Communication-Aware Process and Thread Mapping Using Online Communication Detection

Matthias Diener[a,b], Eduardo H. M. Cruz[a], Philippe O. A. Navaux[a], Anselm Busse[b], Hans-Ulrich Heiß[b]

[a] *Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil*
[b] *Communication and Operating Systems Group, Technische Universität Berlin, Germany*

Abstract

The rising complexity of memory hierarchies and interconnections in parallel shared memory architectures leads to differences in the communication performance. These differences can be exploited to perform a communication-aware mapping of parallel applications to the hardware topology, improving their performance and energy efficiency. To perform the mapping, it is necessary to determine the communication behavior of the processes and threads of the application. Previous methods rely on static communication traces to detect communication, require hardware changes or support only a subset of parallelization models.

We propose CDSM, Communication Detection in Shared Memory, a mechanism that detects communication in from page faults and uses this information to perform the mapping. CDSM works on the operating system level during the execution of the parallel application and supports all parallelization models that use shared memory for communication. It does not require modifications to the applications, previous knowledge about their behavior, or changes to the hardware and runtime libraries. Experiments with the MPI, MPI+OpenMP and OpenMP implementations of the NAS parallel benchmarks, the HPCC benchmark and the PARSEC benchmark suite on a shared memory machine show that CDSM has a high detection accuracy with a negligible overhead. Execution time and processor energy consumption were reduced by up to 35.9% and 18.9%, respectively (10.2% and 7.3%, on average). Experiments on a cluster system, where CDSM optimizes the communication within each node, showed an average execution time reduction of 10.4%.

Keywords: Shared memory, Parallel applications, Communication optimization, Mapping

1. Introduction

In parallel shared memory architectures, the number of cores per processor and number of processors per system are increasing [1]. This leads to new challenges for the memory and interconnection subsystems, increasing their complexity and introducing a hierarchy for the memory requests from cores to the memory. With this hierarchy, memory accesses can be divided into different groups according to their performance. Accesses to local processor caches or local non-uniform memory access (NUMA) memories have a higher performance than accesses to remote caches or memories. For this reason, improving the *locality* of memory accesses is an important design goal for operating systems and applications, as it can increase the performance and energy efficiency of computer systems [2, 3].

Parallel applications need to communicate in order to perform their tasks. This communication can be *explicit*, by using dedicated functions to send and receive data, or *implicit*, through memory accesses to shared memory segments. Programming models for explicit communication include the Message Passing Interface (MPI), while OpenMP and Pthreads use implicit communication. As communication via shared memory has a lower overhead, many common implementations of MPI provide extensions that enable faster intra-node

Email addresses: mdiener@inf.ufrgs.br (Matthias Diener), ehmcruz@inf.ufrgs.br (Eduardo H. M. Cruz), navaux@inf.ufrgs.br (Philippe O. A. Navaux), anselm.busse@tu-berlin.de (Anselm Busse), hans-ulrich.heiss@tu-berlin.de (Hans-Ulrich Heiß)

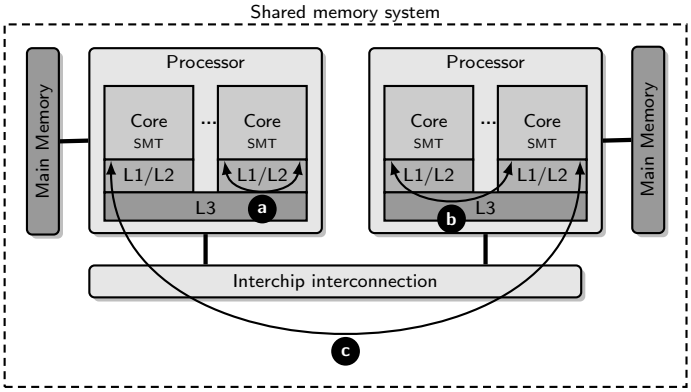


Figure 1: Architecture of a shared memory system with three communication possibilities between two processes *a*, *b* and *c*.

communication through the use of shared memory, such as Nemesis [4] for MPICH2 [5] and KNEM [6] for Open MPI [7]. To exploit several levels of interconnections more efficiently, hybrid applications combine several paradigms, such as MPI and OpenMP [8, 9]. As the communication performance varies greatly on modern system architectures, using the communication pattern to map processes and threads to cores is important to increase performance and save energy. We call this type of mapping *communication-aware mapping*.

To illustrate how the memory hierarchy can affect the communication performance, consider the shared memory architecture shown in Figure 1, which consists of two NUMA nodes. Each node contains a multicore processor that also supports simultaneous multithreading (SMT). Two processes can communicate in three ways in this architecture. They are labeled a, b and c in the figure. a) Processes that are executing on the same core on the SMT siblings can communicate via the fast L1 and L2 caches; b) Processes that are executing on different cores of the same processor have to communicate via the slower L3 cache, but can still benefit from the fast intra-chip interconnection; c) Processes that are executing on different processors need to communicate via the slow off-chip interconnection and have the lowest communication performance.

Communication-aware mapping in shared memory architectures has three benefits. First, it improves the efficiency of the interconnections, reducing inter-chip traffic that has a higher latency and lower bandwidth than intra-chip interconnections. Second, it reduces the number of cache misses of parallel applications. In a read-read situation, where two processes read the same data, the mapping reduces the replication of data in different caches, thereby increasing the cache space available to the application [10]. In the read-write or write-write case, an optimized mapping additionally reduces the number of cache lines that need to be invalidated in case of a write operation. Third, communication-aware mapping improves the memory locality on NUMA machines by keeping communicating processes on the same NUMA node.

In this paper, we propose *Communication Detection in Shared Memory (CDSM)*, a mechanism to perform communication-aware mapping in shared memory architectures. Related work in this area uses communication traces from previous executions to perform the mapping [11, 12, 13]. These mechanisms can not be used if the application changes its communication behavior with different inputs or a different number of processes. Most mechanisms for MPI-based applications focus on optimizing inter-node traffic in cluster environments, but do not improve the mapping inside each node [14, 15, 16]. Many proposals for mapping in shared memory systems use hardware counters that are highly hardware dependent [17], provide only indirect information about the communication behavior [18], or require changes to the hardware [19]. Finally, some related mechanisms only work for applications with specific parallelization models, such as Forest-GOMP [20] for applications based on OpenMP. Our proposal overcomes these limitations. Although CDSM performs mapping in shared memory architectures, it also supports programming models for distributed memory, such as MPI, as long as they use shared memory to communicate.

Since communication is usually performed through memory accesses in shared memory platforms, these accesses can be used to detect the communication between processes and threads. CDSM gathers information

about the memory access behavior of parallel applications and uses this information to map processes and threads to processing units according to their communication behavior. Communication is detected by analyzing the page faults caused by the parallel applications on the operating system level. Page faults in the same memory block caused by different processes or threads are considered as communication. The detected communication pattern is used together with information about the hardware topology to calculate the mapping with the objective of optimizing the communication of the application. We use an efficient mapping algorithm based on graph partitioning. CDSM performs communication-aware mapping completely online, without any previous information about the application's behavior. It also requires no modifications to applications, runtime libraries, or the hardware.

CDSM represents an extension of our previous work [21], which was limited to multi-threaded applications whose communication behavior does not change during execution. We extended it with support for multi-process parallel applications, such as applications based on MPI, and added techniques to detect dynamic behavior during execution. Furthermore, we reduced the overhead due to more efficient page fault handling routines.

To prove that our approach works, we implemented CDSM as a module for the Linux kernel and performed experiments using five sets of parallel benchmarks: three implementations of the NAS Parallel Benchmarks (NPB) (the MPI [22], hybrid MPI+OpenMP [23] and OpenMP [24] implementations), the HPC Challenge benchmark suite (HPCC) [25], and the PARSEC benchmark suite [26]. Experiments on a shared memory system show a high detection accuracy with a low overhead, and substantial improvements in performance and energy consumption that are close to an Oracle mechanism. Experiments on a cluster system, where CDSM optimizes the communication within each cluster node, and with the BRAMS weather prediction model [27] show that CDSM is an efficient and practical solution for communication-aware mapping.

The rest of the paper is organized as follows. The next section presents CDSM, our proposed mechanism to detect communication in shared memory architectures. The mapping algorithm used by CDSM is presented in Section 3. Section 4 describes how CDSM operates during the execution of a parallel application. Evaluation methodology and results are presented in Section 5. We discuss related work in Section 6 and compare it to CDSM. Section 7 summarizes conclusions and outlines future work.

2. CDSM: Communication Detection in Shared Memory

This section describes how CDSM detects the communication of parallel applications. We begin by explaining the general concept of the detection, followed by a description of the implementation in the Linux kernel.

The only requirement of CDSM is that the hardware and OS use virtual memory with paging. When an application causes a page fault, the hardware notifies the OS, which updates the page table of the process that caused the fault. It is important to note that threads of the same process share a common page table in many OS. Different processes have separate page tables. Processes and threads are handled similarly by CDSM. To simplify the explanation, we refer to both of them as processes in the paper and will only refer to threads when they are handled differently from processes.

2.1. Concepts of CDSM

CDSM is based on two fundamental concepts. First, memory access behavior is analyzed through page faults of the parallel application. Second, detection accuracy is increased through insertion of extra page faults during execution. An overview of the operation of CDSM is shown in Figure 2.

2.1.1. Analyzing Memory Access Behavior Through Page Faults

Since communication is usually performed through memory accesses in shared memory architectures, it is necessary to analyze these accesses in order to determine the communication behavior. Memory accesses are normally handled directly by the hardware, without any knowledge of the OS. However, whenever a memory address is accessed and its corresponding page is set as not present or invalid in the page table, the

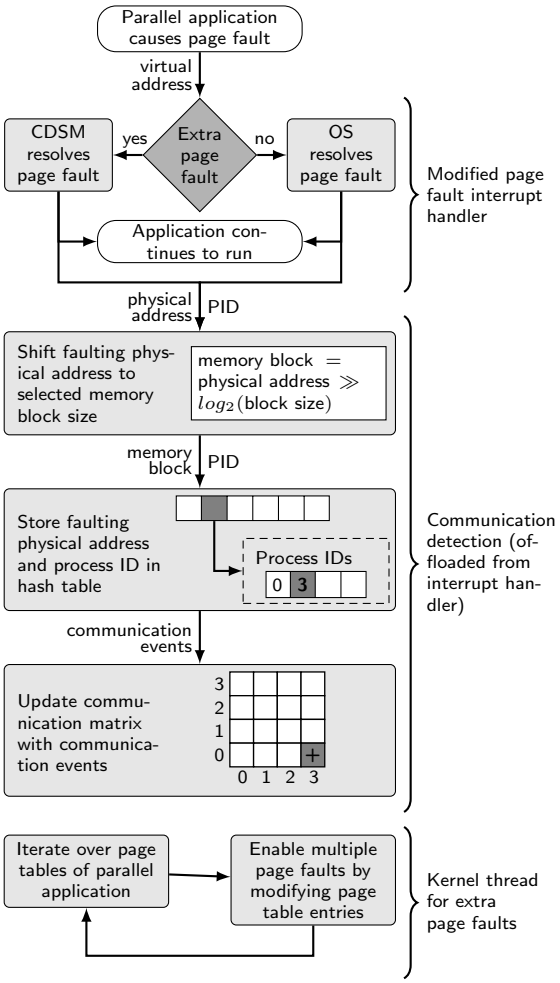


Figure 2: The communication detection mechanism.

OS is notified of the faulting address. We make use of these page faults for the communication detection. The mechanism works as follows.

Whenever a page fault is caused by the parallel application, CDSM stores the ID of the process that generated the page fault in a hash table that is indexed by the physical memory address that caused the fault. It is necessary to use the physical address instead of the virtual address since processes occupy different virtual address spaces. Since the full address is available during the page fault, communication can be detected using different granularities. CDSM separates the memory into *memory blocks* of an adjustable block size. In this way, false sharing is reduced and the communication detection is independent from the page size that the hardware uses. The memory block is calculated with Equation 1, where $\gg$ represents the bit shift operator.

$$memory\ block = physical\ address \gg log_2(block\ size) \tag{1}$$

For each memory block, we maintain a small list with the PIDs that most recently accessed the block. The list has a default size of 4 elements. When another process causes a page fault in the same memory block, we add the PID to the list and consider this fault a *communication event*. In case the list is full, we remove the oldest entry and replace it with the new PID. The amount of communication events of the

parallel application is stored in a *communication matrix*. Each cell (x, y) of the matrix contains the number of communication events between processes x and y.

2.1.2. Increasing Detection Accuracy by Enabling Multiple Page Faults per Memory Page

An important aspect of the detection mechanism is that, under normal circumstances, each process causes only one page fault per page, at the first access to the page. To increase the accuracy of the detection and to detect changes in the communication behavior during the execution, CDSM enables extra page faults, such that more than one fault can happen in the same page. CDSM periodically iterates over the page tables of the parallel application and modifies their entries.

The page table entries can be modified in different ways. A generic solution is to clear the page present bit, which is available in most architectures that support paging. In some architectures, such as x86_64, it is possible to modify reserved bits in the page table entry, which leads to a page fault that is easier to resolve. In any case, it is necessary to remove the page table entry from the translation lookaside buffer (TLB) after it is modified. As the extra page faults do not represent missing information in the page table, they can be resolved quickly by CDSM itself without intervention from the normal kernel routines. In this way, the overhead of these extra page faults is minimized. More details about where extra page faults are inserted will be given in Section 2.3.

2.2. Example of the Operation of CDSM

Figure 3 shows an example of the communication detection for a parallel application consisting of 4 processes. On application start, CDSM allocates data structures for the new application (a hash table and a communication matrix). After the application starts to execute, process 0 tries to access a memory address in a page that was never accessed before and generates a page fault. CDSM calculates the memory block of the address by bit shifting the address with the number of bits that correspond to the block size, and adds process 0 to the list of processes that accessed this block. As the block has not been accessed before, the page fault does not represent a communication event and it is resolved by the kernel.

During the execution, CDSM enables extra page faults for the same page in the page table of process 3. The next time process 3 tries to access the page, it will therefore generate an extra page fault. As the memory block has been accessed before, CDSM will count this fault as a communication event and increment the communication matrix in cell $(0, 3)$, which corresponds to process IDs 0 and 3. This situation is also depicted in Figure 2. CDSM is responsible for this extra page fault and resolves it by removing the modification to the page table. It then returns directly to the application, without calling the page fault handler of the OS.

2.3. Implementing CDSM in the Linux Kernel

CDSM was implemented as a Linux kernel module for the x86_64 architecture. The implementation consists of four parts: (1) data structures to store the communication behavior, (2) a modification to the page fault interrupt handler, (3) the communication detection, which is offloaded from the interrupt handler, and (4) a kernel thread that enables extra page faults. The description follows the sequence shown in Figure 2.

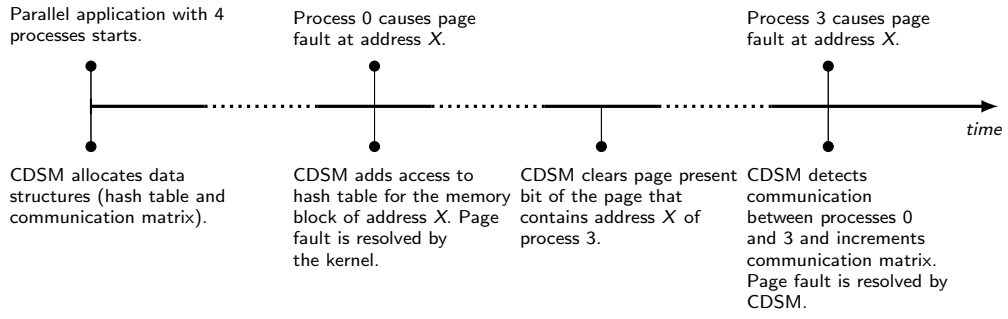


Figure 3: Example of the communication detection mechanism for a parallel application consisting of multiple processes.

2.3.1. Data Structures

For each parallel application, two data structures need to be allocated: A hash table, where CDSM stores information about the page faults and the process IDs that caused them, and a communication matrix to store the number of communication events between processes. The size of the hash table scales linearly with the size of the physical main memory. For each GByte of main memory, it consumes about 2.3 MByte of memory for the configuration used in our experiments (see Section 4). We allocate a communication matrix that is sufficient to store the number of communication events for up to 512 processes. In case the application creates more than 512 processes or threads, we dynamically grow the matrix. Each matrix cell is 4 Bytes wide, for a total memory consumption of 1 MByte per parallel application in case the size of the matrix does not need to be increased.

2.3.2. Analyzing Page Faults

The default page fault behavior of the kernel was changed by adding a kprobe [28] to the page fault interrupt handler. When a parallel application causes a page fault, CDSM determines in the modified handler if it was responsible for the page fault, and resolves the fault itself or lets the normal kernel routines handle it. CDSM then schedules a work item in the kernel work queue. The item consists of the process ID that caused the fault and the physical address of the fault. In the Linux kernel, even threads of the same application have different process IDs. After scheduling the work item, the parallel application continues to run.

The work item is analyzed by the communication detection routine. In the routine, the physical address is bit shifted to the chosen memory block size. With the shifted address, CDSM accesses the hash table and updates the list of process IDs that accessed the memory block. If the memory block has been accessed before by different processes, CDSM increments the number of communication events in the communication matrix for each pair of processes that accessed the memory block.

The handling of the page faults and the communication detection were separated for two reasons. First, it allows the parallel application to continue executing earlier, before detecting the communication. Second, it allows the communication detection to be performed during a non-critical time, such as when the application is stalled due to a lock.

2.3.3. Creating Extra Page Faults

To enable multiple faults per page during the execution of the parallel application, we create a kernel thread that periodically iterates over the page tables of the parallel application and enables extra page faults. To increase the accuracy of the detection, page faults are only enabled in virtual memory areas that can be used for communication. For multi-process applications, memory areas that can be used for communication are marked as *shared* by the virtual memory subsystem of Linux, and the kernel thread only modifies the page table for these shared areas. For multi-threaded or hybrid applications, where threads share the the same virtual address space, communication can happen in more areas, so the kernel thread enables extra page faults in larger part of the virtual memory space. In both cases, extra page faults are only enabled for data segments, not for code segments.

After identifying a suitable virtual memory area, the kernel thread enables the extra faults by setting a reserved bit in the page table entry and clears the corresponding entry in the TLB. In the x86_64 architecture, setting a reserved bit in the page table entry causes a page fault due to an invalid entry on the next memory access to that page [29]. By creating extra faults in this way, it is not necessary to access the page table to determine if CDSM caused the fault, leading to a more efficient communication detection. To limit the overhead of the mechanism, the kernel thread enables extra page faults up to a percentage of the number of pages the application accesses per second. More details about this limit will be given in Section 4.

2.4. Overhead of the Mechanism

To perform the communication detection, CDSM introduces two types of overhead to the execution of the application. The first type of overhead consists of the extra page faults, which require additional context switches to the OS as well as periodic accesses to the page tables of the parallel application to enable the

extra page faults. This overhead is reduced by keeping the amount of computation in the page fault handler low. The second type of overhead is the communication detection itself. It consists of accesses to the hash table and the communication matrix, which have a constant time complexity. In Section 5, we will evaluate the runtime overhead of the communication detection.

2.5. Further Remarks

The communication detection part of CDSM has the following features. CDSM is independent from the parallelization model, supporting any model that can use shared memory to communicate, as well as applications that use several models. CDSM detects communication online, during the execution of the parallel application, and requires no modification to the application, its runtime system or expensive operations such as tracing. Methods that rely on tracing are not able to handle applications in which the communication behavior changes between or during executions. Furthermore, changing the input data or the degree of parallelism can also lead to different communication patterns.

CDSM is also highly OS and hardware independent. It can be applied to all platforms that use virtual memory with paging, which covers most of the current computer architectures. No architecture specific information, such as a special hardware counter, is necessary. As the detection is completely dynamic, CDSM can be combined with other techniques that alter the behavior of a parallel application, such as modifying the number of processes and threads during execution [30], migrating MPI processes between cluster nodes [12] or moving memory pages between NUMA nodes [31].

CDSM is implemented as a Linux kernel module. Although installing the module requires superuser access, it does not require recompilation of the kernel or a reboot of the machine. Implementing the mapping in the operating system has several advantages compared to implementations in user space, such as in runtime libraries or directly in applications. In case different applications are executing at the same time, the user space might have only a partial view of the system state, and mapping decisions might interfere with each other. Since the kernel has a full view of the system, it can perform the mapping in such a way that all running applications are taken into account. Furthermore, mapping in the kernel allows supporting many different parallelization models as well as applications that use several models, as mentioned before. To support programming models that use user-level threads, such as Charm++[32], CDSM could be extended with a mechanism (such as a system call) to inform the kernel of the assignment of user-level to kernel-level threads. If the user-level library has a mapping mechanism, CDSM could provide information (such as the communication matrix) to help perform mapping decisions.

3. Calculating the Mapping

The information provided by the communication detection is used to calculate an optimized mapping from processes and threads to processing units (PUs) of the shared memory machine during the execution of the parallel application. Processes and threads are handled in the same way by the mapping algorithm. The mapping problem is NP-hard [33], therefore it is necessary to use efficient heuristic algorithms to calculate the mapping. Online mapping requires algorithms with a short execution time, since it directly impacts the overhead on the executing application. This section describes the mapping algorithm we use. Section 4 contains information about how the mapping is calculated during the execution of a parallel application.

The mapping problem is modeled with two undirected graphs, a communication graph and a hierarchy graph. In the communication graph, vertices represent processes and edges represent the amount of communication between them. Figure 4a shows an example communication graph for 4 processes A–D. In the hierarchy graph, vertices represent levels of the memory hierarchy and PUs, while edges represent the interconnections between them. Figure 4b depicts an example hierarchy graph for the architecture shown in Figure 1. The communication graph is obtained from the communication matrix during runtime, while the hierarchy graph is generated from the hardware topology provided by the kernel (including information about processors, cores and SMT, as well as the memory hierarchy).

To calculate the mapping, we use the dual recursive bipartitioning algorithm of the Scotch mapping library [34], version 6.0. Other algorithms have been proposed, such as Treematch [35] and Zoltan [36]. We

selected Scotch because it has a short execution time (less than 1 ms to map 32 processes) while providing good results [37, 38, 39, 13, 40], and is therefore suitable for online mapping. Scotch has a complexity of $\mathcal{O}(N^3)$, where N is the number of processes to be mapped [38]. Recent experimental results [35] show that up to 1000 processes, Scotch is the most efficient mapping algorithm in terms of execution time. If the number of processes is higher than 1000, other algorithms could be employed (such as the previously mentioned Treematch algorithm, which scales better for very large numbers of processes), or the parallel version of Scotch could be used.

Scotch seamlessly supports under/overprovisioning, that is, executing with less or more processes than the architecture can execute in parallel and does not assume one process or thread per core. The algorithm receives the communication and hierarchy graphs as input, and outputs the PU for each process such that the total cost of communication is minimized, as shown in Figure 4c. This information is then used to migrate the processes to their assigned PUs.

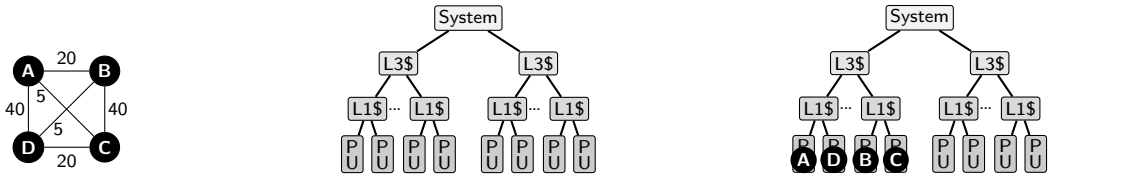
4. Operation of CDSM

This section describes how CDSM operates during the execution of a parallel application. An overview of the operation is shown in Figure 5. Table 1 summarizes the configuration parameters.

Monitor execution of parallel application. CDSM automatically detects the start and end of a parallel application, as well as its processes and threads, by monitoring the `do_fork()` and `do_exit()` kernel functions. When a parallel application starts, CDSM allocates a hash table and communication matrix, and installs a handler that detects its page faults. It also starts the kernel thread that enables multiple faults for the same page. After all processes of the application have exited, it deletes the page fault handler, frees the memory for the hash table and communication matrix and stops the kernel thread.

Detect communication. While the parallel application is executing, CDSM collects information about its page faults and generates the communication matrix, as described in Section 2. The page fault thread enables extra page faults up to a limit of 1% of the pages the application uses per second. This mechanism ensures that more page faults are created for applications with a higher memory usage. The value of 1% was chosen to provide a high level of accuracy with a negligible overhead, as will be shown in Section 5. Higher amounts of page faults did not improve the results and increased the overhead. The page fault thread wakes up periodically, every 10 ms, and performs a page table walk in the parallel application, creating extra page faults for pages that can be used for communication (with the criteria discussed in Section 2.3). The thread terminates the walk when the limit is reached or at the end of the page table. In the next iteration, the thread continues to insert faults from the position at the end of the previous walk.

Calculate mapping. The mapping algorithm is executed periodically (with the mapping interval) and calculates the new mapping using the current communication matrix, as described in Section 3. To prevent unnecessary migrations and to reduce the overhead, CDSM dynamically adjusts the mapping interval. If the calculated mapping does not differ from the previous mapping, we increase the mapping interval by 50 ms, since the mapping is stable. If the mapping differs, we divide the interval by 2. The interval is kept between 50 ms and 2 s to limit the overhead while still being able to react quickly to changes of the communication behavior.



(a) Input: Communication graph of processes A–D.

(b) Input: Hierarchy graph of the architecture shown in Figure 1.

(c) Output: Mapping of processes to the processing units of the architecture.

Figure 4: Inputs and output of the mapping algorithm.

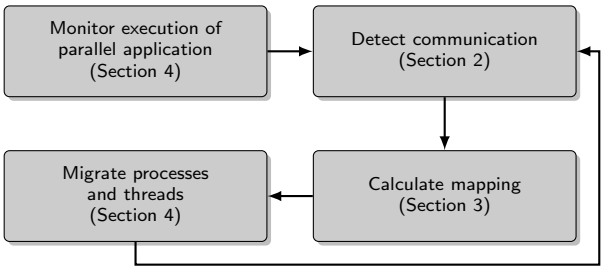


Figure 5: Overview of the operation of CDSM.

Migrate processes and threads. If the calculated mapping differs from the previously calculated one, the processes are migrated to their assigned cores using the `sched_setaffinity()` kernel function. After the migration, we apply an aging technique in the communication matrix, multiplying its values by 0.75, to detect changes in the communication behavior during execution. This operation can be performed efficiently with a subtraction and a bit shift operation: $M_{new}[i][j] = M_{old}[i][j] - (M_{old}[i][j] \gg 2)$, where i and j are the indices of the communication matrices M_{old} and M_{new}. Other values between 0.5 and 0.9 were tested as well, but the results are not sensitive to this value. With this aging technique, CDSM prevents unnecessary migrations, while still being able to handle changing communication patterns.

5. Evaluation

In this section, we describe the evaluation methodology of CDSM. Our main experimental results on a shared memory machine consist of three parts: communication patterns, performance and energy consumption. We also show results for a cluster system. Finally, we analyze the overhead imposed by CDSM.

5.1. Methodology of the Experiments

Our main evaluation machine *Xeon32* consists of 2 Intel Xeon E5-2650 processors based on the Sandy Bridge-EP microarchitecture [41]. The processors contain eight 2-way SMT cores with private L1 and L2 caches and an L3 cache that is shared among all cores. The machine has 32 virtual cores in total. Figure 1 depicts the memory hierarchy of the architecture. We use the MPICH2 [5] framework, which uses Nemesis [4] for shared memory communication. We also verified the correct operation of CDSM with Open MPI [7]. Table 1 summarizes the details of the machine.

For the experiments, we use three implementations of the NAS Parallel Benchmarks (NPB) [22], version 3.3.1, with the *B* input size. From the pure MPI implementation, we execute all benchmarks except BT-MPI and SP-MPI with 32 processes, as the machine can execute 32 processes simultaneously. Because

Table 1: Experimental configuration.

Component	Parameter	Value
Xeon32	Processors	2x Xeon E5-2650 (Sandy Bridge-EP), 8 cores, 2-SMT
	Caches/proc.	8x 32 KByte L1, 8x 256 KByte L2, 20 MByte L3
	Memory	32 GByte DDR3-1600, 4 KByte page size
CDSM	Detection	256 Bytes mem. block size, 4 recent PIDs per block, 16 bit PIDs
	Hash table	4 Million elements, 73 MByte memory usage
	Page faults	max. 1% of pages/second extra, 10 ms interval
	Mapping	dynamically adjusted interval, 50 ms – 2 s
Benchmarks	Applications	NAS–MPI (9), –MZ (3), –OMP (10); HPCC (1); PARSEC (13)
	Input size	*Class B* (NAS), 4000*4000 matrix (HPCC), *native* (PARSEC)

BT-MPI and SP-MPI require a quadratic number of processes, they were executed with 25 processes. We use the BlackHole (BH) variant of the DT-MPI benchmark. The OpenMP implementation of the NAS benchmarks [24] was executed with 32 threads.

To evaluate the detection of communication between threads and processes, we performed experiments using the NAS MPI Multi-Zone (MZ) benchmarks [23], which are hybrid applications that use the MPI and OpenMP parallelization models. These benchmarks can be executed with a variable number of threads and processes. Three configurations were evaluated: 1 process (32 threads), 2 processes (16 threads each) and 4 processes (8 threads each). Since results of the configurations were very similar, only the evaluation with 2 processes will be presented.

The HPC Challenge benchmark suite (HPCC) [25], version 1.4.3, is a single application that consists of 16 phases with different computational and memory access characteristics. The input matrix consists of 4000*4000 elements.

The PARSEC parallel benchmark suite [26], version 2.1, contains 13 applications with a highly dynamic behavior. This dynamic behavior includes creating and stopping threads during execution, as well as allocating and deallocating memory dynamically. PARSEC benchmarks use the Pthreads and OpenMP programming models. We execute all PARSEC benchmarks with the *native* input size, which is the largest size available.

In total, 36 benchmarks with widely different behaviors were used in the main evaluation with a shared memory system. All applications were compiled for the x86_64 architecture. A summary of the benchmarks is shown in Table 1. Additional experiments with the BRAMS weather prediction model and a cluster system will be presented in Sections 5.5 and 5.6, respectively.

5.2. Communication Behavior

Two properties of the communication are important for communication-aware mapping: the communication pattern as well as the amount of communication. Both characteristics will be evaluated in this section. To be able to benefit from communication-aware mapping, a significant amount of communication is required. Furthermore, the *structure* of the communication pattern is important. We expect more gains from an improved mapping if the communication is structured, that is, if not all processes communicate the same amount with all other processes.

The communication patterns detected by CDSM on Xeon32 are shown in Figures 6–10. The x- and y-axes contain the ranks of the MPI processes or thread IDs for the benchmarks that use Pthreads and OpenMP. In the case of the Multi-Zone benchmarks, the two processes have the ranks 0 and 1, while the higher IDs correspond to the thread IDs created by each process. The matrix cells show the number of communication events between two processes or threads, as defined in Section 2.1. A darker shade indicates a higher amount of communication. The average number of detected communication events per process per second is shown in parentheses after the name of each benchmark.

5.2.1. NAS-MPI Benchmarks

Figure 6 shows the communication patterns of the NAS-MPI benchmarks. BT-MPI, LU-MPI and SP-MPI show similar communication patterns, with a large amount of communication between neighboring processes, such as processes (0,1), but also similar amounts of communication between processes that are further apart, such as processes (0,4) and (0,20) in the case of BT-MPI and SP-MPI. EP-MPI and FT-MPI show low amounts of communication between the processes, with maxima for neighboring processes. The pattern of IS-MPI shows similar amounts of communication for all pairs of processes, what we describe as an all-to-all pattern. In the DT-MPI benchmark, two pairs of processes, (16,19) and (16,31), perform the majority of communication. All other processes have a negligible amount of communication. The pairs of processes that communicate change for each execution. In CG-MPI, clusters of 8 processes communicate with each other. For example, ranks 8–15 perform a large amount of communication among themselves. MG-MPI has a pattern similar to BT-MPI and SP-MPI, with a stronger focus on the neighboring processes.

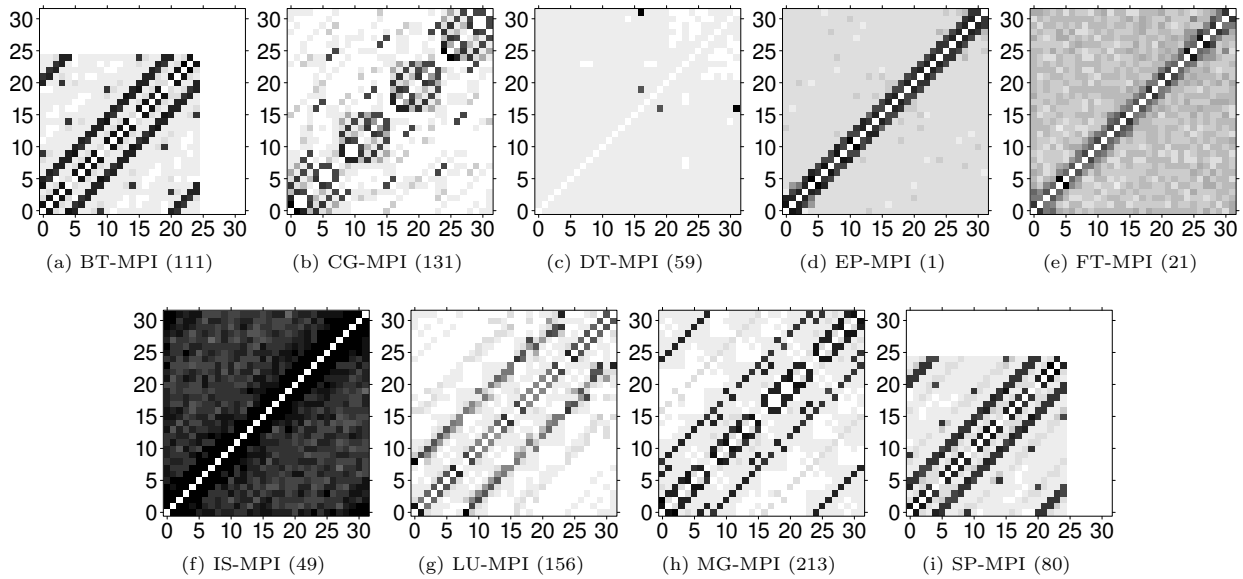


Figure 6: Communication patterns of the NAS-MPI benchmarks detected by CDSM. Axes represent process and thread IDs. Cell shades show the amount of communication events for the pairs of processes/threads, darker cells indicate more communication. The average number of detected communication events per process and second is shown in parentheses.

5.2.2. NAS-MZ Benchmarks

The communication patterns of the NAS-MZ benchmarks are shown in Figure 7. Process IDs 0 and 1 correspond to the two processes, while the higher IDs correspond to the threads created by each process. In the benchmarks, most communication is performed between the two processes and between threads that are not direct neighbors, but farther apart, such as threads (3,5). For all three benchmarks, the communication pattern changes when modifying the number of processes and threads. Additionally, the threads that communicate change on every execution of BT-MZ and SP-MZ.

5.2.3. NAS-OMP Benchmarks

The communication patterns of the NAS-OMP benchmarks are shown in Figure 8. From the benchmarks, BT-OMP and UA-OMP have lots of communication between neighboring threads. For MG-OMP and SP-OMP, the communication distance increases with increasing thread IDs. SP-OMP has the highest amount of communication of all the NAS-OMP benchmarks evaluated. CG-OMP and EP-OMP have an all-to-all pattern, with very little communication overall. DC-OMP, FT-OMP and IS-OMP have lots of communication, with a weak neighboring pattern. The communication pattern of LU-OMP shows lots of communication between distant threads, such as threads 0 and 31. It is interesting to note that even

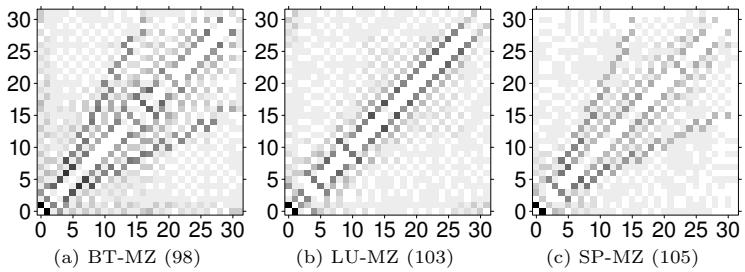
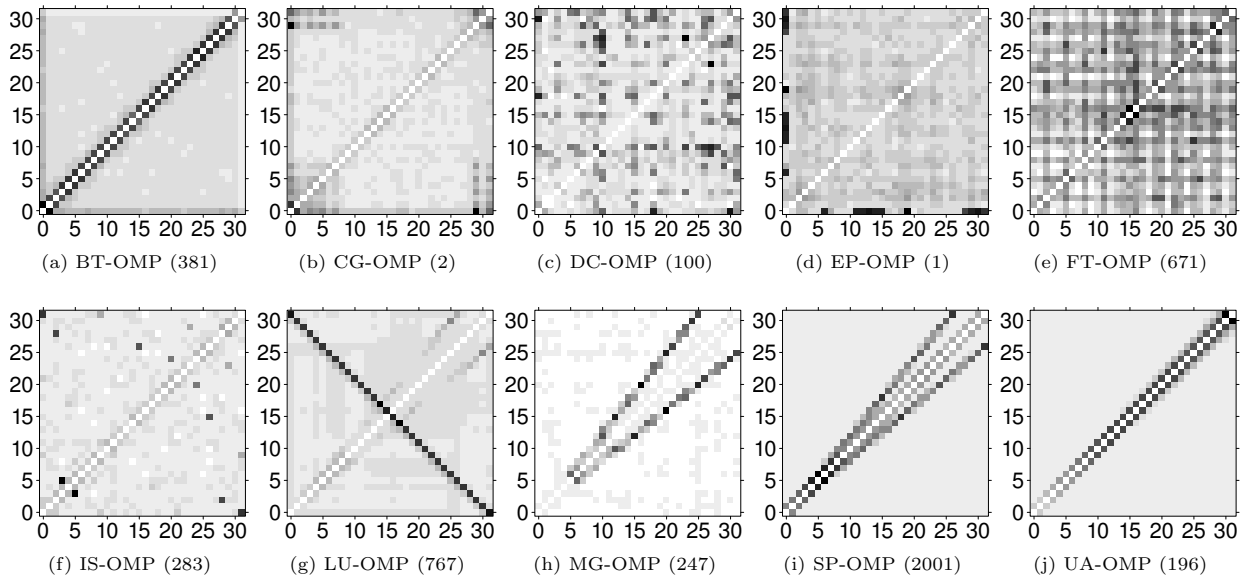


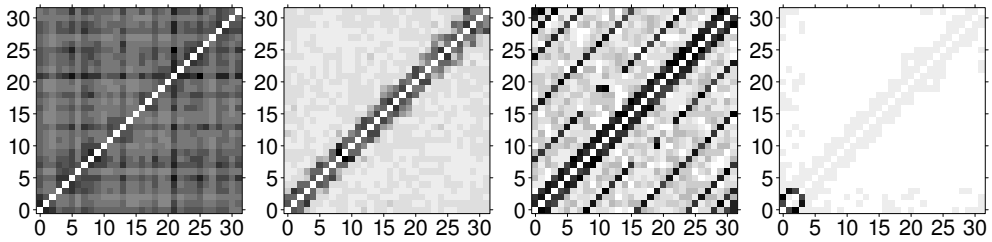
Figure 7: Communication patterns of the NAS-MZ benchmarks.



(a) BT-OMP (381) (b) CG-OMP (2) (c) DC-OMP (100) (d) EP-OMP (1) (e) FT-OMP (671)

(f) IS-OMP (283) (g) LU-OMP (767) (h) MG-OMP (247) (i) SP-OMP (2001) (j) UA-OMP (196)

Figure 8: Communication patterns of the NAS-OMP benchmarks.



(a) HPCC, phase 1 (b) HPCC, phase 6 (c) HPCC, phase 7 (d) HPCC, phase 16

Figure 9: Communication patterns of several HPCC benchmark phases.

when implementing the same algorithms, the NAS benchmarks have different communication patterns. For example, the communication behaviors of BT-MPI, BT-MZ and BT-OMP are completely different.

5.2.4. HPCC

The communication patterns of several phases of the HPCC benchmark are shown in Figure 9. HPCC consists of 16 phases, each with different communication characteristics. On average, 156 communication events per process per second were detected. The patterns can be classified into four groups: all-to-all communication (phases 1, 4), communication between neighbors (phases 3, 6, 8, 10, 15), communication between neighbors and processes that are further apart (phases 2, 5, 7, 9, 11–14) and communication within a small group of processes (phase 16). This shows the highly dynamic communication behavior of HPCC.

5.2.5. PARSEC Benchmarks

Figure 10 shows the communication patterns of the PARSEC benchmarks. From the benchmarks, only Blackscholes shows a clear communication pattern between neighboring threads. Bodytrack only communicates between thread 0 and the other threads, similar to an initialization pattern. Facesim shows both communication between neighboring threads and an initialization pattern. Ferret, Dedup and Streamcluster show a pipeline pattern, where groups of threads (pipeline stages) communicate among themselves. Freqmine, Swaptions and Canneal have all-to-all communication patterns with lots of communication. X264 and Raytrace almost do not communicate. Fluidanimate and Vips show communication between pairs of threads that are not neighbors, but father apart.

5.2.6. Summary of Communication Behavior

We verified the detected communication patterns of the MPI-based benchmarks with related work that uses traditional message tracing methods [42, 43, 44], as well as our own analysis with the eztrace tool [45]. To analyze the communication of the benchmarks based on OpenMP and Pthreads, as well as the NAS-MZ benchmarks, we used a memory tracer based on the Pin Dynamic Binary Analysis Tool [46]. The comparison shows that the communication patterns were detected correctly by CDSM for all applications that were used in the evaluation.

We expect that the benchmarks with a larger amount of communication and a more structured pattern can benefit more from communication-aware mapping. For benchmarks with a static communication behavior, an offline analysis of their behavior can be performed. For applications that change their communication behavior between or during executions, such as DT-MPI, HPCC, as well as most PARSEC and NAS-MZ benchmarks, online mapping has more potential for improvements.

5.3. Performance

To evaluate the performance improvements of CDSM, we measure the execution time as well as the number of L3 cache misses with the perf tool [47] on the Xeon32 machine presented in Section 5.1. All experiments were executed 30 times. We show the average values, as well as the confidence interval for a 95% confidence level in a Student's t-distribution.

We experimented with four mapping mechanisms, the default mapping of the operating system, a round robin mapping, an oracle mapping and CDSM. The *operating system* mapping uses the default scheduler (CFS [48]) of the Linux kernel, version 3.5. It represents the baseline for our experiments. In the *round robin* mapping, neighboring threads or ranks are mapped sequentially to nearby cores in the topology, similar to a compact mapping that is available in some MPI and OpenMP frameworks [49, 50]. This mapping optimizes

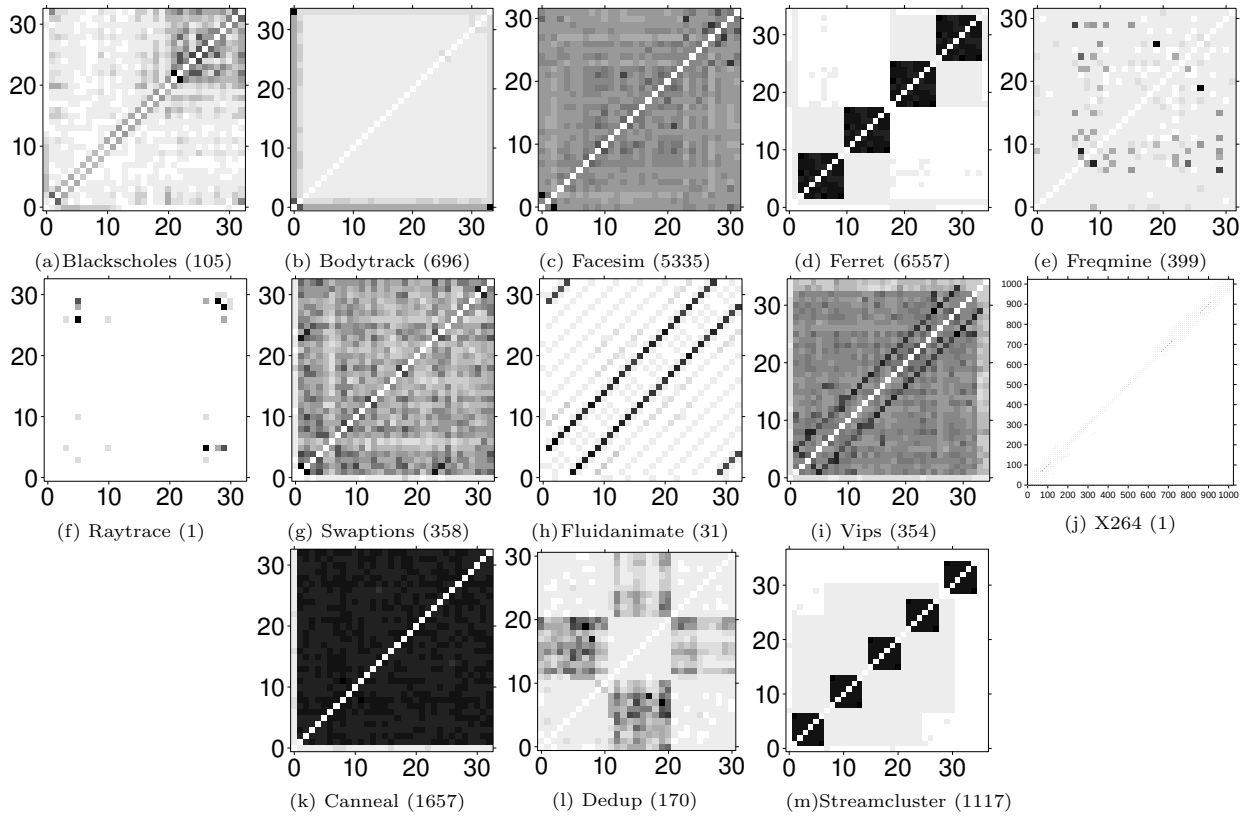


Figure 10: Communication patterns of the PARSEC benchmarks.

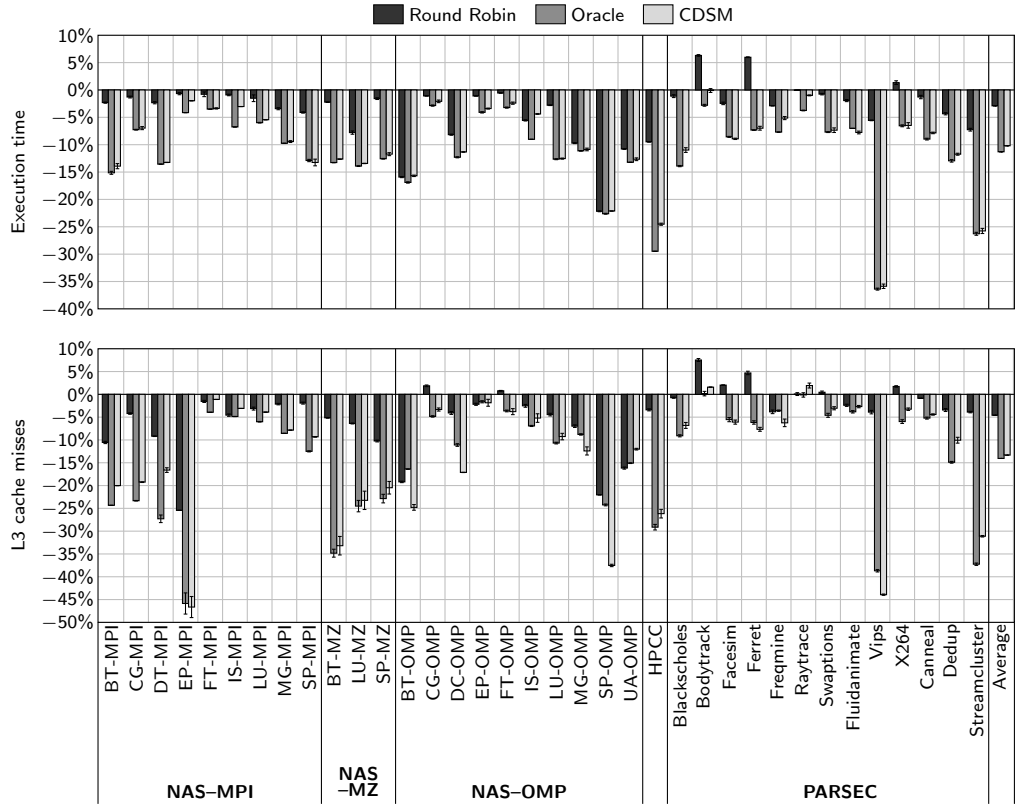


Figure 11: Performance results on Xeon32, normalized to the OS.

communication performance when most of the communication happens between directly neighboring processes. The *Oracle* mapping was generated by collecting communication traces with the help of eztrace [45] (for the benchmarks that use MPI) and a memory tracer based on the Pin dynamic binary analysis tool [46]. The traces were then evaluated to generate an optimized mapping of processes and threads to cores. We evaluated both the Scotch [34] and Treematch [35] algorithms. Since results were very close to each other (less than 1% of difference on average), we show only the results for Scotch. For applications with a dynamic behavior, the mapping is modified during runtime. This mapping mechanism optimizes the execution in terms of the communication between the processes. *CDSM* was executed with the configuration shown in Section 4.

5.3.1. NAS-MPI Benchmarks

Figure 11 shows the results for the execution time and L3 cache misses. The graphics are normalized to the result of the Linux OS, our baseline. From the NAS-MPI benchmarks, we classified BT-MPI, LU-MPI, MG-MPI and SP-MPI as having a large amount of communication between the processes. These applications show large improvements of execution time, reaching up to 13.9% in the case of BT-MPI. In DT-MPI, mapping the two communication process pairs to nearby processing units also shows significant performance improvements. CG-MPI with its cluster communication pattern shows moderate improvements. The EP-MPI and FT-MPI benchmarks, which we classified as having small amounts of communication, only improve their execution time slightly. Likewise, IS-MPI shows only small improvements because the amounts of communication between the processes are very similar.

5.3.2. NAS-MZ Benchmarks

The NAS Multi-Zone benchmarks show similar improvements of the execution time, with up to 13.4% for LU-MZ with CDSM. As LU-MZ has a static communication pattern between threads that are relatively close to each other, it has a substantial speedup even for the round robin mapping. The other two Multi-Zone benchmarks only show small improvements with this mapping.

5.3.3. NAS-OMP Benchmarks

Several NAS-OMP benchmarks, such as BT-OMP, SP-OMP and UA-OMP have large amounts of communication between the directly neighboring threads. In these cases, all three mappings produce similar performance improvements (up to 22.5% for SP-OMP). In the benchmarks where the communication is not performed by direct neighbors, but by threads that are further apart (DC-OMP, LU-OMP and MG-OMP), CDSM achieves much higher speedups than the round robin mapping. CG-OMP, EP-OMP and FT-OMP have a very low amount of communication and therefore present only small improvements. IS-OMP can not benefit from an improved mapping due its all-to-all communication behavior, and shows only small improvements as well.

5.3.4. HPCC

HPCC shows large improvements, reducing execution time by 24.5%. Due to its complex and changing communication behavior, the round robin mapping only presents very small improvements.

5.3.5. PARSEC Benchmarks

From the PARSEC benchmarks, Blackscholes shows moderate improvements of 11.0%. Considering only the parallel phase of the application, the speedup is close to 35%, since it has a very long sequential phase (about 70% of the total execution time). Raytrace, X264 and Canneal show only moderate improvements, due to their small amounts of communication and homogeneous patterns. Fluidanimate with its highly structured communication behavior shows moderate improvements despite the relatively low amount of communication. Vips showed the highest improvements of all the benchmarks evaluated, of 35.9% with CDSM. The pipeline communication patterns of Ferret, Dedup and Streamcluster show also large improvements. Facesim, Freqmine and Swaptions show relatively unstructured communication patterns and have moderate performance improvements.

5.3.6. Summary of Performance Results

It is important to note that in all cases, the improvements achieved by CDSM were close to the improvements of the Oracle mapping. Furthermore, even in applications whose communication behavior can not be exploited for mapping, the performance is not reduced. On average, execution time was improved by 10.2% using CDSM and by 11.3% using the Oracle. The round robin mapping only improves the performance slightly compared to the OS (2.9% on average), since only benchmarks with a clear communication pattern between neighboring threads (such as BT-OMP and SP-OMP) can benefit from it. Most benchmarks show more complex patterns which limit the improvements from the round robin mapping. Moreover, the round robin mapping actually reduces performance for some benchmarks, such as Bodytrack and Ferret from PARSEC.

The reason for the performance improvements can be seen in the reduction of L3 cache misses in Figure 11. On average, CDSM reduced the number of L3 cache misses by 13.3%, compared to 14.1% for the Oracle and 3.9% for the round robin mapping. Overall improvements were higher than the execution time. This is expected, as a communication-aware mapping directly improves cache usage but has only an indirect influence on the execution time, through less cache misses and improved usage of the interconnections. EP-MPI has the highest reduction of L3 cache misses (46.6%), however, the absolute number of misses is very low. The other benchmarks show improvements that are qualitatively similar to their performance improvements.

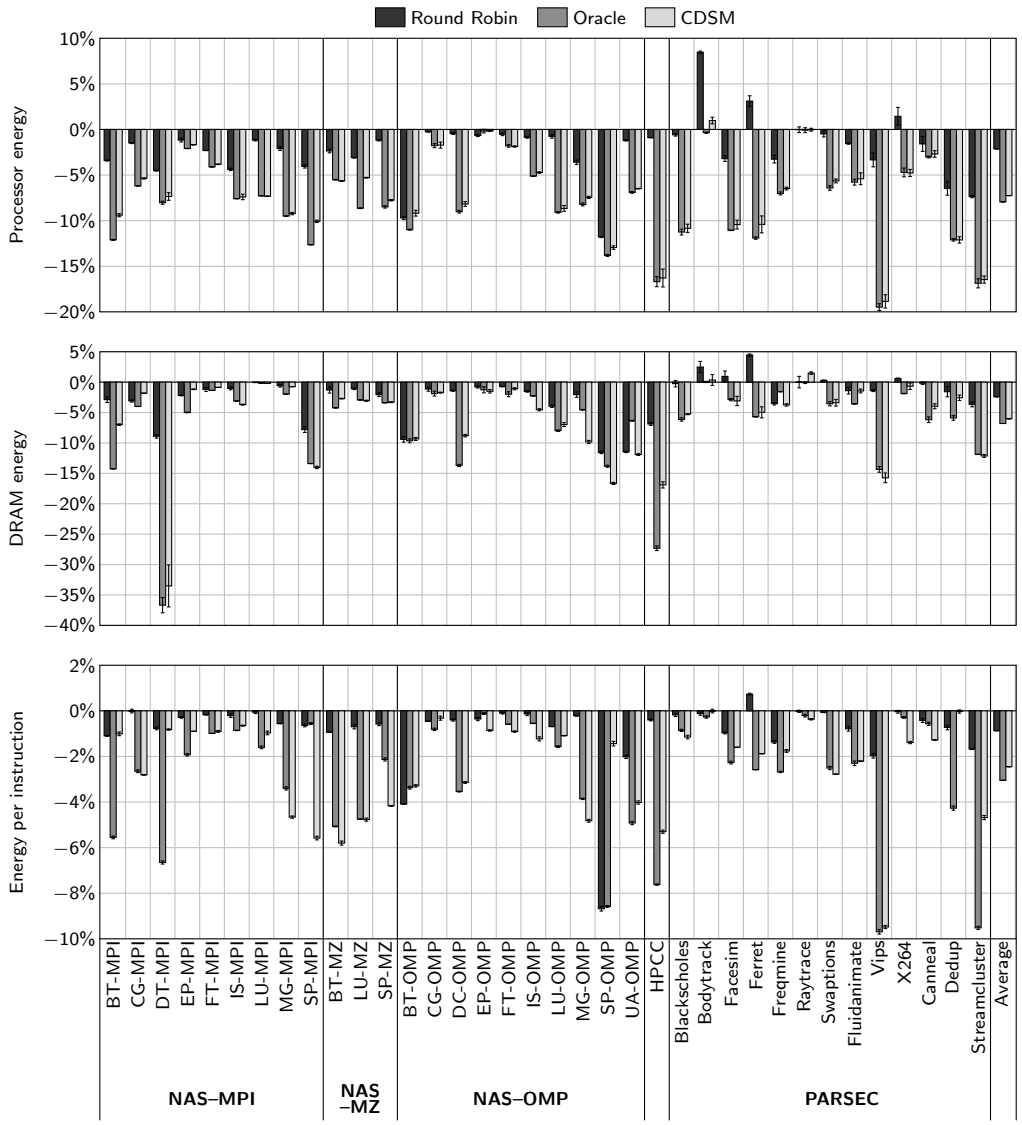


Figure 12: Energy consumption results on Xeon32, normalized to the OS.

5.4. Energy Consumption

Communication-aware mapping can reduce energy consumption of parallel applications for two main reasons. By reducing application execution time, static energy consumption (leakage) will be reduced proportionally in most cases, since the processing cores will be in a high power-consuming state for a shorter time. A shorter execution time itself does not directly affect the dynamic energy consumption, since the amount of work performed by the application will remain about the same. However, reducing the number of cache misses and traffic on the interconnections reduces the dynamic energy consumption, leading to a more energy-efficient execution of parallel applications due to improved mappings. We measure the energy consumption by using the Running Average Power Limit (RAPL) hardware counters introduced in the Intel Sandy Bridge architecture [41]. These counters were evaluated with the help of the Intel Performance Counter Monitor [51], version 2.6. We measure processor and DRAM energy consumption to evaluate the total energy reduction, and use energy per instruction as an indicator for the dynamic energy savings [52]. In the evaluation, we used the same methodology as for the performance experiments.

(a) Execution time in seconds.

(b) Execution time, normalized to the OS.

Figure 13: Performance results of BRAMS on Xeon32.

Figure 12 shows the results for the processor and DRAM energy consumption, as well energy per instruction. All results are normalized to the baseline, the OS. For all benchmarks, CDSM reduces the processor and DRAM energy consumption. Similar to the performance results, we save more energy for benchmarks with a higher amount of communication and a more structured pattern. Processor energy consumption was reduced most in the Vips benchmark, with a reduction of 18.9%. DRAM energy consumption was reduced by 33.5% in DT-MPI, which has a low absolute consumption. In SP-OMP, we save 14.0% of DRAM energy. On average, processor energy consumption was reduced by 7.3% in CDSM and 7.9% with the Oracle, while DRAM energy was reduced by 6.0% and 6.8% respectively. As before, the round robin mapping only reduced energy consumption slightly (2.1% for the processor, and 2.4% for the DRAM).

As expected, the energy per instruction reductions were lower than the total energy reductions, reaching up to 9.5% for Vips with CDSM (2.5% on average). Despite the smaller reductions, the results show that communication-aware mapping can reduce dynamic energy consumption and improve energy efficiency.

5.5. Case Study: Scientific Application

To show the operation of CDSM on a real-world scientific application, we selected the BRAMS weather prediction model [27], which is an extended version of the RAMS Regional Atmospheric Modeling System [53]. BRAMS is designed to simulate atmospheric circulations spanning from the hemispheric scale down to large eddy simulations (LES) of the planetary boundary layer[1]. BRAMS is parallelized with MPI. We evaluated BRAMS with the *light1gr* input set, and executed it on the Xeon32 system used in the previous experiments with 32 processes. All evaluation parameters, including the configuration of CDSM, are the same as in the previous experiments.

The communication pattern of BRAMS consists of two phases. In the beginning of each time step, neighboring processes communicate for about 25% of the duration. For the rest of each time step, more distant processes communicate. The global communication pattern is similar to the pattern of the LU-MPI benchmark presented in Section 5.2. The performance results for the BRAMS application are shown in Figure 13. On average, execution time was reduced by 19.1% with the Oracle and 16.3% with CDSM. Due to the dynamic behavior, the round robin mapping only improves execution time by 8.0%. The results show that CDSM not only can improve smaller benchmarks and application kernels, but also large real-world applications with longer execution times.

5.6. Evaluating CDSM with Larger Systems

We present two additional sets of results with two larger systems than our main evaluation machine: a cluster system to evaluate CDSM in a distributed-memory environment and a shared memory machine with 64 virtual cores to show the scaling behavior of CDSM. The results of these experiments are shown in this section.

[1] http://brams.cptec.inpe.br/

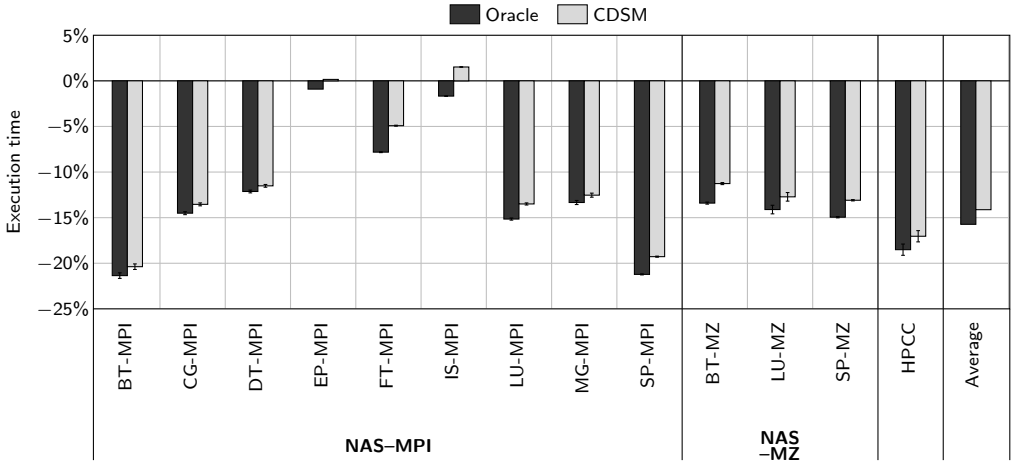


Figure 14: Execution time on the cluster system, normalized to the OS.

5.6.1. Cluster Environment

Although CDSM focuses on shared memory architectures, it can be used in distributed-memory environments (such as clusters) as well, improving the locality of memory accesses and communication in each cluster node. To show the potential of CDSM in such a distributed environment, we executed the MPI-based benchmarks (NAS-MPI, NAS-MZ and HPCC) on a cluster system. NAS-OMP and PARSEC were not evaluated as they operate in shared memory only. An overview of the configuration of the cluster is shown in Table 2. It consists of 8 cluster nodes, each with two Intel Xeon E5310 processors and 16 GByte of main memory. Each processor contains 4 cores, with a private L1 cache and two L2 caches shared between pairs of cores. We execute the NAS-MZ benchmarks with 1 process per cluster node, statically mapped to each node during initialization, and 8 threads per process. The NAS-MPI and HPCC benchmarks were executed with 1 process per core, for 64 processes in total. In all experiments (including the OS baseline), processes are statically mapped to cluster nodes. The processes and threads on each node are mapped by an unmodified version of CDSM, with the same configuration and methodology as in the previous experiments.

Figure 14 shows the execution time results for the benchmarks on the cluster. We can observe that despite the hardware differences, the results on the cluster are qualitatively similar to the results on the shared memory machine shown in the previous sections. As before, the results of CDSM are close to the Oracle mechanism, with an average execution time reduction of 13.0% for the Oracle and 11.4% for CDSM. These results show that CDSM can be used efficiently in a cluster environment.

CDSM could handle inter-node communication as well if combined with a message monitoring technique, such as MPIPP [12]. In this way, intra-node communication is detected by page faults, and inter-node communication is detected by monitoring the messages sent through the network. If another mechanism migrates processes between cluster nodes, CDSM can detect communication and remap these new processes correctly, since CDSM operates during the execution of the parallel application and can react to dynamic changes of the communication behavior.

Table 2: Hardware configuration of the cluster system.

Parameter	Value
Number of cluster nodes	8
Processors/node	2x Intel Xeon E5310 (Clovertown), 4 cores
Caches/processor	4x 32 KByte L1, 2x 4 MByte L2
Memory/node	16 GByte DDR2-667, 4 KByte page size
Interconnection	Gigabit Ethernet

Table 3: Hardware configuration of the Xeon64 system.

Parameter	Value
Processors	4x Intel Xeon X7550 (Nehalem), 8 cores, 2-SMT
Caches/processor	8x 32 KByte L1, 8x 256 KByte L2, 18 MByte L3
Memory	128 GByte DDR3-1333, 4 KByte page size

5.6.2. Large Shared Memory System

To show how the performance improvements of CDSM scale with larger shared memory systems, we run additional experiments on the *Xeon64*, which consists of 4 Intel Xeon X7550 processors with 8 cores and SMT. In total, Xeon64 can execute 64 processes or threads at the same time. Xeon64 has the same memory hierarchy as Xeon32, with L1 and L2 caches private to each core and an L3 cache shared among all cores. The machine configuration is summarized in Table 3. We run the experiments with the same methodology as for the evaluation on the Xeon32 machine, including the same configuration of CDSM, only increasing the number of processes or threads for each benchmark.

Figure 15 shows the execution time results on Xeon64, normalized to the OS. For most benchmarks, the improvements are qualitatively similar to the improvements on Xeon32, with larger improvements for applications with a structured communication pattern and large amounts of communication. Quantitatively, improvements were higher on Xeon64, especially for the PARSEC benchmarks. On average, execution time was reduced by 12.6% by CDSM and 15.2% by the Oracle, which shows that on larger machines, communication-aware mapping becomes even more important and can result in higher improvements.

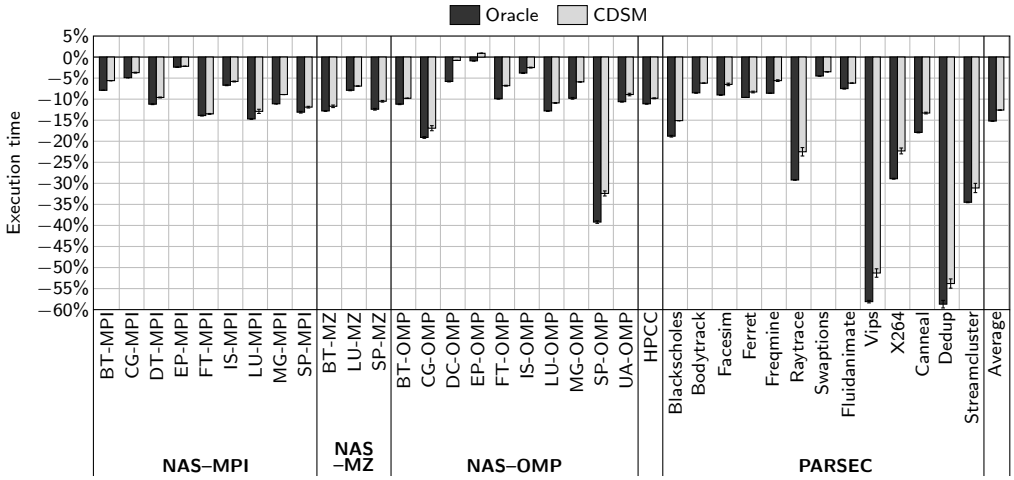


Figure 15: Execution time results on Xeon64, normalized to the OS.

5.7. Overhead of CDSM

Because it operates during the runtime of an application, CDSM imposes an overhead on the execution. The overhead can be classified into three groups, the communication detection (including the extra page faults), the calculation of the mapping and the migration of the threads and processes.

The communication detection consists of accesses to the hash table and accesses to the page tables of the application to enable multiple page faults per page. These extra page faults present an overhead for a small part of the memory accesses by the application. However, extra page faults were generated for less than 0.0001% of memory accesses of each application, corresponding to less than 15% extra faults for each application during execution. The communication detection overhead was measured by statically mapping each application, and executing it only with the communication detection part of CDSM, without the mapping algorithm and the migration.

The mapping overhead consists of the repeated execution of the mapping algorithm during the execution of the application. It was evaluated by measuring the time spent in the mapping function. The migration overhead consists of a small increase in cache misses for the process after the migration. The impact on the execution time was less than 0.1%, and is therefore not included in the discussion here.

The communication detection and mapping overheads on Xeon32 are shown in Figure 16. The values are the percentage of execution time of each application. For all benchmarks, the communication detection and mapping overhead is less than 0.5%. The average overhead of the communication detection is 0.34% and 0.23% for the mapping, for a total average overhead of 0.57%. On Xeon64, the communication detection overhead remained the same, with an average of 0.41%, since the amount of extra faults per process remains constant. The average mapping overhead increased to 0.62%, since more processes and threads need to be mapped. The total average overhead on Xeon64 remains low, at 1.03%.

6. Related Work

Several related mechanisms that focus on thread and process mapping considering the communication have been proposed. In this section, we analyze these mechanisms and compare them to our proposed approach.

6.1. Mapping of MPI-Based Applications

To map parallel applications that use explicit communication, most previous proposals make use of information gathered from MPI primitives to generate a communication pattern and make a static assignment from processes to cluster nodes and cores. The mapping is usually performed in two steps. In the first step, information about the communication behavior is gathered through traces, source code analysis or other

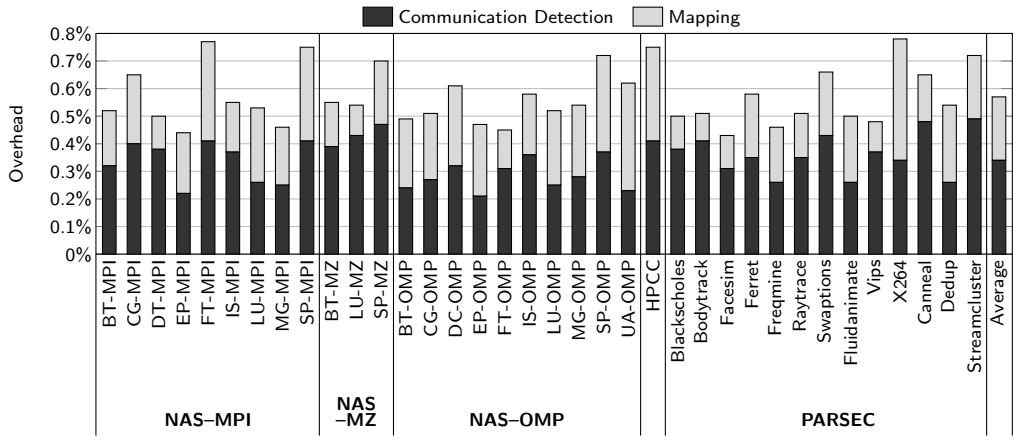


Figure 16: Overhead of CDSM on Xeon32 in % of the total execution time of each application.

methods and a mapping algorithm calculates an optimized static mapping. This information is then used in subsequent executions to assign processes to cores. Most previous mapping techniques for MPI-based applications focus on mapping in distributed-memory environments to optimize the network bandwidth and latency between cluster nodes. These techniques have little awareness of the memory hierarchy and effects of mapping within the nodes. As we have shown in this paper, mapping within cluster nodes is also an important issue and leads to significant performance and energy efficiency improvements.

Mapping improvements in cluster environments focus on reducing message latency, by minimizing the number of hops in the network that each message needs to travel, or by maximizing network bandwidth utilization by using as many links as possible [54]. A large amount of research exists in this context. Subramoni et al. [14, 15] improve communication latency with the help of a network discovery library for the InfiniBand interconnect, increasing the amount of intra-node communication and reducing communication between nodes. However, the mapping within each node is not optimized. As shown in our experiments, a large amount of intra-node communication provides higher possibilities for mapping improvements in each node. Information about the communication pattern is gathered from previous runs of the applications and hints from the application itself.

Rubik [54] is a tool to generate mappings on torus networks, with a focus on increasing network bandwidth and link utilization for applications that use communication collectives over sub-communicators. As before, the communication pattern needs to be known before execution, and the mapping does not take into account the memory hierarchy within the nodes. Agarwal et al. [55] introduce a process mapping technique to reduce network contention by minimizing the amount of hop-bytes communicated. It is implemented as an extension to the Charm++ programming framework, which provides information about the communication behavior. Since Charm++ splits the parallel application into a large number of communicating objects, performance can be increased significantly on large architectures through the mapping. However, parallel applications need to be written specifically for Charm++.

To detect communication in MPI-based applications, many proposals trace MPI messages between processes, using frameworks such as the MPI Parallel Environment (MPE) [56], eztrace [45] and the Intel Trace Analyzer and Collector [57]. In [11], the communication pattern can be either provided by the programmer, or be automatically detected by monitoring the MPI messages of the parallel application. MPIPP [12] also uses MPI communication traces to perform a static mapping of MPI ranks to cores in subsequent executions. Similar techniques are used in [13, 58]. Process mapping of MPI based applications is also evaluated in [16, 40, 39, 59], with the assumption that the communication pattern is previously known. These mechanisms utilize MPI primitives and are therefore not a generic solution for process and thread mapping.

Ito et al. [60] propose a mapping solution for parallel applications that use domain decomposition. MPI ranks that access the same sub-domains are mapped close to each other. Their mechanism requires detailed knowledge about the behavior of the application. To the best of our knowledge, no MPI-based mechanism currently supports online migration of processes between cores due to a changing communication behavior. As CDSM needs to previous information about application behavior and performs the mapping during the execution, it can be easily combined with techniques that optimize the assignment of processes to cluster nodes and can optimize the mapping of processes and threads within each node.

6.2. Mapping of Multithreaded Applications

Previous work that focuses on implicit communication in shared memory architectures generally uses indirect information about memory accesses to perform the mapping. Most approaches are limited to applications based on OpenMP.

Our previous work [21] also uses page faults to determine the communication behavior of parallel applications. This work is limited to multithreaded applications, that is, parallel applications that share a single page table and does not support MPI applications. Furthermore, it is less suitable for mapping applications with a dynamic memory access behavior due to the static mapping interval and the lack of an aging strategy. Additionally, the way that page faults are created and evaluated was much less efficient, limiting the accuracy of the mechanism and increasing its overhead. The ForestGOMP library [20] integrates

into the OpenMP runtime environment and gathers communication statistics from hardware performance counters. The hardware counters the authors used to guide the thread mapping only indirectly estimate the communication patterns. Autopin [18] automatically selects the best mapping from a previously generated set of mappings. It requires the external generation of the mappings using other mechanisms, such as communication traces. Autopin evaluates hardware statistics, such as IPC and cache misses, to select the best mapping from the provided mappings during runtime.

Azimi et al. [17] schedule threads on the IBM Power5 processor by monitoring memory accesses that are resolved by cache memories located on remote processors. Memory accesses that are resolved by local cache memories or the main memory are not considered when detecting the communication, thereby generating an incomplete communication pattern. Cruz et al. [19] use the translation lookaside buffer (TLB) to detect the communication patterns of parallel applications by searching all TLBs for matching entries during execution and use the collected information to perform a static mapping of threads to cores. This mechanism is limited to detecting communication on the page granularity, and most current processor architectures require hardware modifications to be able to use it. Furthermore, the mechanism requires a shared page table and therefore does not support multi-process applications.

6.3. Mapping Algorithms

Calculating a mapping from a given communication pattern is a well-studied problem, and several mapping algorithms to optimize communication have been proposed. Most algorithms are based on graph partitioning and were originally developed for MPI-based applications, such as METIS [61], Zoltan [36], Scotch [34] and the topology framework for MPI/SX [62]. Some of the previously mentioned mapping mechanisms also include their own custom mapping algorithms [19, 12, 40]. In sparse node allocation scenarios, where the nodes assigned to an application are not allocated contiguously, Deveci et al. [63] propose a mapping algorithm based on geometric partitioning. As mentioned in Section 3, we selected Scotch due to its good results with a short execution time, which makes it suitable for online mapping.

TreeMatch [35] focuses on mapping in shared memory architectures, where the memory hierarchy can be represented as a tree instead of a generic graph. For the number of processes and threads used in this paper, Treematch has a higher runtime overhead than Scotch [35] and is therefore less suitable for online mapping.

6.4. Comparing CDSM to Related Work

We compared CDSM to two of the previous mechanisms that were presented in this section. For the MPI-based applications (NAS-MPI and HPCC), we used the MPIPP toolkit [12]. MPIPP uses MPI communication traces to perform a static mapping of MPI ranks to cores. For the benchmarks with a static communication behavior, MPIPP is similar to our Oracle mapping for the MPI-based benchmarks, using only a different mapping algorithm. Processes are not migrated during execution however.

For the multithreaded applications (NAS-OMP and PARSEC), we used Autopin [18]. Autopin periodically migrates the threads of the parallel application using a set of previously generated mappings. For each application, we executed Autopin with 5 different mappings, consisting of the mapping used by our Oracle mechanism, a round robin mapping and 3 random mappings. After an initialization phase of 500ms, each of the 5 mappings is executed for 200ms. The mapping with the highest IPC over all threads is then maintained for the next 10 seconds, after which we evaluate the 5 mappings again. As we are not aware of a previous solution that handles hybrid applications, we do not use the NAS-MZ applications in the comparison.

Figure 17 shows the average execution time for the benchmarks on the Xeon32 machine. As expected, MPIPP shows similar improvements as CDSM for the NAS-MPI benchmarks. However, it is not able to fully take advantage of the dynamic communication behavior of HPCC and achieves results that are not optimal. For NAS-OMP, Autopin produces reasonably good results, due to their static communication behavior. For PARSEC, Autopin results are much worse and actually increase execution time for several benchmarks. Overall, the two related mechanisms achieved improvements of 4.4% on average, compared to 10.0% for CDSM.

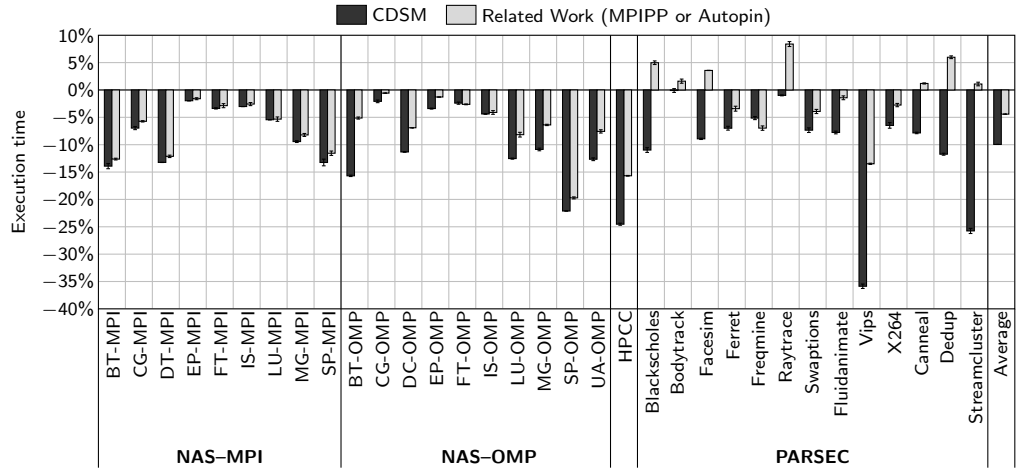


Figure 17: Application execution time when performing mapping with CDSM and related work on the Xeon32 machine. The values are normalized to the results of the OS. We use MPIPP [12] (for NAS-MPI and HPCC) and Autopin [18] (for NAS-OMP and PARSEC) as related work.

7. Conclusions

In shared memory architectures, mapping parallel applications according to their communication behavior can reduce execution time and energy consumption. As communication is performed through memory accesses, it is necessary to efficiently analyze them. We introduced CDSM, a mechanism that performs communication-aware mapping during the execution of the parallel application. It needs no modifications to the application or any previous knowledge about its behavior and works with a wide variety of parallelization paradigms that use shared memory to communicate.

Communication is detected by analyzing the page faults of parallel applications. CDSM enables extra page faults for the application to increase the accuracy of the detection. These page faults can be resolved quickly without involving the normal OS routines. The detected communication behavior is used by the mapping mechanism to calculate an optimized mapping and migrate the processes.

CDSM was evaluated with the MPI, hybrid MPI+OpenMP and OpenMP implementations of the NAS Parallel Benchmarks (NPB), the HPC Challenge (HPCC) benchmark, and the PARSEC benchmark suite. We verified that all communication patterns were detected correctly by comparing them to message and memory tracing methods. CDSM reduced execution time by up to 35.9% (10.2% on average) compared to the OS scheduler. Processor and DRAM energy consumption were reduced by up to 18.9% and 33.5%, respectively (7.3% and 6.0% on average). Improvements were close to an oracle-based mapping and significantly better than previously published mechanisms. Our results also showed that a simple round robin mapping of processes and threads to cores can not achieve optimal performance for most applications. The overhead imposed by CDSM on the applications was less than 0.8%.

Experiments with the BRAMS weather prediction model showed that CDSM is suitable for complex real-world applications, achieving performance improvements of 16.3%. Although CDSM focuses on mapping in shared memory architectures, it can be applied in distributed-memory environments, such as clusters, as well. In this case, CDSM improves communication within each cluster node, and can thereby complement techniques that migrate processes between nodes. Experiments showed that CDSM worked correctly in this scenario and achieved performance improvements of 11.4% on average.

For the future, we intend to port the mechanism to other hardware architectures and operating systems. Currently, CDSM is implemented for the Linux kernel for the x86_64 architecture. We want to evaluate the possibility of porting the mechanism to Sparc and ARM, as well as to other operating systems, such as FreeBSD.

Acknowledgments

This work was partially supported by CNPq and CAPES.

[1] K. Asanovic, B. C. Catanzaro, D. A. Patterson, K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Tech. rep. (2006).
[2] S. Borkar, A. A. Chien, The Future of Microprocessors, Communications of the ACM 54 (5) (2011) 67–77. doi:10.1145/1941487.
[3] P. W. Coteus, J. U. Knickerbocker, C. H. Lam, Y. a. Vlasov, Technologies for exascale systems, IBM Journal of Research and Development 55 (5) (2011) 14:1–14:12. doi:10.1147/JRD.2011.2163967.
[4] D. Buntinas, G. Mercier, W. Gropp, Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2006.
[5] W. Gropp, MPICH2: A new start for MPI implementations, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2002.
[6] B. Goglin, S. Moreaud, KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework, Journal of Parallel and Distributed Computing 73 (2) (2013) 176–188. doi:10.1016/j.jpdc.2012.09.016.
[7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2004.
[8] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, in: Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), no. c, 2009, pp. 427–436. doi:10.1109/PDP.2009.43.
[9] N. Drosinos, N. Koziris, Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), no. C, 2004.
[10] Z. Chishti, M. D. Powell, T. N. Vijaykumar, Optimizing Replication, Communication, and Capacity Allocation in CMPs, ACM SIGARCH Computer Architecture News 33 (2) (2005) 357–368. doi:10.1145/1080695.1070001.
[11] G. Mercier, E. Jeannot, Improving MPI Applications Performance on Multicore Clusters with Rank Reordering, in: European MPI Users' Group conference on Recent advances in the message passing interface (EuroMPI), 2011.
[12] H. Chen, W. Chen, J. Huang, B. Robert, H. Kuhn, MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters, in: International Conference on Supercomputing, 2006.
[13] G. Mercier, J. Clet-Ortega, Towards an efficient process placement policy for mpi applications in multicore environments, Recent Advances in Parallel Virtual Machine and Message Passing Interface.
[14] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, D. K. Panda, Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012. doi:10.1109/SC.2012.47.
[15] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold, D. K. Panda, Design of Network Topology Aware Scheduling Services for Large InfiniBand Clusters, in: International Conference on Cluster Computing (CLUSTER), 2013.
[16] A. Bhatele, Automating Topology Aware Mapping for Supercomputers, Ph.D. thesis (2010).
[17] R. Azimi, D. K. Tam, L. Soares, M. Stumm, Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring, ACM SIGOPS Operating Systems Review 43 (2) (2009) 56–65. doi:10.1145/1531793.1531803.
[18] T. Klug, M. Ott, J. Weidendorfer, C. Trinitis, autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems, High Performance Embedded Architectures and Compilers 3 (4) (2008) 219–235.
[19] E. H. M. Cruz, M. Diener, P. O. A. Navaux, Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2012. doi:10.1109/IPDPS.2012.56.
[20] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, R. Namyst, Structuring the execution of OpenMP applications for multicore architectures, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2010.
[21] M. Diener, E. H. M. Cruz, P. O. A. Navaux, Communication-Based Mapping Using Shared Pages, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2013, pp. 700–711. doi:10.1109/IPDPS.2013.57.
[22] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS Parallel Benchmarks, International Journal of High Performance Computing Applications 5 (3) (1991) 66–73.
[23] R. F. Van der Wijngaart, H. Jin, NAS Parallel Benchmarks, Multi-Zone Versions, Tech. rep. (2003).
[24] H. Jin, M. Frumkin, J. Yan, The OpenMP implementation of NAS Parallel Benchmarks and Its Performance (1999).
[25] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifer, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi, J. Jack, R. Rabenseifner, Introduction to the HPC Challenge Benchmark Suite (2005).
[26] C. Bienia, K. Li, PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors, in: Annual Workshop on Modeling, Benchmarking and Simulation, 2009.
[27] S. R. Freitas, K. M. Longo, M. a. F. Silva Dias, R. Chatfield, P. Silva Dias, P. Artaxo, M. O. Andreae, G. Grell, L. F. Rodrigues, A. Fazenda, J. Panetta, The Coupled Aerosol and Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System (CATT-BRAMS) Part 1: Model description and evaluation, Atmospheric Chemistry and Physics 9 (8) (2009) 2843–2861. doi:10.5194/acp-9-2843-2009.

[28] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, M. Hiramatsu, Probing the Guts of Kprobes, in: Linux Symposium, 2006, pp. 101–116.
[29] Intel, Intel 64 and IA-32 Architectures Software Developer's Manual (2013).
[30] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, D. S. Nikolopoulos, Hybrid MPI/OpenMP Power-Aware Computing, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2010. doi:10.1109/IPDPS.2010.5470463.
[31] J. Marathe, V. Thakkar, F. Mueller, Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces, Journal of Parallel and Distributed Computing 70 (12) (2010) 1204–1219.
[32] L. V. Kale, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based On C++, in: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1993, pp. 91–108.
[33] S. Bokhari, On the Mapping Problem, IEEE Transactions on Computers C-30 (3) (1981) 207–214.
[34] F. Pellegrini, Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs, in: Scalable High-Performance Computing Conference (SHPCC), 1994, pp. 486–493.
[35] E. Jeannot, G. Mercier, F. Tessier, Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques, IEEE Transactions on Parallel and Distributed Systems 25 (4) (2014) 993–1002. doi:10.1109/TPDS.2013.104.
[36] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, U. V. Catalyurek, Parallel hypergraph partitioning for scientific computing, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006. doi:10.1109/IPDPS.2006.1639359.
[37] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, W. Gropp, Multi-core and Network Aware MPI Topology Functions, in: Recent Advances in the Message Passing Interface, 2011.
[38] T. Hoefler, M. Snir, Generic topology mapping strategies for large-scale parallel architectures, in: International Conference on Supercomputing (ICS), 2011. doi:10.1145/1995896.1995909.
[39] E. R. Rodrigues, F. L. Madruga, P. O. A. Navaux, J. Panetta, Multi-core aware process mapping and its impact on communication overhead of parallel applications, in: IEEE Symposium on Computers and Communications (ISCC), 2009. doi:10.1109/ISCC.2009.5202271.
[40] B. Brandfass, T. Alrutz, T. Gerhold, Rank reordering for MPI communication optimization, Computers & Fluids (2012) 372–380doi:10.1016/j.compfluid.2012.01.019.
[41] Intel, 2nd Generation Intel Core Processor Family, Tech. Rep. September (2012).
[42] J. Zhai, T. Sheng, J. He, Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications, IEEE Transactions on Parallel and Distributed Systems 22 (11) (2011) 1862–1870.
[43] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, S. See, An Approach for Matching Communication Patterns in Parallel Applications, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2009.
[44] R. Riesen, Communication Patterns, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006.
[45] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, J. Dongarra, EZTrace: a generic framework for performance analysis, in: International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2011. doi:10.1109/CCGrid.2011.83.
[46] C. Luk, R. Cohn, R. Muth, H. Patil, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 190–200.
[47] A. C. de Melo, The New Linux 'perf' Tools, in: Linux Kongress, 2010.
[48] C. S. Wong, I. Tan, R. D. Kumari, F. Wey, Towards achieving fairness in the Linux scheduler, ACM SIGOPS Operating Systems Review 42 (5) (2008) 34–43. doi:10.1145/1400097.1400102.
[49] Intel, Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs (2012).
URL `https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-8BA55F4A-D5AE-4E27-8C25-058B68D280A4.htm`
[50] Argonne National Laboratory, Using the Hydra Process Manager (2014).
URL `http://wiki.mpich.org/mpich/index.php/Using\_the\_Hydra\_Process\_Manager`
[51] Intel, Intel Performance Counter Monitor - A better way to measure CPU utilization (2012).
URL `http://www.intel.com/software/pcm`
[52] J. Torrellas, Architectures for extreme-scale computing, IEEE Computer 42 (11) (2009) 28–35.
[53] R. A. Pielke, W. R. Cotton, R. L. Walko, C. J. Trembaek, W. A. Lyons, L. D. Grasso, M. E. Nieholls, M. D. Moran, D. A. Wesley, T. J. Lee, J. H. Copeland, A Comprehensive Meteorological Modeling System- RAMS, Meteorology and Atmospheric Physics 91 (1-4) (1992) 69–91.
[54] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. a. Levine, V. Pascucci, M. Schulz, C. H. Still, Mapping applications with collectives over sub-communicators on torus networks, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012. doi:10.1109/SC.2012.75.
[55] T. Agarwal, A. Sharma, L. V. Kalé, Topology-aware task mapping for reducing communication contention on large parallel machines, in: International Parallel and Distributed Processing Symposium (IPDPS), 2006.
[56] A. Chan, W. Gropp, E. Lusk, User's Guide for MPE Extensions for MPI Programs (1998).
[57] Intel, Intel Trace Analyzer and Collector (2013).
URL `http://software.intel.com/en-us/intel-trace-analyzer`
[58] C. Karlsson, T. Davies, Z. Chen, Optimizing Process-to-Core Mappings for Application Level Multi-dimensional MPI Communications, in: IEEE International Conference on Cluster Computing (CLUSTER), 2012, pp. 486–494. doi:10.1109/CLUSTER.2012.47.
[59] J. Hursey, J. Squyres, T. Dontje, Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems, in: IEEE International Conference on Cluster Computing (CLUSTER), 2011. doi:10.1109/CLUSTER.2011.59.

[60] S. Ito, K. Goto, K. Ono, Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments, Computers & Fluids 80 (2013) 88–93. doi:10.1016/j.compfluid.2012.04.024.
[61] G. Karypis, V. Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.
[62] J. L. Träff, Implementing the MPI Process Topology Mechanism, in: ACM/IEEE conference on Supercomputing (SC), 2002.
[63] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Catalyurek, K. Devine, Exploiting Geometric Partitioning in Task Mapping for Parallel Computers, in: International Parallel and Distributed Processing Symposium (IPDPS), 2014, pp. 27–36.