A Template Library to Integrate Thread Scheduling and Locality Management for NUMA Multiprocessors

Zoltan Majo
Department of Computer Science
ETH Zurich

Thomas R. Gross
Department of Computer Science
ETH Zurich

Abstract

Many multicore multiprocessors have a non-uniform memory architecture (NUMA), and for good performance, data and computations must be partitioned so that (ideally) all threads execute on the processor that holds their data. However, many multithreaded applications show heavy use of shared data structures that are accessed by all threads of the application. Automatic data placement and thread scheduling for these applications is (still) difficult.

We present a template library for shared data structures that allows a programmer to express both the data layout (how the data space is partitioned) as well as thread mapping and scheduling (when and where a thread is executed). The template library supports programmers in dividing computations and data for reducing the percentage of costly remote memory accesses in NUMA multicore multiprocessors. Initial experience with `ferret`, a program with irregular memory access patterns from the PARSEC benchmark suite, shows that this approach can reduce the number of remote accesses from 42% to 10% and results in a performance improvement of 3% without overwhelming the programmer.

1 Introduction

The performance of many programs depends closely on the performance of the memory system. To offer better memory system performance, recent multicore-multiprocessors have a non-uniform memory architecture (NUMA). Figure 1 shows a 2-processor 8-core NUMA multicore-multiprocessor.

In NUMA systems each processor has a direct connection to a subset of the total system memory and each processor can access its directly connected memory with high bandwidth and low latency. We call these high performance memory accesses *local accesses*. As NUMA systems are shared memory multiprocessors, each processor can access the memory of other processors as well. These *remote accesses* to memory are transferred through the interconnect that connects processor chips. Transferring data through the cross-chip interconnect causes overhead, and as a result remote memory accesses have a performance penalty over local memory accesses. If, however, good *data locality* is achieved (i.e., the number of costly remote memory accesses is low), the memory system performance of NUMA multiprocessors scales well with increasing core- and processor counts [9]. Therefore, the main goal of performance optimizations targeting NUMA systems is to have good data locality in the system.

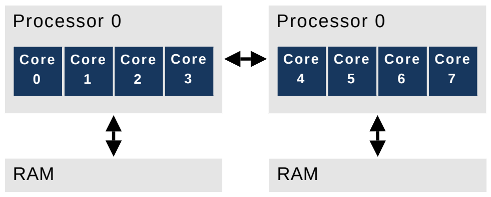


Figure 1: Example NUMA-multicore machine.

As multicore processors are widespread, there is a large body of research on performance optimizations for these systems. Many of these performance optimizations are evaluated on multicore-multiprocessors with a non-uniform memory architecture, however few authors consider the problem of data locality explicitly. Moreover, there is little work on performance optimizations specifically targeting NUMA-multicore systems. In our previous work [8] we describe a prototype thread scheduler that optimizes data locality by scheduling threads so that they run on the same processor that the data has been allocated at. Blagodurov et. al [4] also attack the problem of thread scheduling in NUMA multicores. Their approach supports data migration to improve data locality in the system. However, data migration incurs overhead that needs to be carefully accounted for, otherwise the overhead of data migration cancels the benefit of improved data locality. Data migration is used also to improve the performance of scientific multithreaded computations [10, 11, 18] and Java virtual machines [13, 17].

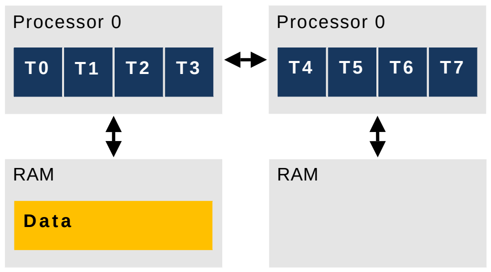
Inter-processor data sharing Although the previously mentioned performance optimizations result in significant performance improvements, there is a shortcoming that affects all these optimizations, namely that these optimizations are evaluated mostly with programs that have little or no data sharing. In programs with little data sharing threads mostly access non-overlapping sets of data; thus the data can be divided between the processor of the system so that each processor accesses data local to it. Optimizing scheduling and memory allocation for multithreaded programs with shared data, however, is a more difficult problem due to *inter-processor data sharing*.

To better illustrate the problem of inter-processor data sharing let us consider a program that is executed on the 2-processor 8-core NUMA machine shown in Figure 2. To guarantee good load balance, four of the program's threads (T0–T3) run on Processor 0 and four threads (T4–T7) run on Processor 1. All threads access the same piece of data.

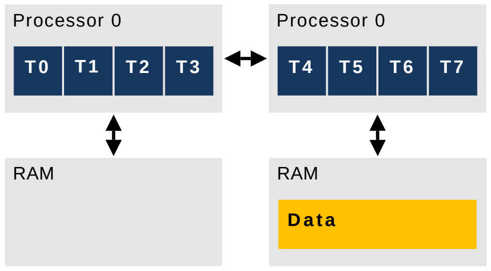
Depending on the allocation of the shared data at the processors of the system we can distinguish three possible scenarios. In the first scenario (Figure 2(a)) all shared data is allocated at Processor 0. In this scenario threads T0–T3 access the shared data locally and threads T4–T7 access the shared data remotely, therefore threads T4–T7 experience a performance penalty. In the second scenario (Figure 2(b)) the data is allocated at Processor 1. This scenario is not better either because in this case the threads running on Processor 0 must access the data remotely. In the third scenario (Figure 2(c)) the data is divided into two chunks (Data 0 and Data 1). The chunks are allocated at Processor 0 and Processor 1, respectively. All threads access both pieces of data, therefore all threads experience remote memory accesses and the corresponding performance penalty.

In summary, in all three previously presented scenarios the multithreaded program experiences performance degradation due to remote memory accesses. The remote memory accesses are the result of inter-processor data sharing caused by all threads of the program accessing the same shared data. This example shows that in the presence of shared data there is no way to allocate data and computations in a NUMA system so that the fraction of remote memory accesses to these shared data is low.

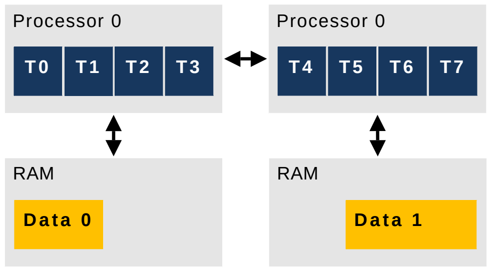
Globally shared data structures The problem of inter-processor data sharing has been previously noticed by several authors [16, 15, 19, 7]. Existing solutions, however, are mostly limited to scientific computations with regular memory access patterns that can be understood statically. In this paper we consider programs with more irregular data access patterns instead.



(a) Data allocated at Processor 0.



(b) Data allocated at Processor 1.



(c) Data allocated at both processors.

Figure 2: Inter-processor data sharing.

This paper has two parts. The first part presents a case study of inter-processor data sharing for the `ferret` program. `ferret` is part of the PARSEC benchmark suite [2] and belongs to the important class of pipeline-parallel programs well discussed in the literature [14, 12]. Low-level hardware performance counter data shows that `ferret` suffers a performance penalty due to the high percentage of remote memory references. Examining the program's memory access patterns and the source-code level suggests that the remote memory accesses of the program are caused by inter-processor data sharing: The threads of the `ferret` program frequently access a data structure shared between all threads of the program.

Optimizing data locality for `ferret` is difficult, because to eliminate inter-processor data sharing the shared data structure must be divided between the processors of the system. Dividing the data structure, however, requires detailed knowledge about the implementation of the data structure, about the way the data structure is accessed, and also about the architecture of the system the data structure is going to be used on. Therefore, in the second part of the paper we discuss a template library that provides high-level primitives for data allocation and computation scheduling that can be used to restructure data structures to avoid inter-processor data sharing. As an example use-case of the template library we rewrite the performance-critical data structure of ferret to use functionality provided by the library. Initial experience with a version of `ferret` based on this NUMA-aware data structure shows a reduction of the percentage of remote memory accesses from 42% to 10% that results in a performance improvement of 3% without imposing an unnecessarily high burden on the programmer.

2 Case study: Inter-processor data sharing

2.1 Experimental setup

This section presents an analysis of the memory system performance of the `ferret` program. `ferret` is executed on a 2-processor 8-core system based on the Intel Nehalem microarchitecture (shown in Figure 1). The Nehalem-based system has a non-uniform memory architecture. The local memory interfaces of the Nehalem-based machine have a total throughput of 51.2 GB/s; the cross-chip interconnect has less throughput (23.44 GB/s). Additionally, local memory accesses experience a significantly lower latency than remote memory accesses do (50 ns vs. 90 ns). The Nehalem system used in the experiments runs Linux 2.6.30.

In its initial configuration the `ferret` benchmark program runs with a number of worker threads larger than the number of cores available in the system [3] (34 threads in the case of our 8-core system). However, running `ferret` with a smaller number of threads avoids excessive context-switching overhead and improves performance, as previously shown in [12, 14]. Therefore, we run `ferret` with only 16 threads. We also fix the thread-to-core affinities of the program's threads to further reduce context-switching overhead and also to reduce measurement variation. These changes result in a 2% performance improvement over the initial configuration of the benchmark. We refer to the configuration with the reduced number of threads and fixed thread-to-core affinities as default setup.

In NUMA systems the OS memory allocation policy determines at which processor each page is allocated. Many OSs (including Linux) use the *first-touch* memory allocation policy, which allocates each page at the processor that first reads from or writes to the page after the page's allocation. In all our experiments memory allocation is based on the first-touch policy. To avoid performance degradation caused by page-level false sharing [5], we use a prototype memory allocator that, similarly to the Hoard allocator [1], uses per-processor memory pools.

2.2 Data address profiling

The memory behavior of programs can be understood by tracing memory accesses. On most modern microarchitectures memory access tracing is supported by hardware performance counters. The Intel Nehalem microarchitecture implements sampling-based data address profiling, a mechanism that records the data address used by each sampled memory access instruction. Using this mechanism we build a memory access trace for the `ferret` program. The trace contains the number of times each page of `ferret`'s address space is accessed. The trace may include only a subset of the pages mapped by `ferret` because data address profiling is sampling-based.

There are two types of pages that appear in the trace: accesses to the stack and accesses to the heap. The stack is private to each thread and the first-touch memory allocation can appropriately allocate pages that store the stack, therefore we exclude stack pages from further analysis and we focus only on heap accesses.

Most heap pages that appear in the trace file have a *dominant processor* that contributes the largest part of the accesses to that page. Based on the percentage of accesses contributed by the dominant processor we divide accesses into six categories; Figure 3 shows this categorization. The first category contains accesses to pages accessed by a single processor (the dominant processor, which contributes 100% of the accesses to that page). Please note that accesses in this category appear to be performed exclusively by a single processor, however, as tracing is sampling-based, the trace file does not capture all accesses (some of which possibly originate at a processor different than the dominant processor). Nevertheless, we can state that accesses in the first category target pages predominantly accessed by a single processor. Figure 3 shows that a large percentage (59%) of all heap accesses of `ferret` are issued to pages that belong to the first category. Pages in this category can be allocated so that all (or most) accesses to these pages are local.

The remaining five categories contain heap pages that are accessed by both processors of the system (shared pages). In each category the dominant processor contributes a different percentage of the total number of accesses (in these categories 90% to 50% of the accesses are contributed by the dominant processor). In total 41% of `ferret`'s heap accesses use shared heap pages. More-

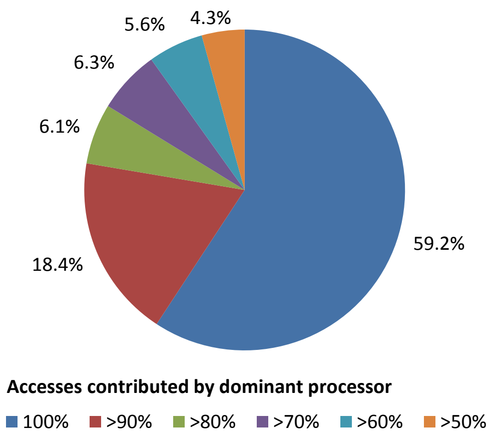


Figure 3: Accesses to shared data in `ferret`.

over, around 16% of the accesses use pages that have a dominant processor of less than 80%, that is, these pages are accessed frequently by both processors in the system and thus it is impossible to place these pages in the system without some of the threads of `ferret` experiencing a high number of remote memory references.

2.3 Cause of inter-processor data sharing: database lookups

A closer examination of `ferret`'s source code reveals the problem that causes the program to exhibit high inter-processor data sharing. `ferret` implements content-based similarity search of images. At first similarity search extracts features of each image given as input to the program. Then an image database is queried for images that have similar features as those of the input images. The output of the program contains a list of images in the program database that are most similar to the image given as input. For more details of the internals of `ferret` please refer to [2, 6].

`ferret` is organized as a pipeline; pipeline stages are interconnected with queues. Both feature extraction and database querying are implemented as a separate pipeline stage. Figure 4 shows the implementation pipeline stage that performs the database queries. For conciseness this code example (and also other examples in this paper) uses a Java-like syntax.

To increase the utilization of parallel machines each pipeline stage is executed in parallel by multiple threads, each thread processing a different input image. Threads run on both processors of the system, however each instance of the querying phase accesses the same globally shared image database. Querying the database is a memory-intensive operation (90% of the main memory bandwidth generated by `ferret` is due to Line 9 in Figure 4). Because the allocation of the image database

```
1  class Indexing implements Runnable {
2    Queue inputQueue, outputQueue;
3    Features features;
4    Image[] images;
5
6    public void run() {
7      while(true) {
8        features = inputQueue.take();
9        images = database.query(features);
10       outputQueue.put(images);
11     }
12   }
13 }
```

Figure 4: Database querying in `ferret`.

is not controlled at all and database queries are executed simultaneously by multiple threads, the execution of database queries in the `ferret` benchmark can correspond to any of the scenarios shown in Figure 2 (the exact scenario is determined by two dynamic factors: the thread-to-core assignment of the OS scheduler and the first-touch memory allocation policy). Due to inter-processor data sharing 42% of the total main memory bandwidth generated by `ferret` consists of remote memory accesses.

3 Template library for integrated data management and thread scheduling

The main cause of performance degradation for `ferret` (and also for other programs of the PARSEC suite we looked at) is that in these programs shared data structures are organized in a NUMA-oblivious way that results in inter-processor data sharing. These data structures are supplied as a library, and the application programmer reuses functionality provided by the library. To optimize data locality for a program using these data structures the programmer must be aware of low-level details about the internal organization of the data structures. Moreover, the data structures do not provide a handle for the application programmer to easily control memory allocation and computation scheduling within the data structure.

For example, `ferret` searches for images in an in-memory database. The search in the image database uses multi-probe locality sensitive hashing (LSH) [6]. This method relies on indexing the database in the form of multiple hash tables. The hash function used for the hash tables maps with high probability similar images into the same hash location. To find the images closest to the image given as input, multiple hash tables are used in the index. Because each lookup can potentially access the whole image database, making multi-probe LSH NUMA-aware requires two actions to be taken: (1) a disjoint subset of the image database must be allocated at each processor of the system, and (2) the computations associated with querying the database must be divided into multiple pieces, where each piece operates on a subset of the data and each piece of computation is executed on the processor that stores the data. Taking these two steps can increase the data locality of database queries, however adding these functionalities to the database requires a significant revision of the database code provided as a library.

In general, there exist many library-based data structures that lack NUMA functionality and thus must be revised to include better control of data distribution and computation scheduling. This section describes a template library that implements these functionalities to aid programmers to write NUMA-aware data structures. Figure 5 lists the class `SplittableData` that forms the basis of the template library by implementing two basic functionalities, *per-processor memory allocation* and *locality-aware task dispatching*.

```
1  abstract class
2  SplittableData<Data,Query,Result> {
3    Map<Processor,Data> map;
4    ThreadPool threadPool;
5
6    Data newAtProcessor(Processor p) {
7      // new Data instance at Processor p
8    }
9
10   Result dispatch(Query query,
11                   Class task) {
12
13     List<Result> results;
14     Data localData;
15
16     for (Processor p : processors) {
17       localData = map.get(p);
18       threadPool.submit(localData,
19                         query,
20                         task);
21     }
22
23     for (Processor p : processors)
24       results.add(threadPool.get(p));
25
26     return Result.merge(results);
27   }
28 }
```

Figure 5: Template data type.

Per-processor memory allocation With per-processor memory allocation it is possible to allocate data at the processor(s) specified by the programmer. For example in the case of multi-probe LSH a subset of the total number of images stored in the database can be allocated at each processor. In our template library the `newAtProcessor()` method implements this functionality (line 6 in Figure 5). This method can be mapped to lower-level OS functionality that supports per-processor memory allocation (e.g., the `numa_alloc()` call in Linux). The data structure keeps track of the allocation site of its elements in the `map` field on line 3 in Figure 5.

Locality-aware task dispatching After the distribution of data in the system is defined, care needs to be taken to dispatch computations to the same processor as where the data has been allocated at. This is handled by the `dispatch()` method (line 10 in Figure 5). This method first determines for each processor the subset of the data allocated at that processor. Then the `dispatch()` method submits a task to the thread pool of the library together with a pointer to the data selected in the previous step. The thread pool consists of worker threads executing at each processor of the system. The `dispatch()` method ensures good data locality by executing tasks at the same processor as the processor holding the data processed by the task. After all tasks have completed, the result produced by each task are collected, merged, and then returned to the caller of the `dispatch()` method.

Initial experience Figure 6 shows a possible way to implement a NUMA-aware version of `ferret`'s image database. The `query()` method required by client code accessing the database is mapped to the `dispatch()` code of the library. The multi-probe LSH lookup is implemented by the task `LSHTask`. `LSHTask` implements a modified version of multi-probe LSH. The modified version accesses all hash tables of the database index, however it does detailed computations only on the subset of the image database given to it as parameter. To implement a full query, the library's `dispatch()` method first creates multiple instances of `LSHTask`, then a task is dispatched to each processor of the system to process the set of images local to that processor. After all tasks finish, results are summarized and are passed on to the next pipeline stage of `ferret`. With the locality-aware implementation of the image database, `ferret` shows a reduction of remote memory accesses from 42% to 10% and a performance improvement of 3% over the default setup of the benchmark.

4 Related work

There are several approaches that optimize data locality on NUMA systems. Blagodurov et al. [4] integrate automatic data migration into an OS scheduler to reduce the number of remote memory accesses. On-line data migration based on hardware performance counter feedback has been also explored by Tikir et al. [18] and Vergh-

```
class
Database<Image[],Features,Image[]> extends
SplittableData<Image[],Features,Image[]> {

  public Image[] query(Features feature) {
    return dispatch(feature, LSHTask);
  }
}
```

Figure 6: Image database based on the template library.

ese et al. [19]. Marathe et al. [10] place memory pages based on profile information to increase the data locality of scientific programs. Data locality optimizations can be beneficial also in a Java virtual machines, as shown by Ogasawara [13] and Tikir et al. [17].

If a program exhibits inter-processor data sharing, program data cannot be distributed in the system without hurting the performance of a subset of the threads of the program. Inter-processor data sharing has been described by several authors. Thekkath et al [16] describe an approach to cluster threads based on the amount of sharing between each thread. They do not achieve performance improvements because the clustering techniques they develop do not decrease the number of cache misses experienced by the benchmark programs they use, mostly due data sharing between threads. Tam et al. [15] present a similar approach based on clustering of threads, however, they consider only programs with low degree of data sharing between threads. Dynamic page migration and replication has been explored by Verghese et al. [19], however the authors do not consider enabling these mechanisms for regions that are shared between threads. In our earlier work [7] we show that data migration is not enough to optimize the performance of programs with inter-processor data sharing, but additional program transformations are required to eliminate inter-processor data sharing.

5 Conclusions

Many current multiprocessor systems have a non-uniform memory architecture. For good application performance in these systems it is crucial that the number of costly remote memory accesses is kept low. However, many multithreaded programs process data that is shared between all threads of the program and this situation renders data locality optimizations difficult.

The reason for global data sharing lies in many cases within data structures that are reused from libraries providing the functionality required by the programmer. As the structure of and the access patterns to these data structures are in many cases complicated and difficult to understand, the data structures themselves should take care of appropriate data distribution and thread scheduling in a NUMA system. This paper presents a template library that offers pre-defined data distribution and thread scheduling primitives. We expect that this template library can provide a basis for future NUMA-aware data structures.

References

[1] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS '00*.

[2] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08*.

[3] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. Tech. Rep. TR-811-08, Princeton University, January 2008.

[4] Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A. A case for NUMA-aware contention management on multicore processors. In *USENIX ATC '11*.

[5] Lee, J. W., and Cho, Y. An effective shared memory allocator for reducing false sharing in NUMA multiprocessors. In *ICAPP '96*.

[6] Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB '07*.

[7] Majo, Z., and Gross, T. R. Matching memory access patterns and data placement for NUMA systems. In *CGO '12*.

[8] Majo, Z., and Gross, T. R. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ISMM '11*.

[9] Majo, Z., and Gross, T. R. Memory system performance in a NUMA multicore multiprocessor. In *SYSTOR '11*.

[10] Marathe, J., Thakkar, V., and Mueller, F. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Parallel Distrib. Comput. 70* (2010), 1204–1219.

[11] McCurdy, C., and Vetter, J. S. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *ISPASS '10*.

[12] Navarro, A., Asenjo, R., Tabik, S., and Cascaval, C. Analytical modeling of pipeline parallelism. In *PACT '09*.

[13] Ogasawara, T. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA '09*.

[14] Suleman, M. A., Qureshi, M. K., Khubaib, and Patt, Y. N. Feedback-directed pipeline parallelism. In *PACT '10*.

[15] Tam, D., Azimi, R., and Stumm, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07*.

[16] Thekkath, R., and Eggers, S. J. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94*.

[17] Tikir, M. M., and Hollingsworth, J. K. NUMA-aware java heaps for server applications. In *IPDPS '05*.

[18] Tikir, M. M., and Hollingsworth, J. K. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput. 68* (2008), 1186–1200.

[19] Verghese, B., Devine, S., Gupta, A., and Rosenblum, M. Operating system support for improving data locality on cc-NUMA compute servers. In *ASPLOS '96*.