# A method to efficiently select the suitable barrier algorithm during the system characterization task

Puneet Sharma and Alan Sussman

Department of Computer Science,
University of Maryland College Park, College Park MD 20742, USA,
{puneet,als}@cs.umd.edu

**Abstract.** Comparing implementations of various thread barrier algorithms is really an impossible task for the system with lots of processors. With brute force approach, we have to measure the performance of each algorithm in ${}_nC_t$ times, where n is the maximum number of processors and t is the number of threads. This task is exponential in nature and impossible to accomplish for machine with huge number of processors. We designed an algorithm, which uses the symmetric characteristics of the underlying system architecture, and filter out the unnecessary and redundant measurements. This algorithm helps us to compare the performance of various thread barrier algorithms for the machine with any number of processors.

**Keywords:** thread barrier, characterization, automation, cache, system

## 1 Introduction

The Architecture-Aware Compiler Environment (AACE) program was funded by DARPA, and Adaptive Environment for Supercomputing with Optimized Parallelism (AESOP), [1], was one of the teams funded under the AACE program. The goal of AESOP project was to build the parallel compilers which can take advantage of the hardware characteristics of the underlying system to generate optimized compiled code. There were various components of this system, which were being built by various teams at different locations. The system characterization (SYSCHAR) component of AESOP was being handled by the BAE Systems and the University of Maryland College Park under the supervision of Prof. Alan Sussman. This component measures the properties of hardware and operating system, which are eventually be used by AESOP compiler and runtime to perform optimization in parallelism. The SYSCHAR system composed of:

1. Various utility programs and libraries for managing results, producing input files for T2 teams scoring algorithms etc.
2. Operating system adaptation libraries (OSAL) which provide the various functionalities like setting thread affinity, thread barrier implementation,

thread broadcast implementation etc. This layer contains the many implementations of the same functionality and chooses best implementation for the given system by running them and measuring the performance.
3. Micro-benchmarks, which measure either directly or through the aggregation of the results from Nano-benchmarks, various properties of the system like cache size, number of cache levels, associativity etc.
4. Platform-benchmarks which measure system properties directly. These benchmark must run on the node where they have been compiled.
5. Nano-benchmarks to measure the specific operation like floating-point divide operations. These benchmarks are generated by the template where micro-benchmarks supply parameters to this template.

We looked into the various barrier implementations in OSAL layer and tried to understand them. The basic motivation behind this task was to understand the methods to measure the performance of different barrier algorithms for different number of threads. The code location is x-ray/osal/source/thread_barrier, where sub-folders, all_static_tree, butterfly, central, dissemination, pthread, rice _tree, and static_tree, provide different kind of thread barrier implementations. File, check.c, exercises these various implementation on different number of threads, measures the timings, and reports them to the systems. System picks the best implementation based on the timings results. All runs are stored at x-ray/osal/temp. We found that:

1. The Splatter function in check.c performs the partitioning of the threads and generates <threadId, processor Index> vector. Each record of this vector specified which thread will be attached to which processor. Moreover, splatter function doesnt use hardware characteristics to filter out the redundant measurements.
2. Although, thread barrier runs are being reported on various number of thread count, but only timings evaluated on thread count equals to maximum number of contexts are being considered while choosing the thread barrier implementation.

So, our task was to come up with an algorithm that can generate optimized <threadId, processorIndex>vectors for various thread counts by using underlying system characteristics.

To retrieve the underlying system characteristics, we used Hardware Locality (hwloc) software, [4], which is already one of the components of AESOP system. This software gathers hardware information about processors, caches, memory nodes and more, and exposes it to applications and runtime systems in a abstracted and portable hierarchical manner. This may significantly help performance by having runtime systems place their tasks or adapt their communication strategies depending on hardware affinities.

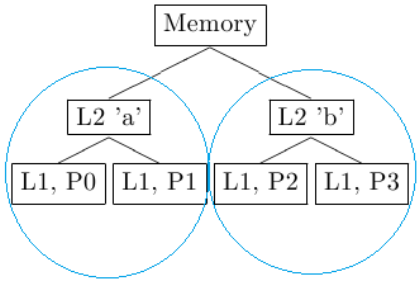
After gathering hardware characteristics, tree like system architecture information, we used it as an input to our algorithm, explained in section 2. This algorithm emits the optimized number of measurements required for any barrier algorithm for the given number of threads. So the different thread barrier implementations can be compared.

Without our algorithm, it is really impossible to compare the various barrier implementations for the machines with lots of processors. That is because, we had to run each barrier implementation ${}_nC_t$ times, where n is the maximum number of processors and t is the number of threads. So as n grows higher, the runs are being increase exponentially, which makes measurement difficult or almost impossible.

Our algorithm filters out many redundant runs by using symmetric characteristics of the underlying system. Our algorithm is generic as it is independent of the memory hierarchy. So, we can use the same algorithm in the future systems, with more than three levels of cache hierarchy. Also, we can use the same algorithm to compare various thread broadcast implementations. In section 2 we describe the algorithm, and provide examples. Section 3 concludes this article with possible future directions.

## 2 Algorithm

We describe few terms below, which should be understood before going through the algorithm. With the help of hwloc software, [4], we can generate a system architecture tree like below.




- **Configuration:** Each subtree rooted at a node above than the leaf level is called configuration. We represent the configuration by labeling the subtree root. So, In the above tree, there are two configurations, each encircled and labeled 'a' and 'b' respectively. Also, the root of the subtree is called the configuration node.
- **Configuration with number of threads:** We represent configuration and number of threads scheduled on this configuration by $a_t$, where 'a' is a configuration and 't' is the number of threads scheduled on 'a'. Also each of these threads can be scheduled to any processor inside the configuration provided that each processor has only one thread.
- **Configuration Set:** A set $\{a_{t1}, b_{t2}, c_{t3}, ...n_{tn}\}$, is a configuration set, where a,b,c,...n are the configurations and t1,t2,t3,...tn are the number of threads scheduled on each configuration. So the total number of scheduled thread on this configuration set is t1+t2+t3+....+tn.

**Algorithm 1** Compute optimum Configuration Sets.

**Require:** System architecture looks like a tree.

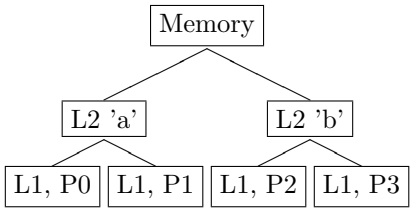
**Input:** A tree structure of the memory hierarchy and processors information attached to lowest level cache/leaf node of the tree.

**Output:** Collection of configuration set for each thread count. Where thread count is $\geq 0$ and $\leq$ number of processors in the system.

1. In the given tree, assign each configuration, a key. The key can be calculated by following expression:
   key = {number of children of the given configuration node, number of children of configuration node's parent, number of children of configuration node's grand parent... root node of tree}.
2. If two configurations have the same key and are from the same hardware vendor, they are same. Here 'same' implies that, only one configuration will be considered in the evaluation (except few scenarios).
3. Label each configuration uniquely.
4. For each of these configurations, calculate how many threads can be scheduled by counting number of leaves each configuration has.
5. For each of these configurations, use label and number of schedulable threads, calculated in step 3 and 4, and generate the following type of set:
   S = $\{a_0, a_1, a_2, a_3\}$,
   where $a_i$ is the 'configuration with number of threads', and $0 \leq i \leq$ maximum number of schedulable threads on the given configuration (number of leaves associated to the configuration).
6. Calculate the cartesian product of the sets, generated in step 5. The expression will look like :
   $a \times b \times c \times ..... \times z = \{a_{p1}, b_{p2}, c_{p3}, .....z_{pn}\}$, where
   a = $\{a_0, a_1.....a_i\}$, b = $\{b_0, b_1.....b_j\}$, c = $\{c_0, c_1.....c_k\}$, z = $\{z_0, z_1.....z_t\}$, and
   $0 \leq p1 \leq i$, $0 \leq p2 \leq j$, $0 \leq p3 \leq k$, $0 \leq pn \leq t$
7. Each set generated in step 6, is called a configuration set, and looks like $\{a_i, b_j, c_k, ....., z_n\}$, where the total number of scheduled threads or thread count is $i + j + k + .....n$.
8. From step 1 to step 7, we get the collection of configuration set for thread count $\geq 0$ and $\leq$ maximum number of schedulable threads on the given architecture. This list is not optimal as there will be records associated to the same configurations as well. So, if there are no configurations having the same key, we go to step 10. Otherwise, we go to step 9 to further prune our results.
9. Find out the the records associated to the same configurations, and merge the duplicate/redundant records. A redundant records can be found by this expression.
   S1 = $\{a_i, b_j, c_k, d_t.....z_n\}$
   S2 = $\{a_j, b_k, c_i, d_t.....z_n\}$
   S3 = $\{a_k, b_i, c_j, d_t.....z_n\}$, where '$d_t....z_n$' is same in all three sets.
   So, set S1, S2 and S3 are duplicate sets and can be merged in one by taking any one of them as a final set.
10. We get the collection of configuration set for each thread count $\geq 0$ and $\leq$ maximum number of schedulable threads on the given architecture, and this collection is optimal.

Now we can go through the Algorithm 1, which is pretty much self explanatory. However, we can understand this algorithm more thoroughly by running it with few examples below.

- **Example 1**




The above tree, shows a common system architecture, where main memory is connected to two level-2 caches and each level-2 cache connected to two level-1 cache. Each level-1 cache is associated with one of the processors, represented by label 'P' and some index. So, the tree has four processors/contexts.
Here is the step by step description when we apply Algorithm 1 to this tree.

1. We calculate keys for each configuration, which is same and equals to {2,2}.
2. So, the two configurations are same (according to point 2 in the algorithm).
3. We label left configuration as 'a' and right configuration as 'b'.
4. We can schedule maximum two threads on 'a' and maximum two threads on 'b'.
5. So, the set associated to each configuration can be given by :
   $S_a = \{a_0, a_1, a_2\}, S_b = \{b_0, b_1, b_2\}$
6. We calculate the cartesian product on the above sets and compute total no. of threads according to step 6 and step 7. The results are shown in Table 1.

**Table 1.** Cartesian product and number of threads calculated in Example 1

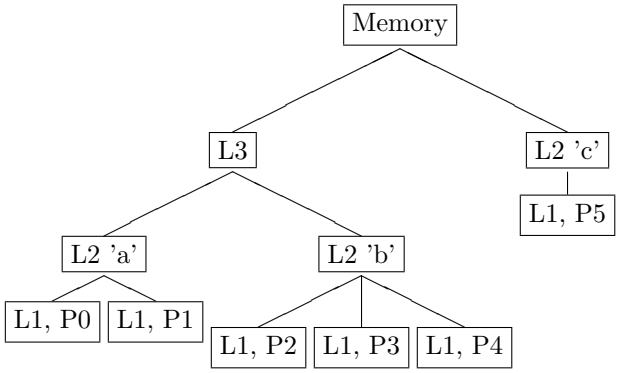
| configuration set | # threads | configuration set | # threads |
|---|---|---|---|
| $(a_0, b_0)$ | 0 | $(a_1, b_2)$ | 3 |
| $(a_0, b_1)$ | 1 | $(a_2, b_0)$ | 2 |
| $(a_0, b_2)$ | 2 | $(a_2, b_1)$ | 3 |
| $(a_1, b_0)$ | 1 | $(a_2, b_2)$ | 4 |
| $(a_1, b_1)$ | 2 | | |

7. So, we get the configuration sets for number of threads $\geq 0$ and $\leq 4$. We still need to optimize this set by removing duplicates according to the step 9 of the algorithm. The results are shown in Table 2.

**Table 2.** Configuration Sets and number of schedulable threads on each Configuration Set.

| configuration set | # threads | configuration set | # threads |
|---|---|---|---|
| $(a_0, b_0)$ | 0 | $(a_1, b_1)$ | 2 |
| $(a_0, b_1)$ | 1 | $(a_1, b_2)$ | 3 |
| $(a_0, b_2)$ | 2 | $(a_2, b_2)$ | 4 |

- **Example 2**




The above tree, shows not-so-common system architecture, where main memory is connected to one level-3 cache and one level-2 cache. Level-2 cache is connected to only one level-1 cache associated to processor 5. The level-3 cache is connected to two level-2 caches, where one conncted to two level-1 caches and another is connected to three level-1 caches. And the above tree has six processors/contexts. Again, the step by step description when we apply Algorithm 1 to this tree is given below.

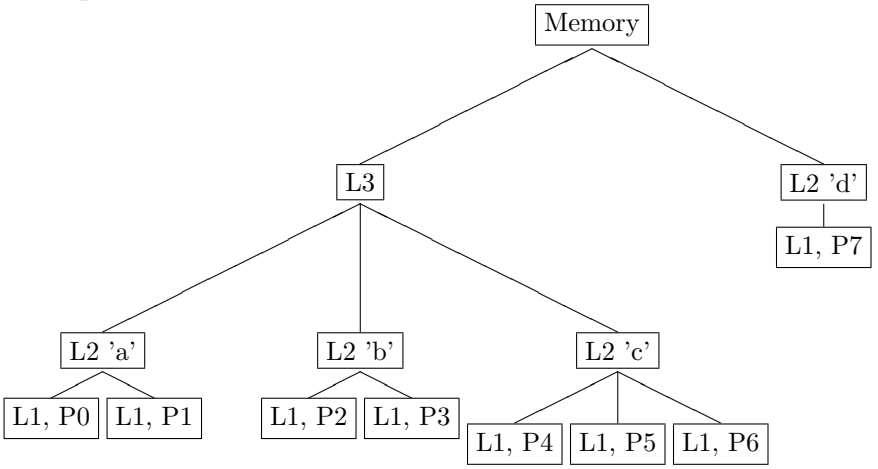
1. We calculate keys for each configuration. For left most configuration key is {2,2,2}, for middle configuration key is {3,2,2}, and for right most configuration cache key is {1,2}.
2. So, none of the configuration is same. (according to point 2 in the algorithm).
3. We label left most configuration as 'a', middle one as 'b' and right most one as 'c'.
4. We can schedule maximum two threads on 'a', maximum three threads on 'b' and maximum one thread on 'c'.
5. So, the set associated to each configuration can be given by :
   $S_a = \{a_0, a_1, a_2\}, S_b = \{b_0, b_1, b_2, b_3\}, S_c = \{c_0, c_1\}$
6. We calculate the cartesian product on the above sets and compute total no. of threads according to step 6 and step 7. The results are shown in Table 3.

**Table 3.** Configuration Sets and number of schedulable threads on each Configuration Set.

| configuration set | # threads | configuration set | # threads | configuration set | # threads |
|---|---|---|---|---|---|
| $(a_0, b_0, c_0)$ | 0 | $(a_1, b_0, c_0)$ | 1 | $(a_2, b_0, c_0)$ | 2 |
| $(a_0, b_0, c_1)$ | 1 | $(a_1, b_0, c_1)$ | 2 | $(a_2, b_0, c_1)$ | 3 |
| $(a_0, b_1, c_0)$ | 1 | $(a_1, b_1, c_0)$ | 2 | $(a_2, b_1, c_0)$ | 3 |
| $(a_0, b_1, c_1)$ | 2 | $(a_1, b_1, c_1)$ | 3 | $(a_2, b_1, c_1)$ | 4 |
| $(a_0, b_2, c_0)$ | 2 | $(a_1, b_2, c_0)$ | 3 | $(a_2, b_2, c_0)$ | 4 |
| $(a_0, b_2, c_1)$ | 3 | $(a_1, b_2, c_1)$ | 4 | $(a_2, b_2, c_1)$ | 5 |
| $(a_0, b_3, c_0)$ | 3 | $(a_1, b_3, c_0)$ | 4 | $(a_2, b_3, c_0)$ | 5 |
| $(a_0, b_3, c_1)$ | 4 | $(a_1, b_3, c_1)$ | 5 | $(a_2, b_3, c_1)$ | 6 |

7. So, we get the configuration sets for number of threads $\geq 0$ and $\leq 6$. And this result is optimized as we don't have duplicate configurations in our tree.

- **Example 3**




The above tree looks like a tree in example 2, except it's level-3 cache has one more level-2 cache. Also this tree has eight processors/contexts. And we have already labeled all configurations. Again we show the step by step process of applying Algorithm 1 to this tree.

1. We calculate keys for each already labeled configuration. The key of 'a' is {2,3,2}, key of 'b' is {2,3,2}, key of c is {3,3,2}, and key of 'd' is {1,2}.
2. So, the two configurations, 'a' and 'b', are same. (according to point 2 in the algorithm).
3. We can schedule maximum two threads on 'a' and 'b' each, maximum three threads on 'c', and maximum one thread on 'd'.
4. So, the set associated to each configuration can be given by :
   $S_a = \{a_0, a_1, a_2\}, S_b = \{b_0, b_1, b_2\}, S_c = \{c_0, c_1, c_2, c_3\}, S_d = \{d_0, d_1\}$

5. We calculate the cartesian product on the above sets and compute total no. of threads according to step 6 and step 7 of the algorithm. The results are shown in Table 4.

**Table 4.** Cartesian product and number of threads calculated in Example 3

| Conf. Set | # tds | Conf. Set | # tds | Conf. Set | # tds | Conf. Set | # tds |
|---|---|---|---|---|---|---|---|
| $(a_0, b_0, c_0, d_0)$ | 0 | $(a_0, b_2, c_1, d_0)$ | 3 | $(a_1, b_1, c_2, d_0)$ | 4 | $(a_2, b_0, c_3, d_0)$ | 5 |
| $(a_0, b_0, c_0, d_1)$ | 1 | $(a_0, b_2, c_1, d_1)$ | 4 | $(a_1, b_1, c_2, d_1)$ | 5 | $(a_2, b_0, c_3, d_1)$ | 6 |
| $(a_0, b_0, c_1, d_0)$ | 1 | $(a_0, b_2, c_2, d_0)$ | 4 | $(a_1, b_1, c_3, d_0)$ | 5 | $(a_2, b_1, c_0, d_0)$ | 3 |
| $(a_0, b_0, c_1, d_1)$ | 2 | $(a_0, b_2, c_2, d_1)$ | 5 | $(a_1, b_1, c_3, d_1)$ | 6 | $(a_2, b_1, c_0, d_1)$ | 4 |
| $(a_0, b_0, c_2, d_0)$ | 2 | $(a_0, b_2, c_3, d_0)$ | 5 | $(a_1, b_2, c_0, d_0)$ | 3 | $(a_2, b_1, c_1, d_0)$ | 4 |
| $(a_0, b_0, c_2, d_1)$ | 3 | $(a_0, b_2, c_3, d_1)$ | 6 | $(a_1, b_2, c_0, d_1)$ | 4 | $(a_2, b_1, c_1, d_1)$ | 5 |
| $(a_0, b_0, c_3, d_0)$ | 3 | $(a_1, b_0, c_0, d_0)$ | 1 | $(a_1, b_2, c_1, d_0)$ | 4 | $(a_2, b_1, c_2, d_0)$ | 5 |
| $(a_0, b_0, c_3, d_1)$ | 4 | $(a_1, b_0, c_0, d_1)$ | 2 | $(a_1, b_2, c_1, d_1)$ | 5 | $(a_2, b_1, c_2, d_1)$ | 6 |
| $(a_0, b_1, c_0, d_0)$ | 1 | $(a_1, b_0, c_1, d_0)$ | 2 | $(a_1, b_2, c_2, d_0)$ | 5 | $(a_2, b_1, c_3, d_0)$ | 6 |
| $(a_0, b_1, c_0, d_1)$ | 2 | $(a_1, b_0, c_1, d_1)$ | 3 | $(a_1, b_2, c_2, d_1)$ | 6 | $(a_2, b_1, c_3, d_1)$ | 7 |
| $(a_0, b_1, c_1, d_0)$ | 2 | $(a_1, b_0, c_2, d_0)$ | 3 | $(a_1, b_2, c_3, d_0)$ | 6 | $(a_2, b_2, c_0, d_0)$ | 4 |
| $(a_0, b_1, c_1, d_1)$ | 3 | $(a_1, b_0, c_2, d_1)$ | 4 | $(a_1, b_2, c_3, d_1)$ | 7 | $(a_2, b_2, c_0, d_1)$ | 5 |
| $(a_0, b_1, c_2, d_0)$ | 3 | $(a_1, b_0, c_3, d_0)$ | 4 | $(a_2, b_0, c_0, d_0)$ | 2 | $(a_2, b_2, c_1, d_0)$ | 5 |
| $(a_0, b_1, c_2, d_1)$ | 4 | $(a_1, b_0, c_3, d_1)$ | 5 | $(a_2, b_0, c_0, d_1)$ | 3 | $(a_2, b_2, c_1, d_1)$ | 6 |
| $(a_0, b_1, c_3, d_0)$ | 4 | $(a_1, b_1, c_0, d_0)$ | 2 | $(a_2, b_0, c_1, d_0)$ | 3 | $(a_2, b_2, c_2, d_0)$ | 6 |
| $(a_0, b_1, c_3, d_1)$ | 5 | $(a_1, b_1, c_0, d_1)$ | 3 | $(a_2, b_0, c_1, d_1)$ | 4 | $(a_2, b_2, c_2, d_1)$ | 7 |
| $(a_0, b_2, c_0, d_0)$ | 2 | $(a_1, b_1, c_1, d_0)$ | 3 | $(a_2, b_0, c_2, d_0)$ | 4 | $(a_2, b_2, c_3, d_0)$ | 7 |
| $(a_0, b_2, c_0, d_1)$ | 3 | $(a_1, b_1, c_1, d_1)$ | 4 | $(a_2, b_0, c_2, d_1)$ | 5 | $(a_2, b_2, c_3, d_1)$ | 8 |

6. So, we get the configuration sets for number of threads $\geq 0$ and $\leq 8$. We still need to optimize this set by removing duplicates according to the step 9 of the algorithm. The number of results came down from 72 to 48 due to duplicate removal. The results are shown in Table 5.

## 3 Conclusion and Future work

We designed an algorithm which can compare the performance of various thread barrier implementations in an optimal way. Instead of exponentially running the measurements, our algorithm filters out the unnecessary and redundant measurement by using underlying system characteristics, thus making it possible to find out the suitable or best barrier algorithm for the given number of threads.

Initially, instead of finding suitable algorithm for each and every thread count, we can find out the suitable algorithm for the representative number of threads like thread count equals to the power of two. And during the system run, if request arises for the different number of threads for which we dont have measurements, we can always go back and find out the suitable barrier algorithm. We also save this result so that it can be used later.

**Table 5.** Configuration Sets and number of schedulable threads on each Configuration Set.

| Conf. Set | # tds | Conf. Set | # tds | Conf. Set | # tds |
|---|---|---|---|---|---|
| $(a_0, b_0, c_0, d_0)$ | 0 | $(a_0, b_0, c_0, d_1)$ | 1 | $(a_0, b_0, c_1, d_0)$ | 1 |
| $(a_0, b_1, c_0, d_0)$ | 1 | $(a_0, b_0, c_1, d_1)$ | 2 | $(a_0, b_0, c_2, d_0)$ | 2 |
| $(a_0, b_1, c_0, d_1)$ | 2 | $(a_0, b_1, c_1, d_0)$ | 2 | $(a_0, b_2, c_0, d_0)$ | 2 |
| $(a_1, b_1, c_0, d_0)$ | 2 | $(a_0, b_0, c_2, d_1)$ | 3 | $(a_0, b_0, c_3, d_0)$ | 3 |
| $(a_0, b_1, c_1, d_1)$ | 3 | $(a_0, b_1, c_2, d_0)$ | 3 | $(a_0, b_2, c_0, d_1)$ | 3 |
| $(a_0, b_2, c_1, d_0)$ | 3 | $(a_1, b_1, c_0, d_1)$ | 3 | $(a_1, b_1, c_1, d_0)$ | 3 |
| $(a_1, b_2, c_0, d_0)$ | 3 | $(a_0, b_0, c_3, d_1)$ | 4 | $(a_0, b_1, c_2, d_1)$ | 4 |
| $(a_0, b_1, c_3, d_0)$ | 4 | $(a_0, b_2, c_1, d_1)$ | 4 | $(a_0, b_2, c_2, d_0)$ | 4 |
| $(a_1, b_1, c_1, d_1)$ | 4 | $(a_1, b_1, c_2, d_0)$ | 4 | $(a_1, b_2, c_0, d_1)$ | 4 |
| $(a_1, b_2, c_1, d_0)$ | 4 | $(a_2, b_2, c_0, d_0)$ | 4 | $(a_0, b_1, c_3, d_1)$ | 5 |
| $(a_0, b_2, c_2, d_1)$ | 5 | $(a_0, b_2, c_3, d_0)$ | 5 | $(a_1, b_1, c_2, d_1)$ | 5 |
| $(a_1, b_1, c_3, d_0)$ | 5 | $(a_1, b_2, c_1, d_1)$ | 5 | $(a_1, b_2, c_2, d_0)$ | 5 |
| $(a_2, b_2, c_0, d_1)$ | 5 | $(a_2, b_2, c_1, d_0)$ | 5 | $(a_0, b_2, c_3, d_1)$ | 6 |
| $(a_1, b_1, c_3, d_1)$ | 6 | $(a_1, b_2, c_2, d_1)$ | 6 | $(a_1, b_2, c_3, d_0)$ | 6 |
| $(a_2, b_2, c_1, d_1)$ | 6 | $(a_2, b_2, c_2, d_0)$ | 6 | $(a_1, b_2, c_3, d_1)$ | 7 |
| $(a_2, b_2, c_2, d_1)$ | 7 | $(a_2, b_2, c_3, d_0)$ | 7 | $(a_2, b_2, c_3, d_1)$ | 8 |

Our algorithm is very generic and can accommodate any level of hierarchies. The same algorithm can be used to compare various thread broadcast implementations. We can also explore the opportunities to apply this algorithm in other fields like networking as well.

We are thankful to DARPA and BAE systems for continuous funding and support for the AESOP project. We are also thankful to Department of Computer Science and University of Maryland, College Park for providing us the resources to accomplish this task.

## References

1. AESOP : Adaptive Environment for Supercompiling with Optimized Parallelism. http://www.cs.umd.edu/projects/hpsl/chaos/LocalResources/funding.html
2. Yotov, K., Pingali, K. and Stodghill, P.: X-ray: a tool for automatic measurement of hardware parameters. Quantitative Evaluation of Systems, 2005. Second International Conference on the, 168–177, (2005)
3. Yotov, K., Pingali, K. and Stodghill, P.: Automatic measurement of memory hierarchy parameters. ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 181–192, (2005)
4. F., Broquedis, J., Clet-Ortega, S., Moreaud, N., Furmento, B., Goglin, G., Mercier, S., Thibaul and R. Namyst: hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 180–186 (2010)
5. Duchateau, A., Sidelnik, A., Garzarn, M. and Padua, D.: P-Ray: A Software Suite for Multi-core Architecture Characterization. Lecture Notes in Computer Science, 187–201 (2008)

6. Gonzalez-Dominguez, J., Taboada, G.L., Fraguela, B.B., Martin, M.J. and Tourio, J.: Servet: A benchmark suite for autotuning on multicore clusters. Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 1–9 (2010)
7. Nanjegowda, R. and Hernandez, O.: Scalability Evaluation of Barrier Algorithms for OpenMP. Lecture Notes in Computer Science, 42–52 (2009)
8. Berger, S.A and Stamatakis, A.: Assessment of barrier implementations for fine-grain parallel regions on current multi-core architectures . Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on , 1–8 (2010)
9. Mellor-Crummey, John M. and Scott, Michael L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst., 21–65 (1991)