Exploiting Thread-Data Affinity In OpenMP with Data Access Patterns *

Andrea Di Biagio, Ettore Speziale **, and Giovanni Agosta

Dipartimento di Elettronica ed Informazione, Politecnico di Milano
andrea.dibiagio@gmail.com, {speziale,agosta}@elet.polimi.it

Abstract. In modern NUMA architectures, preserving data access locality is a key issue to guarantee performance. We define, for the OpenMP programming model, a type of architecture-agnostic programmer hint to describe the behaviour of parallel loops. These hints are only related to features of the program, in particular to the data accessed by each loop iteration. The runtime will then combine this information with architectural information gathered during its initialization, to guide task scheduling, in case of dynamic loop iteration scheduling. We prove the effectiveness of the proposed technique on the NAS parallel benchmark suite, achieving an average speedup of 1.21x.

Current trends in computer architectures tend to increase the number of cores per chip to cope with the power and frequency walls while exploiting the transistor density increase. This is driving designers towards multi- and many-core architectures, where *Non-Uniform Memory Access* (NUMA) designs are needed [9,2]. In NUMA architectures, cores incur in greater delays when accessing non-local memories. Since NUMA machines preserve the shared memory abstraction, it is possible to program them using programming models such as OpenMP [3], which hide the complexity of the underlying memory hierarchy.

To achieve performance in NUMA architectures, it is essential to provide *data access locality*, that is, data located in a given node are accessed as much as possible from the cores of the same node, and as little as possible from the other ones [14,21]. Recent works targeting OpenMP on Linux focus on exploiting specialized page allocation policies [4] such as explicit data distribution, which allows the programmer to select a precise distribution to be implemented at initialization time. The *next-touch* policy, introduced in [8,6], allows dynamic data relocation by exploiting memory protection mechanisms.

However, such works incur in one or more of the following drawbacks: they rely on programmer knowledge of the underlying architecture, thus negating a major benefit of OpenMP, architecture independence [15]; they lack dynamism, since they provide only a single data distribution strategy which might not cover

* This work was supported in part by the European Commission under Grant 2PARMA FP7-248716 and ARTEMIS-SMECY.
** This author was supported in part by a grant from ST Microelectronics.

all the access patterns the program employs during different phases of its execution; or, they do not deal with workload balancing, which in turn adversely affects irregular parallel applications.

In this work, we take into account these issues, providing a solution to maintain thread-data affinity across the lifetime of the application, which relies on programmer hints describing only the application behavior, and exploiting them through a specialized runtime, balancing the workload by means of work-stealing.

The rest of this paper is organized as follows. Section 1 introduces the syntax and semantics of the proposed hints, while Section 2 provides details on our runtime design and implementation, and Section 3 provides an experimental evaluation. Finally, Section 4 provides a brief survey of related works, and Section 5 draws some conclusions and highlights future research directions.

1 The Data Access Pattern Approach

The current OpenMP standard provides support for parallel loops through the `omp for` and `omp do` directives [1]. The parallel loop syntax is restricted to force the loop bounds to be loop invariants, since the runtime must always be able to evaluate the iteration space. Once the iteration space has been computed, iterations are first grouped into *chunks* [2] and then mapped to the active threads of the parallel team according to the scheduling policy implemented by the runtime. Programmers can influence the behaviour of the runtime system only by forcing a iteration scheduling policy and specifying a minimum chunk size.

Even though OpenMP allows the programmer to choose among different scheduling strategies, to address the problem of mapping iterations over the threads in a team, there is no support for expressing thread-data affinity [5,19].

The key idea of our approach is to allow the runtime to identify the portion of data which will be accessed by the iterations of a parallel loop. These iterations will then be scheduled to threads according to a novel dynamic scheduling policy, which will try to preserve locality as much as possible.

To this end, we extend the existing OpenMP parallel loop directive through a new clause representing the *data access pattern*, that is the way loop iterations access the data. The runtime will then use the thread-data affinity information derived from the data access pattern to improve the existing dynamic iteration scheduling policy, by scheduling threads on the cores nearest to the memory where the related data are stored. While automated approaches to page placement do not require changes to the API, identifying and exploiting thread-data affinity at compile time might not be feasible, and is in general a very complex task [5]. By contrast, a skilled programmer is able to identify more effectively the patterns used by threads when accessing data, and thus provide precise hints

[1] `omp for` and `omp do` model the same type of parallel loop, in C and Fortran respectively. For brevity in the rest of the paper we will refer to `omp for` but the same considerations apply to `omp do` as well.

[2] We use the term *chunk* to refer to a set of iterations as specified in OpenMP [3].

to the runtime. This is, anyway, mandatory if a fine-tuning of the application performances is desired [4,20,16].

A key difference with respect to previous works [4], including *PGAS* languages [18,1], is that to minimize the programming efforts when writing parallel programs, our approach does not rely on explicit data distribution and exploitation of the processor space.

1.1 Data Access Pattern Definition

A data access pattern binds iterations in a parallel loop with the portion of memory accessed at runtime. We formally define the data access pattern and the OpenMP syntactic extension needed to support it as follows.

Definition 1. *A* data access pattern *is an equivalence relation over the elements of a k-dimensional array data structure. An equivalence class under the data access pattern relation is called* tile*. Data access pattern relations are described by means of* pattern clauses*, defined by the grammar in Figure 1 and its associated semantics.*

$$
\begin{array}{lll}
\mathit{Axiom} \rightarrow \texttt{pattern(}\ \mathit{Clause}\ \texttt{)} & \mathit{PatternExpr} \rightarrow \mathit{RangeExpr} \mid \mathit{SliceExpr} \\
\mathit{Clause} \rightarrow \mathit{DataStructure}\ \texttt{[}\ \mathit{PESeq}\ \texttt{]} & \mathit{RangeExpr} \rightarrow \mathit{Expr}\ \texttt{:}\ \mathit{Expr} \\
\mathit{PESeq} \rightarrow \mathit{PESeq}\ \texttt{,}\ \mathit{PatternExpr} & \quad \mid \mathit{Expr} \mid \texttt{*} \\
\quad \mid \mathit{PatternExpr} & \mathit{SliceExpr} \rightarrow \texttt{\^{}}\ \mathit{Expr}
\end{array}
$$

Fig. 1: Pattern Clause Syntax. *Expr* is any expression of runtime constants, while *DataStructure* can be any array or pointer variable name

In our OpenMP extension, a *pattern clause* (or, for brevity, a pattern) is associated to a loop directive. The first argument of a pattern clause is a reference to the shared data structure that is concurrently accessed by iterations in the loop. The rest of the pattern clause consists of a sequence of *pattern expressions*, one for each dimension. A pattern expression can be either a *range expression* or a *slice expression*. A range expression is used to identify a range of indices in a given dimension of the data structure, that are associated to all tiles. A slice expression identifies the size of each tile in a given dimension.

A range expression has the form `[n:m]`. Both `n` and `m` must be loop invariant. Their value is thus known at runtime before the loop execution starts. The lower bound of a range expression may be omitted when it matches exactly the lower bound of the associated dimension. Hence, a pattern expression `m` is an alias for `[lb:m]`, where `lb` is the lower bound of the index for the dimension considered. The `*` operator is also a shorthand for `[lb:ub]`, where `lb` and `ub` are the lower bound and the upper bound values of the index for a given dimension. The latter range expression variants allow a more compact definition of the pattern clause in many practical cases, but do not add any expressive power.

```
#pragma omp for collapse(2) \
 pattern(A[^RSLICE,^CSLICE])
for(i = 0; i < ROWS; i += RSLICE)
  for(j = 0; j < COLS; j += CSLICE)
    for(k = 0; k < RSLICE; ++k)
      for(h = 0; h < CSLICE; ++h)
        A[i+k][j+h] = ...;
```

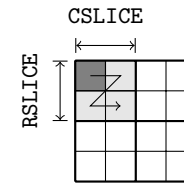


Fig. 2: Pattern example. Matrix `A` is accessed in a block-wise fashion by the collapsed parallel loop

A slice expression takes the form `^n`, where `n` is a runtime constant.

Figure 2 demonstrates the data access pattern semantics. The two slice expressions define bi-dimensional tiles of size `RSLICE` × `CSLICE` on matrix `A`, thus representing the block-wise accesses performed by the loop nest.

The mapping between tiles and iterations is defined as follows: if there is no slice expression in the pattern there is a single tile which is accessed by all iterations; otherwise, tiles and iterations are associated by a bijective relation, that depends on both the iteration indices and the sign of the loop increment expressions. In a normalized loop nest, each slice expression is associated to one loop index i_l and the tiles can be ordered with respect to the indices i_d of the dimension d associated to the slice expression divided by the tile size n. Iterations of loop index i_l are mapped to tiles with $i_d/n = i_l$.

Back to the example in Figure 2, assuming `RSLICE` = `CSLICE` = 2, and `A` a 4×4 square matrix, the pattern identifies four tiles. The iterations with index `i` = 0 are associated to data items of indices 0 and 1 on the first dimension. The same holds for loop index `j`, which is associated to the slice expression corresponding to the second dimension of `A`. Thus, iteration `i,j` = $\langle 0, 0 \rangle$ is mapped to the data in `A[0][0]`, `A[0][1]`, `A[1][0]`, and `A[1][1]`.

2 Runtime Extensions to Exploit Patterns

To employ the information encoded in the pattern clauses, we propose an extension of the OpenMP runtime. The runtime analyzes each pattern expressions to identify the size of the memory tiles accessed by iterations. The tile information can then be exploited at runtime to group together iterations that will probably touch the same set of virtual memory pages. Since at runtime the base address of the patterned data structure is known, it is always possible to identify the set of memory pages that are expected to be touched by the iterations of the loop. This is true also for dynamically allocated data-structures for which the size can be assumed equal to the tile size times the size of the iteration space.

Since the runtime aims at maximizing the number of local accesses, while avoiding, if possible, to incur in the penalty of long latency due to remote memory accesses, the information obtained analyzing pattern clauses is used to identify

groups of iterations (*blocks*) that need to be scheduled together on the same node. Iterations that access the same memory pages (or different pages physically mapped to the same node) are grouped within the same block. The dynamic scheduling policy is thus driven by the collected pattern information.

The implementation used in this work is based on the *libgomp* [7] OpenMP runtime and uses the Linux NUMA API [12] to detect virtual page mappings.

2.1 Iteration Space Partitioning

To exploit the hints provided by the pattern information, the runtime has to partition the iteration space so to minimize the number of remote accesses.

Finding an optimal partition is known to be NP-complete. Obviously, such complexity cannot be handled at runtime even with moderate numbers of iterations. Therefore, we propose a straightforward heuristic approach to minimize the time spent by the runtime in analyzing pattern information while still providing a good, even if potentially sub-optimal, partitioning. To further reduce the overhead, we base the partitioning of the iterations of each loop on the information obtained from a single pattern.

The algorithm implemented in the proposed heuristic approach performs a linear scan of the iteration space in search of opportunities for grouping adjacent iterations. Let a and b be two adjacent iterations of the analyzed parallel loop. Both a and b will be mapped to the same block if at least one of the following conditions is satisfied: iteration a accesses to the same set of memory pages touched by iteration b; the set of pages touched by a are physically mapped to a node that is the same for the pages touched by b; pages touched by both a and b are not physically mapped to any node in the system.

Let us now formally introduce the concept of iteration block.

Definition 2. *Let lb and ub be respectively the lower and upper bound of the iteration space I of the analyzed loop. A block of iterations is defined as a range of indices of the form $[base, last]$, where $base \geq lb$ and $last \leq ub$.*

Let B be the set of blocks obtained from the partitioning phase, and let $b \in B$ be a block of iterations. We call $r(b)$ the range of indices described by b.

The runtime limits the maximum number of blocks to reduce the algorithm complexity while maintaining the required flexibility to cope with irregular workloads. The limit has been set, considering the outcome of an experimental campaign, to twice the number of available nodes in the system. To cope with the imposed constraints, different blocks of iterations may be merged.

When no pattern clause is specified for a given parallel loop, the iteration space is evenly partitioned into a number of blocks equal to the number of available nodes. Since the output of the partitioning algorithm is not necessarily the optimal partition, we later introduce a runtime work stealing mechanism to reduce the effects of an unbalanced distribution of the workload.

2.2 A Dynamic Scheduling Policy for Pattern Enabled OpenMP Runtimes

At the end of the partitioning stage, the iteration space of the parallel loop is divided into blocks of iterations. When a loop has associated pattern information, the runtime knows exactly which pages are touched by each iteration block. The runtime assigns a *work queue* to each NUMA node. The work queue is used to store information about iteration blocks. A global work queue is reserved for those blocks that are not related to any of the active NUMA nodes.

The algorithm that maps blocks to work queues uses the iteration-data affinity information coming from the analysis of the pattern. Each thread of the parallel team analyzes the set of blocks in parallel. Let b be a block and let P_b be the set of pages touched by iterations of b. The algorithm counts how many pages in P_b are mapped to each node. The node with the highest number of mapped pages is finally selected as the target node for the block b. If none of the nodes is related to any of the pages in P_b, b is assigned to the global queue.

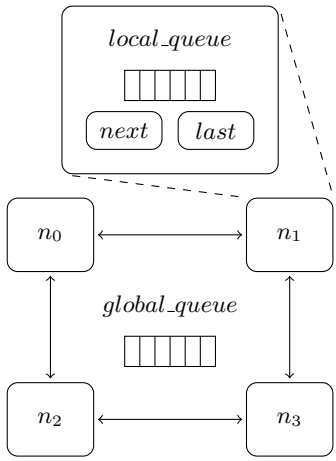


Fig. 3: Runtime system with four distributed work queues and a global queue

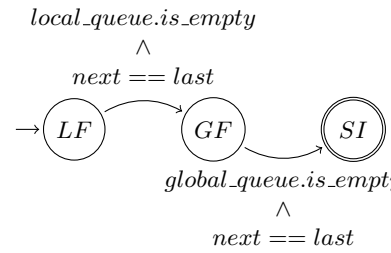


Fig. 4: Runtime behaviour of a sub-team. *Local Fetch* (*LF*): fetch blocks from the *local_queue* of the local node; *Global Fetch* (*GF*): fetch blocks from the *global_queue*; *Steal Iterations* (*SI*): steal blocks from the *local_queue* of neighbour nodes

Figure 3 shows the internal state of the runtime system in the case of a cc-NUMA architecture with four nodes. The internal state of each node is composed of a working queue called *local_queue* and two integer fields *next* and *last*, used respectively to store the lower bound and the upper bound index of the range of iteration indices associated to the *current block*.

At runtime, parallel teams are split into sub-teams, each associated to a distinct NUMA node. A sub-team associated to a node n is composed only by

threads of the team that are running on node n. Threads are mapped to sub-teams at runtime when a new team starts. The runtime behaviour of a sub-team can be formally described by a finite state automaton as shown in Figure 4.

Each sub-team starts executing in the initial state LF. Threads of a sub-team whose working state is LF, are only allowed to fetch blocks from the local work queue of the node in which they are running. When the local block queue is empty and no iterations are available in the current block, the sub-team moves from LF to GF, where the sub-team fetches blocks from the global queue. In both states, iterations are selected using the Guided Self Scheduling algorithm [17].

The idea is to exploit the locality of accesses preventing when possible threads from accessing remote pages. To this end, at first threads are forced to execute iterations from the local queue to maximize the probability of local accesses. Only when there are no more iterations associated to the local node, threads start fetching iterations from the global queue. Since global queue only stores blocks related to virtual pages that are still not mapped, there is an high probability that threads accessing the global queue will own those pages because of the *first-touch* policy implemented by the OS. The first-touch policy is the default policy for NUMA-aware Linux systems. It consists of placing memory pages on those nodes that first access the data during the program execution.

2.3 Work Stealing Strategy

When the global queue is empty and there are no iterations available in the local queue, the sub-team transitions from *GF* to *SI*. While in *SI*, threads start stealing blocks of iterations from the queues associated to other nodes. According to the implemented work stealing policy, threads in *SI* start stealing from the work queues of the nearest neighbour nodes. Since the runtime is aware of the distance between nodes (identified by means of calls to the Linux NUMA API), each sub-team knows which nodes are the best candidates for stealing.

The work stealing procedure iterates over the neighbours set of a node n in search of available blocks of iterations. By default the current neighbour node (n_{neigh}) is initially set equal to the node that hosts the current sub-team (n).

As long as there are iterations to fetch from the work queue of n_{neigh}, threads fetch new iterations from their work queues. Eventually, when the work queue of the current neighbour becomes empty, a new neighbour is selected.

The selection strategy is based on the *NUMA distance* between nodes of the underlying architecture. In the case shown in Figure 3, $\langle dist(n_0, n_i) | i \in [0:3] \rangle = \langle 0, 1, 1, 2 \rangle$. The distance relation $dist(n, n_i)$ imposes a partial ordering of the nodes $n_i \in K$. We need, for each node, a sequence of nodes to poll for the next neighbour, called a *neighbours vector*. To obtain the vectors, we make this a total ordering by imposing that, when $dist(n, n_i) = dist(n, n_j)$, $n_i \prec n_j$ if $i < j$.

3 Experimental Results

In this Section, we provide an experimental validation of our approach. The main findings are that the proposed approach based on pattern clauses is able

to consistently reduce the number of remote memory accesses, and that the reduction directly translates into a significant performance improvement.

The experimental campaign has been conducted on a AMD ccNUMA machine with four nodes, each a quad core Opteron 8378 processor. Each core has a two-level private cache hierarchy. L1 cache is composed by a 64KBytes data cache and by a 64KBytes instruction cache. L2 cache is an unified 512KBytes cache. All cores within a node share an unified 6144KBytes L3 cache. Inter-node communication is supported by a ring network topology.

AMD event based counters have been used to measure memory accesses. Separate runs have been used for performance and memory access profiling, to avoid memory access counter sampling overhead in timing measurements.

3.1 Benchmark Suite

We employ the NAS Parallel Benchmark suite, OpenMP version 3.3 [11]. We do not report on *DC* and *EP*, since these benchmarks do not have any OpenMP loop constructs (`omp for` and `omp do`). The benchmarks have been modified in order to make use of dynamic scheduling. Table 1 shows the number of total loops, dynamically scheduled loops, and loops tagged with the `pattern` clause.

Table 1: Benchmark characterization

Bench	Parallel loops	Dynamic loops	Patterns	Bench	Parallel loops	Dynamic loops	Patterns
bt.c	28	14	9	lu.c	26	10	9
cg.c	18	16	16	mg.b	14	11	11
ft.b	8	6	6	sp.c	33	20	20
is.c	9	2	2	ua.c	68	56	56

We compare the baseline *libgomp* runtime implementation opportunely extended to support a Guided Self Scheduling strategy for dynamically scheduled loop iterations with our optimized runtime. This choice is dictated by the fact that the *libgomp* dynamic scheduler provides only poor performance, thus comparing with it would result in a significant bias due to Guided Self Scheduling.

For all experiments we use 16 threads, each pinned on a different core.

3.2 Performance Analysis

Table 2 describes the runtime behaviour of the benchmarks, showing the percentage of blocks fetched in each of the states of the automaton in Figure 4 along with the percentage of the execution time spent in loops tagged with `pattern` clauses. A high percentage of fetches from local queues denotes a good distribution of the data structures, which is effectively exploited by the iteration scheduling thanks

to correct pattern information. On the other hand, blocks fetched through work stealing have higher probability of resulting in remote accesses since they were originally intended to be executed on a different node.

Table 2: Runtime behaviour

Bench	Blocks fetched from			Time in opt. loops
	Local [%]	Global [%]	Steal [%]	[%]
bt.c	65.72	0.01	34.27	90.55
cg.c	99.61	0.03	0.36	87.26
ft.b	76.40	0.00	23.60	66.69
is.c	66.67	0.00	33.33	51.30
lu.c	80.21	0.21	19.58	26.49
mg.b	35.16	22.26	42.58	66.82
sp.c	70.03	0.00	29.97	91.92
ua.c	88.36	0.13	11.51	78.28

Table 3: Speedups

Bench	Speedup		Δ
	Worst	Best	
bt.c	1.14	1.27	0.13
cg.c	1.81	1.82	0.01
ft.b	1.12	1.19	0.07
is.c	1.00	1.00	0.00
lu.c	1.02	1.05	0.03
mg.b	1.00	1.00	0.00
sp.c	1.18	1.23	0.05
ua.c	1.07	1.08	0.01

Table 3 shows the speedups obtained by our optimized runtime with respect to the baseline. Two scenarios are provided: *Best*, where the proposed work stealing policy based on NUMA distances is used; and *Worst*, where neighbour vectors are reversed. This shows that the order of the neighbours counts: the last column (Δ) shows the maximum performance loss in case of random neighbours selection. However, the results also show that the impact of this policy is not so large as to make the runtime less effective than the baseline. Thus, the *Worst* scenario shows the impact of the iteration scheduling optimization, while the *Best* scenario adds the impact of an effective work-stealing policy.

We can see that, for most benchmarks, we obtain a speedup between 1.05x and 1.27x for the *Best* scenario. There are three exceptions: *MG*, *IS* and *CG*.

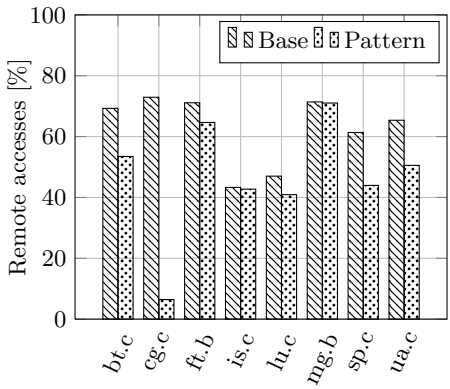
MG is the only benchmark where the initial distribution of frequently accessed data structures is performed by the master thread alone. Since we rely on the first-touch policy to provide the initial distribution, a large number of remote accesses is generated regardless of the iteration scheduling policy. Note that the pattern definition leads the runtime to place most of the iterations on the node where the master thread resides, thus leading to a reduced amount of blocks fetched from local queues.

IS benchmark implements a bucket sort algorithm. Excluding the time spent in initializing data structures, most of the time is spent on a fast data parallel loop used to sort keys of each bucket. There are several instances of non-linear accesses where array indices are obtained from table lookups. This type of access cannot be optimized, since it is by design hard to predict, to provide the required randomness. While the proposed technique cannot obtain a speedup, it still does not impose an overhead with respect to the baseline.

CG obtains the highest speedup, a remarkable 1.82x. It performs sparse matrix multiplication, which can easily lead to irregular accesses. However, the benchmark provides an initial data distribution that combined with the data access pattern information allows a massive improvement in data access regularity, which immediately translates into a performance improvement.

3.3 Remote Memory Access Analysis

Figure 5 shows the reduction in remote memory accesses obtained by our runtime with respect to the baseline. Memory access reduction is at the base of performance improvement, so these results mirror the performance speedups.



Bench	Base	Pattern	Savings [%]
bt.c	70,777.56	54,787.53	22.59
cg.c	40,969.86	3,592.73	91.23
ft.b	4,824.42	4,494.31	0.90
is.c	851.71	844.01	6.84
lu.c	37,674.95	32,731.91	13.12
mg.b	2,504.46	2,486.34	0.72
sp.c	269,485.03	192,971.48	28.39
ua.c	115,912.76	85,196.04	26.50

(a) Remote accesses percentage (b) Raw results (millions of accesses)

Fig. 5: Memory accesses performed by benchmarks

It is especially interesting to consider the reduction in *CG*, where remote memory accesses are strongly minimized thanks to the pattern information.

In *IS*, the data access patterns are mostly unpredictable, as memory accesses are defined through non-affine array functions. This makes it hard to find good pattern information for most of the parallel loops in the code. While the savings in terms of remote accesses are small, they are sufficient to offset the overhead imposed by the pattern evaluation and iteration space partitioning phases.

In *MG*, most frequently accessed data structures are allocated on a single node, which forces all threads on other nodes to perform remote accesses. Thus, no significant reduction is obtained. Moreover, the high amount of global fetches shows that part of data structures were not preallocated at all.

4 Related Work

Several different approaches are proposed in literature to mitigate the memory latency penalty due to remote accesses. Some of these approaches rely on the ability of the runtime system [6] or the OS itself [10] to implicitly trigger the migration of worker threads to avoid the cost of remote accesses.

Other approaches, such as PGAS languages [18,1] rely on the ability of the programmer to manually distribute data structures concurrently accessed by threads at runtime. These languages provide the programmer a mean to force a specific dynamic page placement policy for those shared data structures that will be heavily accessed by loops. On the other hand, our solution does not rely on explicit distribution hints, though it can take advantage of an initial data distribution provided by means of the *first-touch* policy.

Dynamic data distribution based on memory protection mechanisms has been introduced in [8,6,13]. Memory pages forming shared data structures can be dynamically tagged, to trigger a page migration to the next node touching them (*next-touch* strategy). Our approach is orthogonal with respect to this strategy, since we reduce the number of remote accesses without triggering redistributions.

In [15] the authors propose a dynamic data redistribution solution similar to [8,6] but based on information akin to our proposed data access pattern, which is, contrary to our solution, computed at runtime by means of profiling.

5 Conclusions

We propose an optimized OpenMP runtime design for NUMA machines to exploit thread-data affinity in parallel programs by means of programmer hints that take into account only the application behavior. Our experimental campaign shows a reduction in the number of remote accesses for most NAS benchmarks.

The approach could be further improved by removing unnecessary pattern evaluations when multiple subsequent loops share the same pattern. Moreover, opportunities for data redistribution could be automatically detected at compile-time by analysing pattern variations between subsequent loops.

Future extensions could include adding thread migration to handle the cases of multiple concurrent applications as well as the case of applications with multiple phases, alternating I/O bound phases with CPU bound ones. We also expect that combining our technique with a next-touch strategy would further reduce the remote accesses, while limiting the number of pages moved.

Furthermore, identifying patterns requires skill and time. It would be worth exploring both static analysis and profiling based techniques to provide recommended patterns to the programmer.

References

1. Allen, E., Chase, D., Hallet, J., Luchangco, V., Maessen, J., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress Language Specification. Sun Microsystems (2008)

2. AMD: AMD Direct Connect Architecture (2010), http://www.amd.com/us/products/technologies/direct-connect-architecture
3. ARB: OpenMP Application Program Interface, version 3.0 (2008), http://www.openmp.org
4. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., Offner, C.D.: Extending OpenMP for NUMA Machines. In: SC (2000)
5. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.A.: Scheduling Dynamic OpenMP Applications over Multicore Architectures. In: IWOMP. LNCS, vol. 5004, pp. 170–180. Springer (2008)
6. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.A.: Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In: IWOMP. LNCS, vol. 5568, pp. 79–92. Springer (2009)
7. GNU: GNU libgomp (2010), http://gcc.gnu.org/onlinedocs/libgomp/
8. Goglin, B., Furmento, N.: Enabling High-performance Memory Migration for Multithreaded Applications on LINUX. In: IPDPS. pp. 1–9. IEEE (2009)
9. Intel: Intel QuickPath Architecture (2010), www.intel.com/technology/quickpath/whitepaper.pdf
10. Jenks, S., Gaudiot, J.L.: Exploiting Locality and Tolerating Remote Memory Access Latency Using Thread Migration. Int. J. Parallel Program. 25(4), 281–304 (1997)
11. Jin, H., Frumkin, M.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Tech. rep., NASA (1999)
12. Kleen, A.: An NUMA API for Linux (2004), http://www.halobates.de/numaapi3.pdf
13. Lankes, S., Bierbaum, B., Bemmerl, T.: Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM (1). LNCS, vol. 6067, pp. 576–585. Springer (2009)
14. Marathe, J., Mueller, F.: Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: PPOPP. pp. 90–99. ACM (2006)
15. Nikolopoulos, D.S., Artiaga, E., Ayguadé, E., Labarta, J.: Scaling Non-regular Shared-memory Codes by Reusing Custom Loop Schedules. Scientific Programming 11(2), 143–158 (2003)
16. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E.: A Transparent Runtime Data Distribution Engine for OpenMP. Scientific Programming 8(3), 143–162 (2000)
17. Polychronopoulos, C.D., Kuck, D.J.: Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Trans. Computers 36(12), 1425–1439 (1987)
18. Rice University: High Performance Fortran Language Specification. SIGPLAN Fortran Forum 12(4), 1–86 (1993)
19. Robertson, N., Rendell, A.P.: OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In: International Conference on Computational Science. LNCS, vol. 2660, pp. 648–656. Springer (2003)
20. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: MAW '08: Proceedings of the 2008 workshop on Memory access on future processors. pp. 377–384. ACM (2008)
21. Tikir, M.M., Hollingsworth, J.K.: Using Hardware Counters to Automatically Improve Memory Performance. In: SC. p. 46. IEEE Computer Society (2004)