It is not part of the current or any proposed MPI standard. Continued inclusion of the Java interface is contingent upon active user interest and continued developer support. The decision to add a Java interface to Open MPI was motivated by a request from the Hadoop community. Hadoop is a Java-based environment for processing extremely large data sets. Modeled on the Google enterprise system, it has evolved into its own rapidly growing open-source community. Although executed in parallel, Hadoop processes are independent and thus would only possibly use MPI as a base for efficient messaging. Of greater interest, however, is the ability to add MPI-based follow-on processing to a Hadoop map-reduce operation. Since Hadoop map-reduce is typically done in Java, any subsequent MPI operations would best be done in that language.

MPJ Express [23]: It is an open source Java message passing library that allows application developers to write and execute parallel applications for multi-core processors and compute clusters/clouds. MPJ Express enables HPC using Java. Earlier efforts for building a Java messaging systems have typically followed either the JNI approach, or the pure Java approach. On commodity platform like Fast Ethernet, advances in JVM technology now enable networking applications written in Java to rival their C counterparts. On the other hand, improvements in specialized networking hardware have continued, cutting down the communication costs to a couple of microseconds. Keeping both in mind, the key issue at present is not to debate the JNI approach versus the pure Java approach, but to provide a flexible mechanism for applications to swap communication protocols

Intel Threading Building Blocks [24–29]: Intel TBB can download from https://www.threadingbuildingblocks.org. It is a C++ template-based library for loop-level parallelism that concentrates on defining tasks rather than explicit threads. The components of TBB include generic parallel algorithms, concurrent containers, low-level synchronization primitives, and a task scheduler. Programmers using TBB can parallelize the execution of loop iterations by treating chunks of iterations as tasks and allowing the TBB task scheduler to determine the task sizes, number of threads to use, assignment of tasks to those threads, and how those threads are scheduled for execution. The task scheduler will give precedence to tasks that have been most recently in a core with the idea of making best use of the cache that likely contains the task's data. The task scheduler utilizes a task-stealing mechanism to load balance the execution.

Intel TBB assisted us create applications that collect the benefits of new processors with more and more cores as they become available. It uses templates for common parallel iteration patterns, enabling us to attain increased speed from multiple processor cores without having to be experts in synchronization, load balancing, and cache optimization. Intel TBB promotes scalable data parallel programming. The data parallelism has a special significance in the era

of the multi and many-core computing, where huge numbers of cores are available on single chip devices. The data parallelism is concerned mainly with operations on arrays of data. The data parallelism involves sharing common data among executing processes through memory coherence, improving performance by reducing the time required to load and access memory.

We exercised Intel TBB win32 [30] through JNI to utilize in Java. This contribution exploited painless usage of Intel TBB parallel algorithms used in Java. In this research, we especially focused on utilizing parallel_for algorithms with blocked_range, which supply an iterator that determines how to make a task split in half when the task is considered large enough. In turn, Intel TBB will then divide large data ranges repeatedly to help spread work evenly among processor cores. The parallel_for loop constructs deserved overhead cost for every chunk of work that it schedules. It chooses chunk sizes automatically, depending upon load balancing needs. The blocked_range supported to work with single dimensional array. Thus, we illustrated manipulating array on single dimensional and the speedup and performance improvements demonstrated. If some other operations required performing on one dimensional array, we are expected to change the code on NativeTBB.cpp program. At present, there are opportunities to work with two dimensional arrays through using blocked_range2d and three-dimensional arrays through using blocked_range3d. Not only the parallel_for loop can be used in java, but also there are huge openings for utilizing the entire Intel TBB parallel library template that can be used for various purposes.

5. Explicit Threading

Explicit threading libraries [7] require the programmer to control all aspects of threads, including creating threads, associating threads to functions, and synchronizing and controlling the interactions between threads and shared resources. The two most prominent threading libraries in use today are POSIX threads (Pthreads) and Windows Threads by Microsoft. While the syntax is different between the two APIs, most of the functionality in one model can be found in the other. Each model can create and join threads, and each features synchronization objects to coordinate execution between threads and control the access to shared resources by multiple threads executing concurrently. Let's start with Pthreads for readers who use Linux.

Pthreads [31,32]: Pthreads has a thread container data type of pthread_t. Objects of this type are used to reference the thread (borrowing terms from Windows Threads, I tend to call this object the handle of the created thread). To create a thread and associate it with a function for execution, use the pthread_create() function. A pthread_t handle is returned through the parameter list. When one thread needs to be sure that some other thread has terminated before proceeding with execution, it calls pthread_join(). The calling thread uses the handle of the thread to be waited on as a parameter to this function. If the thread of interest has terminated prior to the call, pthread_join() returns immediately with the threaded function's exit code (if any) from the terminated thread; otherwise, the calling thread is blocked until that currently executing thread has completed.

Pthreads for Java is Pthreads-w32 [33], which can download from https://www.sourceware.org/pthreads-win32/index.html. The POSIX 1003.1-2001 standard defines an application programming interface (API) for writing multithreaded applications. This interface is known more commonly as Pthreads. A good number of modern operating systems include a threading library of some kind: Solaris (UI) threads, Win32 threads, DCE threads, DECthreads, or any of the draft revisions of the Pthreads standard. The trend is that most of these systems are slowly adopting the Pthreads standard API, with application developers following suit to reduce porting woes. Win32 does not, and is unlikely to ever, support Pthreads natively. This project seeks to provide a freely available and high-quality solution to this problem. Various individuals have been working on independent implementations of this well-documented and standardized threading API, but most of them never see the light of day. The tendency is for people to only implement what they personally need, and that usually does not help others. This project attempts to consolidate these implementations into one implementation of Pthreads for Win32.

We exercised Pthread-w32 [34] through JNI to utilize in Java. The native threads are scheduled by the operating system that is hosting the virtual machine. The user level Pthreads has used the kernel of the operating system lies at system level threads. The kernel is accountable for managing system calls on behalf of programs that run at user level Pthreads. This research finding focused on how Java can facilitate Pthreads through JNI, which enables Java threads and native threads to schedule and execute in hybrid mode. As a result, this research brings up opening to exploit Pthreads within Java program, it is strongly recommending for Flynn's Multiple Program Multiple Data (MPMD) and Multiple Program and Single Data (MPSD) through method level concurrency.

Java support flexible and easy use of threads; yet, java does not contain methods for thread affinity to the processors. Setting an affinity [35] thread to multiprocessor is not new to research, since it was already sustained by other