Thread and Memory Placement on NUMA Systems
Asymmetry Matters

BAPTISTE LEPERS, VIVIEN QUÉMA, AND ALEXANDRA FEDOROVA

Baptiste Lepers is a postdoc at Simon Fraser University. His research topics include performance profiling, optimizations for NUMA systems, and multicore programming. He likes to spend his weekends in the mountains, hiking and biking. baptiste.lepers@gmail.com

Vivien Quéma is a Professor at Grenoble INP (ENSIMAG). His research is about understanding, designing, and building (distributed) systems. He works on Byzantine fault tolerance, multicore systems, and P2P systems. vivien.quema@grenoble-inp.fr

Alexandra Fedorova is an Associate Professor at the University of British Columbia. Her research focuses on building systems software that facilitates synergy between applications and hardware. In her spare time, she consults for MongoDB. sasha@ece.ubc.ca

Industry uses NUMA multicore machines for its servers. On NUMA machines, the conventional wisdom is to place threads close to the memory they access, and to collocate the threads that share data on the same CPU nodes. However, this is often not optimal. Indeed, modern NUMA machines have asymmetric interconnect links between CPU nodes, which can strongly affect performance, with best placement outperforming worst placement on nodes by a factor of almost two. We present the *AsymSched* algorithm, which uses CPU performance counters to measure performance and dynamically migrate threads and memory to achieve the best placement.

Modern Computers Are Asymmetric

Modern multicore machines are structured as several CPU/memory nodes connected via an interconnect. These architectures are usually characterized by non-uniform memory access times (NUMA), meaning that the latency of data access depends on *where* (which CPU-cache or memory node) the data is located. For this reason, the placement of threads and memory plays a crucial role in performance. To that end, both researchers and practitioners designed a variety of NUMA-aware thread and memory placement algorithms [8, 7, 5, 13, 14, 4]. Their insight is to place threads close to their memory, to spread the memory pages across the system to avoid the overload on memory controllers and interconnect links, to collocate data-sharing threads on the same node while avoiding memory controller contention, and to segregate threads competing for cache and memory bandwidth on different nodes. These algorithms assume that the interconnect between nodes is symmetric: given any pair of nodes connected via a direct link, the links have the same bandwidth and the same latency. *On modern NUMA systems this is not the case.*

Figure 1 depicts an AMD Bulldozer NUMA machine with eight nodes, each hosting eight cores. Interconnect links exhibit many disparities:

1. Links have different bandwidths: some have 16-bit width, some have 8-bit width.
2. Some links can send data faster in one direction than in the other (i.e., one side sends data at 3/4 the speed of a 16-bit link, while the other side can only send data at the speed of an 8-bit link). We call these links 16/8-bit links.
3. Links are shared differently. For instance, the link between nodes 4 and 3 is only used by these two nodes, while the link between nodes 2 and 3 is shared by nodes 0, 1, 2, 3, 6, and 7.
4. Some links are unidirectional. For instance, node 7 sends requests directly to node 3, but node 3 routes its answers via node 2. This creates an asymmetry in read/write bandwidth: node 7 can write at 4 GB/s to node 3, but can only read at 2 GB/s.

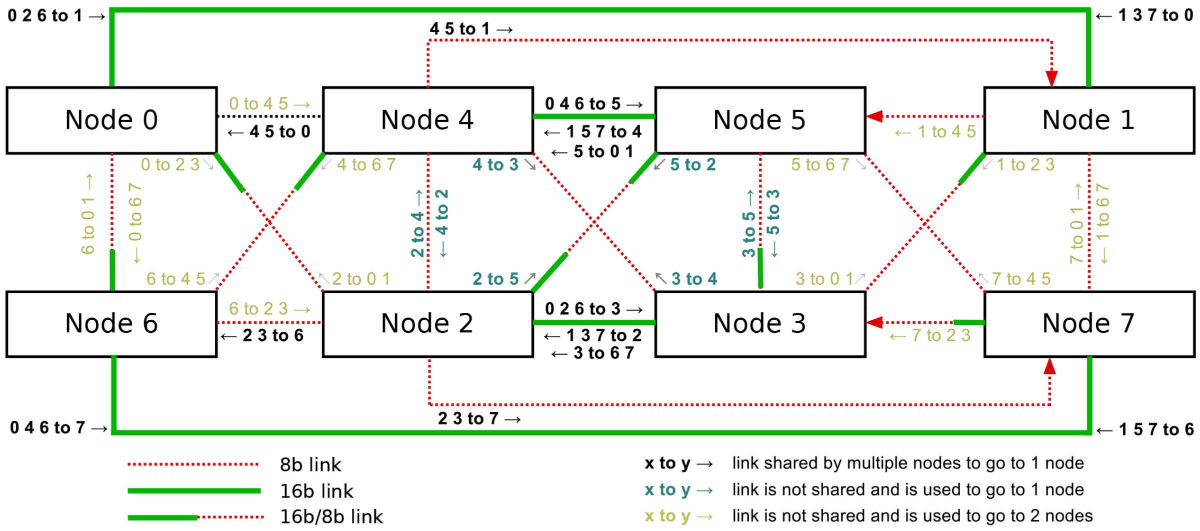


Figure 1: Modern NUMA systems, with eight nodes. The width of links varies; some paths are unidirectional (e.g., between 7 and 3), and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node—not represented in the picture), and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

Impact of Asymmetry on Performance

The asymmetry of interconnect links has dramatic and at times surprising effects on performance. Figure 2 shows the performance of 20 different applications on the 64-core machine shown in Figure 1. Each application runs with 24 threads, so it needs three nodes to run on. We vary *which* three nodes are assigned to the application and hence the *connectivity* between the nodes. The relative placement of threads and memory on those nodes is *identical* in all configurations. The only difference is *how the chosen nodes are connected*. The figure shows the performance on the best-performing and the worst-performing subset of nodes for that application compared to the average (obtained by measuring the performance on all 336 unique subsets of nodes and computing the mean). We make several observations. First, the performance on the best subset is up to 88% faster than the average, and the performance on the worst subset is up to 44% slower. Second, the maximum performance difference between the best and the worst subsets is 237% (for FaceRec). Finally, the mean difference between the best and worst subsets is 40% and the median 14%.

We measured that the memory accesses performed by FaceRec are approximately 600 cycles faster when running on the best subset of nodes relative to the average, and 1400 cycles faster relative to the worst. The latency differences are tightly correlated with the performance difference between configurations.

To further understand the cause of very high latencies on "bad" configurations, we analyzed streamcluster, an application from

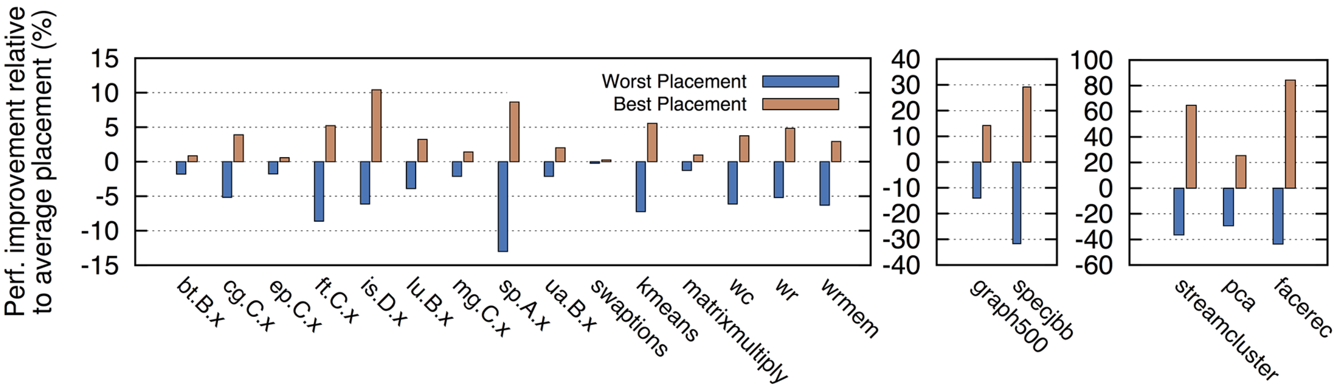
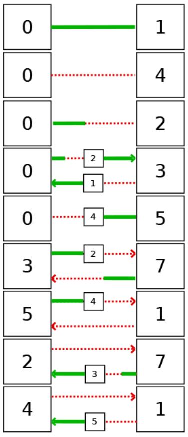


Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, SPECjbb, streamcluster, PCA, and FaceRec are highly affected by the choice of nodes and are shown separately with a different y-axis range.



Master thread node	Execution time (s)	Diff with 0-1 (%)	Latency of memory accesses (cycles) (compared to 0-1(%))	% accesses via 2-hop links	Bandwidth to the "master" node (MB/s)
-	148	0%	750	0	5598
-	228	56%	1169 (56%)	0	2999
0	228	56%	1179 (57%)	0	2973
2	168	15%	855 (14%)	0	4329
0	340	133%	1527 (104%)	98	1915
3	**185**	27%	1040 (39%)	98	3741
0	340	133%	1601 (113%)	98	1903
5	**228**	56%	1206 (61%)	98	2884
3	185	27%	1020 (36%)	0	3748
7	338	132%	1614 (115%)	98	1928
1	338	132%	1612 (115%)	98	1891
5	230	58%	1200 (60%)	0	2880
2	**167**	15%	867 (16%)	98	3748
7	225	54%	1220 (63%)	0	3014
4	230	58%	1205 (60%)	0	2959
1	**226**	55%	1203 (60%)	98	2880

Table 1: Performance of streamcluster executing with 16 threads on two nodes on machine A. The performance depends on the connectivity between the nodes on which streamcluster is executing and on the node on which the master thread is executing. Numbers in bold indicate two-hop configurations that are as fast or faster than some one-hop configurations.

the PARSEC [11] benchmark suite, which is among the most affected by the placement of its threads and memory. We ran streamcluster with 16 threads on two nodes. Table 1 presents the salient metrics for each possible two-node subset. Depending on which two nodes we chose, we observe large (up to 133%) disparities in performance. The data in Table 1 leads to several crucial observations:

- Performance is correlated with the latency of memory accesses.
- Surprisingly, the latency of memory accesses is not correlated with the number of hops between the nodes: some two-hop configurations (shown in bold) are faster than one-hop configurations.
- The latency of memory accesses is actually correlated with the *bandwidth* between the nodes. Note that this makes sense: the difference between one-hop vs. two-hop latency is only 80 cycles when the interconnect is nearly idle. So a higher number of hops alone cannot explain the latency differences of thousands of cycles.

As a summary, we can say that **bandwidth between the nodes matters more than the distance between them**.

Computers Are Increasingly Asymmetric

Asymmetric interconnect is not a new phenomenon. Nevertheless, its effects on performance are increasing as machines are built with more nodes and cores. We measured the performance of streamcluster on four different asymmetric machines: two recent machines with 64 and 48 cores, respectively, and eight nodes (Machines A and B, Figure 1), and two older machines with 24 and 16 cores, respectively, and four nodes (Machines C and D, not depicted). All of these machines use AMD Opteron processors. Machines A and B have highly asymmetric interconnect. Machines C and D have a less pronounced asymmetry. Machine C has full connectivity, but two of the links are slower than the rest. Machine D has links with equal bandwidth, but two nodes do not have a link between them.

Table 2 shows the performance of streamcluster with 16 threads on the best-performing and the worst-performing set of nodes on each machine. The performance difference between the best and worst configurations increases with the number of cores in the machine: from 3% for the 16-core machine to 133% for the 64-core machine. We explain this as follows:

1. On the 16-core Machine D, the only difference between configurations is the longer wire delay between the nodes that are not connected via a direct link. This delay is not significant compared to the extra latency induced by bandwidth contention on the interconnect.
2. The CPUs on 24-core Machine C have a low frequency compared to the other machines. As a result, the impact of longer memory latency is not as pronounced. More importantly, the network on this machine is still a fully connected mesh, so there is less asymmetry than on Machines A and B.
3. The 48- and 64-core Machines B and A offer a wider range of bandwidth configurations, which increases the difference between the best and the worst placements. The 64-core machine is more affected than the 48-core machine because it has more cores per node, which increases the effects of bandwidth contention.

Intel machines are currently built using symmetric interconnect links, but we believe that, as the number of nodes in systems increases, this will no longer remain true in the future.

Machine	Best Time	Worst Time	Difference
A (64 cores)	148s	340s	133%
B (48 cores)	149s	277s	85%
C (24 cores)	171s	229s	33%
D (16 cores)	255s	262s	3%

Table 2: Performance of streamcluster executing on two nodes on machine A, B, C, and D. The performance of streamcluster depends on the placement of its threads. The impact of thread placement is more important on recent machines (A and B) than on older ones (C and D).

Handling Asymmetry: The Challenges

To take into account interconnect asymmetry, the operating system should choose a "good" subset of nodes for each application. More precisely, the operating system should try, for each application, to **place threads and memory pages on a well-connected subset of nodes**. When an application executes on only two nodes on a machine similar to the one used in Table 1, the placement on the nodes connected with the widest (16-bit) link is always the best because it maximizes the bandwidth and minimizes the latency between the nodes. However, when an application needs more than two nodes to run, no configuration exists with 16-bit links between every pair of nodes, so the operating system must decide which nodes to pick. Besides, when there is more than one application running, the operating system needs to decide how to allocate the nodes among multiple applications. Designing such a thread and memory placement algorithm raises several challenges that we list below.

Nodes	% Perf. Relative to Best Subset	
	Streamcluster	SPECjbb
0, 1, 3, 4, and 7	-64%	0% (best)
2, 3, 4, 5, and 6	0% (best)	-9.4%

Table 3: Performance of streamcluster and SPECjbb on two different set of nodes on machine A, relative to the best set of nodes for the respective application

Efficient online measurement of communication patterns is challenging: The algorithm must measure the volume of CPU-to-CPU and CPU-to-memory communication for different threads in order to determine the best placement. This measurement process must be very efficient, because it must be done continuously in order to adapt to phase changes.

Changing the placement of threads and memory may incur high overhead: Frequent migration of threads may be costly, because of the associated CPU overhead, but most importantly because cache affinity is not preserved. Moreover, when threads are migrated to "better" nodes, it might be necessary to migrate their memory in order to avoid the overhead of remote accesses and overloaded memory controllers. Migrating large amounts of memory can be extremely costly. Thus, thread migration must be done in a way that minimizes memory migration.

Accommodating multiple applications simultaneously is challenging: Applications have different communication patterns and are thus differently impacted by the connectivity between the nodes they run on. As an illustration, Table 3 presents the performance of streamcluster and SPECjbb executing on two different sets of five nodes (the best set of nodes for the two applications, respectively). The two applications behave differently on these two sets of nodes: streamcluster is 64% slower on the best set of nodes for SPECjbb than on its own best set. The algorithm must, therefore, determine the best set of nodes for every application. Furthermore, it cannot always place each application on its best set of nodes, because applications may have conflicting preferences.

Selecting the best placement is combinatorially difficult: The number of possible application placements on an eight-node machine is very large (e.g., 5040 possible configurations for four applications executing on two nodes). So, (1) it is not possible to try all configurations *online* by migrating threads and then choosing the best configurations, and (2) doing even the simplest computation involving "all possible placements" can still add a significant overhead to a placement algorithm.

The AsymSched Algorithm

We designed *AsymSched* [9], a thread and memory placement algorithm that takes into account the bandwidth asymmetry of asymmetric NUMA systems. *AsymSched*'s goal is to maximize the bandwidth for *CPU-to-CPU communication,* which occurs between threads that exchange data, and *CPU-to-memory communication,* which occurs between a CPU and a memory node upon a cache miss. To that end, *AsymSched* places threads that perform extensive communication on relatively well-connected nodes, and places the frequently accessed memory pages such that the data requests are either local or travel across high-bandwidth paths.

AsymSched is implemented as a user-level process and interacts with the kernel and the hardware using system calls and `/proc` file system, but could also be easily integrated with the kernel scheduler if needed.

AsymSched relies on three main techniques to manage threads and memory:

1. **Thread migration**: changing the node where a thread is running
2. **Full memory migration**: migrating all pages of an application from one node to another
3. **Dynamic memory migration**: migrating only the pages that an application actively accesses as done in [7]

The operating principle of *AsymSched* is the following: *AsymSched* continuously gathers hardware counter values on the number of memory requests. Every second, *AsymSched* takes a thread placement decision. Roughly speaking, it groups threads of the same application that share data in virtual weighted *clusters*. The weight of a cluster represents the intensity of memory accesses performed between threads belonging to the cluster. Then *AsymSched* computes possible *placements* for all the clusters. A placement is an array mapping clusters to nodes. For each placement, *AsymSched* computes the maximum bandwidth that each cluster would receive if it were put in this placement. *AsymSched* selects the placement, ensuring that clusters with the highest weights will be scheduled on the nodes with the best connectivity. Finally, *AsymSched* estimates the overhead of memory migration induced by the new placement. If the overhead is deemed too high, the new placement will not be applied. Otherwise, *AsymSched* performs thread and memory migration to apply the new placement.

AsymSched implements two main optimizations. The first optimization allows **fast memory migrations**. When *AsymSched* performs full memory migration, all the pages located on one node are migrated to another node. The applications we tested have large working sets (up to 15 GB per node), and migrating pages is costly. Migrating 10 GB of data using the standard `migrate_pages` system call takes 51 seconds on average, making the migration of large applications impractical.

We therefore designed a new system call for memory migration. This system call performs memory migration without locks in most cases, and exploits the parallelism available on multicore machines. Using our system call, migrating memory between two nodes is on average 17x faster than using the default Linux system call and is only limited by the bandwidth available on interconnect links. Unlike the Linux system call, our system call can migrate memory from multiple nodes simultaneously. So if we are migrating the memory simultaneously between two pairs of nodes that do not use the same interconnect path, our system call will run about 34x faster.

The second optimization **avoids evaluating all possible placements.** It is based on two observations:

1. A lot of thread placement configurations are “obviously” bad. For instance, when a communication-intensive application uses two nodes, we only consider configurations with nodes connected with a 16-bit link.
2. Several configurations are equivalent (e.g., in the machine depicted in Figure 1, the bandwidth between nodes 0 and 1 and between nodes 2 and 3 is the same). To avoid estimating the bandwidth of *all* placements, we create a hash for each placement. The hash is computed so that equivalent configurations have the same hash.

Using simple dynamic programming techniques, we only perform computations on non-equivalent configurations. Our optimization allows skipping between 67% and 99% of computations in all tested configurations with clusters of two, three, or five nodes (e.g., with four clusters of two nodes, we only evaluate 20 configurations out of 5040).

AsymSched Assessment

We evaluated the performance achieved when using *AsymSched* on Machine A. The latter is equipped with four AMD Opteron 6272 processors, each with two NUMA nodes and eight cores per node (64 cores in total). The machine has 256 GB of RAM, uses HyperTransport 3.0, and runs Linux 3.9. We used several benchmark suites: the NAS Parallel Benchmarks suite [3], which is composed of numeric kernels; MapReduce benchmarks from Metis [10]; parallel applications from PARSEC [11]; Graph500 [1], a graph processing application with a problem size of 21; FaceRec from the ALPBench benchmark suite [6]; and SPECjbb [2] running on OpenJDK7.

Our goal was to evaluate the impact of asymmetry-aware thread placement *in isolation* from other effects, such as those stemming purely from collocating threads that share data on the same node. Performance benefits of sharing-aware thread clustering are well known [13]. *AsymSched* clusters threads that share data; the Linux thread scheduler, however, does not. We experimentally observed that Linux performed worse than clustered configurations. For instance, when Graph500 and SPECjbb are scheduled simultaneously, both run 23% slower on Linux than on an average clustered placement.

Since comparing Linux to *AsymSched* would not be meaningful because of that, we instead compare *AsymSched* to the best and the worst static placements of data-sharing thread clusters. When running *AsymSched*, thread clusters are initially placed on a randomly chosen set of nodes. We also compare the average performance achieved under all static placements that are unique in terms of connectivity. We obtain all unique static placements with respect to connectivity by examining the topology of the machine. There are 336 placements for single-application scenarios and 560 placements for multi-application scenarios.

Further, we want to isolate the effects of thread placement with *AsymSched* from the effects of dynamic memory migration. To that end, we compare *AsymSched* to the subset of our algorithm that performs the dynamic placement of memory only, turning off the parts performing thread placement.

The results are presented in Figure 3. *AsymSched* always performs close to the best static thread placement. In a few cases where it does not, the difference is not statistically significant. For applications that produce the highest degree of contention on the interconnect links (streamcluster, PCA, and FaceRec),

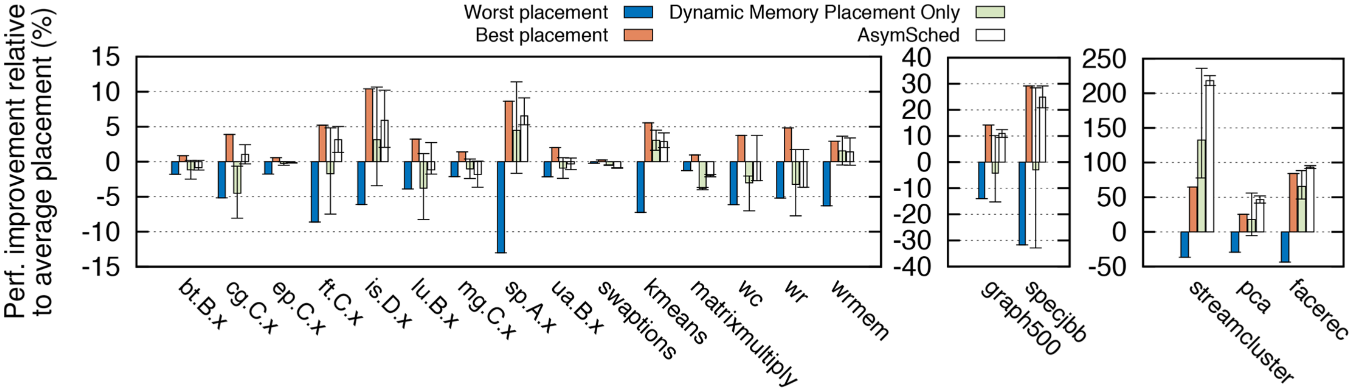


Figure 3: Performance difference between the best and worst static thread placement, dynamic memory placement, and *AsymSched* relative to the average thread placement on Machine A. Applications run with 24 threads on three nodes.

AsymSched achieves much better performance than the best thread placement, because the dynamic memory migration component balances memory accesses across nodes, thus reducing contention on interconnect links and memory controllers.

We also observe that dynamic memory migration without the migration of threads is not sufficient to achieve the best performance. More precisely, dynamic memory migration alone often achieves performance close to average. Moreover, it produces a high standard deviation for many benchmarks: the minimum and maximum performance often being the same as that of the best and worst static thread placement. For instance, on SPECjbb, the difference between the minimum and maximum performance with dynamic memory migration alone is 91%.

In contrast, *AsymSched* produces a very low standard deviation for most benchmarks. Two exceptions are is.D and SPECjbb. This is because in both cases, *AsymSched* migrates a large amount of memory. Both applications become memory intensive after an initialization phase, and *AsymSched* starts migrating memory only after the entire working set has been allocated. For instance, in the case of is.D, *AsymSched* migrates between 0 GB and 20 GB, depending on the initial placement of threads.

Conclusion

Asymmetry of the interconnect in modern NUMA systems drastically impacts performance. We found that the performance is more affected by the bandwidth between nodes than by the distance between them. We developed *AsymSched*, a new thread and memory placement algorithm that maximizes the bandwidth for communicating threads.

As the number of nodes in NUMA machines increases, the interconnect is less likely to remain symmetric. We believe that the clustering and placement techniques used in *AsymSched* will scale and be well adapted to these machines. Indeed, with very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future machines, as machines are usually made of multiple identical cores/sockets (e.g., our 64-core machine has four identical sockets). On machines that offer a wider diversity of thread placements, a possibility will be to use statistical approaches, such as that of Radojković et al. [12] to find good thread placements with a bounded overhead.

References

[1] Graph500 reference implementation: http://www.graph500.org/referencecode.

[2] SPECjbb2005 industry-standard server-side Java benchmark (j2se 5.0): http://www.spec.org/jbb2005/.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-Aware Contention Management on Multicore Systems," in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*, 2011.

[5] Timothy Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors," in *Proceedings of the USENIX Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993.

[6] CSU Face Identification Evaluation System: http://www.cs.colostate.edu/evalfacerec/index10.php.

[7] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2013, pp. 381–394.

[8] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma, "MemProf: A Memory Profiler for NUMA Multicore Systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*, 2012.

[9] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*, Santa Clara, CA, July 2015, pp. 277–289.

[10] Metis MapReduce Library: http://pdos.csail.mit.edu/metis/.

[11] PARSEC Benchmark Suite: http://parsec.cs.princeton.edu/.

[12] Petar Radojković, Vladimir Cakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero, "Optimal Task Assignment in Multithreaded Processors: A Statistical Approach," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, March 2012, pp. 235–248.

[13] David Tam, Reza Azimi, and Michael Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys, 2007), pp. 47–58.*

[14] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's Warehouse Scale Computers: The NUMA Experience," in *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13)*, Feb. 2013, pp. 188–197.