Locality and Balance for Communication-Aware Thread Mapping in Multicore Systems

Matthias Diener[1,2](✉), Eduardo H.M. Cruz[1], Marco A.Z. Alves[1], Mohammad S. Alhakeem[2], Philippe O.A. Navaux[1], and Hans-Ulrich Heiß[2]

[1] Informatics Institute, Federal University of Rio Grande Do Sul, Porto Alegre, Brazil
{mdiener,ehmcruz,mazalves,navaux}@inf.ufrgs.br

[2] Communication and Operating Systems Group, Technische Universität Berlin, Berlin, Germany
{alhakeem,hans-ulrich.heiss}@tu-berlin.de

Abstract. In multicore architectures, deciding where to execute the threads of parallel applications is increasingly a significant challenge. This thread mapping has a large impact on the application's performance and energy consumption. Recent research in this area mostly focuses on improving the locality of memory accesses and optimizing the use of shared caches by mapping threads that frequently communicate with each other to processing units that are closer to each other in the memory hierarchy. However, locality-based policies can lead to a substantial performance reduction in some cases due to communication imbalance. In this paper, we perform a comprehensive exploration of communication-aware thread mapping policies in multicore architectures. We develop a set of metrics to evaluate the communication behavior of parallel applications, and describe how these metrics can be used to favor locality-based or balance-based mapping policies. Based on these metrics, we introduce a novel mapping policy that combines locality and balance aspects and achieves the highest overall improvements. We provide an experimental evaluation of the performance gains using different mapping policies as well as a detailed analysis of the sources of energy savings.

1 Introduction

Due to the rising parallelism in modern multicore architectures, deciding where to execute each thread of a parallel application is becoming increasingly important to improve the application's performance as well as its energy consumption. The assignment of threads to processing units (PUs), which is called *thread mapping*, can take into account several characteristics of the parallel application and the underlying hardware architecture, such as utilization of PUs, contention on functional units, memory usage or memory access patterns. Recent research in this area mostly focuses on threads' memory accesses to shared data, which we call *communication* between threads, and uses a thread mapping policy that puts threads closer to each other in the memory hierarchy if they communicate frequently. In this way, threads can make better use of shared caches, and the overall memory access *locality* increases [2,9].

DOI: 10.1007/978-3-662-48096-0_16

However, these *communication-aware* mapping policies based on increasing locality can actually reduce the performance in some cases. We identify two conditions under which a locality-based policy has no improvements or is even detrimental to performance; when communication is imbalanced between threads, and when the ratio of communication to private data memory accesses is low. In such cases, it can be better to balance the communication or to scatter threads, such that there is less contention on caches or interconnections.

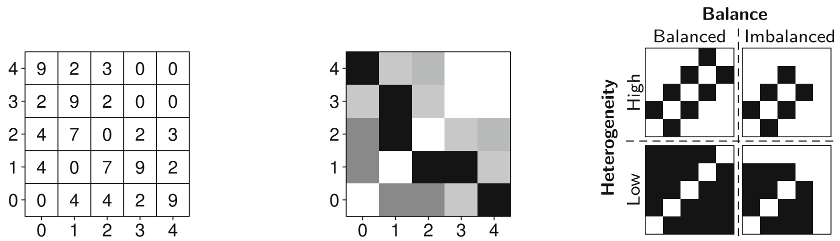
In this paper, we make the following contributions to the thread mapping problem: (1) We develop a set of metrics that are based on shared memory accesses and comprehensively represent the communication behavior of the threads in parallel applications. These metrics describe the structure and volume of communication. (2) We discuss which characteristics are suitable for each type of thread mapping, and introduce policies that optimize the mapping for each metric, as well as a policy that combines locality and balance. (3) We evaluate parallel applications in terms of the metrics and show the performance improvements of the different mapping policies. We also analyze the sources of performance improvements and energy savings by using a microarchitectural simulator.

2 Communication in Shared Memory

In parallel applications based on shared memory programming models, such as OpenMP and Pthreads, communication is *implicit* and is performed via memory accesses to shared memory areas. By observing accesses to memory addresses at the cache line granularity, we can define a *communication event* as two memory accesses from two different threads to the same cache line. With this definition, we create a *communication matrix* that represents the communication behavior of a parallel application by grouping the communication events [8,14].

In a communication matrix, the axes represent the thread IDs, while each cell in the matrix contains the number of communication events for the corresponding thread pair. For example, Fig. 1a shows a communication matrix for an application that consists of 5 threads. Figure 1b shows a visualization of this matrix, where darker matrix cells illustrate more communication. Based on the communication matrix, we introduce metrics to describe the communication behavior formally. Our goal is to determine the most appropriate mapping policy for a particular communication behavior depending on these metrics. Four metrics are presented: *heterogeneity* and *balance* describe the structure of communication, while *amount* and *ratio* describe the volume of communication.

Communication Heterogeneity. For policies that focus on improving the locality of communication, it is necessary to have groups of threads that communicate more within the group than with threads outside the group. Based on this intuition, a higher variation in the number of communication events for thread pairs in the communication matrix indicates opportunities to increase the overall locality. We refer to this variation as the *heterogeneity* of communication. We adapt previous work [6,10] to formulate the metric H_{Comm}, which evaluates



(a) Communication matrix. Axes show thread IDs. Cells contain the number of communication events.

(b) Visualization of the communication matrix (a). Darker cells indicate more communication.

(c) Communication matrices with different values for heterogeneity and balance.

Fig. 1. Communication behavior of a parallel application consisting of 5 threads.

this heterogeneity. As shown in (1), H_{Comm} is calculated by first normalizing the communication matrix M to its highest value, and then calculating the average variance of the number of communication events per thread. The *max* and *var* functions calculate the maximum and variance, respectively, and T represents the number of threads. A locality-based thread mapping policy is more suitable for higher values of H_{Comm}.

$$M_{norm} = \frac{M}{max(M)} \cdot 100, \qquad H_{Comm} = \frac{\sum_{i=1}^{T} var(M_{norm}[i][1...T])}{T} \tag{1}$$

Communication Balance. For mapping policies that are based on balance, it is necessary to determine if some threads are performing more communication than others. To evaluate this property, we introduce the metric B_{Comm}, which we refer to as the *balance* of the threads' communication behavior. To calculate B_{Comm}, we first calculate the total amount of communication per thread in a *communication vector CommV*, where each element i of $CommV$ contains the number of communication events of thread i. Then, similar to traditional load balance [15], B_{Comm} is calculated by (2).

$$CommV[i] = \sum_{j=1}^{T} M[i][j], \quad B_{Comm} = \left(\frac{max(CommV)}{\sum_{i=1}^{T} CommV[i]/T} - 1 \right) \cdot 100\,\% \tag{2}$$

A value of B_{Comm} that is close to 0 indicates a highly balanced communication between threads, while higher values indicate more imbalance in the threads' communication behavior, suggesting that communication balance-based mapping policies are more beneficial. A comparison of communication matrices with different values of *heterogeneity* and *balance* is shown in Fig. 1c.

Communication Amount. Improvements according to a specific thread mapping policy depend on how much threads are communicating. We expect higher gains for parallel applications that communicate more. To describe the *amount* of communication, we introduce the A_{Comm} metric, defined as the average number of communication events per thread, which is calculated by (3).

$$A_{Comm} = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} M[i][j]}{T^2} \tag{3}$$

Communication Ratio. The amount of communication itself is not sufficient to evaluate if an application is suitable for communication-aware thread mapping. If threads have much more memory accesses to private data than communication, a communication-aware mapping might not affect the overall memory access behavior. For this reason, we define the *communication ratio* metric R_{Comm}, which is the ratio of the communication accesses to the total number of memory accesses of the application threads. R_{Comm} is calculated by (4), where $AccV[i]$ is the number of memory accesses performed by thread i.

$$R_{Comm} = \frac{A_{Comm}}{\sum_{i=1}^{T} AccV[i]} \tag{4}$$

3 Communication Behavior of the Benchmarks

In this section, we analyze the communication behavior of two sets of parallel applications in terms of the metrics introduced in the previous section.

3.1 Methodology of the Experiments

Benchmarks. We chose two parallel benchmark suites for the evaluation. *NAS-OMP* [11] is the OpenMP implementation of the NAS Parallel Benchmarks (NPB), which consists of 10 applications from the HPC domain. We use three input sizes for the characterization, *W*, *A*, and *B* (from smallest to largest), to show how the behavior changes with increasing input sizes. All applications were executed with 64 threads. *PARSEC* [3] is a suite of 13 benchmarks that focus on emerging workloads and are implemented using OpenMP and Pthreads. All benchmarks were executed with the *native* input size. The number of threads is different for each application, but most of them use 64 threads.

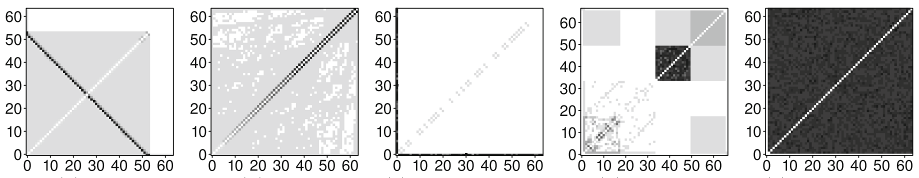
Profiling Environment. To characterize the benchmarks' communication behavior, we use a memory tracer based on our `numalize` technique [10], built with the Pin DBI tool [13]. We calculate the communication behavior in a simplified way to characterize the applications independently from a particular hardware architecture. We collect all memory accesses of the application's threads at a granularity of 64 byte-wide memory blocks and within time intervals of 10 ms. During each time interval, every time a memory block is accessed by a thread, we record a communication event between this thread and the other threads that have been involved in memory accesses to the same block since the beginning of the current time interval. By aggregating these events, we generate a communication matrix. Comparison with different time intervals, as well as a full cache simulator, showed that the detected behavior is stable in such a way that our characterization remains the same.

3.2 Results of the Communication Characterization

We begin with a discussion of several common types of communication matrices of the benchmarks, followed by an analysis of the metrics introduced in Sect. 2.

Communication Matrices. Figure 2 shows the communication matrices of selected benchmarks that represent the most common types of behavior. For the NAS-OMP benchmarks (LU and UA), we show the matrices of the *B* input. In LU, threads that are far apart communicate mostly with each other, e.g., the threads 0 and 53. UA has a nearest neighbor pattern, where neighboring threads perform most communication. In Blackscholes, thread 0 communicates with all other threads, indicating that communication is due to initialization or reduction of data. Ferret has a pipeline pattern, where one stage (threads 34–49) performs most of the communication of the application. Swaptions has an all-to-all pattern with similar amounts of communication for all threads.

From the communication matrices, it is possible to develop an idea of which applications can benefit from which type of mapping. In LU, UA, and Ferret, groups of threads perform substantial amounts of communication among themselves and only little communication with threads outside the group. Therefore, a locality-based policy can increase the overall locality by mapping threads that communicate closer to each other. Blackscholes and Swaptions can not benefit from such a policy, as no mapping can improve the overall locality. LU and Ferret can also benefit from a balance-based policy, as some threads perform very little communication, such as threads 53–63 of LU and threads 1–33 of Ferret.

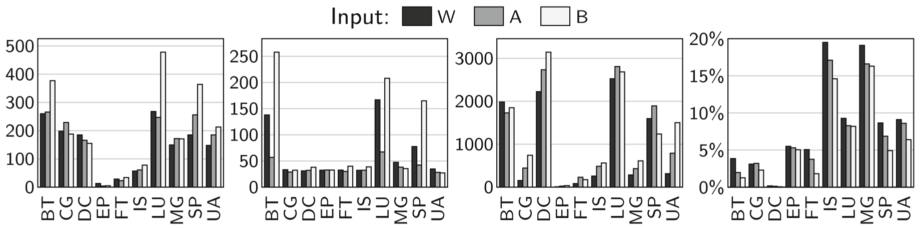


(a) LU. (b) UA. (c) Blackscholes. (d) Ferret. (e) Swaptions.

Fig. 2. Communication matrices of several parallel applications.

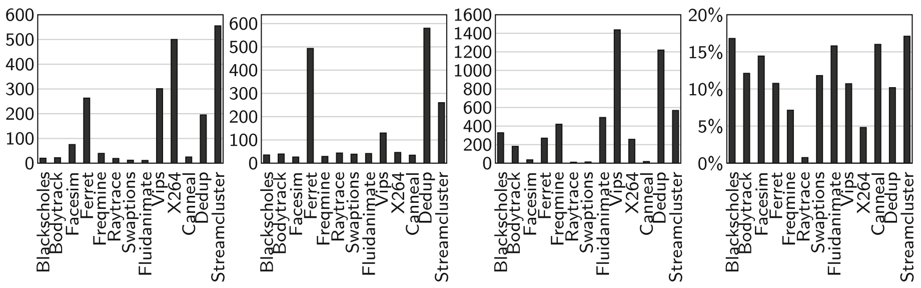
Communication Metrics. The values of the communication metrics introduced in Sect. 2 for the two benchmark suites are shown in Figs. 3 and 4.

All NAS-OMP applications except EP, FT, and IS have a high *heterogeneity*, indicating their suitability for locality-based mapping. In BT, LU, SP, and UA, the *heterogeneity* increases with larger input sizes. Evaluating the *communication balance* shows that only BT, LU, and SP are significantly imbalanced and show a suitability for balance-based policies. The reason for the imbalance of these applications is shown in Fig. 2a, as some of the threads are not communicating at all. This behavior changes with the input size: inputs *W* and *B* are imbalanced, while *A* is much more balanced. However, despite this communication imbalance, there is no significant load imbalance for these benchmarks according to our measurements, showing that the threads that communicate less



(a) Heterogeneity. (b) Balance. (c) Amount. (d) Ratio.

Fig. 3. Communication characteristics of the NAS-OMP benchmarks.



(a) Heterogeneity. (b) Balance. (c) Amount. (d) Ratio.

Fig. 4. Communication characteristics of the PARSEC benchmarks.

still perform substantial amounts of computation. For example, SP with the *B* input has a load balance of only 3.8, while the *communication balance* metric is much higher (16.5, higher values indicate a higher imbalance). The *communication amount* increases slightly with larger input sizes for most benchmarks. The *communication ratio* presented in Fig. 3d shows that with increasing input sizes, less communication in comparison to the total number of memory accesses is performed. This indicates that larger input sizes of NAS-OMP are less suitable for communication-aware thread mapping in most cases. Although DC has a high *amount* of communication, its *ratio* is very low.

Only a minority of the PARSEC benchmarks have a high *heterogeneity*, indicating that PARSEC applications are generally less suitable for locality-based thread mapping than those from NAS-OMP. Three PARSEC benchmarks, Ferret, Dedup, and Streamcluster, are significantly imbalanced. These three applications have a pipeline communication pattern, similar to the one shown in Fig. 2d. The load balance is again much lower than the *communication balance* (13.1 and 58.0 for Ferret, respectively). The *communication amount* differs widely between applications, but PARSEC benchmarks have a high *communication ratio* in general compared to NAS-OMP.

Summary. Summarizing our analysis, we find that a majority of the applications have a high heterogeneity and are therefore suitable for locality-based thread mapping. Some of these applications show varying degrees of communication imbalance and should therefore benefit also from balancing policies. On the other hand, few applications appear to require only a balance-based mapping policy, i.e., none of the imbalanced benchmarks have a low heterogeneity.

4 Mapping Policies

Several mapping policies that optimize different characteristics will be evaluated: *OS*, *Compact*, *Scatter*, *Locality*, *Distance*, *Balance* and *Balanced Locality*. The three last policies are introduced in this paper.

OS. The mapping performed by the operating system represents the baseline for our experiments. We use the Linux kernel, version 3.8, which uses the Completely Fair Scheduler (CFS) [18]. The scheduler focuses mostly on fairness and load balance [18], and has no means for improving communication locality or balance.

Compact. The Compact mapping performs a round-robin scheduling of threads to PUs such that neighboring threads are placed close to each other in the memory hierarchy. This mapping can increase the locality of communication behaviors where neighboring threads communicate frequently with each other.

Scatter. The Scatter policy represents the opposite of Compact. In this mapping, neighboring threads are placed far from each other in the hierarchy. In this way, performance can be improved for applications with little communication or a low communication ratio, by reducing competition for cache space. Compact and Scatter do not take the actual communication behavior into account.

Locality. The Locality policy optimizes the communication behavior by mapping threads that communicate frequently close to each other in the memory hierarchy. The mapping algorithm receives as input the communication matrix and a description of the memory hierarchy of the system, generated with `hwloc` [4]. It outputs a thread mapping that maximizes the overall locality of communication. Several algorithms have been proposed to calculate this mapping. We use the recently-proposed EagerMap algorithm [7] to calculate the Locality policy.

Distance. The Distance policy represents the opposite of Locality, placing threads that communicate far apart in the memory hierarchy. We calculate this mapping by inverting the communication matrix, subtracting each cell by the maximum value of the matrix. We then apply the same mapping algorithm as for the Locality mapping to the inverted matrix. This mapping can be useful when the heterogeneity is high, but the communication ratio is low, similar to the Scatter policy, but taking the actual communication behavior into account.

Balance. The Balance policy focuses on maximizing the communication balance for the application. The mapping algorithm receives the communication vector (introduced in Sect. 2) and the description of the memory hierarchy as input. The mapping is calculated by selecting the thread with the highest amount of communication that has not been mapped to a PU yet. This thread is then mapped to the PU which currently has the lowest amount of communication mapped to it. This process is repeated until all threads are mapped to a PU. This policy focuses only on balance and does not take locality into account.

Balanced Locality. The Balanced Locality policy focuses on increasing locality while still maintaining the balance of the communication. First, it maps threads that communicate frequently to nearby PUs, similar to the Locality policy. Second, for each level of the memory hierarchy, it keeps a similar amount of communication for each cache memory of that level. We model the memory hierarchy as a tree, where the leaves represent the PUs, and the other levels of the tree represent cache levels and their nodes represent specific cache memories. Our algorithm groups threads with high amounts of communication to the leaves of the tree, propagating this mapping to the parent nodes up to the root node. We add threads to the leaves until the amount of communication is higher than the average amount of communication per leaf. Summarizing, this policy maps threads that communicate frequently to close PUs whose amounts of communication are lower than the average amount of communication per PU.

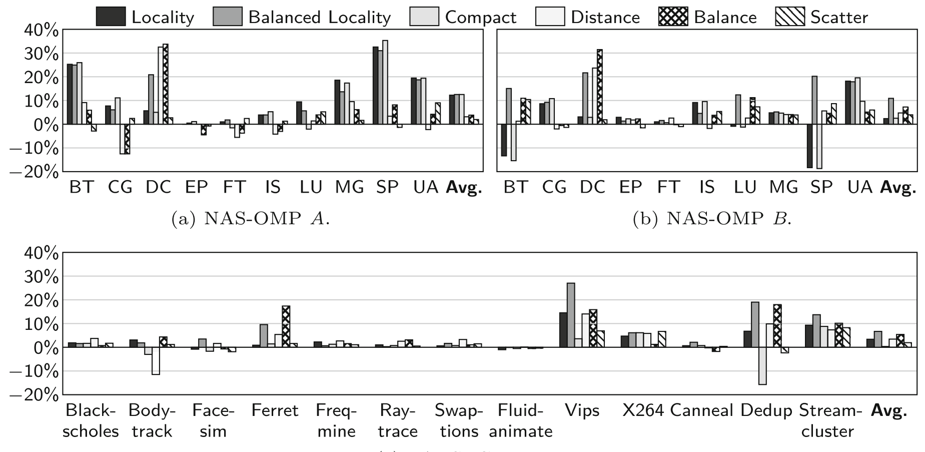
Load Balance. We also evaluate the Load Balance of selected benchmarks to compare it to the *Communication Balance* metric introduced in Sect. 2. We use the number of executed instructions per thread as the metric for the load.

5 Performance Evaluation on a Real Machine

In this section, we evaluate the performance improvements that are achieved by the thread mapping policies proposed in the previous section on a real machine.

Methodology. We run the experiments on a 4-socket system consisting of 4 Intel Xeon X7550 processors, with 8 cores and 2-SMT each (64 processing units in total). Each core has private L1 and L2 caches, while the L3 cache is shared among all the cores of the same processor. The same benchmarks with the same number of threads and input sizes (NAS-OMP only with A and B) as in the previous sections were evaluated. For each mapping policy presented in Sect. 4, we show the average execution time of 10 runs. The OS mapping policy is our baseline, and results are presented in terms of performance gains over this policy. In all policies except OS, no thread migrations during execution were performed.

Results. Figure 5 shows the performance gains compared to the OS mapping. For NAS-OMP with the A input, most benchmarks profit from the Locality policy, as predicted by our analysis. With the A input, this policy never reduces performance. The Balanced Locality policy has similar results as Locality for all benchmarks except DC. For the benchmarks that have a nearest neighbor communication pattern, the Compact policy improves performance, but reduces it in some cases, such as LU. The Distance and Balance policies only show performance improvements close to the Locality policy for the DC benchmark, which benefits from a better balance due to its low communication ratio. The Scatter policy never results in significant performance gains and reduces it in many cases. On average, the Locality, Balanced Locality, and Compact policies show improvements of more than 12 %, the other policies gain less than 4 %.



(a) NAS-OMP *A*.

(b) NAS-OMP *B*.

(c) PARSEC *native*.

Fig. 5. Performance improvements on the real machine compared to the OS mapping.

For the NAS-OMP benchmarks with the *B* input, the Locality policy reduces performance for the benchmarks that are imbalanced (BT, LU, and SP). On the other hand, Balanced Locality achieves the highest gains, proving that only taking locality into account is not sufficient for applications with this characteristic. The Load Balance policy (not shown in the figure) has improvements of less than 5 % for the 3 benchmarks, indicating that balancing the load is not as effective as balancing the communication in these cases. The other benchmarks show a similar behavior as the *A* input, with lower average gains. This echoes our discussion of the communication ratio, were we expected lower improvements when the ratio decreases. On average, Balanced Locality achieved the highest improvements, of 10.9 %. As several benchmarks benefit from balancing, the Balance policy has the second-highest improvements, of 7.3 %. The other policies gain less than 5 %.

As discussed in Sect. 3, the PARSEC benchmarks generally have lower metrics than the NAS-OMP benchmarks, which is reflected in the performance results. Five benchmarks (Ferret, Vips, X264, Dedup, and Streamcluster) benefit from communication-aware thread mapping. Most of them benefit from both the Locality and the Balance polices, but the Balanced Locality policy, which combines both, results in the highest improvements in most cases. The Compact, Distance, and Scatter policies do not improve performance consistently and result in performance losses in several cases. On average, Balanced Locality has again the highest gains of 6.7 %, followed by Balance (5.4 %) and Locality (3.4 %).

Summary. We conclude that increasing locality is the most important way to perform communication-aware thread mapping for most parallel applications. However, many applications can benefit from improving the balance of the communication, achieving higher performance gains and avoiding the performance reduction that a locality-based policy can cause. Simple mapping policies that do not take the communication behavior into account only improve performance in some cases and provide no consistent improvements over the OS.

6 Performance and Energy Consumption in a Simulator

Apart from performance, thread mapping can also improve the energy efficiency of parallel applications, for two main reasons. By reducing execution time, static energy consumption (leakage) will be reduced proportionally, since the processor is in a high power-consuming state for less time. Additionally, reducing the number of cache misses and traffic on the interconnections reduces the dynamic energy consumption, leading to a more energy-efficient execution. This section investigates the architectural impacts of thread mapping on the performance and energy consumption using a microarchitecture simulator.

Table 1. Parameters of the simulated machine.

Parameter	Value
System	2x 4-core processors; L1I/L1D cache per core; L2 cache shared between 2 cores
Execution cores	OoO; 1.8 GHz, 65 nm; 12 stages; 16 B fetch size; 96-entry ROB; PAs branch predictor
L1I/L1D caches	32 KB, 8-way, 64 B line size; LRU policy; 1 cycle; MOESI protocol; stride prefetch
L2 caches	2 MB, 8-way, 64 B line size; LRU policy; 4 cycles; stream prefetch
Interconnection	Cache line transfer: 2 cycles L2-to-L2; 32 cycles L2-to-DRAM
DRAM	DDR2 667 MHz (5-5-5); 8 DRAM banks/channel; 2 channels; 1 KB row buffer

Methodology. We use an in-house, cycle-accurate x86 processor simulator [1]. The execution statistics of the simulator are fed into McPAT [12] to calculate the energy consumption. Table 1 shows the simulation parameters. As benchmark, we chose SP from NAS-OMP, and run it with input *W* and 8 threads. We compare the Locality and Distance mappings in depth, which have the highest performance difference for this configuration of SP in the simulated machine.

Results. Figure 6 presents the results for execution time, performance statistics and energy consumption. The results are normalized to the values of the Distance mapping. Regarding the performance, the execution time was reduced by 10.1 %, caused by the reduction of the number of L2 cache misses (32.0 %) and a reduction of the number of DRAM accesses (39.9 %). The processor memory read time was reduced by 22.6 %. The higher data locality also led to a reduction of

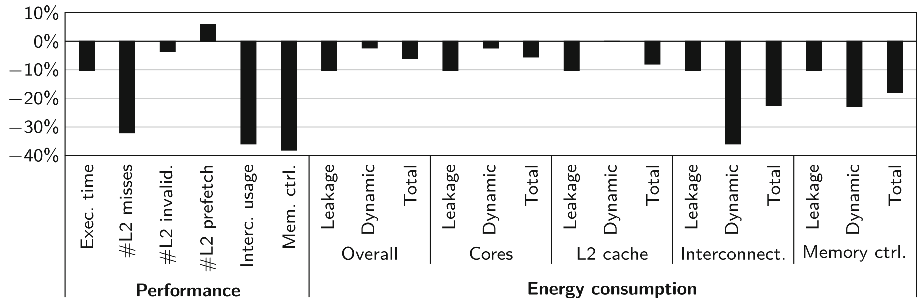


Fig. 6. Results of executing SP in the simulator, normalized to the Distance mapping.

the number of L2 invalidation messages and off-chip interconnection usage. The reduction of the interconnection traffic and L2 misses enabled the L2 prefetcher to issue 5.8 % more requests, which also contributed to the overall performance gains.

The more efficient execution also reduced energy consumption. Leakage was reduced by 10.1 % for all components, the same amount as the execution time. As expected, dynamic energy consumption was reduced less, by 2.4 % overall, leading to a total energy reduction of 6.1 %. Although there are reductions of the energy consumed by the cores and L2 caches, of 5.5 % and 8.0 % respectively, the extra prefetches reduced potential reductions of the L2 dynamic energy, which comprises 23.2 % of the total L2 energy. The highest energy reductions were achieved by the memory controller and the interconnections between the L2 caches, and between the processors and the main memory, with reductions of 17.9 % and 22.4 % respectively. The interconnection savings are caused by less off-chip searches, as well as less cache-to-cache and DRAM data transfers.

7 Related Work

Many techniques for thread mapping have been investigated previously, focusing on balance-based or locality-based policies. Most balance-based policies depend on characteristics of the parallel application and the underlying architecture, such as memory usage or core utilization. Sasaki et al. [16] develop a scheduling scheme for multi-threaded applications based on predicting the application scalability to balance the resource utilization. The Extended Lowest Load technique [17] uses a heuristic that is based on the amount of time spent by each core doing useful work to find the optimal target core for each thread. Pearce et al. [15] argue that the limitation in load balancing-based mapping policies is related to inaccurate load information. Depending on information about the work units of the application and dependencies between them, they develop load metrics and a cost model for re-correcting load imbalance.

In these approaches, locality issues and the communication behavior are not considered. On the other hand, policies that perform communication-aware mapping mostly focus on improving the locality of communication without evaluating balance. For parallel applications that communicate through MPI, most previous research focuses on methods to trace the messages and uses the information to perform a process mapping. MPIPP [5] is a framework for process mapping, consisting of a message tracer and mapping algorithm. Some papers evaluate process mapping for particular applications, such as the NAS-MPI benchmarks [14]. For applications that use OpenMP or Pthreads, most mapping solutions focus on analyzing memory accesses to map threads that communicate on shared caches, but do not address the balance issue [6,8].

8 Conclusions

Communication-aware thread mapping can improve the performance of parallel applications on multicore systems. In this paper, we introduced metrics to describe the communication behavior and determine if an application can benefit from mapping policies that focus on the locality or the balance of communication. We presented a mapping policy that increases locality while still maintaining the balance. Our evaluation on a real system showed that this policy can provide the highest improvements and avoids the performance losses that may occur using a pure locality-based policy. We also provided an in-depth analysis of performance and energy efficiency gains from thread mapping in a hardware simulator.

References

1. Alves, M.A.: Increasing energy efficiency of processor caches via line usage predictors. Ph.D. thesis, Federal University of Rio Grande do Sul (2014)
2. Barrow-Williams, N., Fensch, C., Moore, S.: A communication characterisation of Splash-2 and Parsec. In: International Symposium on Workload Characterization (2009)
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: International Conference on Parallel Architectures and Compilation Techniques (2008)
4. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: International Conference on Parallel, Distributed and Network-based Processing (2010)
5. Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In: International Conference on Supercomputing (2006)
6. Cruz, E.H.M., Diener, M., Alves, M.A.Z., Navaux, P.O.A.: Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. J. Parallel Distrib. Comput. **74**(3), 2215–2228 (2014)
7. Cruz, E.H.M., Diener, M., Pilla, L.L., Navaux, P.O.A.: An efficient algorithm for communication-based task mapping. In: International Conference on Parallel, Distributed, and Network-Based Processing (2015)
8. Diener, M., Cruz, E.H.M., Navaux, P.O.A.: Communication-based mapping using shared pages. In: International Parallel and Distributed Processing Symposium (2013)

9. Diener, M., Cruz, E.H.M., Navaux, P.O.A., Busse, A., Heiß, H.U.: kMAF: automatic kernel-level management of thread and data affinity. In: International Conference on Parallel Architectures and Compilation Techniques (2014)
10. Diener, M., Cruz, E.H.M., Pilla, L.L., Dupros, F., Navaux, P.O.A.: Characterizing communication and page usage of parallel applications for thread and data mapping. Perform. Eval. **88–89**, 18–36 (2015)
11. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, October 1999
12. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing. ACM Trans. Archit. Code Optim. (TACO) **10**(1), 5 (2013)
13. Luk, C., Cohn, R., Muth, R., Patil, H.: Pin: building customized program analysis tools with dynamic instrumentation. In: SIGPLAN Conference on Programming Language Design and Implementation (2005)
14. Mercier, G., Clet-Ortega, J.: Towards an efficient process placement policy for MPI applications in multicore environments. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 104–115. Springer, Heidelberg (2009)
15. Pearce, O., Gamblin, T., de Supinski, B.R., Schulz, M., Amato, N.M.: Quantifying the effectiveness of load balance algorithms. In: Supercomputing (2012)
16. Sasaki, H., Tanimoto, T., Inoue, K., Nakamura, H.: Scalability-based manycore partitioning. In: International Conference on Parallel Architectures and Compilation Techniques (2012)
17. Tousimojarad, A., et al.: An efficient thread mapping strategy for multiprogramming on manycore processors. In: International Conference on Parallel Computing (2013)
18. Wong, C.S., Tan, I., Kumari, R.D., Wey, F.: Towards achieving fairness in the Linux scheduler. SIGOPS Oper. Syst. Rev. **42**(5), 34–43 (2008)