

# Matching Memory Access Patterns and Data Placement for NUMA Systems

Zoltan Majo  
Department of Computer Science  
ETH Zurich

Thomas R. Gross  
Department of Computer Science  
ETH Zurich

## ABSTRACT

Many recent multicore multiprocessors are based on a non-uniform memory architecture (NUMA). A mismatch between the data access patterns of programs and the mapping of data to memory incurs a high overhead, as remote accesses have higher latency and lower throughput than local accesses. This paper reports on a limit study that shows that many scientific loop-parallel programs include multiple, mutually incompatible data access patterns, therefore these programs encounter a high fraction of costly remote memory accesses. Matching the data distribution of a program to the individual data access patterns is possible, however it is difficult to find a data distribution that matches *all* access patterns.

Directives as included in, e.g., OpenMP provide a way to distribute the computation, but the induced data partitioning does not take into account the placement of data into the processors' memory. To alleviate this problem we describe a small set of language-level primitives for memory allocation and loop scheduling. Using the primitives together with simple program-level transformations eliminates mutually incompatible access patterns from OpenMP-style parallel programs. This result represents an improvement of up to 3.3X over the default setup, and the programs obtain a speedup of up to 33.6X over single-core execution (19X on average) on a 4-processor 32-core machine.

## Categories and Subject Descriptors

C.4 [Performance of Systems]: Performance attributes, Measurement techniques; D.1.3 [Concurrent Programming]: Parallel programming

## General Terms

Measurement, Performance, Languages

## Keywords

NUMA, scheduling, data placement

## 1. INTRODUCTION

Multicore multiprocessors are an attractive and popular platform that are used as stand-alone systems or as building blocks in supercomputers. These systems usually have a *non-uniform memory access* (NUMA) architecture to allow scaling. Figure 1 shows an example 2-processor system.

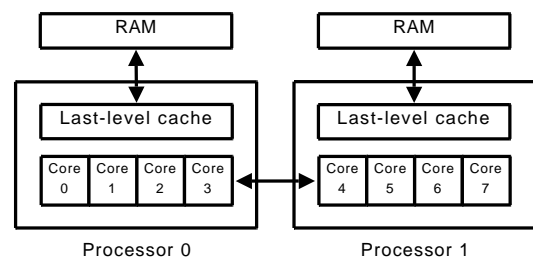


Figure 1: Example NUMA-multicore machine.

In a NUMA multiprocessor system, each processor enjoys a direction connection to *some* of the memory modules that together make up the system's memory. Accessing these *locally* connected modules has high throughput and low latency. However, as NUMA systems still offer the shared memory programming paradigm, each processor must be able to access the memory modules of other processors as well. The processors of a NUMA system are therefore connected with a cross-chip interconnect, and cross-chip (or shortly *remote*) memory accesses experience the additional overhead of being transferred through this interconnect. Reducing the number of remote memory accesses is critical for the performance of NUMA systems.

In this paper we focus on the performance of loop-based parallel code within a shared-memory multiprocessor with NUMA characteristics. In the first part of the paper we analyze the memory behavior of multi-threaded scientific computations of the NAS Parallel Benchmark (NPB) suite [6]. Experiments show that in many of these computations the threads access a large number of memory locations, and we characterize the applications with regard to their access patterns. A consequence of these access patterns is that it is difficult to obtain good data locality on NUMA systems, and thus it is difficult to get good performance. The experiments show furthermore that OS data migration and locality-aware iteration scheduling can in some cases soften the problems caused by global accesses, however in many cases the overhead of using these mechanisms cancels the benefits of good

data locality (or worse).

The mismatch between the data layout in the processor’s memory and the accesses to these data as performed by the processors’ cores is immediately felt by users of popular parallelization directives like OpenMP. These directives allow partitioning of computations but do not allow a programmer to control neither the mapping of computations to cores, nor the placement of data in memory. We present a small set of user-level directives that, combined with simple program transformations, can almost completely eliminate remote memory accesses for the NPB suite and thus result in a performance improvement of up to 3.3X over the default setup in a 4-processor 32-core machine.

## 2. MOTIVATION

### 2.1 Experimental setup

This section presents an analysis of the memory behavior of data-parallel programs of the NPB suite [6]. Table 1 shows a short characterization of the NPB programs.

The programs are executed on a 2-processor quad-core NUMA machine based on the Intel Nehalem microarchitecture (Figure 1) with 6 GB DRAM/processor. The total theoretical throughput of both local memory interfaces of the system is 51.2 GB/s, and the cross-chip interconnect has a total theoretical throughput of 23.44 GB/s. Local memory accesses have a lower memory access latency than remote memory accesses (50 ns vs. 90 ns). The experiment platform runs Linux 2.6.30. patched with perfmon2 for performance monitoring [5].

Benchmark	Class	Working set size	Run time
bt	B	1299 MB	125 s
cg	B	500 MB	26 s
ep	C	72 KB	85 s
ft	B	1766 MB	19 s
is	B	468 MB	10 s
lu	C	750 MB	368 s
mg	C	3503 MB	33 s
sp	B	423 MB	82 s

Table 1: Benchmark programs.

All benchmark programs are configured to execute with 8 worker threads (the total number of cores in the system) for all experiments described in this paper (if not explicitly stated otherwise). Similarly, for all experiments the thread-to-core mapping of the worker threads is fixed to *identity affinity*. Identity affinity maps each worker thread to the core with the same number as the thread’s number (e.g., Thread T0 is mapped to Core 0). As threads are mapped to cores with identity affinity, the terms core and thread are used interchangeably in this paper. Fixing the thread-to-core affinity disables OS reschedules and thus measurement readings are more stable. Additionally, as the programs of the NPB suite are data-parallel, using thread-to-core affinities other than identity affinity does not result in any increase or decrease of performance (as noted previously by Zhang et al. [19]). In NUMA systems memory allocation happens on a per-processor basis, therefore we refer to processors when we describe the distribution of data in the system: Processor 0 contains the memory local for threads

T0–T3, and Processor 1 contains memory local for threads T4–T7.

Figure 2 shows a breakdown of the total measured memory bandwidth into local and remote memory bandwidth for all benchmark programs of the NPB suite. For these measurements the *first-touch* page placement policy was in effect. This policy places every page at the processor that first reads from/writes to this page after page allocation. We refer to the combination of first-touch policy and identity affinity as *default setup*. As shown in Figure 2, the total memory bandwidth generated by *ep* is negligible. As a result, the performance of this program does not depend on the memory system, and the program is excluded from further investigation. The *is* and *mg* programs are excluded as well because both programs exhibit a low percentage of remote memory accesses with the default setup (on average 3% and 2% of the total bandwidth, respectively). Nonetheless, the other benchmarks generate a significant amount of bandwidth (up to 23 GB/s) and also show a significant contribution of remote memory accesses (11%–48% of the total bandwidth). These high percentages suggest that there is a need for approaches that reduce the number of remote memory accesses.

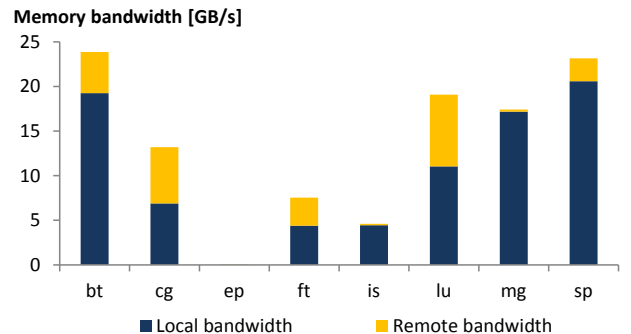


Figure 2: Memory bandwidth generated by the NPB suite.

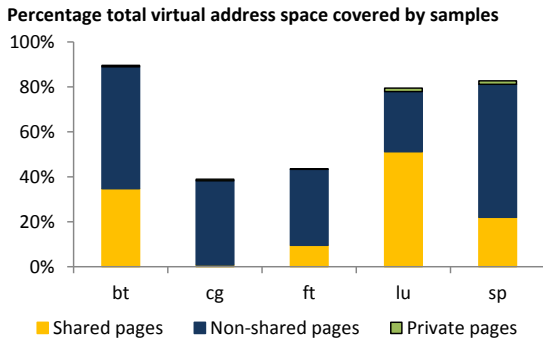
### 2.2 Profiling

To explore the memory performance issues in a limit study, we profile the execution of the NPB programs using the latency-above-threshold profiling mechanism of the Intel Nehalem microarchitecture. This hardware-based mechanism samples memory instructions with access latencies higher than a predefined threshold and provides detailed information about the data address used by each sampled instruction. Based on the sampled data addresses it is straightforward to determine the number of times each page of the program’s address space is accessed by each core of the system. To account for accesses to all levels of the memory hierarchy we set the latency threshold to 3 cycles (accesses to the first level cache on the Nehalem-based system have a minimum latency of 4 cycles). The profiling technique is portable to many different microarchitectures, because most recent Intel microarchitectures support latency-above-threshold profiling. Additionally, AMD’s processors support Instruction-Based Sampling [2], a profiling mechanism very similar to that of Intel’s.

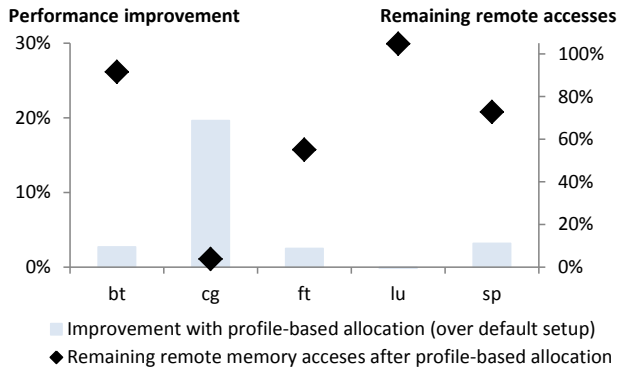
As memory access profiling on the Intel Nehalem is sampling-based, not all pages of the program’s address space appear

in the profiles. Figure 3(a) shows for each program the percentage of the virtual address space covered by samples. A page of the address space is covered if there is at least one sample in the profiles to an address that belongs to the page in question. Samples cover a large portion of the address space (up to 90%). The coverage is low in the case of *cg* because around 50% of the program’s address space is occupied by data structures that store temporary data. As these structures are used only in the initialization phase of the program, very few samples are gathered for pages that store these structures.

A program’s address space can be divided into three categories. The first category contains *private* pages that belong to each thread’s stack. As the stack is private for each thread, the first-touch policy (remember that we profile program execution with the default setup) can allocate the pages for thread-private stacks on each thread’s processor. The first category also contains pages that are reserved for dynamically linked shared objects and the program’s image in memory. These latter regions are small and are shared by all threads. As not much can be done for the first category of pages as far as data placement is concerned, these pages are excluded from further analysis and optimizations.



(a) Sampled and covered address space.



(b) Performance improvement; remaining remote accesses.

Figure 3: Hardware-based profiling.

Most data accessed by the computations is allocated on the heap, which holds the second and third category of pages. Some heap pages are exclusively accessed by a single thread, thus these pages are *non-shared*. Other heap regions, however, are accessed by multiple threads, and are thus *shared*. Figure 3(a) shows the percentages of the address space that

belong to the three categories previously described. The NPB programs we consider exhibit various degrees of sharing, ranging from almost no sharing (e.g., *cg*) to intense sharing when up to 51% of the program address space is accessed by multiple threads (e.g., *lu*).

### 2.3 Frequency-based page placement

Profiles provide a great deal of information about the access patterns of a program, but the usefulness of profiles for optimizing data locality remains to be determined. To optimize data locality we adapt the profile-based page-placement approach described by Marathe et al. [10], and we use a simple heuristic to determine for each page the processor it should be allocated at: On a profile-based run of a benchmark we allocate each page on the processor whose cores accessed the page the most frequently. Figure 3(b) shows the performance improvement over the default setup when profile-based memory allocation is used. The performance measurements include the overhead of reading the profiles and placing data into memory. Profile-based allocation performs well for only the *cg* benchmark, which improves 20%. However, the remaining benchmarks show minor or no performance improvement over the default setup.

The same figure (Figure 3(b)) shows also the percentage of the programs’ default remote memory accesses remaining after profile-based allocation. The performance improvement correlates well with the reduction of the remote memory traffic measured with the default setup. For *cg* (the program with the largest performance improvement) a small percentage of the default remote memory accesses remains after profile-based allocation. For programs with little or no performance improvement (e.g., *lu*) remote memory traffic is the same as with the default setup, because profile-based memory allocation was not able to determine a distribution of pages in memory that reduces or removes remote memory traffic (that was originally observed with the default setup).

The information about the number of shared pages (Figure 3(a)) provides additional insights into the performance improvement due to profile-based allocation (Figure 3(b)): the *cg* benchmark, which has a small number of shared pages improves with profile-based memory allocation, but the other benchmarks, which have a high number of shared pages, do not. Benchmarks with high degree of sharing have the same number of remote memory accesses both with profile-based memory allocation and the default setup (Figure 3(b)). These measurements show that profile-based allocation cannot correctly allocate exactly those pages that are shared between multiple processors. In this paper we describe various techniques to improve the performance of programs with a high degree of data sharing.

## 3. UNIFORM DATA SHARING

NUMA systems introduce another aspect into the problem of managing a program’s data space. The program’s computation determines the memory locations that are accessed; directives (e.g., OpenMP) or compilers and the runtime system determine how computations are partitioned respectively mapped onto cores. A simple characterization of memory access patterns provides a handle to understand the memory behavior of programs. We start with the memory allocation and data access patterns that frequently appear

in scientific computations (and thus in the NPB programs). Then we analyze the ability of different mechanisms (in-program data migration and loop iteration distribution) to improve data locality.

### 3.1 Access and distribution patterns

Many scientific computations process multidimensional matrices. Figure 4 shows a three-dimensional matrix stored in memory (in row-major order, addresses increase from left to right and from top to bottom, respectively). The matrix contains  $NX = 5$  rows (the size of the matrix along the x-dimension, shown vertically in the figure). Each row contains  $NY = 4$  columns (the size of the matrix along the y-dimension, shown horizontally). As the matrix has a third, z-dimension, the matrix contains a block of  $NZ$  items for each  $(x,y)$  element. The memory layout of matrices with dimensionality higher than three is analogous to the three-dimensional case, therefore we do not discuss such matrices in detail. Moreover, to keep the following discussion simple, we focus on two-dimensional matrices. Nonetheless, the principles we describe in this section apply to matrices with any dimensionality.

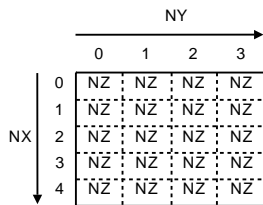


Figure 4: Memory layout of 3D matrix (row-major order).

When we study processor memories that are accessed by threads of scientific computations, we encounter two frequent data access- and data distribution patterns for two-dimensional matrices. We call these the *x-wise* and *y-wise* pattern, respectively.

***x-wise pattern.*** Figure 5(a) shows a simple loop that accesses all elements of a two-dimensional matrix. In this example the outermost loop (the loop that iterates along the x dimension of the matrix) is parallelized with a standard OpenMP `parallel for` construct. The loop iterations are distributed across all worker threads (here: eight threads). Each thread gets a chunk of size  $D = NX / 8$  of the total iteration space (assuming that  $NX$  is an integer multiple of 8). Figure 5(c) shows the assignment of iterations to worker threads for the code shown in Figure 5(a). Each thread  $T_0$  to  $T_7$  accesses  $D$  non-overlapping rows of the matrix. Figure 5(c) shows the distribution of the matrix's memory in the system for an *x-wise* data access pattern. As we map threads to cores with identity affinity, the first-touch policy allocates the first  $4 \times D$  rows at Processor 0 and the second  $4 \times D$  rows at Processor 1. We call this distribution of the matrix in memory an *x-wise* data distribution.

***y-wise pattern.*** If the second `for` loop (the one that sweeps the y-dimension) is parallelized (as shown in Figure 5(b)), the result is a different, *y-wise*, data access and distribution

pattern. Figure 5(d) shows both the allocation of the iteration space to worker threads and the distribution of the matrix memory to processors (based on a first-touch allocation). With the *y-wise* access pattern, the first  $4 \times D$  columns are allocated at Processor 0, and the second  $4 \times D$  columns are allocated at Processor 1.

Some programs that operate on data with a dimensionality higher than two have higher order access patterns. E.g., some programs of the NPB suite have a *z-wise* pattern. The iteration and data distribution in this case is analogous to the *x-wise* and *y-wise* patterns, respectively. The only difference is that the blocks of data are distributed among the threads and processors of the system along the *z*-dimension.

***Uniform data sharing.*** The problem of many data-parallel programs is that in many cases programs access a single memory region with multiple data access patterns. Let us assume that a memory region is accessed in two stages by eight worker threads ( $T_0$ – $T_7$ ). In the first stage (Figure 5(c)) the worker threads access the memory region with an *x-wise* access pattern, followed by an *y-wise* access pattern in the second stage (Figure 5(d)). Figure 5(e) shows the iteration distribution (i.e., mapping of rows resp. columns to threads for processing) for both stages: the iteration space distribution of the two stages overlaps.

If we divide the memory region into four equal-sized memory sub-regions (quadrants), then in both stages the upper left and lower right quadrants of the memory region are accessed exclusively by threads  $T_0$ – $T_3$  and threads  $T_4$ – $T_7$ , respectively. Therefore, these quadrants are non-shared and allocating these quadrants at Processor 0 (Processor 1) guarantees good data locality for threads  $T_0$ – $T_3$  ( $T_4$ – $T_7$ ).

The upper right and lower left quadrants, however, are accessed by all threads, and we call these quadrants *uniformly shared*. A memory region is considered to be uniformly shared if it is accessed by multiple threads from more than one processor (without concern about the number of accesses from each processor). For our example, uniformly shared quadrants constitute 50% of the region presented in this example. In this case the shared quadrants of the matrix are accessed equally often by the threads of all processors of the NUMA system, therefore the profile-based optimization technique described in Section 2 cannot determine the processor memory that should store these quadrants. If the processor of threads  $T_0$ – $T_3$  is preferred in allocation, then threads  $T_4$ – $T_7$  must access these regions remotely (or threads  $T_0$ – $T_3$  access their regions remotely if threads  $T_4$ – $T_7$  are preferred). For many programs that perform scientific computations uniform data sharing is a major performance limiting factor, as we have seen in Section 1. In the next section we describe in more depth a real-world example of uniform data-sharing, the `bt` benchmark from NPB.

### 3.2 Example: bt

The `bt` benchmark is a 3D computational fluid dynamics application [6] that iteratively solves partial differential equations using the alternating direction implicit method (implemented by the `adi()` function in Figure 6). The `adi()` function calls several other functions (also listed in Fig-

```

#pragma omp parallel for
for (i = 0; i < NX; i++)
  for (j = 0; j < NY; j++)
    // access m[i][j]

```

(a) x-wise data access pattern.

```

for (i = 0; i < NX; i++)
#pragma omp parallel for
  for (j = 0; j < NY; j++)
    // access m[i][j]

```

(b) y-wise data access pattern.

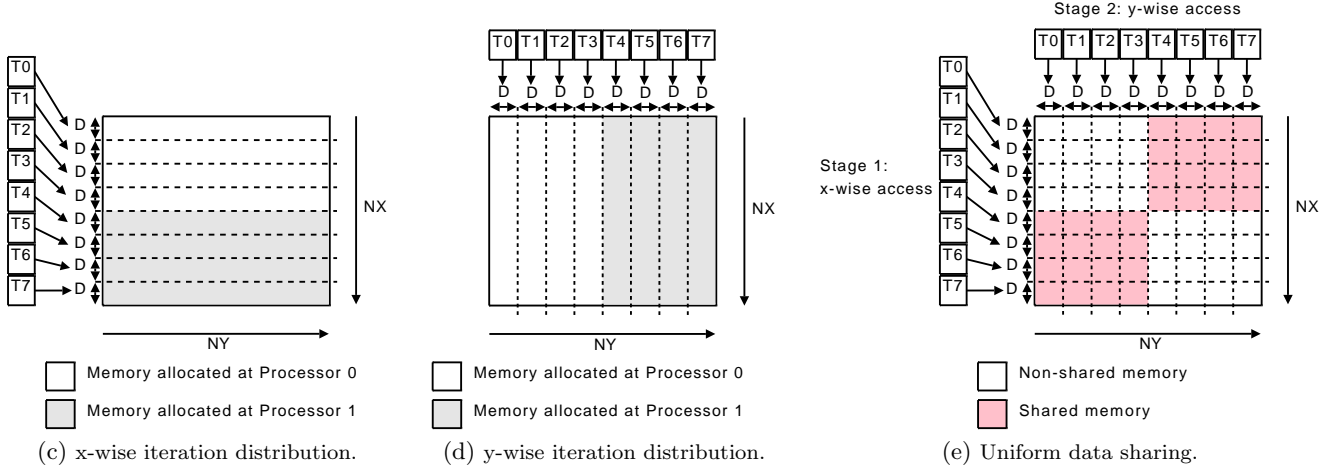


Figure 5: Data access patterns and uniform data sharing.

ure 6). These functions operate mostly on the same data, but they have different access patterns: The functions `compute_rhs()`, `y_solve()`, `z_solve()`, and `add()` have an x-wise access pattern, but the function `x_solve()` accesses memory y-wise. Moreover, as the `adi()` function is also used for data initialization in the first stage of `bt`, some of the regions (the ones that are first touched in `compute_rhs()`) are laid out x-wise in memory, and some of the regions (first used in `x_solve()`) are laid out y-wise. As a result of this mismatch in data access and memory allocation patterns a high percentage of this program’s virtual address space is uniformly shared (35%), and the program has a high percentage of remote memory accesses (19%).

```

1: adi() {
2:   compute_rhs(); // x-wise pattern
3:   x_solve();    // y-wise pattern
4:   y_solve();    // x-wise pattern
5:   z_solve();    // x-wise pattern
6:   add();        // x-wise pattern
7: }

```

Figure 6: `bt` data access patterns.

**In-program data distribution changes.** The simplest approach to reduce the fraction of remote memory accesses of programs with uniform data sharing is to change the distribution of the program’s data on-the-fly so that it matches each different access pattern that appears during program execution. A similar approach has been described in a cluster context for a program that calculates the fast Fourier transform (FFT) of 3D data [1]. To evaluate the effectiveness of this approach we insert into the code of the `bt` program calls to functions that change the data distribution in the system (line 4 and 7 in Figure 7). These func-

tions are based on the migration primitives offered by the OS (see Section 4.3 for more details about the implementation). The data distribution of the regions accessed by `x_solve()` is changed to y-wise data distribution before the call to `x_solve()` and then changed back to x-wise after the `x_solve()` function completes. The `adi()` function is executed in a tight loop, therefore starting with the second iteration of the loop the `compute_rhs()` function will encounter the correct, x-wise, data layout it requires (due to the previous iteration of `adi()`).

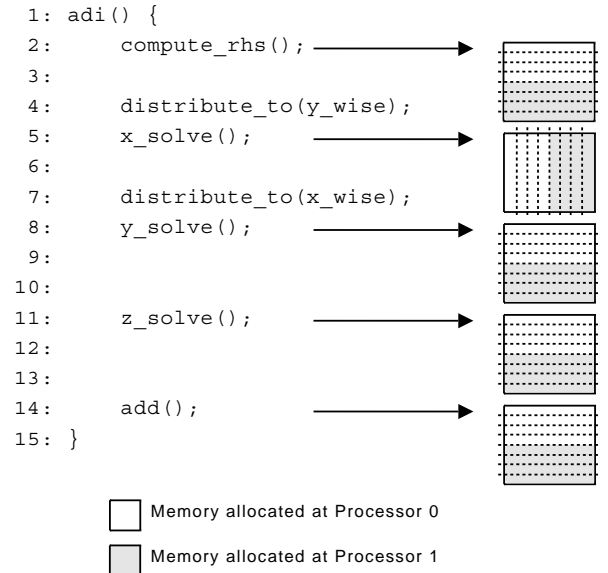


Figure 7: `bt` with in-program data migration.

Similar in-program data layout changes can be done to the `ft` benchmark. Figure 8(a) compares the bandwidth distri-

bution of the default setup (first-touch memory allocation) with dynamic in-program data layout changes for **ft** and **bt**. The figure reports the percentage of remote accesses relative to the total number of accesses, as measured in user-mode. As data migration is performed by the kernel, the bandwidth generated by data migrations is not included in the figure. The measurements show that in-program data layout changes significantly improve data locality for these two programs (**bt**: reduction of the percentage remote accesses from 19% to 1%, **ft**: reduction from 43% to 8%). This reduction of the fraction remote accesses results in a reduction of execution time of the computations, as shown in Figure 8(b) (this figure shows execution time relative to the default setup, therefore lower numbers are better). For the computation part, **bt** experiences a performance improvement of around 10%, and **ft** speeds up 16%. However, if we account for the overhead of data migration, the total performance is worse than without data migration (**bt** slows down 2.5X, **ft** slows down 1.4X).

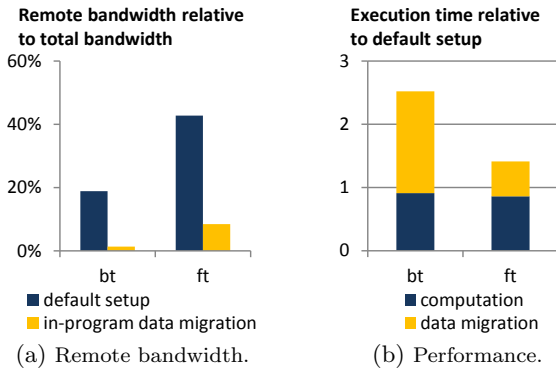


Figure 8: In-program data migration.

In conclusion, in-program data migration can eliminate remote memory accesses by guaranteeing the data distribution required by each access pattern of the program. However, if access patterns change frequently (as it is in the case of **bt**), the overhead of data migration cancels the benefits of data locality. Moreover, data migrations serialize the execution of multithreaded code, because access to the page tables shared by the threads requires acquiring a shared lock (parallel migrations do not result in any improvement relative to serial migrations). As a result, in-program data layout changes should be avoided for programs with frequent access pattern changes (but data migration could be still a viable option for coarser-grained parallel programs).

*Iteration redistribution.* Changing the data distribution causes too much overhead to be a viable option. As changing the distribution of computations has far less overhead than redistributing data, in this section we explore the idea of changing the schedule of loop iterations so that the access patterns of the program match the distribution of data in memory. We take the following simple approach for the **bt** benchmark: As the majority of the functions executed by **bt** has an x-wise access pattern, we change the distribution of *all* program data to x-wise. As a result, most functions experience good data locality and their performance improves. However, as the function `x_solve()` has a y-wise

access pattern, the number of remote memory accesses it must perform increases due to the change of the data distribution to x-wise. At the end, the performance of the whole program (**bt**) does not improve.

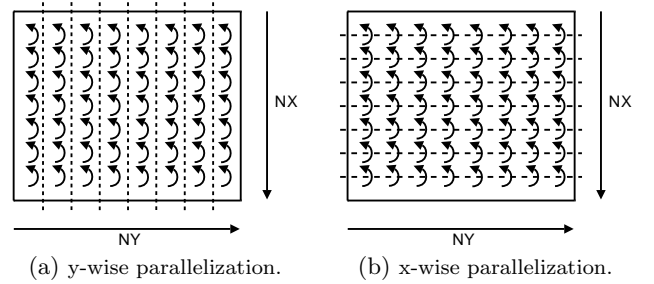


Figure 9: Access patterns of code with y-wise dependences.

To make `x_solve()` have also an x-wise access pattern, we take an approach similar to [7, 8], and we change the distribution of the iterations of the loops in `x_solve()` to x-wise. This transformation reduces the percentage of remote memory accesses from 19% to 8%, and improves the performance of **bt** by 6% relative to the default setup. Although changing the data distribution and the iteration distribution of **bt** eliminates half of the remote memory accesses of the program, further reduction is not possible without inserting extra synchronization operations due to loops in `x_solve()` that have loop-carried dependences along the y-dimension of the processed data. Usually iterations of loop-parallel code are distributed among worker threads so that loop-carried dependences are within the iteration space assigned to each thread for processing. The main reason for this decision is to keep the number of inter-thread synchronization operations low: if data dependences are within the iteration space assigned to each thread, threads do not need to synchronize. For example, Figure 9 shows the distribution of the iteration space for two different parallelizations of a computation that has y-wise data dependences. In the first, y-wise, parallelization (Figure 9(a)) the loop dependences do not cross per-thread iteration space boundaries. Nonetheless, in the second, x-wise parallelization of the code (Figure 9(b)) the data dependences cross iteration space boundaries, thus inter-thread synchronization is required. The overhead of inter-thread synchronization is potentially high, thus it can cancel the benefit of data locality achieved with redistributing iterations.

#### 4. FINE-GRAINED DATA MANAGEMENT AND WORK DISTRIBUTION

In-program data migration can be used to match the data distribution in the system to the access patterns of programs with uniform data sharing, however the high cost of the data migration cancels the benefit of improved data locality. Similarly, it is also possible to redistribute computations so that the data access patterns of the program matches a given data distribution, however this method is also incapable of completely eliminating remote memory accesses without introducing potentially large synchronization overhead. To address the dual problem of distributing computations and data, we describe a simple system API that can be used to change *both* the distribution of computations and the distribution of data in the system. Using the described API

together with simple program-level transformations can almost completely eliminate remote memory accesses in programs with uniform data sharing and thus helps to better exploit the performance potential of NUMA systems. The API presented here is implemented as an extension to GCC version 4.3.3 and offers two kinds of language primitives: primitives to manipulate the distribution of a program's data, and additional schedule clauses for distribution of loop iterations.

## 4.1 Data distribution primitives

**Block-cyclic data distribution.** Figure 10(a) shows the function `create_block_cyclic_distr()` that is used to define a block-cyclic data distribution. The block size given as parameter to this function influences the data distribution. Consider the case of a memory region that stores a two-dimensional matrix of size  $NX \times NY$  (as shown in Figure 5(c)). If the block size is  $NX \times NY / 2$ , the data region will be x-wise distributed as shown in Figure 5(c). If, however, the size of the block is  $NY / 2$ , the region will be y-wise distributed (Figure 5(d)). By further varying the block size the function can be used to set up a block-cyclic distribution for matrices with dimensionality higher than two as well.

```
data_distr_t *create_block_cyclic_distr(
    void *m, size_t size, size_t block_size);
```

(a) Block-cyclic data layout.

```
data_distr_t *create_block_exclusive_distr(
    void *m, size_t size, size_t block_size);
```

(b) Block-exclusive data layout.

```
void distribute_to(data_distr_t *distr);
```

(c) Apply data layout.

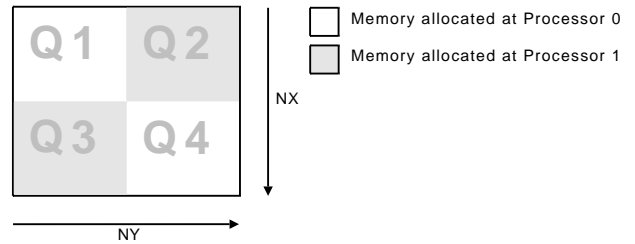
Figure 10: Data distribution primitives.

**Enforcing data distributions.** The primitives of the API decouple the description of the data distribution from the operations to implement a chosen data distribution. The function `create_block_cyclic_distr()` defines only a *description* of the distribution in memory of region `m` passed to it as parameter. The function `distribute_to()` shown in Figure 10(c) takes this description and then applies it (i.e., the function migrates data to enforce the distribution). For example, the in-program data distribution described in Section 3.2 uses this call to change the distribution of data in the system at runtime (see lines 4 and 7 of Figure 7).

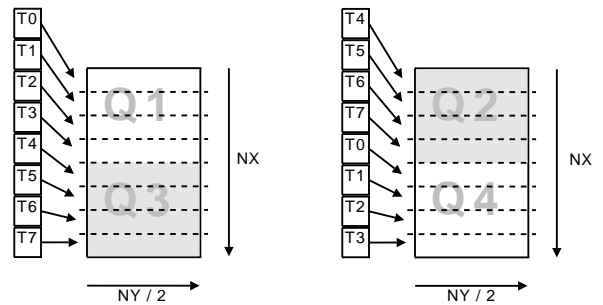
**Block-exclusive data distribution.** Changing the access pattern of code with loop-carried dependences requires frequent inter-thread synchronization. In many cases inter-thread synchronization has a high overhead so that the data locality gained from changing access patterns does not increase performance. Alternatively, the distribution of a data region can be changed to *block-exclusive* (see Figure 11(a)). The key idea behind using a block-exclusive distribution is that a region with a block-exclusive distribution can be

swept by two different data access patterns (e.g, with an x-wise and a y-wise access pattern) with no parts of the memory region shared between the processors of the system. As a result, a block-exclusive data region can be placed at processors so that all accesses to the region are local.

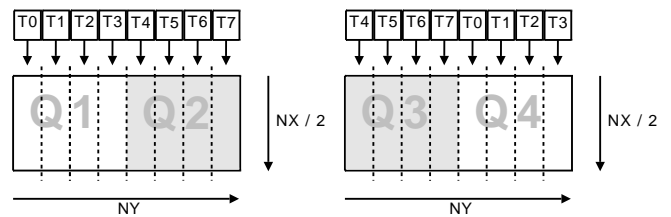
To illustrate the advantages of the block-exclusive data distribution let us consider the simple example when eight worker threads access the block-exclusive memory region in Figure 11(a). To simplify further discussion, we divide the memory region into four equal-sized quadrants (Q1–Q4). The eight worker threads considered in this example traverse the memory region with two different access patterns, first with an x-wise access pattern followed by a y-wise access pattern.



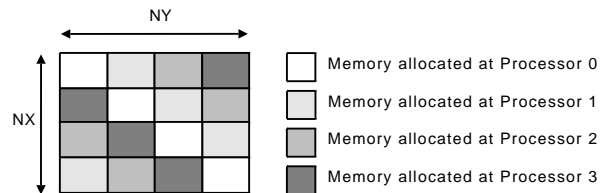
(a) Block-exclusive data distribution on two processors.



(b) x-wise traversal: phase 1. (c) x-wise traversal: phase 2.



(d) y-wise traversal: phase 1. (e) y-wise traversal: phase 2.



(f) Block-exclusive data distribution on four processors.

Figure 11: Block-exclusive data distribution.

The x-wise traversal of the memory region must be processed

in two phases due to the block-exclusive data distribution of the region. These two phases are illustrated in Figure 11(b) resp. Figure 11(c). In the first phase (Figure 11(b)) the worker threads process in parallel the left half of the data region (quadrants Q1 and Q3). The iteration distribution is standard OpenMP static scheduling: threads get their iterations in the ascending order of their respective thread number (threads T0–T3 process Q1, threads T4–T7 process Q3). As a result all threads access memory locally. In the second phase (Figure 11(c)) the threads process quadrants Q2 and Q4, which are allocated to processors in a different order than quadrants Q1 and Q3. To guarantee data locality in the second phase as well, the distribution of iterations between the threads is different from the first processing phase: threads T0–T3 process quadrant Q4, and threads T4–T7 process quadrant Q2 (as shown in Figure 11(c)). Between the two phases of the x-wise traversal thread synchronization is required, however as there is only one place in the traversal where cross-thread data is accessed (when execution transitions from processing the first two quadrants to processing the second two), the cost of this synchronization is negligible.

Uniform data sharing is in many cases caused by a memory region being traversed with an x-wise pattern followed by a y-wise pattern. So far we have seen an x-wise traversal of a block-exclusive memory region. A y-wise traversal of the same memory region is also possible, and it proceeds in two phases. In the first phase (Figure 11(d)) the worker threads process quadrants Q1 and Q2, followed by quadrants Q3 and Q4 in the second phase (Figure 11(e)). As the distribution of loop iterations in the phases of the y-wise traversal is similar to the case of the x-wise traversal, we do not discuss it any further. Please note, however, that the y-wise traversal of the memory region guarantees data locality as well.

Figure 10(b) shows the programming language primitive to create a block-exclusive data distribution. So far we have discussed the block-exclusive layout only for two processors. Using the `create_block_exclusive_distr()` on a system with a number of processors higher than two results in a *latin-square* [15, 4] distribution of memory. The latin-square distribution of a two-dimensional matrix on a 4-processor system is shown in Figure 11(f). The data distribution primitives described in this section can be used in a system with any number of cores/processors. The primitives only require that information about the number of cores/processors is available at runtime (which is the case in many recent OSs).

## 4.2 Iteration distribution primitives

Well-known methods for iteration distribution, like static scheduling are inflexible and cannot always follow the distribution of data in the system. For example, the assignment of loop iterations to data shown in Figure 11(c) or Figure 11(e) is impossible in an OpenMP-like system. To alleviate this problem we define `inverse static` scheduling as shown in Figure 12(a). With this simple extension data locality can be guaranteed also for code accessing regions with distribution as in Figure 11(c) or Figure 11(e).

We have seen in the previous section that the block-exclusive data distribution can be generalized to any number of processors (see Figure 11(f) for the 4-processor case). As the

distribution of loop iterations must match the distribution of data in the system if good data locality is desired, inverse scheduling must be generalized to a high number of processors as well. We call the generalized scheduling primitive *block-exclusive scheduling* (syntax shown in Figure 12(b)).

In the general case the block exclusive data distribution partitions a data region into blocks. In a system with  $p$  processors the data distribution can be seen either as  $p$  columns of blocks, or as  $p$  rows of blocks (see Figure 11(f) for a block-exclusive data distribution on a system with  $p = 4$  processors). For example on an x-wise traversal the data distribution can be viewed as  $p$  columns of blocks and on a y-wise distribution the data distribution can be viewed as  $p$  rows of blocks. For both views of a block-exclusive data distribution there are  $p$  different iteration-to-core distributions, one iteration distribution for each different configuration of columns of blocks resp. rows of blocks (so that the iterations are allocated to the processors that hold the block of data processed). As a result, a block-exclusive scheduling clause is defined as a function of two parameters. The first parameter specifies the view taken (columns of blocks resp. row of blocks), and the second parameter specifies the identifier `id` ( $0 \leq \text{id} < p$ ) of the schedule used. Section 5 presents a detailed example of using the block-exclusive loop iteration distributions primitive.

```
#pragma omp parallel for schedule(static-inverse)
    (a) Inverse static scheduling.

#pragma omp parallel for
    schedule(block-exclusive, dir, id)
    (b) Block-exclusive static scheduling.
```

Figure 12: Loop iteration scheduling primitives.

## 4.3 Implementation and portability

The proposed system-level API uses system calls to migrate memory between the processors of the system and also to set the affinity of threads to processor cores. Linux exposes these OS functionalities through the `move_pages()` and `pthread_setaffinity_np()` system calls. Most recent OSs support memory migration and affinity scheduling, and we expect that the primitives we propose are straightforward to implement on these OSs as well. Moreover, as the presented API masks from the programmer the exact way to use these facilities, programs that use the API can be compiled on several platforms with minimal modifications.

## 5. EXAMPLE: PROGRAM TRANSFORMATIONS

This section shows how two NPB programs can be transformed with the previously described API to execute with better data locality.

*bt*. Figure 13(a) shows code that is executed in the main computational loop of `bt` (for brevity we show only an excerpt of the complete `bt` program). The code has a y-wise access pattern and accesses two matrices, `lhs` and `rhs`, in the `matvec_sub()` function. There are loop-carried dependencies in the code: The computations performed on the

current row  $i$  depend on the result of the computation on the previous row  $i - 1$ .

```

for (i = 1; i < NX - 1; i++)
#pragma omp parallel for schedule(static)
  for (j = 1; j < NY - 1; j++)
    for (k = 1; k < NZ - 1; k++)
      matvec_sub(lhs[i][j][k][0],
                rhs[i - 1][j][k],
                rhs[i][j][k]);

```

(a) Original **bt** code.

```

for (i = 1; i < (NX - 2) / 2; i++) {
#pragma omp parallel for schedule(static)
  for (j = 1; j < NY - 1; j++) {
    for (k = 1; k < NZ - 1; k++) {
      matvec_sub(lhs[i][j][k][0],
                rhs[i - 1][j][k],
                rhs[i][j][k]);
    }
  }
}
for (i = (NX - 2) / 2; i < NX - 1; i++) {
#pragma omp parallel for schedule(static-inverse)
  for (j = 1; j < NY - 1; j++) {
    for (k = 1; k < NZ - 1; k++) {
      matvec_sub(lhs[i][j][k][0],
                rhs[i - 1][j][k],
                rhs[i][j][k]);
    }
  }
}

```

(b) 2-processor version of **bt** code.

```

for (p = 0; p < processors; p++) {
  for (i = p * (NX - 2) / processors;
       i < (p + 1) * (NX - 2) / processors;
       i++) {
#pragma omp parallel for
  schedule(block-exclusive, X_WISE, p)
  for (j = 1; j < NY - 1; j++)
    for (k = 1; k < NZ - 1; k++) {
      matvec_sub(lhs[i][j][k][0],
                rhs[i - 1][j][k],
                rhs[i][j][k]);
    }
  }
}

```

(c) Generalized version of **bt** code.

Figure 13: Program transformations in **bt**.

The code is transformed in two steps. At the start of the program the distribution of both matrices **lhs** and **rhs** is changed to block-exclusive using the memory distribution primitives described in Section 4.1. In the second step the outermost loop of the computation (the loop that iterates through the rows of the arrays using variable  $i$ ) is split into two halves. The transformed code is shown in Figure 13(b). Both of the resulting half-loops iterate using variable  $i$ , but the first loop processes the first  $(NX - 2) / 2$  rows of the matrices, and the second loop processes the remaining  $(NX - 2) / 2$  rows. The work distribution in the first loop is standard static block scheduling, but in the second loop inverse scheduling is required so that the affinity of the worker threads matches the data distribution. Splitting the outermost loop results in the two-phase traversal previously shown in Figure 11(d) and Figure 11(e). Note that the transformation does not break any of the data depen-

```

#pragma omp parallel for schedule(static)
for (i = 0; i < NX; i++)
  for (j = 0; j < NY; j++)
    // ...

```

(a) **lu** lower triangular part.

```

#pragma omp parallel for schedule(static)
for (i = NX - 1; i >= 0; i--)
  for (j = NY - 1; j >= 0; j--)
    // ...

```

(b) **lu** upper triangular part.

```

#pragma omp parallel for schedule(static-inverse)
for (i = NX - 1; i >= 0; i--)
  for (j = NY - 1; j >= 0; j--)
    // ...

```

(c) **lu** upper triangular part with inverse scheduling.

Figure 14: Program transformations in **lu**.

dences of the program, therefore only one synchronization operation is required for correctness (at the point where the first half of the iteration space has been processed). This program transformation can be generalized to systems with an arbitrary number of processors as shown in Figure 13(c).

The main computational loop of **bt** executes other loops as well (not just the loop shown in Figure 13(a)). Many of these other loops have a different, x-wise, access pattern. To match the access patterns of these loops to the block-exclusive data distribution of the program's data, the code must be transformed in a manner similar to the transformations shown in Figure 13(b) and 13(c). These transformations require programmer effort, but at the end the access patterns of all loops of the program match the blocked-exclusive distribution of shared data, and data distribution is required only once (at the start of the program).

**lu**. The **lu** program solves the Navier-Stokes equations in two parts: a *lower* (Figure 14(a)), and an *upper* part (Figure 14(b)). Both parts have an x-wise access pattern, therefore the memory regions used by the program are initialized x-wise. However, as the upper triangular part (Figure 14(b)) traverses rows in descending order of row numbers, the static scheduling clause of OpenMP distributes loop iterations between worker threads so that each worker thread operates on remote data. To increase data locality we match the access pattern of **lu** to its data distribution by using the **static-inverse** scheduling clause for the upper triangular part, as shown in Figure 14(c).

## 6. PERFORMANCE EVALUATION

This section presents a performance evaluation of the previously described program transformations. We evaluate the performance of the benchmark programs on the 2-processor 8-core machine described in Section 2.1 (in Section 7 we look at the scalability of program transformations onto a larger, 4-processor 32-core machine). The benchmark programs **ep**, **is**, and **mg** are excluded from the evaluation for the reasons explained in Section 2.1.

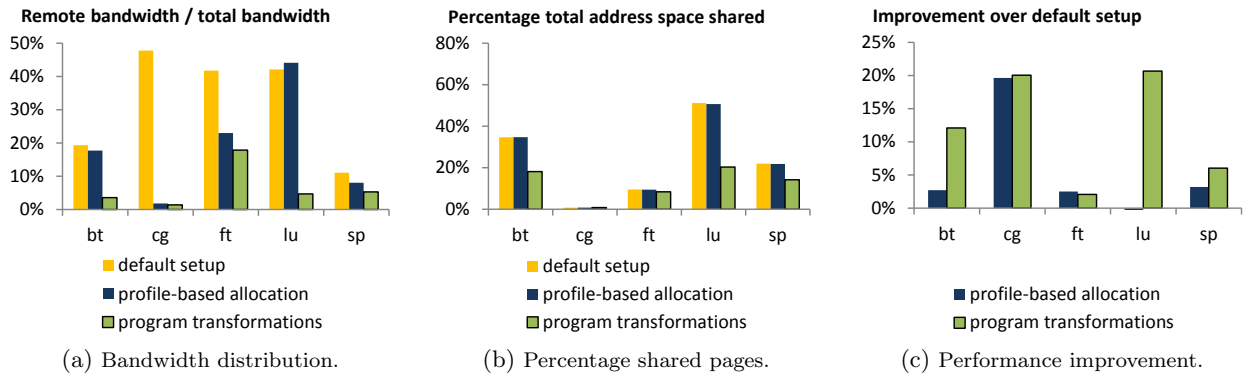


Figure 15: Performance with program transformations (2-processor 8-core machine).

The original version of `sp` is synchronization-limited due to a single triple-nested loop that is parallelized on the innermost level. We eliminate the synchronization boundedness of this loop by moving the parallelization primitive to the outermost loop. We also store data accessed by the loop in multi-dimensional arrays (instead of single-dimensional ones), so that loop-carried data dependencies are eliminated. This simple change gives a 10X speedup on our 2-processor 8-core machine; as a result `sp` is memory bound, and can profit further from data locality optimizations.

Figure 15(a) shows the percentage of remote memory bandwidth relative to the total bandwidth for each program we consider. For `bt`, `lu` and `sp` program transformations reduce the percentage of remote accesses significantly (to around 4% on average). For the `cg` benchmark profile-based memory allocation almost completely eliminates remote memory accesses (see Figure 3(b)). However, we are able to achieve the same effect without profiling by inserting data distribution primitives into the program. In `ft` most remote memory accesses are caused by a single loop. Transforming this loop is possible only by inserting fine-grained inter-thread synchronization that results in high overhead that cancels the benefits of the improved data locality. Nonetheless, we obtain a reduction of the fraction of remote memory accesses relative to profile-based memory allocation (from 23% to 18%) by distributing memory so that the distribution matches the access patterns of the most frequently executing loop of the program.

Figure 15(b) shows for each program we consider the percentage of program address space shared with the default setup, with profile-based memory allocation, and with program transformations. For three programs (`bt`, `lu`, and `sp`) program transformations reduce the number of shared pages, thus it is possible to place more pages appropriately than with the original version of the program (either by using the proposed data distribution primitives, or by profiling the program). Finally, Figure 15(c) shows the performance improvement with profile-based memory allocation and program transformations relative to the default setup. By eliminating sharing we obtain performance improvements also when profile-based allocation does not. Moreover, for `cg` (the program that improves with profile-based allocation) we are able to match the performance of profile-based allocation by distributing memory so that the distribution of

memory matches the access patterns of the program.

## 7. SCALABILITY

Adding more processors to a NUMA system increases the complexity of the system. Standard OSs and runtime systems handle the increase of the number of processors gracefully, however, if good program performance is also desired, action must be taken (e.g., by the programmer). This section evaluates the scalability of the proposed program transformations on a 4-processor 32-core NUMA machine larger than the 2-processor 8-core machine previously examined.

The benchmark programs of the NPB suite we consider are executed on a system equipped with four 8-core Intel processors based on the Westmere microarchitecture. Each processor has 16 GB directly-connected RAM and a last-level cache shared by all processor cores. In total, the system has 32 cores and 64 GB RAM. For all benchmark programs, size C (the largest size) is used.

To look at the scalability of the benchmark programs each program is executed in four different configurations (i.e., with 4, 8, 16, and 32 threads, respectively). As cache contention can cause significant performance degradations in NUMA systems [9, 2], we fix the thread-to-core affinities so that the same number of threads is executed on each processor of the system and thus the cache capacity available to each thread is maximized (remember that each processor has a single last-level cache shared between all processor cores). E.g, in the 4-thread configuration one thread is executed on each processor of the 4-processor system; in the 32-thread configuration each processor runs eight threads.

Figure 16 shows the speedup of the benchmark programs over their respective sequential version. The figure compares two version of the benchmark for each runtime configuration. The default setup relies on the first-touch memory allocation policy and uses the previously discussed thread-to-core assignment (which corresponds to identity affinity in the 32-threads configuration). With program transformations the memory distribution and the scheduling of loop iterations is changed so that data locality maximized. Performance improves with program transformations, with results that are similar to those obtained on a 2-processor 8-core system. In the 32-thread configuration we measure a performance improvement of up to 3.3X (and 1.7X on average) over the

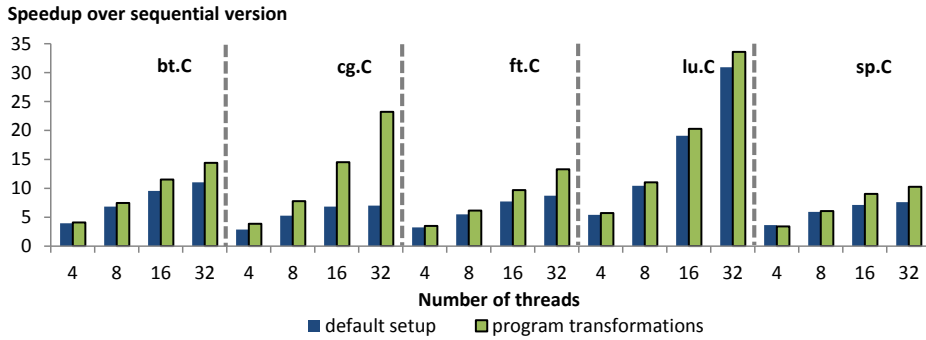


Figure 16: Performance with program transformations (4-processor 32-core machine).

default setup. Program performance also scales better with program transformations than with the default setup. (For the programs of the NAS suite studied here, the programs obtain a speedup of up to 33.6X over single-core execution, with a mean speedup of 19X.)

## 8. RELATED WORK

Several authors have noticed the problem of uniform data sharing. Thekkath et al. [16] show that clustering threads based on the level of data sharing introduces load imbalances and cancels the benefits of data locality. Tam et al. [14] schedule threads with a high degree of data sharing onto the same last-level cache. The authors admit that their method works only with programs without uniform data sharing. Verghese et al. describe a OS-level dynamic page migration [18] that migrates thread-private pages but does not consider uniformly shared pages. Therefore, their results show significant amount of remaining remote memory accesses. An approach similar to the work of Verghese et al. is described by Nikolopoulos et al. [12]. Remote memory accesses are not eliminated in this approach either. Marathe et al. [10] describe the profile-based memory placement methodology we use, but they do not discuss how uniform data sharing influences the effectiveness of their method. Tikir et al. [17] present a profile-based page placement scheme. Although successful in reducing the percentage of remote memory accesses for many benchmark programs, their method is not able to eliminate a significant portion of remote memory accesses for some programs, possibly due to uniform data sharing.

In [13] Nikolopoulos et al. claim that data distribution primitives are not necessary for languages like OpenMP, because dynamic profiling can place pages correctly. We find that even with perfect information available, runtime data migrations have too much overhead to be beneficial for programs with uniform data sharing. Bikshandi et al. [1] describe in-program data migrations in a cluster environment, however their language primitives are too heavyweight in small-scale NUMA machines. Darté et al. [4] present generalized multipartitioning of multi-dimensional arrays, a data distribution very similar to the block-exclusive data distribution. Multipartitioning, however, relies on message-passing, while we consider direct accesses to shared memory. Zhang et al. [19] take an approach similar to ours: They show that simple program transformations that introduce non-uniformities into inter-thread data sharing improve perfor-

mance on multicore architectures with shared caches. Chandra et al. [3] describe language primitives similar to ours, but they do not show how programs must be transformed to eliminate uniform data sharing. McCurdy et al. [11] show that adjusting the initialization phase of programs (so that data access patterns of the initialization phase are similar to that of the computations) gives good performance with the first-touch policy. We find that in programs with multiple phases that have conflicting access patterns, first-touch cannot place pages so that each program phase experiences good data locality.

Tandir et al. [15] present an automatic compiler-based technique to improve data placement. It is not clear how their tool distributes loop iterations that access shared memory regions and have loop-carried dependences at the same time. Kandemir et al. [7] describe an automatic loop iteration distribution method based on a polyhedral model. Their method can optimize for data locality, however in the presence of loop-carried dependences it inserts synchronization operations. We show that the number of additional synchronization operations can be kept at low if the memory distribution of the program is carefully adjusted.

## 9. CONCLUSIONS

In NUMA systems the performance of many multithreaded scientific computations is limited by uniform data sharing. In-program memory migration, a conventional method in large cluster environments, does not help because of its high overhead. In some cases it is possible to redistribute loop iterations so that the access patterns of a loop match the data distribution. However, in the presence of loop-carried dependences inter-thread synchronization reduces (or completely eliminates) the benefits of data locality.

This paper presents a simple system API that is powerful enough to allow for programming scientific programs in a more architecture-aware manner, yet simple enough to be used by programmers with a reasonable amount of knowledge about the underlying architecture. Using this API together with program transformations reduces the number of shared pages and remote memory accesses for many programs of the NPB suite. This API allows a programmer to control the mapping of data and computation and realizes a performance improvement of up to 3.3X (1.7X on average) compared to the “first-touch” policy for NAS benchmark programs. Future multiprocessors are likely to see a widening

performance gap between local and remote memory accesses. For these NUMA systems, compilers and programmer must collaborate to exploit the performance potential of such parallel systems. The techniques described here provide an approach to attempts to strike a balance between complexity and performance.

## 10. ACKNOWLEDGMENTS

We thank Albert Noll, Michael Pradel and the anonymous referees for their helpful comments.

## 11. REFERENCES

- [1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguera, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06*, pages 48–57, New York, NY, USA, 2006. ACM.
- [2] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *USENIX ATC '11*, Berkeley, CA, USA, 2011. USENIX Association.
- [3] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljkovic, and J. M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *PLDI '97*, pages 334–345, New York, NY, USA, 1997. ACM.
- [4] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *J. Parallel Distrib. Comput.*, 63:887–911, September 2003.
- [5] S. Eranian. What can performance counters do for memory subsystem analysis? In *MSPC '08*, pages 26–30, New York, NY, USA, 2008. ACM.
- [6] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA Ames Research Center, 1999.
- [7] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhang. Cache topology aware computation mapping for multicores. In *PLDI '10*, pages 74–85, New York, NY, USA, 2010. ACM.
- [8] H. Li, H. L. Sudarsan, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *ICPP '93*, pages 140–147. CRC Press, Inc, 1993.
- [9] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ISMM '11*, pages 11–20, New York, NY, USA, 2011. ACM.
- [10] J. Marathe, V. Thakkar, and F. Mueller. Feedback-directed page placement for cc-NUMA via hardware-generated memory traces. *J. Par. Distrib. Comput.*, 70:1204–1219, December 2010.
- [11] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *ISPASS '10*, pages 87–96, 2010.
- [12] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A case for user-level dynamic page migration. In *ICS '00*, pages 119–130, New York, NY, USA, 2000. ACM.
- [13] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in OpenMP? In *Supercomputing '00*, Washington, DC, USA, 2000. IEEE Computer Society.
- [14] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07*, pages 47–58, New York, NY, USA, 2007. ACM.
- [15] S. Tandri and T. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *ICPP '97*, pages 64–73, August 1997.
- [16] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94*, pages 176–186, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [17] M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Supercomputing '04*, Washington, DC, USA, 2004. IEEE Computer Society.
- [18] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS '96*, pages 279–289, New York, NY, USA, 1996. ACM.
- [19] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *PPoPP '10*, pages 203–212, New York, NY, USA, 2010. ACM.