High Performance Java

Jordan J. Ruloff, James A. Ross,[*] David A. Richie,[†] Song J. Park, Dale R. Shires, and Brian J. Henz[‡]
[*]Dynamics Research Corp.
Andover, MA
{jruloff, jaross}@drc.com
[†]Brown Deer Technology
Forest Hill, MD
drichie@browndeertechnology.com
[‡]U.S. Army Research Laboratory (ARL)
Aberdeen Proving Ground, MD
{song.j.park.civ, dale.r.shires.civ, brian.j.henz.civ}@mail.mil

***Abstract*—At this point in time, it is apparent that future programming paradigms will be based around many-core processors and heterogeneous computing. Diversity in new processor architectures has led to a large variety of processors, which were designed to address different issues found in past architectures while, unfortunately and unintentionally, burdening programmers to use these new architectures effectively. As more programming libraries and languages are developed, programmers will be able to design algorithms for these different architectures to maximize their code efficiency, whether to maximize performance or minimize power usage. Unfortunately, not all code can scale efficiently in many-core architectures nor can all code efficiently utilize heterogeneous architectures. Sometimes, a programmer may even have to deal with a task that is inherently serial in nature. Even if the task is trivially parallel, a programmer may even find that, due to limiting constraints of a particular architecture, like memory or interconnect speed, an algorithm best suited for a particular problem may not be the most desirable to maximize performance. In order to efficiently utilize the computing hardware, programmers must have a basic understanding of the fundamental differences between the various architectures and how to best utilize them. This paper covers the methods that were employed for addressing task and data parallelism within the Java language to maximize performance of the World Wind Java Ballistic Interface code, Java 7's fork/join framework and AMD's Aparapi Java bindings as well as the importance of parallel execution time and how to map it to the various execution frameworks.**

I. Introduction

The future of high performance computing will be based around many-core processors and heterogeneous computing. While Moore's law has continued to demonstrate itself ever since the invention of the integrated circuit in 1958, Dennard scaling has failed to keep up with the scaling of transistor size[1]. At the moment, the only attainable means for performance growth in a serial application is advancements in processor architecture, hence, it is best to transition from a serial programming paradigm towards a parallel programming paradigm. The task at hand is a difficult one for programmers, especially if one wants to maximize efficiency, but programming in parallel will continue to get easier as more programming libraries and languages are developed. In addition to the task of programming with parallelism, since processor architecture is now playing a larger role in determining performance of a program, a developer must carefully consider the requirements of their program and how to maximize the efficiency of their code on a particular architecture of choice. A quick glance at the plethora of processing architectures available, each designed with a different goal in mind promising high performance, should make it apparent that this is no easy task. Regardless, tools and application programming interfaces for multi-core and heterogeneous computing are maturing and making it easier to maximize performance of one's program.

The Ballistic Trajectory Field Calculator (BTFC) Interface is a user interface to put the power of a supercomputer in the hands of the U.S. Army warfighters. The interface allows for the creation, loading, and saving of scenarios, which involve the placements of threats and watch points on a map. A warfighter can create a scenario, as shown in Figure 1, and submit it to a high performance computer connected to the same network and get back safe positions where they could observe the watch points without being detected by the threats, as shown in Figure 2. It is critical that the interface maximize performance in order to minimize energy use on a portable device and to keep the interface responsive. The interface was coded in Java on top of NASA World Wind and makes use of the latest parallel programming additions in the Java language as well as Aparapi, a Java library that maps Java code to OpenCL™ code.

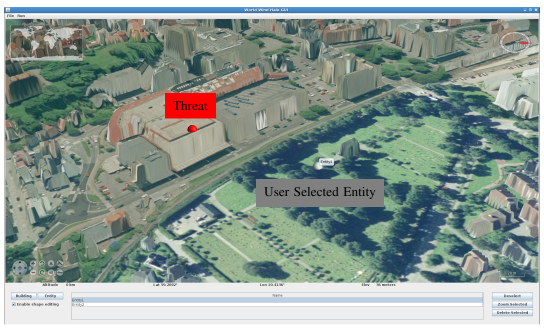

Fig. 1: Setting Up A Scenario

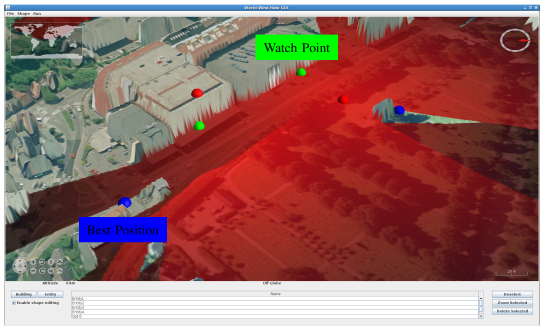

Fig. 2: Scenario With Calculated Best Positions

II. Java Parallelism

One of the features that the BTFC Interface supports is the loading of GeoTIFF files for viewing landscapes with higher resolution than what is currently provided by the NASA World Wind server for use in the scenario. The process of loading GeoTIFFs can occur in parallel except for a small portion which adds the generated data from the files to the interface. It is important the execute the process in the least amount of time possible so that the task does not inhibit the ability of the warfighter to interface with the program. In order to accelerate this process, the BTFC Interface was programmed to efficiently utilize an abstract number of processors through Java thread pooling.

In order to ease multi-threaded programming in Java, the ThreadPoolExecutor class was added in Java 5[2]. It was capable of executing tasks asynchronously and could return values from the other task upon completion and, while it was very efficient and useful, it still failed to handle all the possible cases of parallelism that may arise in a Java program with maximum efficiency. In Java 7, ForkJoinPool was added to ease multi-threaded programming in Java under conditions that ThreadPoolExecutor could not efficiently handle: unbalanced workloads and recursively subdividing tasks[3]. It is important to note that ForkJoinPool does not replace ThreadPoolExecutor, but provides an alternative to ThreadPoolExecutor. In order to evaluate the difference in performance between ThreadPoolExecutor and ForkJoinPool, a benchmark was created which loads a specified list of GeoTIFF files multiple times and the benchmark is timed in order to find which Java thread pool more efficiently utilizes its processors. The benchmark was run on two Intel® Xeon® X5675 processors.

In the first case, which is shown in Table I and Figure 3, one GeoTIFF is loaded multiple times. Since the workload for each task is equivalent, all the tasks execute in an equal amount of time and no thread spends a long time idling while another thread is executing work. The graph shows that the two thread pools have similar performance and that there is no advantage to using a ForkJoinPool.

GeoTIFFs	ThreadPoolExecutor(s)	ForkJoinPool(s)
1	0.62	0.62
2	0.63	0.64
4	0.67	0.66
8	0.72	0.72
16	0.97	0.98
32	1.43	1.43
64	3.07	3.06

TABLE I: Execution Time For Loading Same GeoTIFF Multiple Times

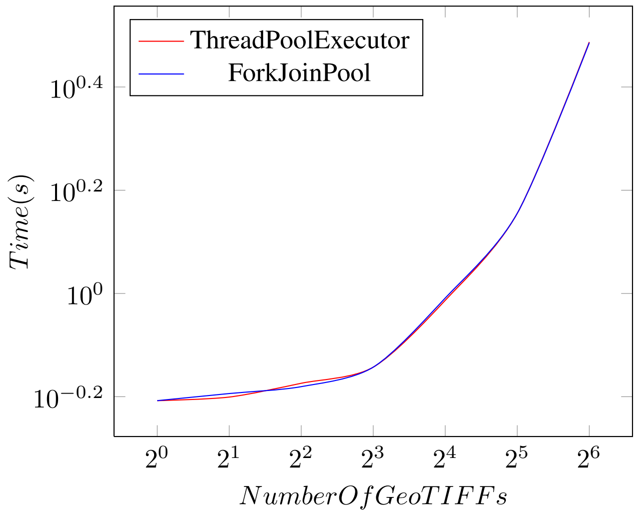

Fig. 3: Execution Time For Loading Same GeoTIFF Multiple Times

In the second case, which is shown in Table II and Figure 4, four GeoTIFFs are loaded multiple times. The GeoTIFFs vary in both type and file size, so the workload assigned across all the threads is unbalanced. If there is an enough of an imbalance, the idle threads in the ForkJoinPool can steal work from the queue of the busier threads, which more effectively utilizes the thread pool. In unbalanced workloads, it is more effective to use ForkJoinPools due to work stealing.

GeoTIFFs	ThreadPoolExecutor(s)	ForkJoinPool(s)
4	1.15	1.17
8	1.24	1.23
16	1.31	1.31
32	1.67	1.69
64	2.64	2.63
128	5.30	4.22
256	9.99	7.12

TABLE II: Execution Time For Loading Multiple GeoTIFFs Multiple Times

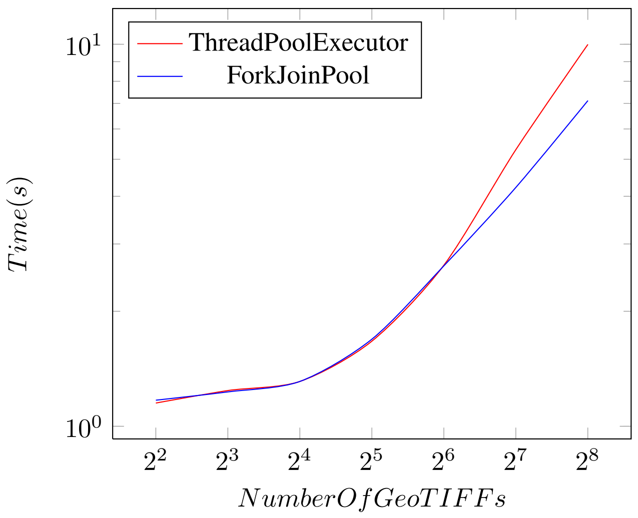

Fig. 4: Execution Time For Loading Multiple GeoTIFFs Multiple Times

Since the BTFC Interface needs to remain responsive and execute its tasks as quick as possible, it was decided to use ForkJoinPool since it offered the best time for the worst case scenario. When designing a user interface, it is important to always consider worst case as one worst case can ruin the user experience.

III. Aparapi

While NASA World Wind may be capable of loading GeoTIFFs, not every file a user may want to use for BTFC Interface may be a GeoTIFF. In this case, one may have to convert a file to a GeoTIFF in order to properly render the object in NASA World Wind. This benchmark parses a triangle data file, builds a Bounding Volume Hierarchy (BVH) tree[4], and then does some intersection tests in order to generate height values which could be put into a GeoTIFF. Due to this benchmark's data parallel nature, Aparapi was chosen as the library of choice. Aparapi is a Java library which can convert Java byte code to OpenCL™ code while hiding the complexities of running the OpenCL™ code and binding it to Java from the programmer[5]. While it is still a rather new library and still has not fully matured, it has demonstrated some interesting performance benefits over native Java threading when the algorithm is data parallel. At this time, there are a few restrictions when using Aparapi, such as ensuring that there are no new arrays or structures and no passing local memory objects into a function. Fortunately, all the kernels used in this benchmark could be modified to avoid the restrictions imposed by Aparapi[6]. The benchmark was run on two Intel® Xeon® X5675 processors and one AMD Radeon™ HD 6970.

The construction of the BVH tree can be broken down into two major steps; the first of which is the preparation step. The preparation step for building the BVH tree consists of constructing bounding boxes for each individual triangle as well as sorting the bounding boxes by dimension. Construction of the bounding boxes executes in constant parallel time while the bitonic sort occurs in $(\log N)^2$ parallel time[7], [8]. The execution times for the preparation kernels are shown in Table III and Figure 5. As one can see from the figure, there is a significant amount of overhead associated with Java threading; in Aparapi, each OpenCL™ thread is equivalent to a spawned Java thread, which are synchronized by cyclic barriers. By default, Aparapi may attempt to run as many as 256 threads, but it is usually best to specify a thread number no greater than the number of cores in your machine if executing the kernel in Java. Additionally, one can see from the figure that there is a relatively small increase in execution time as the number of triangles increases for the OpenCL™ execution cases. Since the slope is so small for the first few test cases, one can infer that a significant portion of the execution time is spent compiling the kernel and moving memory for these first few test cases; this is the reason why the native Java implementation is faster than the OpenCL™ implementations for a small number of triangles.

Triangles	JTP(s)	CPU(s)	GPU(s)
65536	0.717	0.951	0.972
131072	1.028	0.989	0.977
262144	1.407	1.068	1.014
524288	2.693	1.315	1.107
1048576	4.960	2.126	1.325

TABLE III: Execution Time For Preparation Kernels

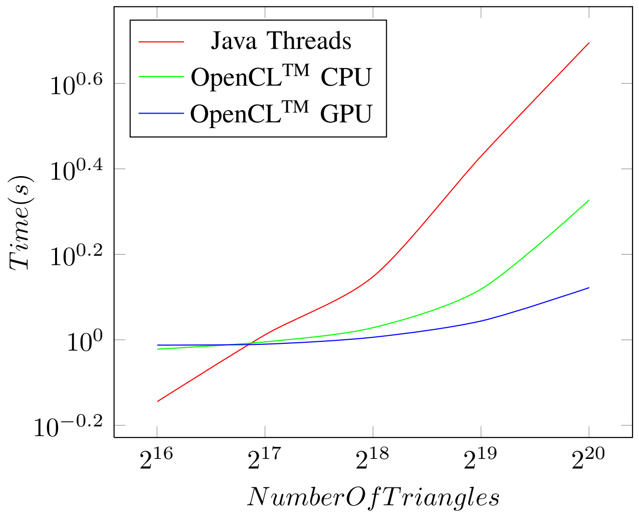

Fig. 5: Execution Time For Preparation Kernels

The second step in constructing the BVH tree is the actual building of the BVH tree. The algorithm for construction of a BVH tree, shown below in Algorithm 1, can be divided into multiple pieces of parallel portions with varying degrees of parallelism[9], [10], [11]. Initially, the portion of the algorithm with a limited amount of parallelism is choosing the dimension along which to split the structures, which may adversely affect architectures designed for parallelism initially, but increasing the size of the tree should show an increased overall performance as more splits occur in parallel and the algorithm begins to spend less relative time splitting the node and more relative time reordering.

Algorithm 1 The BVH Tree Construction Algorithm

```
1:  procedure BUILDBVHTREE
2:      $SetUpInitialNode$
3:      while $NotFullySplit$ do
4:          for all $NodesWhichExist$ do
5:              for all $Dimension \in Dimensions$ do
6:                  $ChooseBestSplit$
7:              end for
8:              $SplitNode$
9:              $SetupNodeForDataParallelPortion$
10:             for all $Dimension \in Dimensions$ do
11:                 if $DimensionWasNotSplitDimension$ then
12:                     for all $Indexes \in SortedIndexes$ do
13:                         $CalculateNewPosition$
14:                         $Reorder$
15:                     end for
16:                 end if
17:             end for
18:         end for
19:     end while
20:     for all $Node \in Nodes$ do
21:         $CalculateAxisAlignedBoundingBoxes$
22:     end for
23:     for $Values \in Sorted$ do
24:         $SetToSortedIndexInFirstDimension$
25:     end for
26: end procedure
```

Table IV and Figure 6 shows the execution time for building a BVH tree. It is interesting to note that, as the number of structures to organize gets larger, the difference in execution time between Java and OpenCL™ for the CPU decreases. This is due to the majority of the execution time being spent choosing the best split plane, which is the least parallel portion of the algorithm and has the least amount of overhead associated with the cyclic barriers.

Triangles	JTP(s)	CPU(s)	GPU(s)
65536	2.017	0.944	1.146
131072	4.079	2.545	2.596
262144	10.380	8.717	7.475
524288	34.065	32.765	24.936
1048576	126.286	127.443	89.760

TABLE IV: Execution Time For Build Tree Kernel

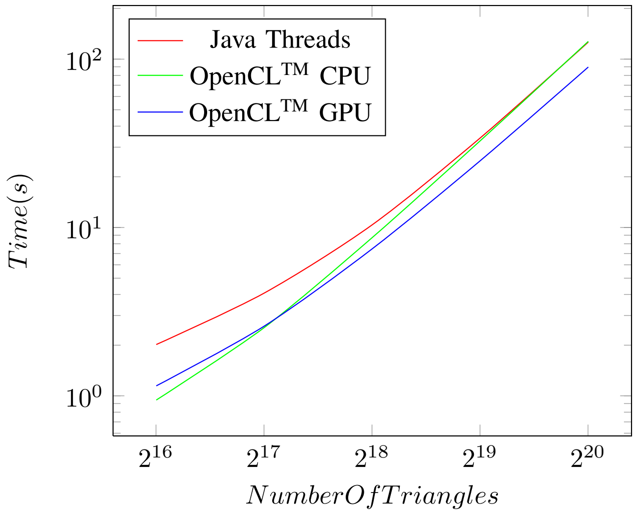

Fig. 6: Execution Time For Build Tree Kernel

Lastly, this benchmark calculates which structure is the closest structure to be intersected by a particular ray for a set of rays. The algorithm for this portion is parallel across all the rays as well as parallel across all the resulting bounding boxes in the leaf nodes of the BVH tree. The algorithm is shown in Algorithm 2 and executes in $\log N$ parallel time for N number of structures. It is important to note that, while testing against each leaf node is less efficient than traversing the tree, finding an intersection in this manner parallelizes across more cores and executes in less parallel time, which would mean a faster execution time if the machine can process enough threads simultaneously. Additionally, this algorithm has a better worst case execution time than the worst case execution time for traversing the tree.

Algorithm 2 The BVH Tree Intersection Algorithm

```
1:  procedure INTERSECTBVHTREE
2:      for all $Ray \in Rays$ do in Parallel
3:          for all $Node \in LeafNodes$ do in Parallel
4:              if $RayHitsBoundingBox$ then
5:                  $FindClosestStructureHit$
6:              end if
7:          end for
8:          $FindClosestNodeHit$
9:      end for
10: end procedure
```

Due to the relatively limited amount of memory present in the machine, the amount of parallelism expressed in the intersection test kernel had to be restricted when the BVH tree became exceptionally large. Despite this constraint, Table V and Figure 7 clearly shows that the speed of execution is dependent on the manner in which the threads were executed as well as the amount processing power of a particular architecture. Native Java threads execute the task an order of magnitude slower than OpenCL™ threads, which is expected since Java has been shown to be less efficient at utilizing the hardware available than C[12], [13]. Additionally, a GPU is a throughput orientated architecture which is designed to maximized overall performance by hiding latency through context switching at the expense of additional latency[14], [15]. Through hiding latency and maximizing performance of its many simpler cores, GPUs are shown to have more processing power than CPUs[16].

Triangles	JTP(s)	CPU(s)	GPU(s)
65536	96.618	2.441	0.310
131072	162.488	4.192	0.385
262144	229.051	7.843	0.554
524288	481.334	15.640	0.876
1048576	933.093	31.352	1.529

TABLE V: Execution Time For Intersection Kernel

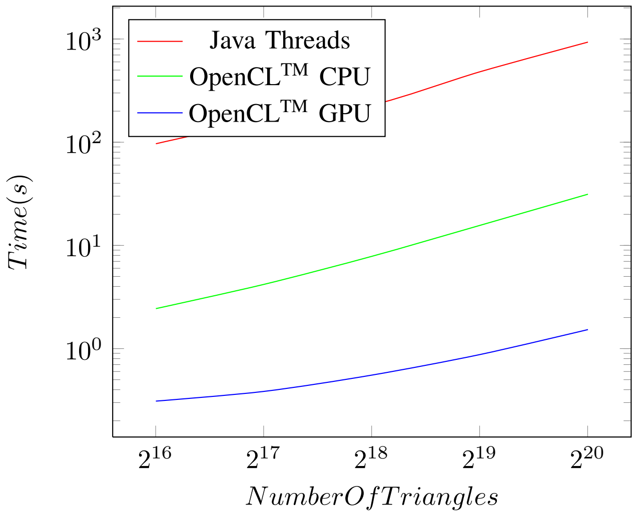

Fig. 7: Execution Time For Intersection Kernel

Aparapi has demonstrated itself as a high performance library which easily integrates with the BTFC Interface. The additional performance from executing data parallel algorithms efficiently across multiple processors as well as on processors which native Java can not interact with enables the BTFC Interface to execute compute intensive tasks rapidly without hanging.

IV. Conclusion

When programming for manycore and heterogeneous architectures, it is critical to consider how effectively the code utilizes all the processing cores. In the case of the BTFC Interface, it is critical to minimize execution time and keep the interface responsive since the interface is the only way for the program and the warfighter to interact. By utilizing the ForkJoinPool introduced in Java 7, it is possible to maximize CPU utilization and minimize execution time of task parallel workloads. Additionally, it is possible to efficiently assign work to a GPU or maximize the performance of a CPU for data parallel workloads by utilizing Aparapi. Through proper usage of these libraries, it is possible to not only minimize program execution time, but also maximize the efficiency of the user's time, which is ultimately the goal of any software system. In conclusion, incorporating these libraries into the BTFC Interface assists in achieving the overall goal of minimizing system time where the server, the interface, and the warfighter are all part of the complete system.

Acknowledgments

Support for this work is provided in part by the Army High Performance Computing Research Center (AHPCRC).

References

[1] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," 2007.

[2] H. Rajan, S. M. Kautz, and W. Rowcliffe, "Concurrency by modularity: Design patterns, a case in point," in *OOPSLA*, pp. 790–805, 2010.

[3] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, (New York, NY, USA), pp. 36–43, ACM, 2000.

[4] C. Lauterbach, M. Garland, S. Sengupta, D. P. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[5] S. Joshi, "Leveraging aparapi to help improve financial java application performance," 2012.

[6] G. Frost, "Aparapi java kernel guidelines," Sept. 2011.

[7] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computing Conference*, pp. 307–314, 1968.

[8] P. Kipfer and R. Westermann, "Improved GPU sorting," in *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (M. Pharr, ed.), pp. 733–746, Addison-Wesley, 2005.

[9] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, pp. 1170–1183, Dec. 1986.

[10] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pp. 61–70, 2006.

[11] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Trans. Graph.*, vol. 27, pp. 126:1–126:11, Dec. 2008.

[12] S. Taylor, D. D. Cook, and M. D. Levine, "A comparison of multithreading implementations," in *In Yale Multithreaded Programming Workshop*, 1998.

[13] R. Hundt, "Loop recognition in c++/java/go/scala," in *Proceedings of Scala Days 2011*, 2011.

[14] K. Fatahalian and M. Houston, "A closer look at gpus," *Commun. ACM*, vol. 51, pp. 50–57, Oct. 2008.

[15] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Commun. ACM*, vol. 53, pp. 58–66, Nov. 2010.

[16] K. A. Hawick, A. Leist, and D. P. Playne, "Mixing multi-core cpus and gpus for scientific simulation software," Tech. Rep. CSTN-091, Institute of Information and Mathematical Sciences, Massey University, Auckland, New Zealand, 2009.