Asian Journal of Information Technology 13 (8): 431-437, 2014
ISSN: 1682-3915


# Java Native Intel Thread Building Blocks for Win32 Platform

[1]Bala Dhandayuthapani Veerasamy and [2]G.M. Nasira
[1]Research Scholar, Manonmaniam Sundaranar University, Tirunelveli, India
[2]Department of Computer Science, Chikkanna Govt. Arts College, Tirupur, India

**Abstract:** Threads can be accessed by different programming interfaces. Many software libraries provide an interface for threads usually based on POSIX threads, Windows threads, OpenMP 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. Nonnumeric code is usually implemented with task parallelism rather than data parallelism. The data parallelism is concerned mainly with operations on arrays of data. 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. Multi-core processors have expressed parallel programming subject matter in interesting way for every programmer. Every program written in multi-core processor will run on the many-core processor in future will be difficult task. However, Intel Threading Building Blocks is a C++ template library that provides tasks, parallel algorithms and containers to support for scalable parallel programming using standard C++ code. This research finding focused on how Java can facilitate Intel TBB through JNI which can exploit painless usage of Intel TBB parallel algorithms that can be used in Java.

**Key words:** JNI, Intel TBB, parallel_for, partitioners, range, speedup, performance

## INTRODUCTION

Parallel computers can be separated into two kinds of major categories, they are control flow and data flow (Veerasamy and Nasira, 2014). The control flow parallel computers are called task parallelism, it basically work based on the similar principles as the sequential or von Neumann computer except that multiple instructions can be executed at any given time. Programs are already decomposed into individual parts statements, methods can be run in parallel. Task parallelism takes and extends the pre-existing functional partitioning that already exists and runs independent pieces in parallel with respect to one another. The data-flow parallel computers occasionally referred to as non-von Neumann is completely dissimilar in that they have no pointer to active instructions. The data parallelism uses the input data (Duffy, 2009) to some operation as the means to partition into smaller pieces either large amount of data to process or combination of both. Data is divided up among the available hardware processors in order to achieve parallelism. This partitioning step is often followed by replicating and executing some mostly independent program operation across these partitions. The data parallelism approach is also nice for scalability. Scalable parallelism can make use of additional processors to solve larger problems. The upper limit on parallelism is typically much larger because loop iteration counts are often quite large and dependent on the dynamic size of data that must be operated upon. Growth in data sizes in a data parallel program translates into the exposure of more parallelism opportunities that can scale to use many processors as they become available. Because of this, many industry experts believe that data parallelism is the most scalable and future-proof way of building parallel programs-programs that will not be inherently limited by their construction.

The earlier researches concentrated on task parallelism; they are setting as affinity for task (Veerasamy and Nasira, 2012a, b), Parallel: One Time Pad using Java (Veerasamy and Nasira, 2012a, b), JNT-Java Native Thread for Win32 Platform (Veerasamy and Nasira, 2013) and Java Native Pthread for Win32 Platform (Veerasamy and Nasira, 2014). This research finding focused on how Java can facilitate Intel TBB through JNI which can exploit painless usage of Intel TBB Parallel algorithms that can be used in Java. Intel TBB emphasizes data-parallel programming enabling multiple

**Corresponding Author:** Bala Dhandayuthapani Veerasamy, Research Scholar, Manonmaniam Sundaranar University, Tirunelveli, India

threads to work on different parts of a collection. Data-parallel programming scales well to larger numbers of processors by dividing the collection into smaller pieces. With data-parallel programming, program performance increases as we add processors. Intel TBB benefits specifying logical parallelism instead of threads, targets threading for performance, compatible with other threading packages, emphasizes scalable and data parallel programming, relies on generic programming.

Intel Threading Building Blocks (Intel TBB) (Reinders, 2007) is a C++ template library developed by Intel to purposely concentrate on programming in multi and many-core systems. It supports Linux, Windows and MacOSX and all major C++ compilers. Intel TBB presents algorithms and data structures to describe tasks in a parallel program. These tasks are mapped by an internal scheduler to worker threads. In contrast to other thread programming packages the programmer has no access to these threads, only to the tasks. Intel TBB uses C++ templates widely to minimize runtime overhead. Intel TBB uses generic programming to be proficient.

Intel TBB includes parallel algorithms such as parallel_for, parallel_reduce, parallel_deterministic _reduce, parallel_scan, parallel_do, parallel_for_each, parallel_pipeline, parallel_sort and parallel_invoke, concurrent containers such as concurrent_hash_map, concurrent_vector, locks and atomic operations, a task scheduler and a scalable memory allocator.

Intel TBB uses templates (Reinders, 2007) for common parallel iteration patterns, enabling programmers to attain increased speed from multiple processor cores without having to be specialist in synchronization, load balancing and cache optimization. Programs using Intel TBB will run on systems with a single processor core as well as on systems with multiple processor cores. Intel TBB promotes scalable data parallel programming. As well, it fully supports nested parallelism, so researchers can build larger parallel components from smaller parallel components easily.

To use Intel TBB library (Reinders, 2007), researchers specify tasks not threads and let the library map tasks into threads in capable manner. The result enables us to specify parallelism far handily with better results than using raw threads. Programming in Intel TBB offers an opportunity to avoid thread management. This will result in code that is easier to create, easier to maintain. Though, it does require algorithms in terms of what work can be divided and how data can be divided. The proper degree of dividing a problem is called grain size. Grain size started as a strange manual process which has since been facilitated with some automation. Recursively dividing a problem is better than static division of work. It fits absolutely with the use of task stealing instead of a global task queue. Reliance on task stealing is a critical design decision that avoids implementing something as important as a task queue as a global resource that becomes a bottleneck for scalability.

**Downloading and installing TBB:** Researchers have to download the Intel TBB current Version 4.2 update 2 stable releases from http://threadingbuildingblocks.org. Of course, Intel TBB release available for Windows, Linux and Mac OS X, however, researchers should choose windows stable release. In order to use TBB, researchers have to install Microsoft Visual C++ 2005 or later version and copy all TBB associated files to Microsoft Visual C++ folders. The files to copied are, all include files of TBB should be copied to Microsoft Visual C++ include folder under like include\tbb*.h, all .lib files from TBB should be copies to Microsoft Visual C++ lib folder and all .dll files of TBB should be copies to Microsoft Visual C++ bin folder. These .dll files are supportive or dependent libraries for future java runtime system hence these files should also copied to windows system32 folder.

**Initializing and terminating the library:** Intel TBB components are defined in the tbb namespace. Any program that utilizes an algorithm template from the Intel TBB library (Reinders, 2007) must be initialized with tbb::task_scheduler_init object. The task scheduler shuts down when all task_scheduler_init objects terminate. By default, the constructor for task_scheduler_init does the initialization and the destructor does the termination.

To use the tbb::task_scheduler_init object, researchers must include "tbb/task_scheduler_init.h" header file. The constructor for task_scheduler_init has an optional parameter that identifies the number of desired threads including the calling thread. The optional parameter can be one of the following:

- The value task_scheduler_init::automatic which is the default when the parameter is not specified. It exists for the sake of the method task_ scheduler_init::initialize
- The value task_scheduler_init::deferred which defers the initialization until the task_scheduler_ init::initialize(n) method is called. The value n can be positive integer specifying the number of threads to use

**Parallel_for function:** The parallel_for function template (Reinders, 2007) performs parallel iteration over a range of values. The header for parallel_for is "tbb/parallel_for.h". The syntax is follows: void parallel_for (const Range& range, const Body&body, [,partitioner[,task_group_ context&group]]).

A parallel_for (range, body, partitioner) provides a more general form of parallel iteration. It represents parallel execution of body over each value in range. The optional partitioner specifies a partitioning strategy. The range must model the range concept.

**Range:** The blocked_range (Reinders, 2007) represents the entire iteration space from 0 to n-1 which parallel_for divides into subspaces for each processor. The general form of the constructor is blocked_range<T>(begin, end, grainsize). A range can be recursively subdivided into two parts. It is recommended that the division be into nearly equal parts but it is not required. Splitting as evenly as possible typically yields the best parallelism. Ideally, a range is recursively splittable until the parts represent portions of work that are more efficient to execute serially rather than split further. The amount of work represented by a range typically depends upon higher level context hence a typical type that models a Range should provide a way to control the degree of splitting.

Type blocked_range models a one-dimensional range: a blocked_range<Value> represents any integral type that is convertible to size_t. The value requirements are integral types, pointers and STL random-access iterators whose difference can be implicitly converted to a size_t. The header for blocked_range is "tbb/blocked_range.h".

Type blocked_range2d models a two-dimensional range: a blocked_range2d<RowValue, ColValue> represents RowValue and ColValue, must meet the requirements in the table in the blocked_range Template Class section. A blocked_range is splittable if either axis is splittable. A blocked_range models the Range concept. The header for blocked_rane2d is "tbb/blocked_range2d.h".

Type blocked_range3d models a three-dimensional range: a blocked_range3d<PageValue, RowValue, ColValue> is the three-dimensional extension of blocked_range2d. The header for blocked_range3d is "tbb/blocked_range3d.h".

**Partitioners:** The default behaviour of the loop templates parallel_for, parallel_reduce and parallel_scan tries to recursively split (Reinders, 2007) a range into enough parts to keep processors busy not necessarily splitting as finely as possible. There are different partitioners are listed.

**Auto_partitioner (default):** Performs sufficient splitting to balance load, not necessarily splitting as finely as Range::is_divisible permits. When used with classes such as blocked_range, the selection of an appropriate grain size is less important and often acceptable performance can be achieved with the default grain size of 1.

**Affinity_partitioner:** Similar to auto_partitioner but improves cache affinity by its choice of mapping subranges to worker threads. It can improve performance significantly when a loop is re-executed over the same data set and the data set fits in cache.

**Simple_partitioner:** Recursively splits a range until it is no longer divisible. The Range::is_divisible function is wholly responsible for deciding when recursive splitting halts. When used with classes such as blocked_range, the selection of an appropriate grain size is critical to enabling concurrency while limiting overheads.

**Quick introduction to lambda functions:** Adding lambda functions to C++ would let a programmer write a loop body in place instead of having to write a separate STL-style function object. Similar capability is found in the anonymous method in C#, in the inner class in Java and in the primordial lambda expression of LISP. Lambda expressions make the Intel TBB parallel_for much easier to use. A lambda expression (Reinders, 2007) lets the compiler do the tedious work of creating a function object.

## MATERIALS AND METHODS

**Method:** JNI permits us to develop native code when an application cannot be written entirely in the Java language. It want to implements time-critical code in a lower-level and faster programming language. It has legacy code or code libraries that researchers want to get into from Java programs. It desires platform dependent features not supported in the standard Java class library. In order to create and work with native threads by means of Java Native Interface application (Liang, 1999) that calls a C++ function with the following steps:

**Declare the native method in Java class:** JNI begin by writing the following program in the Java programming language. The Program 1 defined a class named NativeTBB that contains native TBB parallel for method. The native keyword notifies the Java compiler that a method is applied in native code outside of the Java class in which it is being declared. Native methods can only be declared in Java classes, not implemented, so native methods do not have a body. The native methods have implemented using Microsoft Visual Studio 2010 Program in the study.

**Program 1 (Native TBB Method declarations):**

```
   //Java Nativity Threads(JNT)
package JNT.Win32.Kernel;
public class NativeTBB{
public native void parallelFor(
                final int FirstArray[],
                final int SecondArray[],
                final int ResultArray[],
                char Operator);
}
```

The native TBB is defined in package named JNT.Win32.Kernel. The class native TBB declared with native Pthread Methods which will be implemented in C++ using JNI. The parallel for(..) declared with final int FirstArray[] will get the first array, final int SecondArray[] will get the second array and the result will be store on final int ResultArray[] through native TBB implementation based on the arithmetic operator mentioned in char Operator.

**Compiling Java class and creating native method header file:** Researchers have compiled the Java code down to byte code. One way to do this is to use the Java compiler javac which comes with the SDK. The command researchers used to compile the Java code to byte code is: javac NativeTBB.Java.

This command generated a NativeTBB.class file in the JNT.Win32.Kernel directory. The next step, researchers created C/C++ header file that defines native function signatures. One way to do this is researchers used the native method C stub generator tool javah.exe which comes with the SDK. This tool is designed to create a header file that defines C-style functions for each native method it found in a Java source code file. The command we used on JNT.Win32.Kernel directory is: Javah NativeTBB.

The name of the header file is the class name with ".h" appended to the end of it. The command shown above generates a file named JNT_Win32_Kernel_NativeTBB.h which is shown in Program 2.

**Program 2 (JNT_Win32_Kernel_NativeTBB.h):**

```
#include <jni.h>
#ifndef _Included_JNT_Win32_Kernel_NativeTBB
#define _Included_JNT_Win32_Kernel_NativeTBB
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JNT_Win32_Kernel_NativeTBB
 * Method:    parallelFor
 * Signature: ([I[I[IC)V
 */
JNIEXPORT void JNICALL Java_JNT_Win32_Kernel_NativeTBB_
parallelFor
  (JNIEnv *, jobject, jintArray, jintArray, jintArray, jchar);
#ifdef __cplusplus
}
#endif
#endif
```

The C/C++ function signature in JNT_Win32_Kernel_NativeTBB.h is quite different from the Java Native Method declarations in NativeTBB.java. JNIEXPORT and JNICALL is compiler-dependent specifier for export functions. The return types are C/C++ types that map to Java types. The parameter lists of all these functions have a pointer to a JNIEnv and a jobject in addition to normal parameters in the Java declaration. The pointer to JNIEnv is actually a pointer to a table of function pointers. These functions provide the various faculties to manipulate Java data in C and C++. The jobject parameter refers to the current object. Thus, if the C or C++ code needs to refer back to the Java side, this jobject acts as a reference or pointer, back to the calling Java object. The function name itself is made by the "Java_" prefix, followed by the fully qualified package name followed by an underscore and class name followed by an underscore and the method name.

**Implementing native methods and creating native library:** Microsoft Visual C++ 2010 Express allows creating Dynamic Link Library (DLL) is a shared library that contains the native TBB code. In order to create DLL, researchers should create a Dynamic Link Library (DLL) project. Microsoft Visual C++ 2010 Express, on the menu bar, chooses File, New, Project. In the left pane of the New Project dialog box, expand Installed, Templates, Visual C++ and then select Win32. In the center pane, select Win32 Console Application. Researchers should specify a name for the project, native TBB and we can specify a name for the solution as NativeTBBDLL.

In source file folder of the project contains by default dllmain.cpp and stdafx.cpp. Here, researchers should add new cpp file as NativeTBB.cpp. In header file section, researchers should add JNT_Win32_Kernel_NativeTBB.h file. The JNI-style header file generated by javah helped us to write C++ implementations for the native method. When it comes to writing the C++ function implementation, the important thing to keep in mind is that our signatures must be exactly like the function declarations from JNT_Win32_Kernel_NativeTBB.h. In stdafx.h, researchers should include all necessary header files which will be used in the NativeTBB.cpp file. Reseachers implemented the method in C++ file named NativeTBB.cpp as shown in the following Program 3.

**Program 3 (NativeTBB.cpp):**

```
#include<tbb\blocked_range.h>
#include<tbb\parallel_for.h>
#include<tbb\task_scheduler_init.h>
#include<iostream>
using namespace std;
using namespace tbb;
#pragma warning (disable: 588)
#include "stdafx.h"
#include <jni.h>
```

```
#include "JNT_Win32_Kernel_NativeTBB.h"
JNIEXPORT void JNICALL Java_JNT_Win32_Kernel_NativeTBB_
parallelFor (JNIEnv *env, jobject obj, jintArray IPArray1, jintArray
IPArray2, jintArray Result, jchar ope){
        const jsize length1 = env->GetArrayLength (IPArray1);
   jint *IParr1 = env->GetIntArrayElements (IPArray1, NULL); //input array
        const jsize length2 = env->GetArrayLength (IPArray2);
   jint *IParr2 = env->GetIntArrayElements(IPArray2, NULL); //input array
        const jsize length3 = env->GetArrayLength(Result);
 jint *Res = env->GetIntArrayElements(Result, NULL); //input array
tbb::task_scheduler_init init();        // Automatic number of threads
tbb::tick_count parallel_start = tbb::tick_count::now();
if ((length1 == length2) & (length1 == length3))
tbb::parallel_for( tbb::blocked_range<size_t>(0,length1),
  [=](const tbb::blocked_range<size_t>& range) {
for (size_t i = range.begin(); i! = range.end(); ++i){
  if(static_cast<char>(ope) == '+')
   Res[i] = IParr1[i]+IParr2[i];
 else if(static_cast<char>(ope) =='-')
   Res[i] = IParr1[i]-IParr2[i];
else if(static_cast<char>(ope) =='*')
   Res[i] = IParr1[i]*IParr2[i];
else if(static_cast<char>(ope) =='/')
    Res[i] = IParr1[i]/IParr2[i];
   }
 },tbb::affinity_partitioner() );
else
std::cout<<"Array length are not equals";
env->ReleaseIntArrayElements(Result, Res, NULL);
tbb::tick_count parallel_end = tbb::tick_count::now();
std::cout << "Parallelism completed at" << (parallel_end-parallel_start).
seconds() <<"Sec";
}
```

The NativeTBB.cpp Program 3 included with tbb\blocked_range.h,tbb\parallel_for.h, tbb\task_scheduler_init.h. A Java_JNT_Win32_Kernel_NativeTBB_parallelFor (JNIEnv *env, jobject obj, jintArray IPArray1, jintArray IPArray2, jintArray Result, jchar ope) method creates tbb::parallel_for(…). The GetArrayLength(…) Method will get length of arrays. Here, all the arrays are expected in equal length. The GetIntArrayElements(…) Method will get array elements and will assign to pointer variable such as *IParr1, *IParr2 and *Res. The *Res will be used for storing the results which can be used at the java program. The TBB is scheduler initialized with automatic number of threads using tbb::task_scheduler_init init() or using tbb::task_scheduler_init init(tbb::task_scheduler_init::automatic). Researchers can also used to specify number of thread explicitly using tbb::task_scheduler_init init(100). The variable parallel_start allows to find the staring time of parallel_for(…) using tbb::tick_count::now() likewise the parallel_end allows us to find the end time of parallel_for(…) using tbb::tick_count::now(). Hence, researchers can calculate the actual time used for our parallel_for(…).

Before to execute the parallel_for(…), researchers have checked the arrays lengths are equal or not. Once arrays are equal, researchers used to call tbb::parallel_for(…) Method. Inside tbb::parallel_for(…), researchers declared the array range in tbb::blocked_range<size_t>(0, length1) and the blocked_range starts with range.begin() and ends with range.end(). Based on the arithmetic operator passed in the Java_JNT_Win32_Kernel_NativeTBB_parallelFor(…) method, the results are manipulated and stored in Res[] array.

After all implementing the NativeTBB.cpp Program, researchers can build the dynamic link library by choosing Build menu then Build Solution on the menu bar. This will create NativeTBB.dll which can be accessed in Java program through System.load("NativeTBB.dll").

**Testing Native Intel TBB Program:** Currently, researchers have NativeTBB.class file, NativeTBB.dll native TBB library and Intel TBB supportive or dependent library files that researchers already described in downloading and installing TBB section. The following Program 4 is the testing native TBB program, TestNativeTBB class allowed us to run the program on Win32 platform because researchers have used TBB libraries through JNI. First of all the program should be imported with JNT.Win32.Kernel.NativeTBB; this package library included a line of code that loaded a native library into the program through System. load ("NativeTBB.dll"). When program started execution the public static void main(String[] args) is create TestNativeTBB object which will automatically call the TestNativeTBB constructor.

Inside the TestNativeTBB class constructor, researchers declared sa[], sb[], sc[] and pa[], pb[], pc[] array. As usual way sa[], sb[], pa[] and pb[] arrays are initialized. Here, sa[], sb[] and sc[] arrays will be used for sequential array manipulations, like wise pa[], pb[] and pc[] arrays will be used for parallel array manipulations. However, both of the array manipulations are preformed based on what operator declared on 'ope' variable. The actual differentiation only illustrated in array manipulations. The sequential array is manipulated in usual java code; however the parallel array is manipulated using native TBB code.

The variable startTime allows to find the staring time of sequential array manipulation using system. currentTimeMillis() likewise the stopTime allows us to find the end time of sequential array manipulation using System.currentTimeMillis(). Hence, researchers can calculate the actual time used for the sequential array manipulation which is stored on elapsedTime variable. The variable N used to declare array size. Of course, researchers can create array length based on the computer system memory stack availability.

Table 1: Performance analysis

| Processors | Array size | Sequential executions (msec) | Parallel executions (msec) | Speedup | Performance improvements |
|---|---|---|---|---|---|
| Core 2 Duo 2.10 GHz with RAM 2 GB | 100000 | 1 | 0.00529292 | 188.9316 | 0.994707 |
| | 1000000 | 8 | 0.03282340 | 243.7286 | 0.995897 |
| | 10000000 | 71 | 0.29542000 | 240.3358 | 0.995839 |
| Core i5 3.30 GHz with RAM 4 GB | 100000 | 1 | 0.00181435 | 551.1616 | 0.998186 |
| | 1000000 | 4 | 0.00864509 | 462.6904 | 0.997839 |
| | 10000000 | 34 | 0.07854530 | 432.8712 | 0.997690 |
| Core i7 3.40 GHz with RAM 4 GB | 100000 | 1 | 0.00223593 | 447.2412 | 0.997764 |
| | 1000000 | 16 | 0.01001130 | 1598.194 | 0.999374 |
| | 10000000 | 31 | 0.09127830 | 339.6207 | 0.997056 |

**Program 4 (TestNativeTBB.Java):**

```
package JNT.Win32. Kernel;
import JNT.Win32. Kernel.*;
public class TestNativeTBB extends NativeTBB{
static {
        System.loadLibrary("NativeTBBDLL");
      }
 public TestNativeTBB() {
try{
        int N = 10000000;
        int sa[] = new int[N];
        int sb[] = new int[N];
        int sc[] = new int[N];
        char ope = '*';
        for(int i = 0; i<N; i++){
                        sa[i] = i;
                        sb[i] = i;
        }
 long startTime = System.currentTimeMillis();

        for(int i = 0;i<N; i++){
                        if(ope == '+')
                          sc[i] = sa[i]+sb[i];
                        else if(ope == '-')
                          sc[i] = sa[i]-sb[i];
                        else if(ope == '*')
                          sc[i] = sa[i]*sb[i];
                        else if(ope == '/')
                          sc[i] = sa[i]/sb[i];
        }
        long stopTime = System.currentTimeMillis ();
long elapsedTime = stopTime-startTime;
System.out.println ("Serialization completed at "+elapsedTime+" Sec");
        //parallel operatons
        int pa[] = new int[N];
        int pb[] = new int[N];
        int pc[] = new int[N];
        for(int i = 0;i<N; i++){
                        pa[i] = i;
                        pb[i] = i;
        }
        parallelFor(pa, pb, pc, ope);
}catch (OutOfMemoryError e){
        System. out. println ("Error: "+e);
}
 }
public static void main (String[] args) {
  try{
     TestNativeTBB Test1 = new TestNativeTBB();
  }catch(Exception e){
         System.out.println("Error "+e); }
 }
}
```

## RESULTS AND DISCUSSION

The evaluation of the parallel execution performance is measured with respect to speedup, performance improvement and efficiency with reference to the time taken for both sequential and parallel processing. Speed up measures how much a parallel algorithm is faster than a Corresponding Sequential algorithm. The speedup calculation is based on the following equation:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

The performance improvement depicts measurements relative improvement that the parallel system has over the sequential process. This performance is measured based on the following equation:

$$\text{Performance improvements} = \frac{\text{Sequential extcution time-Parallel execution time}}{\text{Sequential execution time}}$$

The Program 4, experienced in dissimilar processor core environment and array size; the execution time of sequential array manipulations results and execution time of parallel array manipulations results are arranged in the Table 1. In addition, the actual speedup and the performance impairments are computed and listed in the Table 1.

## CONCLUSION

Multi-core processors are becoming familiar these days yet writing an efficient scalable program is much harder. Scalability represents the concept that a program should get benefits of the performance as the number of processor cores increases. 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.

This research finding exercised on how Java can facilitate Intel TBB through JNI. This contribution exploited painless usage of Intel TBB Parallel algorithms used in Java. In this research, researchers 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, researchers illustrated manipulating array on single dimensional and the speedup and performance improvements demonstrated. If some other operations required performing on one dimensional array, researchers 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.

## REFERENCES

Duffy, J., 2009. Concurrent Programming on Windows. Pearson Education Inc., New York, USA.

Liang, S., 1999. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, Boston, MA., USA., ISBN-13: 9780201325775, Pages: 302.

Reinders, J., 2007. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media Inc., New York, USA., ISBN-13: 9781449390860, Pages: 336.

Veerasamy, B.D. and G.M. Nasira, 2012a. Setting CPU affinity in windows based SMP systems using Java. Int. J. Sci. Eng. Res., 3: 893-900.

Veerasamy, B.D. and G.M. Nasira, 2012b. Parallel: One time pad using Java. Int. J. Sci. Eng. Res., 3: 1109-1117.

Veerasamy, B.D. and G.M. Nasira, 2013. JNT-Java native thread for Win32 platform. Int. J. Comput. Appl., 71: 1-9.

Veerasamy, B.D. and G.M. Nasira, 2014. Java native threads for Win32 platform. Proceedings of the World Congress on Computing and Communication Technologies, February 27-March 1, 2014, Tamil Nadu, India.