Analyzing and Improving Memory Access Patterns of Large Irregular Applications on NUMA Machines

Artur Mariano*, Matthias Diener†, Christian Bischof*, Philippe O. A. Navaux†
*Institute for Scientific Computing, Technische Universität Darmstadt, Germany
Email: {artur.mariano,christian.bischof}@sc.tu-darmstadt.de
†Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
Email: {mdiener,navaux}@inf.ufrgs.br

***Abstract*—Improving the memory access behavior of parallel applications is one of the most important challenges in high-performance computing. Non-Uniform Memory Access (NUMA) architectures pose particular challenges in this context: they contain multiple memory controllers and the selection of a controller to serve a page request influences the overall locality and balance of memory accesses, which in turn affect performance. In this paper, we analyze and improve the memory access pattern and overall memory usage of large-scale irregular applications on NUMA machines. We selected HashSieve, a very important algorithm in the context of lattice-based cryptography, as a representative example, due to (1) its extremely irregular memory pattern, (2) large memory requirements and (3) unsuitability to other computer architectures, such as GPUs. We optimize HashSieve with a variety of techniques, focusing both on the algorithm itself as well as the mapping of memory pages to NUMA nodes, achieving a speedup of over 2x.**

***Keywords*—Memory accesses, NUMA, irregular applications**

I. Introduction

Modern parallel shared-memory architectures are characterized by a Non-Uniform Memory Access (NUMA) behavior. Such systems contain multiple memory controllers, dividing the available main memory into several NUMA nodes [1]. Each node can access its local memory directly, but has to access memory from remote nodes through an interconnection such as Intel's QPI. Remote accesses incur a higher latency and energy consumption than local ones. For this reason, taking NUMA characteristics into account can substantially improve the efficiency of parallel applications. Two basic strategies to optimize the memory access behavior can be performed: First, increasing the *locality* of memory accesses, by maximizing the number of accesses to local NUMA nodes and minimizing the accesses to remote ones [2]. Second, by *balancing* the number of memory accesses among all memory controllers [3], [4].

HashSieve is a recently proposed algorithm for the Shortest Vector Problem (SVP) [5], a key problem in lattice-based cryptography. HashSieve serves as a representative example of an application that can substantially benefit from memory optimizations, due to multiple reasons. First, it has large memory footprints (experiments were reported requiring over 100 GB of memory and, according to a memory model, some inputs would require about 2.5 TB of memory [6]). Second, it is highly irregular (previous assessments suggested that L2 and L3 cache misses are very close to 100% for large dimensions). Third, the algorithm is not suited for computer architectures other than multi-core shared-memory CPUs, such as GPUs. Since HashSieve can be used as a model of other parallel, irregular applications that are hard to optimize, optimizing HashSieve might also suggest general optimizations for other applications. Thus, understanding and optimizing the performance of HashSieve on NUMA systems is of high importance.

In this paper, we perform an in-depth optimization of HashSieve, focusing on improving its memory access pattern as well as the mapping of memory pages to NUMA nodes. The memory access behavior is improved by source code changes, such as re-ordering operations to leverage cache locality and prefetching data that the compiler could not prefetch. Due to its irregular access pattern, HashSieve is not very suitable for data mapping mechanisms that focus on increasing memory access locality, but can benefit from improving the balance of memory accesses from NUMA nodes. In an evaluation with a NUMA machine, we obtained speedups of over 2x, with respect to the baseline implementation. These findings indicate that HashSieve has considerable room for improvement if the underlying architecture is known.

II. Related Work

Several tools to analyze and improve the memory access behavior of applications on NUMA architectures have been propose, such as MemProf [7] and Numalize [8]. To this day, only a limited number of papers proposed and discussed memory access improvements for specific parallel applications. Majo et al. [9] analyze three benchmarks from the PARSEC benchmark suite [10] and propose specific improvements for them. However, compared to large scientific applications, the used benchmarks have low memory footprints with relatively regular access patterns.

Traditional data mapping mechanisms include interleaving, where pages with concurrent addresses are mapped to NUMA nodes in a round-robin way, and first-touch, where pages are mapped to the first NUMA node that performs an access to them [11]. The first-touch policy is the default policy in current operating systems. The current research trend in operating systems is to perform an online characterization of the memory access behavior, and performing page migrations to improve the behavior. Both locality of memory accesses [2] and their balance can be taken into account [3], [12]. To our knowledge, this is the first paper to address this problem for a real-world, large and irregular scientific application, with code improvements and better mapping policies.

III. The HashSieve Algorithm

This section describes the HashSieve algorithm, its parallel implementation and gives an overview of its memory access challenges in NUMA architectures.

A. Description

The core idea behind sieving algorithms is the reduction of multiple vectors against one another [13]. Reducing one vector **v** against (a multiple of) another vector **w** means that **w** is either subtracted or added to **v**, in such a way that **v** becomes shorter (in regards to its Euclidean norm). The number of vectors necessary for convergence is not known upfront, at least for the majority of sieving algorithms.

HashSieve uses popular a method from nearest neighbor search, known as *locality-sensitive hashing*, to reduce vectors against one another. From a high-level perspective, this method organizes different vectors in hash tables according to the distance spanned between them (which is a strong indicator of the success of a reduction between two vectors). For example, the vectors that are close in space to a given vector **v** are stored in the same hash bucket of **v**, for a given hash table. As each bucket is essentially some part of the lattice, the algorithm uses many hash tables to avoid missing vectors that lie in the borders of those spaces. With this setup, the algorithm samples many vectors, as in every sieving algorithm, reducing them against the vectors with the same hash value, for every hash table. For a complete, mathematical description of this process, we refer the reader to [14]. For a computational description, we refer to [6].

HashSieve is shown in Algorithm 1. After the initialization, the algorithm repeats the following four-step process: (i) sample a random lattice vector **v** (or get one from the stack `S`); (ii) find nearby candidate vectors **w** in the `T` hash tables to reduce **v** with; (iii) use the reduced vector **v** to reduce other vectors **w** in the hash tables (and if such a vector **w** is reduced, move it onto the stack); and finally (iv) add **v** to the stack or hash tables. This process aims to build a large set of short, pairwise reduced lattice vectors until the shortest vector is found. After that, the size of the set does not increase any more, and collisions, which happen when vectors are reduced to the zero-vector, are generated instead. The algorithm terminates when a given number of collisions c is reached.

The number of hash tables is `T`, and the number of hash buckets in each table is $2^{\mathtt{K}}$, where `K` is the number of random hash functions of the hash family that is used. It was shown, in [14], that the asymptotic optimal choices of `K` and `T` are $0.2209n + o(n)$ and $2^{0.1290n+o(n)}$, respectively.

Algorithm 1: The HashSieve algorithm

```
1  Input: (Reduced) basis B, collision threshold c;
2  Initialize stack S ← {}, collisions cl ← 0
3  Initialize T empty hash tables H_1, ..., H_T

4  while cl < c do
5      Pop vector v from S or sample it if |S| = 0
6      while ∃w ∈ H_1[h_1(v)], ..., H_T[h_T(v)] : ||v ± w|| < ||v|| do
7          for each Hash table H_i, ..., H_T do
8              Obtain the set of candidates C = H_i[h_i(v)]
9              for each w ∈ C do
10                 Reduce v against w
11                 Reduce w against v
12                 if w has changed then
13                     Remove w from all T hash tables H_i
14                     if w == 0 then cl++
15                     else Add w to the stack S
16     if v == 0 then cl++
17     else Add v to all T hash tables H_i
```

B. Parallel Implementation

In this section, we briefly review the baseline implementation published in [6], written in C. Essentially, the application spawns a team of threads, each of which samples a vector **v**, and tries to reduce **v** against **w** and vice versa, where **w** is one of the candidate vectors of **v**. The set of candidate vectors of a given vector **v** comprises the vectors that have the same hash value (i.e. are stored in the same hash bucket) of **v**, for each and every hash table in the system. If **v** is reduced against any of the candidate vectors, the thread restarts the process (i.e. goes through all the hash tables, looking for candidates to reduce **v** against, most likely on different buckets). If **w** is reduced, it is removed from all the hash tables and moved onto the stack (which is private per thread). If **w** is reduced to the zero vector, a collision is counted instead.

Whenever a sample has gone through all the hash tables once without being reduced, the thread inserts it in every hash table and samples another vector. The implementation employs a light-weight probable lock-free mechanism to handle concurrent accesses to the hash tables: whenever a thread is to access a hash bucket, it sets a variable to 1, atomically, turning it to 0 when the visited bucket is no longer needed. The same system is used for regular vectors in the system, because as every hash table has one pointer to every vector in the system, the same vector might be accessed (and modified) from different hash buckets, by different threads (cf. Figure 1 in [6]). If a thread encounters one vector under use, the vector is simply disregarded, and the thread proceeds to the next iteration.

C. Data Structures

The main data structures of our implementation are the `T` hash tables employed by the algorithm. Each hash table consists of an array of $2^{\mathtt{K}}$ positions, which holds pointers to the buckets, which are in turn arrays that the algorithm extends (with `realloc()`) as needed. These arrays are used in order to improve cache locality within each bucket and avoid an excessive number of allocations when vectors are to be added to buckets [6]. As `K` grows exponentially with the input dimension, memory consumption is a crucial challenge in the algorithm. For example, in dimension $n = 80$, the algorithm employs $\mathtt{T} = 1278$ hash tables, each of which has with 131072 buckets. As each bucket sums up to 17 bytes plus the size of the pool ($20 \times 8 = 160$) [6], the algorithm spends over 27.61 GBs only on the allocation of the hash tables (disregarding structure padding).

IV. Optimizing HashSieve for NUMA Systems

In this section, we introduce a number of optimizations that can be applied to the HashSieve algorithm. In Section VI, we show the impact of these optimizations on the overall performance and memory usage of the algorithm. These optimizations have been integrated in the baseline implementation, a modified version of the HashSieve implementation presented in [6], which scales linearly with the number of cores on shared-memory CPUs.

A. Improving Locality of Reference

Both temporal and spatial locality are considerably low in HashSieve, due to its inherent work-flow. First, for one

particular sample, the algorithm consists in visiting several hash buckets of different hash tables, which generates high cache miss ratios since hash tables are quite big even for moderate lattice dimensions, thereby not fitting in cache. Second, only a small part of the hash buckets is accessed, since the reduction process starts all over again after the first valid reduction of the sample **v**. Third, except for the sample **v**, which might be accessed several times during its reduction process, each candidate vector **w** is only accessed once.

Previous experiments with the baseline implementation showed high L2 and L3 cache miss ratios even for low dimensional input lattices, and it was conjectured that these ratios are close to 100% for lattices in dimension 88 and onwards. An implementation detail that contributes largely to these figures is that vectors are always accessed via pointers, stored in each hash table. This means that accessing a candidate vector incurs multiple cache misses (access the hash bucket where the vector resides at, access the pool of that hash bucket, via pointer dereference, and dereference the pointer to each candidate vector).

In order to improve locality, we changed the work-flow of the algorithm slightly. Instead of jumping to another hash bucket after the first successful reduction of **v**, we keep the original vector **v** in an auxiliary variable and we test every candidate vector **w** of the bucket against **v**. During this test, many candidate vectors can be successfully reduced against **v**. Those are moved onto the stack. As for **v**, we only commit its minimal version, i.e. the successful reduction of **v** against one candidate **w** that generated the shortest among all its reductions. This way, we leverage spatial locality since we visit the whole bucket pool, which is likely to be in cache.

It is important to note that this modification did not change the solution of the implementation, since any order for vector reduction in sieving algorithms is possible.

B. Lowering Memory Access Latency

Given that HashSieve is memory-bound, memory access latency contributes largely to the overall execution time of the algorithm.

Adding or removing one vector from all the hash tables serve as good examples of operations incurring high memory latency overhead. These operations happen at different points of the application: (1) when it is known that a candidate vector **w** will change, it must be removed from all the `T` hash tables, and (2) at the end of the reduction of a sample or a stack vector **v**, the vector is inserted in all the `T` hash tables. The removal of **w** is done as follows:

```
for(int t=0; t<T; t++){
  hash_value = hash(w,t);
  bkt_remove(&HashTables[t][hash_value], w);
}
```

To lower memory latency, we apply memory prefetching, with hand-inserted code directives (we use compiler intrinsic operations - present both in GCC and Intel's ICC). The idea is that, as the algorithm accesses different hash buckets of different hash tables in subsequent iterations, which requires different memory locations in each iteration, we would benefit from requesting memory upfront, in such a way that the memory request for the subsequent operation is fulfilled beforehand. This way, the memory request of the memory using in iteration $k+1$ is overlapped with the operation in iteration k, thus reducing the impact of the memory latency the algorithm incurs. Although compilers are able to realize some prefetching throughout the application, hand-inserted prefetching directives are necessary in this case, because the compiler does not know the hash value of vector **w** in the subsequent iteration (since the function "hash" needs to be executed), and cannot therefore prefetch data. The resultant code is as follows:

```
long hash_values[T];
for(int t=0; t<T; t++){
  hash_values[t] = hash(w,t);
}
for(int t=0; t<T-1; t++){
  //Prefetch HashTables[t+1][hash_values[t+1]
  bkt_remove(&HashTables[t][hash_values[t]], w);
}
bkt_remove(&HashTables[T][hash_values[T]], w);
```

The last removal is done outside of the loop because, due to prefetching, the main loop spans from 0 to T-1.

Another improvement we have conducted was the integration of a memory pool for the vectors that are sampled at every iteration. Instead of requiring memory allocation for every vector that is sampled, we now store the vector in a pool (private per thread), whose memory is allocated at launch time. This saves concurrent memory allocations.

V. Analysis of Memory Access Behavior

This section provides an in-depth analysis of the memory access behavior of HashSieve. The goal of this analysis is to shed some light on the main problems of the current memory access pattern of the algorithm, and to investigate how they can be resolved or, when not possible, mitigated. This analysis is a crucial tool to determine the suitability of the application for different data mapping policies. We begin with a brief overview of the fundamentals of memory access analysis, followed by a description of the methodology and a presentation of the findings of the analysis.

A. Memory Access Concepts on NUMA Machines

Three concepts are relevant for the description of the memory access behavior in NUMA architectures: *communication* between threads, memory access *locality* and memory access *balance*. The placement of threads in the hardware hierarchy is influenced by communication, whereas the mapping of memory pages to NUMA nodes is influenced by locality and balance. This section discusses these concepts briefly, further information is presented in [2], [8].

1) Communication: In shared memory systems, threads communicate when they access shared memory areas. The pattern of the communication affects the efficiency of cache memories. By placing threads that communicate more on shared caches, fewer cache misses and less traffic on interconnections are expected. For this placement, it is necessary to have groups of threads that communicate more within the group than to threads outside the group. Furthermore, the improvements from data mapping can be increased by placing groups of threads that communicate on the same NUMA node, as the accesses from all threads on the same NUMA node are local. The communication behavior is usually represented as a communication matrix, where matrix axes show the thread

IDs and the matrix cells contain the amount of communication between all pairs of threads.

2) Locality: Memory pages that are accessed mostly from a single NUMA node have high *exclusivity* and are suitable for *locality-based* mapping policies. In such a policy, mapping a page to the NUMA node that performs the most accesses to it reduces the overall memory access latency, as accesses to local nodes have a lower latency than accesses to remote ones [1]. For a page that is shared between NUMA nodes, that is, where several nodes access it in a similar way, a locality-based policy has a lower impact on the overall memory access performance. The exclusivity is calculated by dividing the highest number of memory accesses from a single NUMA node by the total number of accesses from all nodes. Its value is expressed as a percentage, with a maximum of 100% and a minimum of $1/N$, where N is the number of NUMA nodes in the system. By performing this procedure for all pages that an application uses, we can calculate the application exclusivity as the average exclusivity of all pages, weighted with the number of memory accesses to each page.

3) Balance: The balance of memory accesses is, together with locality, a crucial parameter when performing data mapping [3], [4]. The goal is to balance the load on the memory controller of each NUMA node, such that each controller handles a similar number of memory accesses. This policy is appropriate when there is an imbalance of the load on the memory controllers. This can happen if many pages are allocated on a single NUMA node, for example due to the default first-touch policy, which places pages on the NUMA node that accessed them first. If a large amount of memory is first accessed from a single thread, this can lead to a severe imbalance of memory accesses, which can be corrected through a *balance-based* mapping policy.

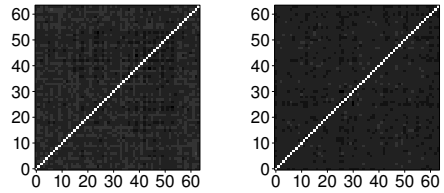
B. Characterization Methodology

The memory access characterization is performed with a custom memory tracer [8] built with the Pin Dynamic Binary Instrumentation (DBI) tool. We execute HashSieve with 64 threads and record all memory accesses at all threads on the page granularity. We then analyze the memory access behavior based on the trace. Since our main evaluation system contains 4 NUMA nodes, we also perform this analysis considering 4 nodes and assign threads that are neighbors to the same NUMA node. For example, thread IDs 0–15 are assigned to the first node. For the sake of discussion, we assume that pages are mapped with a first-touch policy in the discussion.

C. Application Evaluation

We begin the discussion with the communication behavior, followed by the analysis of the locality and balance of the memory accesses of both HashSieve versions.

1) Communication Behavior: The communication matrices of HashSieve are shown in Figure 1. In the matrices, darker cells illustrate larger amounts of communication between threads. The baseline version has very similar amounts of communication between all threads, indicating an unstructured behavior. As both matrices are normalized to the same value, the slightly darker pattern of the optimized version indicates a small increase in the amount of communication. Although difficult to see in the figure, the optimized version also has a more structured communication behavior, with more communication between neighboring threads. For the baseline version, we expect only small gains from a thread mapping policy, as no thread mapping can optimize the overall communication. However, the optimized version is more suitable for thread mapping due to its slightly more structured behavior. This claim will be evaluated with a performance evaluation of an optimized thread mapping in Section VI.



(a) Baseline version. (b) Optimized version.

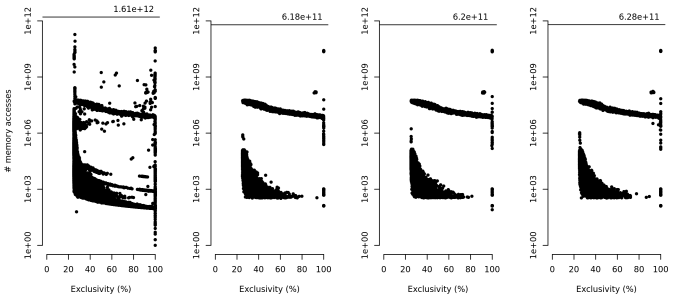
Fig. 1: Communication patterns of HashSieve. Axes show the thread IDs. Cells show the amount of communication between all pairs of threads. Darker cells indicate more communication.

2) Locality of Memory Accesses: To determine the locality of memory accesses of HashSieve, we calculate the exclusivity of the memory accesses to each page, as well as the average (weighted) exclusivity for the whole application. Higher exclusivity values indicate a higher suitability for locality-based data mapping. The results of the exclusivity analysis are shown in Figure 2. In the figure, each dot represents a page. THe horizontal axis shows the exclusivity, while the vertical axis shows the number of memory accesses to each page.

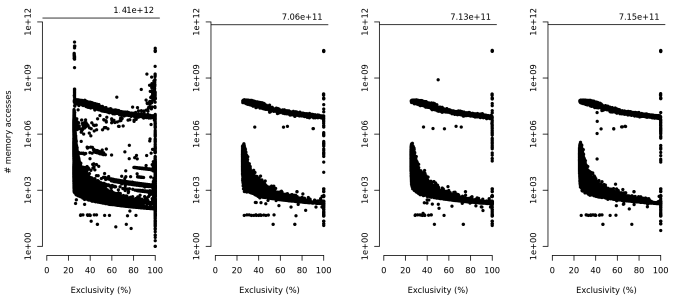
For the baseline version of HashSieve, the results show a lot of pages with a very low exclusivity (towards the minimum exclusivity of 25%). Most of these shared pages are allocated on the first NUMA node and tend to have more memory accesses than the pages with a higher exclusivity. Some pages have the maximum exclusivity of 100%. These pages correspond to memory areas private to, such as memory pools of each thread. In the optimized version of HashSieve, the average exclusivity increases slightly, from 62.2% to 64.3%. We can see that the overall access distribution is pushed towards pages with a higher exclusivity. We also note that the balance of the pages is improved with the optimized version, which will be discussed in the next section. Despite the increased exclusivity, the value is still low enough such that we expect only limited improvements from a locality-based policy.

3) Balance of Memory Accesses: To analyze memory access balance, we evaluate the how many pages are allocated on each NUMA node with a first-touch policy, as well as the number of memory accesses to the nodes. Higher imbalance indicates higher suitability for balance-based data mapping. The balance is also illustrated in Figure 2. The total number of memory accesses by the nodes is shown above each plot.

For the baseline version of HashSieve, Figure 2a shows that the majority of pages (about 92%) will be allocated on NUMA node 1 with a first-touch policy. These pages also receive a large number of memory accesses, leading to a severely imbalanced distribution of memory accesses between NUMA nodes (47% of memory accesses are handled by node 1). In the optimized version, the distribution of pages is considerably better, since both the number of pages decreased on node 1 and increased on the other nodes, as shown in Figure 2b. This leads to an improved distribution of memory accesses (41% are handled by node 1). Despite these improvements, the overall



(a) Baseline version.



(b) Optimized version.

Fig. 2: Memory access pattern of HashSieve. Each dot represents a page. Pages are distributed to NUMA nodes according to a first-touch policy. The vertical axis displays the number of memory accesses to each page in logarithmic scale. The horizontal line on top of the graph indicates the total number of memory accesses performed by each node.

behavior is still significantly imbalanced, suggesting that an improved mapping could result in further in further gains.

4) Summary: The results showed that both page exclusivity and memory access balance have been improved in the optimized version of HashSieve. However, exclusivity and balance remain quite low, since most scientific applications have exclusivities well above 80% [8]. Therefore, we do not except large improvements from locality-based data mapping policies. Balance-based policies might deliver better results, since the application incurs considerable memory access imbalance.

VI. Performance Evaluation

This section presents results of the performance evaluation of both HashSieve versions with different mapping policies.

A. Methodology of the Experiments

This section presents the system and the mapping policies.

1) Hardware Architecture: Experiments were performed on the *Xeon64* system, whose configuration is presented in Table I. The NUMA factor, defined as the latency between memory accesses to remote NUMA nodes divided by the latency to access local nodes, was measured with the Lmbench tool. HashSieve is executed with the number of threads the machine can execute in parallel (64).

2) Mapping Policies: We compare five mapping policies: OS, Compact, Interleave, NUMA Balancing, and kMAF. The three first policies perform no page migrations during execution. NUMA Balancing and kMAF are dynamic mechanisms that migrate pages between NUMA nodes at runtime.

For the *OS* mapping, we run an unmodified Linux kernel (version 3.8), and use its default first-touch mapping policy. The NUMA Balancing mechanism [12] is disabled in this configuration. The *Compact* thread mapping is a simple mechanism to improve memory affinity by placing threads with neighboring IDs (such as threads 0 and 1) close to each other in the memory hierarchy, such as on same cores. In the *Interleave* policy, pages are assigned to NUMA nodes according to their address. This policy ensures that memory accesses to consecutive addresses are distributed among the nodes. Interleave is available on Linux with the `numactl` tool.

We also use the *NUMA Balancing* mechanism [12] of the Linux kernel. The mechanism uses a sampling-based next-touch migration policy. NUMA Balancing gathers information about memory accesses from page faults and uses them to balance the memory pressure between NUMA nodes. To increase detection accuracy, the kernel periodically introduces extra page faults during execution. The *kMAF* mechanism [2] performs a locality-based thread and data mapping of parallel applications. Similar to NUMA Balancing, this mechanism uses page faults to determine memory accesses behavior, but keeps an access history to reduce unnecessary page migrations. Pages are migrated to nodes which access them the most.

B. Results

Figure 3 shows the execution time results of HashSieve. All shown values are averages of 3 executions. The results were very stable, with a difference between the maximum and minimum of less than 2% of the total execution time. The code optimizations we conducted resulted in a speedup between 40-70%, depending on the mapping policy. Compared to the OS, the Compact thread mapping only reduces execution time slightly for both versions of HashSieve (less than 7%). This is due to the unstructured memory access pattern and low exclusivity figures of the implementations, as reported in Section V. For the baseline version of HashSieve, which has very low exclusivity, kMAF is not able to improve performance, since both threads and data are moved around based on locality. As the mechanism detects a low page exclusivity, only few actions are taken. Both NUMA Balancing and the Interleave policies reduce execution time substantially (by 17% and 21%, respectively) for the baseline version, since as we showed in Figure 2, the implementations incur a high imbalance, which is resolved by these policies.

The optimized version of HashSieve is substantially faster, even with the OS mapping. Nevertheless, the optimized version can benefit further from better data mapping. kMAF and NUMA Balancing deliver better results than OS mapping. kMAF delivers about 17% speedup (which compares to ≈5% for the baseline version) whereas NUMA Balancing delivers

TABLE I: Overview of the *Xeon64* system.

Property	Value
NUMA	4 NUMA nodes, 1 processor/node, NUMA factor: 1.5
Processors	4×Intel Xeon X7550, 2.0 GHz, 8 cores, 2-way SMT
Caches	8×32 KB+32 KB L1, 8×256 KB L2, 18 MB L3
Memory	128 GB DDR3-1066, page size 4 KB

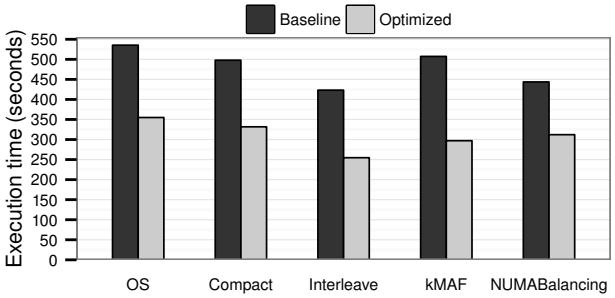


Fig. 3: Execution time results of HashSieve.

about 11%, in comparison to about 17% for the baseline version. The reason for this is that, as locality became better, kMAF is more effective, but since balancing became also better, there is less room for improvement for the NUMA Balancing policy. Similarly to the baseline version, the Interleave policy provides the highest improvements among all policies, reducing execution time by 28%. Overall, the performance was improved by a factor of 2.1×, from the baseline version running with the OS mapping to the optimized version running with the Interleave policy.

C. Generalization of Results

Based on the findings of the previous sections, we can infer a simple, yet generic, methodology to optimize the memory access behavior of large, real-world memory-bound applications on NUMA architectures. The first step to optimize the memory access behavior of applications is profiling. This should include the quantification and characterization of (i) the communication behavior, (ii) locality of memory accesses and (iii) balance of memory accesses, as we described in Section V. This data can be obtained with tools such as Numalize [8].

Communication can be structured or unstructured. The first means that there is a pattern of how threads communicate with one another. Threads that communicate a lot should be placed physically close to one another; this improves the efficiency of caches and memory access behavior on NUMA architectures. If communication is highly unstructured, such as in the baseline version of HashSieve, thread mapping usually leads to small improvements.

For data mapping policies, some guidelines can be followed. If exclusivity is high, the kMAF mechanism is likely to perform well. For instance, cumulative speedups could be achieved with code improvements to boost locality (e.g. with memory pools and better data access partitioning) and the simultaneous use of kMAF. As for balance, the more imbalanced applications are, the more they can benefit from the NUMA Balancing policy. For unbalanced applications where pages are likely to be moved around a lot with the NUMA Balancing policy, the Interleave policy might be a good alternative because there is no runtime overhead, while a good balance is achieved, as happened with HashSieve.

VII. Conclusions

Optimizing parallel applications for NUMA architectures is an important challenge. In this paper, we optimized HashSieve, which exhibits a very irregular work flow and very large memory footprint. We showed how to optimize HashSieve both via (1) an improved memory access pattern (for which we reordered memory accesses and performed data prefetching at specific points in the kernels, which the compiler could not prefetch) and (2) improved data mapping policies. Our experiments showed that the Interleave policy delivered the best results for the HashSieve implementation, as it can resolve the highly imbalanced memory access behavior without a runtime overhead. Altogether, we were able to speed up a parallel implementation of HashSieve by a factor of over 2×. Our findings are also relevant on a more general context, since many scientific applications exhibit an unstructured memory access pattern, which can usually not be improved in traditional ways. For such applications, developers could evaluate similar optimizations we conducted.

Acknowledgments

This work has been co-funded by the DFG as part of project P1 within the CRC 1119 CROSSING. The authors also gratefully acknowledge the support of CNPq and HP.

References

[1] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[2] M. Diener, E. H. M. Cruz, P. O. A. Navaux, A. Busse, and H.-U. Heiß, "kMAF: Automatic Kernel-Level Management of Thread and Data Affinity," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[3] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[4] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Locality vs . Balance: Exploring Data Mapping Policies on NUMA Systems," in *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2015.

[5] D. Micciancio and S. Goldwasser, "Shortest vector problem," in *Complexity of Lattice Problems*. Springer, 2002, pp. 69–90.

[6] A. Mariano, T. Laarhoven, and C. Bischof, "Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP," *International Conference on Parallel Processing (ICPP)*, 2015.

[7] R. Lachaize, B. Lepers, and V. Quéma, "MemProf: A Memory Profiler for NUMA Multicore Systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[8] M. Diener, E. H. M. Cruz, L. L. Pilla, F. Dupros, and P. O. A. Navaux, "Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping," *Performance Evaluation*, vol. 88-89, no. June, 2015.

[9] Z. Majo and T. R. Gross, "(Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[11] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott, "Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems," in *International Parallel Processing Symposium (IPPS)*, 1995.

[12] J. Corbet, "Toward better NUMA scheduling," 2012. [Online]. Available: http://lwn.net/Articles/486858/

[13] M. Ajtai, R. Kumar, and D. Sivakumar, "A Sieve Algorithm for the Shortest Lattice Vector Problem," in *Annual ACM Symposium on Theory of Computing (STOC)*, 2001, pp. 601–610.

[14] T. Laarhoven, "Sieving for shortest vectors in lattices using angular locality-sensitive hashing," Cryptology ePrint Archive, Report 2014/744, 2014, http://eprint.iacr.org/.