Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 981759, 16 pages
http://dx.doi.org/10.1155/2015/981759

Research Article

Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors

Ananya Muddukrishna,[1] Peter A. Jonsson,[2] and Mats Brorsson[1,2]

[1] *KTH Royal Institute of Technology, School of Information and Communication Technology, Electrum 229, 164 40 Kista, Sweden*
[2] *SICS Swedish ICT AB, Box 1263, 164 29 Kista, Sweden*

Correspondence should be addressed to Ananya Muddukrishna; ananya@kth.se

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

Performance degradation due to nonuniform data access latencies has worsened on NUMA systems and can now be felt on-chip in manycore processors. Distributing data across NUMA nodes and manycore processor caches is necessary to reduce the impact of nonuniform latencies. However, techniques for distributing data are error-prone and fragile and require low-level architectural knowledge. Existing task scheduling policies favor quick load-balancing at the expense of locality and ignore NUMA node/manycore cache access latencies while scheduling. Locality-aware scheduling, in conjunction with or as a replacement for existing scheduling, is necessary to minimize NUMA effects and sustain performance. We present a data distribution and locality-aware scheduling technique for task-based OpenMP programs executing on NUMA systems and manycore processors. Our technique relieves the programmer from thinking of NUMA system/manycore processor architecture details by delegating data distribution to the runtime system and uses task data dependence information to guide the scheduling of OpenMP tasks to reduce data stall times. We demonstrate our technique on a four-socket AMD Opteron machine with eight NUMA nodes and on the TILEPro64 processor and identify that data distribution and locality-aware task scheduling improve performance up to 69% for scientific benchmarks compared to default policies and yet provide an architecture-oblivious approach for programmers.

1. Introduction

NUMA systems consist of several multicore processors attached to local memory modules. Local memory can be accessed both faster and with higher bandwidth than remote memory by cores within a processor. Disparity between local and remote node access costs increases both in magnitude and nonuniformity as NUMA systems grow. Modern NUMA systems have reached such size and complexity that even simple memory-oblivious parallel executions such as the task-based Fibonacci program with work-stealing scheduling have begun to suffer from NUMA effects [1]. Careful data distribution is crucial for performance irrespective of memory footprint on modern NUMA systems.

Data distribution is also required on manycore processors which exhibit on-chip NUMA effects due to banked shared caches. Cores can access their local cache bank faster than remote banks. The latency of accessing far-off remote cache banks approaches off-chip memory access latencies. Another performance consideration is that cache coherence of manycore processors is software configurable [2]. Scheduling should adapt to remote cache bank access latencies that can change based on the configuration.

Scheduling decisions of the runtime system are key to task-based program performance. Scheduling decisions are made according to scheduling policies which until now have focused mainly on load-balancing—distributing computation evenly across threads. Load-balancing is a simple decision requiring little information from task abstractions used by the programmer and has been effective for several generations of multicore processors.

However, scheduling policies need to minimize memory access costs in addition to load-balancing for performance on NUMA systems and manycore processors. Strict load-balancing policies lose performance since they neglect data locality exposed by tasks. Neglecting data locality violates design principles of the complex memory subsystems that support NUMA systems and manycore processors. The subsystems require scheduling to keep cores running uninterrupted and provide peak performance by exploiting data locality.

Despite rising importance of data distribution and scheduling, OpenMP—a popular and widely available task-based programming paradigm—neither specifies data distribution mechanisms for programmers nor provides scheduling guidelines for NUMA systems and manycore processors even in the latest version 4.0.

Current data distribution practices on NUMA systems are to either use third-party tools and APIs [3–5] or repurpose the OpenMP `for` work-sharing construct to allocate and distribute data to different NUMA nodes. The third-party tools are fragile and might not be available on all machines and the clever use of the `parallel for` work-sharing construct [6] relies on a particular OS page management policy and requires the programmer to know about the NUMA node topology on the target machine.

Similar data distribution effort is required on manycore processors. For example, programmers directly use system API to distribute data on shared cache banks on the TILEPro64. There are no third-party tools to simplify data distribution effort. Programmers additionally have to match data distribution choice with numerous configurations available for the cache hierarchy for performance.

Expert programmers can still work around existing data distribution difficulties, but even for experts the process can be described as fragile and error-prone. Average programmers who do not manage to cope with all the complexity at once pay a performance penalty when running their programs, a penalty that might be partially mitigated from clever caching by the hardware. The current situation will get increasingly worse for everybody since NUMA effects are exacerbated by growing network diameters and increased cache coherence complexity [7] that inevitably follow from increasing sizes of NUMA systems and manycore processors.

We present a runtime system assisted data distribution scheme that allows programmers to control data distribution in a portable fashion without forcing them to understand low-level system details of NUMA systems and manycore processors. The scheme relies on the programmer to provide high-level hints on the granularity of the data distribution in calls to `malloc`. Programs without hints will work and have the same performance as before, which allows gradual addition of hints to programs to get partial performance benefits. Our runtime system assisted distribution scheme requires nearly the same programmer effort as regular calls to `malloc` and yet doubles the performance for some scientific workloads on NUMA systems.

We also present a locality-aware scheduling algorithm for OpenMP tasks which reduces memory access times by leveraging locality information gained from data distribution and task data footprint information from the programmer. Our scheduling algorithm improves performance over existing schedulers by up to 50% on our test NUMA system and 88% on our test manycore processor in programs where NUMA effects degrade program performance and remains competitive for other programs. Performance of scientific programs—blocked matrix multiplication and vector cross product—improves by 14% and 69%, respectively, when the locality-aware scheduler is used.

Table 1: Simple data distribution policies for the programmer.

Policy	Behavior
Standard	Delegate data distribution to the OS.
Fine	Distribute data, unit-wise round-robin, across all locations.
Coarse	Distribute data units, per-allocation round-robin, across all locations.

Table 2: Data distribution policy abstractions.

System	Unit	Location
NUMA system	Page	NUMA node
TILEPro64	Cache line	Home cache

The paper is an extension of our previous work on NUMA systems [8] and manycore processors [9]. We provide common data distribution mechanisms (Tables 1 and 2) and unify the presentation of locality-aware scheduling mechanisms (Algorithms 1, 2, and 3) for both NUMA systems and manycore processors. The new experimental setup for manycore processors enables L1 caching (Section 5.2) for a more realistic scenario. We disabled L1 caching in previous work to isolate locality-aware scheduling effects. We provide new measurements for manycore processors with a work-stealing scheduler as the common baseline (Figures 9 and 10). Previous work used a central queue-based scheduler as the baseline for manycore processors. We demonstrate the impact of vicinity sizes while stealing tasks (Figure 11), which is not done in previous work.

2. Potential for Performance Improvements

We quantify the performance improvement from data distribution by means of an experiment conducted on an eight-NUMA node system with four AMD Opteron 6172 processors. The topology of the system is shown in Figure 1. The maximum NUMA distance of the system according to the OS is 22, which is an approximation of the maximum latency between two nodes. NUMA interconnects of the system are configured for maximum performance with an average NUMA factor of 1.19 [11]. Latencies to access 4 MB of memory from different NUMA nodes measured using the BenchIT tool are shown in Figure 2. Detailed memory latencies of a similar system are reported by Molka et al. [12].

We execute task-based OpenMP programs using Intel's OpenMP implementation with two different memory allocation strategies: the first strategy uses malloc with the *first-touch* policy and the second distributes memory pages evenly across NUMA nodes using the numactl tool [5]. We use first-touch as a short hand for malloc with first-touch policy in the rest of the paper. We measure execution time of the parallel section of each program and quantify the amount of time

```
Procedure deal-work(task T, queues $Q_1, \ldots, Q_N$, current node n, cores per node C)
    Populate $D[1:N]$ with bytes in T.depend_list;
    if $sum(D) > sizeof(LLC)/C$ and $Standard\_Deviation(D) > 0$ then
        find $Q_l$ with least NUMA distance-weighted cost to $D$;
        enqueue($Q_l$, T);
    else
        enqueue($Q_n$, T);
    end
end
```

Algorithm 1: Work-dealing algorithm for NUMA systems.

```
Procedure find-work(queues $Q_1, \ldots, Q_N$, current node n, cores per node C)
    if empty $Q_n$ then
        for $Q_i$ in (Sort $Q_1, \ldots, Q_N$ by NUMA distance from n) do
            if $sizeof(Q_i) > distance(i, n) * C$ then
                Run dequeue($Q_i$);
                break;
            end
        end
    else
        Run dequeue($Q_n$);
    end
end
```

Algorithm 2: Work-finding algorithm for NUMA systems.

```
Procedure deal-work(task T, queues $Q_1, \ldots, Q_N$, current home cache n,
current data distribution policy p, access-intensive dependence index a)
    if $p == coarse$ then
        if exists $a$ then
            find $Q_a$ containing T.depend_list[$a$];
            enqueue($Q_a$, T);
        else
            Populate $D[1:N]$ with bytes in T.depend_list;
            if $sum(D) > sizeof(L1)$ then
                find $Q_l$ with least home cache latency cost to $D$;
                enqueue($Q_l$, T);
            else
                enqueue($Q_n$, T);
            end
        end
    else
        enqueue($Q_n$, T);
    end
end
```

Algorithm 3: Work-dealing algorithm for TILEPro64.

spent waiting for memory by counting dispatch stall cycles which includes load/store unit stall cycles [13].

Several programs show a reduction in execution time when data is distributed across NUMA nodes as shown in Figure 3. The reduction in dispatch stall cycles contributes to the reduction in execution time. Performance is maintained with data distribution for all remaining programs except Strassen.

We can explain why benchmarks maintain or lose performance with data distribution. Alignment scales linearly which implies low communication. Data distribution does not relieve the memory subsystem for FFT, Health, SparseLU,

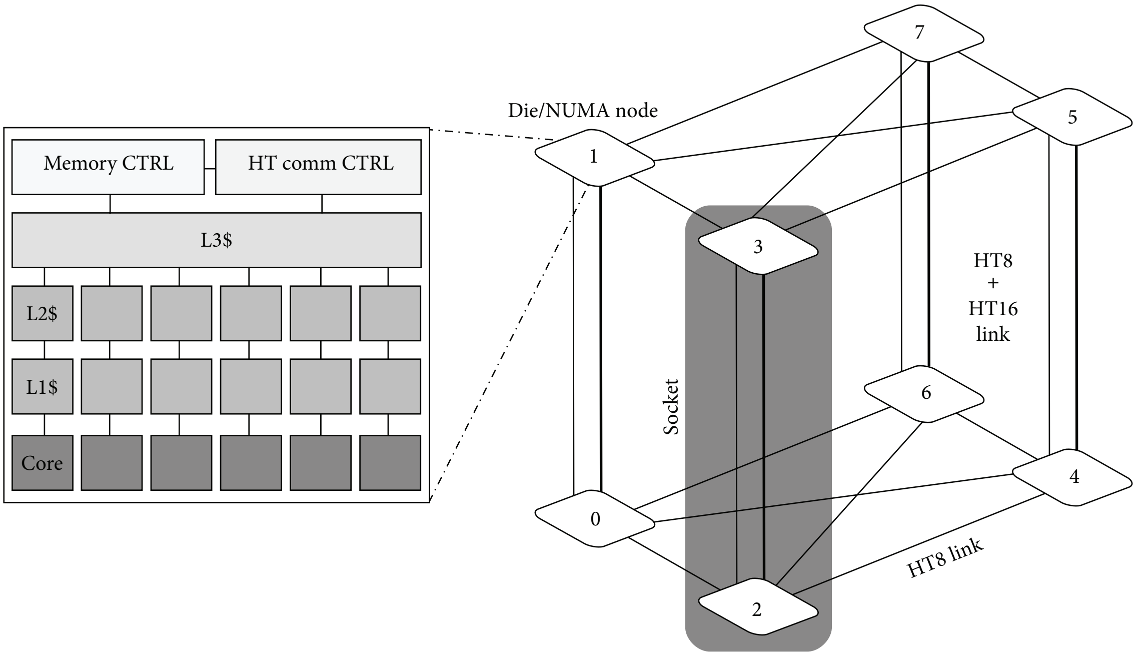


Figure 1: Topology of eight NUMA node, 48-core system with four AMD Opteron 6172 processors. Each processor has a 64 KB DL1 cache, a 512 KB L2 cache and a 5 MB L3 cache.

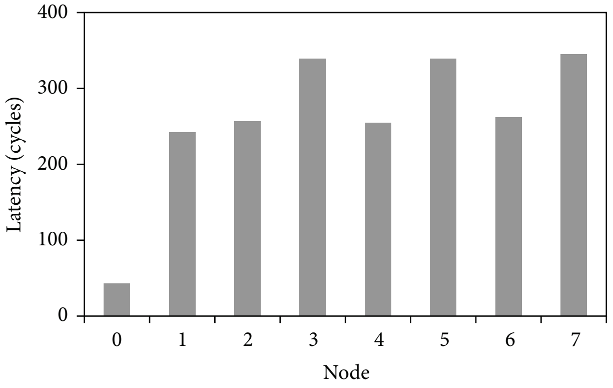


Figure 2: Latencies measured while accessing 4 MB of data allocated on different NUMA nodes from node 0 of the eight-node Opteron system. Remote node access is expensive.

and Strassen benchmarks. Execution time of Health surprisingly improves despite increased dispatch stall cycles implying bandwidth improvements with data distribution. Strassen is a counter-example whose performance degrades from data distribution. Strassen allocates memory inside tasks. Distributing the memory incurs higher access latencies than first-touch.

We demonstrate how locality-aware task scheduling used in conjunction with data distribution can further improve performance by means of an experiment on the TILEPro64 manycore processor. We explain the experiment after introducing key locality features of the TILEPro64 architecture.

The TILEPro64 is a 64-core tiled architecture processor with a configurable directory-based cache coherence protocol

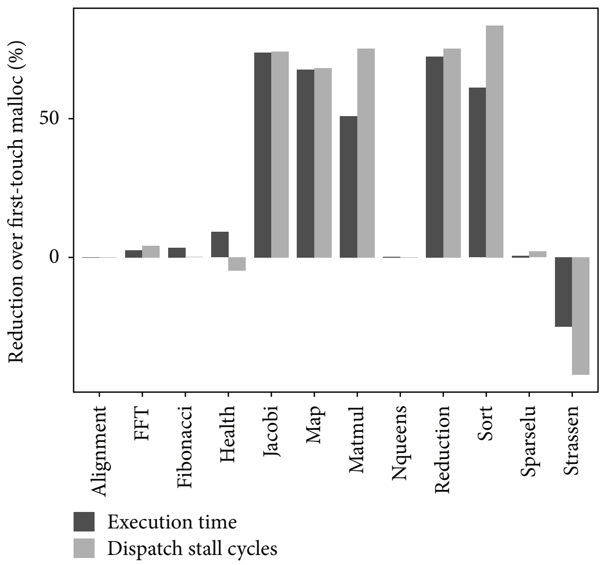


Figure 3: Performance impact of data distribution compared to first-touch in programs taken or derived from the Barcelona OpenMP Task Suite (BOTS) [10] and executed on the eight-node Opteron system. Execution time corresponds to the critical path of parallel section. Dispatch stall cycles are aggregated over all program tasks. Most programs improve or maintain performance when data is distributed across NUMA nodes.

and topology as shown in Figure 4. Load and store misses from cores are handled by a specific L2 bank called the *home cache*. A cache line supplied by the home cache can be allocated selectively in the local L2 bank (inclusive) and the L1 cache depending on software configuration. Stores in a tile are

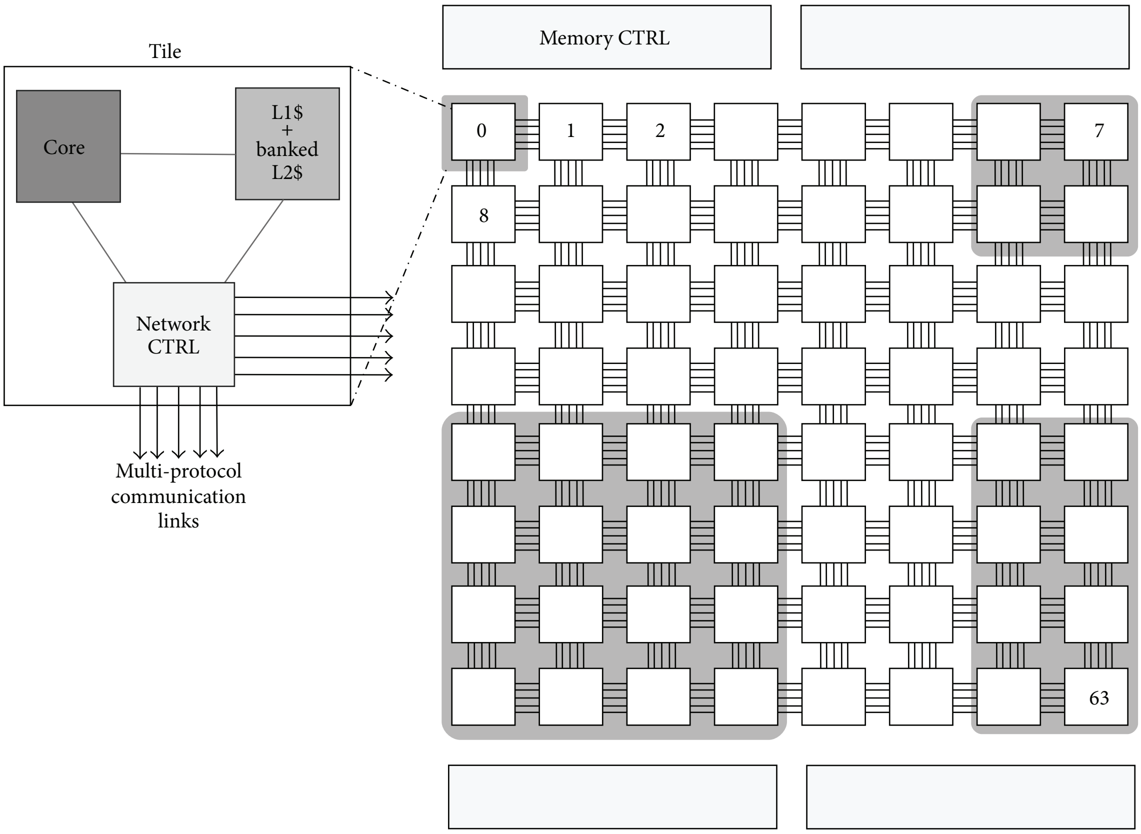


Figure 4: TILEPro64 topology. Tiles are connected by an 8 × 8 mesh on-chip network. Each tile contains a 32-bit VLIW integer core, a private 16 KB IL1 cache, a private 8 KB DL1 cache, a 64 KB bank of the shared 4 MB L2 cache and a network controller.

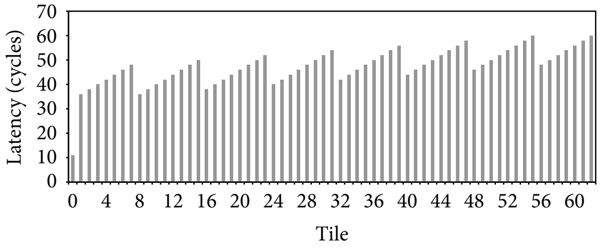


Figure 5: Latencies measured while accessing a cache line from different home cache access from tile 0 under isolation. Latencies increase in the presence of multiprogrammed workloads, OS, and hypervisor interference. Tile 63 runs dedicated system software and is excluded from measurement. Off-tile access takes 4–6 times longer.

always write-through to the home cache with a store update if the line is found in the L1 cache. Load latency of a cache line depends on home cache location and is nonuniform as shown in Figure 5. Remote home caches take four to six times longer to access than local home caches.

TILEPro64 system software also provides data distribution mechanisms. Cache lines in a main memory page can be distributed uniformly to all home caches or to a single home cache. The home cache of an allocated line can additionally be changed at a high cost through migration [14]. The performance impact of data distribution on the TILEPro64 is similar to NUMA systems. Memory allocations through malloc are uniformly distributed to all home caches.

We now explain our experiment to demonstrate locality-aware scheduling effectiveness. Consider *map* [15]—a common parallelization pattern as shown in Listing 1. Tasks operate on separate chunks of data in the map pattern. We execute the map program using two different strategies. Data is uniformly distributed to all home caches and work-stealing scheduling is used to assign tasks to idle cores in the first strategy. Data is distributed *per-allocation* to separate home caches and locality-aware scheduling is used to assign tasks to cores such that data is supplied by the local home cache in the second strategy.

The top and bottom graphs in the first column of Figure 10 show performance of the map program under the two strategies, respectively. The second strategy outperforms the first. Tasks execute faster under locality-aware scheduling since data is supplied by the local home cache. Selectively assigning home caches to cache lines rather than uniformly distributing them is beneficial for locality-aware scheduling. Task performance suffers from nonuniform home cache access latencies due to uniform data distribution and work-stealing scheduling.

We conclude that overheads from memory access latencies are significant in OpenMP programs. Proper choice of data distribution and scheduling is crucial for performance.

```
/* Allocate and initialize data */
for(int i=0; i<N; i++) {
     list[i] = malloc(sizeof(int) * SZ);
     initialize(i, list[i], SZ);
}
/* Work in parallel */
for(int i=0; i<N; i++) {
#pragma omp task input(list[i][0:SZ-1])
     map(list[i], SZ);
}
#pragma omp taskwait
```

Listing 1: Parallel map implemented using OpenMP tasks.

Our goal is to provide simple and portable abstractions that minimize memory access overheads by performing data distribution and enabling scheduling that can exploit locality arising from data distribution.

3. Runtime System Assisted Data Distribution

Runtime system assisted data distribution is one mechanism for increasing performance portability. Handling specific OS and hardware details can be delegated to an architecture-specific runtime system which has a global view of program execution.

We propose a memory allocation and distribution mechanism controlled by a simple data distribution policy that is chosen by the programmer. The distribution policy choice is deliberately kept simple with only a few choices in order to provide predictable behavior and be easy to understand for the programmer, just like process binding hints in OpenMP are defined. There are two different policies available to the programmer as shown in Table 1. *Unit* and *location* abstractions used in policy descriptions are explained in Table 2.

We demonstrate how the data distribution policies work and propose preliminary interfaces for policy selection using an example program in Listing 2. The program makes memory allocation requests A–E which span eight units of memory. Requests A and B use a proposed interface called `omp_malloc` whose signature is similar to `malloc`. The user selects the data distribution policy for requests A and B by setting a proposed environment variable called OMP_DATA_DISTRIBUTION to one of `standard`, `fine`, or `coarse` prior to the program invocation. The `standard` data distribution policy choice refers to the machine default—first-touch for NUMA systems and uniform distribution for TILEPro64. Memory requested using `omp_malloc` is distributed to different locations based on the global data distribution policy selected. Requests C–E use `omp_malloc_specific`—an extension of `omp_malloc`—to override the global policy and distribute specifically instead. Machine level results of policy actions are shown in Figure 6.

We provide heuristics in Table 3 to assist in the choice of data distribution policy. The heuristics are based on the number of data allocations through `malloc` in the original program and the number of tasks operating on those allocations. Programs with many tasks and a single malloc call will benefit from using the fine policy since cores can issue multiple outstanding requests to different nodes/home caches. Programs with a single task and many malloc calls can use the coarse policy to improve bandwidth since memory is likely to be fetched from different network links. Programs with many malloc calls and many tasks that operate on allocated data are likely to improve performance with the coarse policy due to reduced network contention assuming tasks work on allocations in isolation.

Table 3: Heuristics to select data distribution scheme.

		Number of tasks operating on data	
		One	Many
Number of `malloc` calls	One	Regular `malloc`	Fine
	Many	Coarse	Coarse

We have built the runtime system assisted distribution scheme using readily available support—libnuma on NUMA systems and special allocation interfaces on the TILEPro64. The overhead of the distribution scheme is low since our implementation wraps the system API with a few additional book-keeping instructions. The book-keeping instructions track the round-robin node selection counter for the coarse distribution policy and cache location affinity of data when requested by the locality-aware scheduling policy described in Section 4.

Programmers do not need to be concerned about NUMA node/home cache identifiers and topology in order to use our data distribution scheme. The distribution policy choice is kept simple with only those choices that are easy to predict and understand for the programmer. Programmers can also incrementally distribute data by targeting specific memory allocation sites. We also provide precise control for expert programmers in our implementation allowing them to override the global data distribution policy and request fine or coarse data distribution for a specific allocation. We have implemented two simple distribution policies to demonstrate the potential of our data distribution scheme. Runtime system developers can use the extensibility of our

```
int main(...) {
    ...
    /* Allocate data */
    size_t sz = 8 * UNIT_SIZE;
    /* NUMA system: UNIT_SIZE = PAGE_SIZE */
    /* TILEPro64: UNIT_SIZE = PAGE_SIZE/CACHE_LINE_SIZE */
    void* A = omp_malloc(sz);
    void* B = omp_malloc(sz);
    void* C = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void* D = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void* E = omp_malloc_specific(sz, OMP_MALLOC_FINE);
    /* Initialize data */
    init(A, B, C, D, E, sz, ...);
    /* Work in parallel */
    #pragma omp parallel
    {
    ...
}
...
}
$ export OMP_DATA_DISTRIBUTION=<standard | fine | coarse>
$ <invoke program>
```

LISTING 2: Program using the proposed interface for selecting data distribution policies.

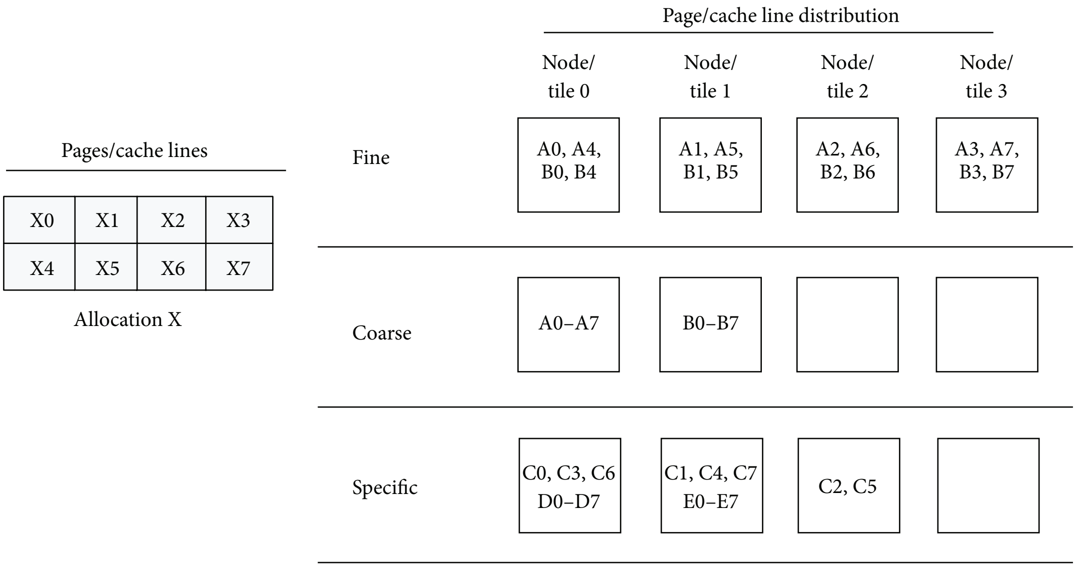


FIGURE 6: Data distribution results on an example four-node/four-tile ma-chine. We simplify illustration by using eight cache lines per page. In reality, over 64 cache lines typically make up a page.

scheme to provide more advanced distribution policies as plug-ins. Programmers can be educated about distribution policies in a manner similar to existing education about `for`-loop scheduling policies within the OpenMP specification.

4. Locality-Aware Task Scheduling

Our implementation of locality-aware scheduling aims to further leverage the performance benefits of data distribution. The main idea behind our locality-aware scheduler is to schedule tasks to minimize memory access latencies. The locality-aware scheduler uses an architecture-specific task queue organization and takes locality-aware decisions both during work-dealing and work-stealing. Work-dealing refers to actions taken at the point of task creation and work-stealing is actions taken when threads are idle.

Knowing the data footprint of tasks is crucial for the scheduler we expect data footprint information to come from the programmer through task definition clauses which do not yet exist in the OpenMP specification. We currently estimate the data footprint of each task through the information provided by the depend clause in the OpenMP 4.0 specification. The estimate is fragile when programmers specify an incomplete depend clause that is sufficient for scheduling decisions but underestimate the data footprint. The limitation can be overcome if programmers use low-effort expressive constructs such as array-sections to express a large fraction of the data footprint in the depend clause in return for improved performance.

The locality-aware scheduler binds task queues to architectural locations to which data can be distributed. There is a task queue per NUMA node on NUMA systems and per home cache on the TILEPro64. Tasks are added at the front and removed from the back of task queues. The scheduler binds one thread to each core.

4.1. NUMA Systems. We describe the work-dealing algorithm of the locality-aware scheduler in Algorithm 1. The scheduler deals a task at the point of task creation to the node queue having the least total memory access latency for pages not in the last-level cache (LLC). The individual access latencies are computed by weighting NUMA node distances with the node-wise distribution D of the data footprint of the task.

NUMA node distances are obtained from OS tables which are cached by the scheduler for performance reasons. The distribution D is calculated using page locality information cached by the data distribution mechanism. The complexity of the access cost computation is $O(N^2)$ where N is the number of NUMA nodes in the system, typically a small number.

Tasks are immediately added to the local queue when scheduling costs outweigh the performance benefits decided by two thresholds. The first threshold—$Sum(D) > sizeof(LLC)/C$—ensures that tasks have a working set size exceeding the LLC size per core. The second threshold—$Standard\_Deviation(D) > 0$—ensures that scheduling effort will not be wasted on tasks with a perfect data distribution.

Distributed task queues may lead to load-imbalance and in our experience the performance benefits from load-balancing often trumps those from locality. We have therefore implemented a work-stealing algorithm to balance the load. Stealing is still preferred over idle threads although cycles spent dealing tasks are wasted.

We show the stealing algorithm of the scheduler in Algorithm 2. Threads attempt to steal when there is no work in the local queue. Candidate queues for steals are ranked based on NUMA node distances. The algorithm includes a threshold which prevents tasks from being stolen from nearly empty task queues which would incur further steals for threads in the victim node. There is an exponential back-off for steal attempts when work cannot be found.

4.2. Manycore Processors. We describe the work-dealing algorithm of the locality-aware scheduler in Algorithm 3. The scheduler deals a task to the home cache queue having the least total memory access latency for cache lines not in the private L1 cache. The individual access latencies are computed by weighting home cache access latencies with the home-cache distribution D of the data footprint of the task. Home cache access latencies are calculated by benchmarking the interconnection during runtime system initialization. The scheduler avoids recalculation by saving latencies across runs. The distribution D is calculated using home cache locality information cached by the data distribution mechanism. The complexity of the access cost computation is $O(N^2)$ where N is the number of home caches in the system.

Tasks are immediately added to the local queue if scheduling costs outweigh the performance benefits. The algorithm ignores distribution policies which potentially distribute data finely to all home caches (condition $p == coarse$). Only tasks with a working set exceeding the L1 data cache are analyzed (condition $sum(D) > sizeof(L1)$).

Another condition—*exists a*—minimizes scheduling effort by using programmer information about the access intensity to data dependences in the list T.depend_list. The index a denotes the most intensely accessed data dependence in the list. The scheduler queues tasks with intensity information in the queue associated with the home cache containing the intensively accessed dependence. Note that we rely on a custom clause to indicate intensity since existing task definition clauses in OpenMP do not support the notion.

We have implemented a work-stealing algorithm to balance load on task queues. Queues are grouped into fixed size vicinities and idle threads are allowed to steal tasks from queues in the same vicinity. Cross-vicinity steals are forbidden. Threads additionally back off when work cannot be found. The size of the vicinity is selected by the programmer prior to execution. We allow vicinity sizes of 1, 4, 8, 16, and 63 tiles in our implementation as shown by tile groups in Figure 4. A vicinity size of 1 only includes the task queue of the member thread; vicinity size of 63 includes task queues of all threads.

5. Experimental Setup

We evaluated data distribution assistance and locality-aware scheduling using benchmarks described in Table 4. The benchmarks were executed using MIR, a task-based runtime system library which we have developed. MIR supports the OpenMP tied tasks model and provides hooks to add custom scheduling and data distribution policies which allows us to compare different policies within the same system. We programmed the evaluation benchmarks using the runtime system interface directly since MIR does not currently have a source-to-source translation front-end.

We ran each benchmark in isolation 20 times for all valid combinations of scheduling and data distribution policies. We recorded the execution time of the critical path of the parallel section and collected execution traces and performance counter readings on an additional set of runs for detailed analysis.

We used a work-stealing scheduler as the baseline for comparing the locality-aware scheduler. The work-stealing scheduler binds one thread to each core and uses one task queue per core. The task queue is the lock-free dequeue by Chase and Lev [16]. The implementation is an adaption of the queue from the Glasgow Haskell Compiler version 7.8.3

Table 4: Pattern-based [15] and real-world benchmarks.

Benchmark	Behavior	Data distribution heuristic guidance
Map (pattern-based)	1D vector scaling	Coarse
Reduction (pattern-based)	Iterative implementation of merge phase of BOTS Sort	Fine
Vecmul	Vector cross product	Coarse
Matmul	Blocked matrix multiplication with BLAS operations in task computation	Coarse
Jacobi	Blocked 2D heat equation solver	Fine
SparseLU	LU factorization of sparse matrix. Derived from BOTS SparseLU.	Coarse

runtime system. Each thread adds newly created tasks to its own task queue. Threads look for work in their own task queue first. Threads with empty task queues select victims for stealing in a round-robin fashion. Both queuing and stealing decisions of the work-stealing scheduler are fast but can result in high memory latencies during task execution since the scheduling is oblivious to data locality and NUMA node/remote cache access latencies.

5.1. NUMA System. We used the Opteron 6172 processor based eight-NUMA node system described in Section 2 for evaluation. Both runtime system and benchmarks were compiled using the Intel C compiler v13.1.1 with -O3 optimization. We used per-core cycle counters and dispatch stall cycle counters to, respectively, measure execution time and memory access latency of tasks.

5.2. Manycore Processor. Both runtime system and benchmarks were compiled using the Tilera GNU compiler with -O3 optimization. We used integer versions of evaluation benchmarks to rule out effects of slow software-emulated floating-point operations. Benchmark inputs were selected to minimize off-chip memory access. We also minimized the adverse effect of evicting local home cache entries to memory by disabling local L2 (inclusive) caching. We used per-core cycle counters and data cache stall cycle counters to, respectively, measure execution time and memory access latency of tasks.

The locality-aware scheduler avoids long home cache access latencies. The L1 cache also mitigates the impact of long home cache access latencies. We separated effects of locality-aware scheduling by disabling L1 caching in previous work [9] but enabled L1 caching in the current setup for a more realistic scenario.

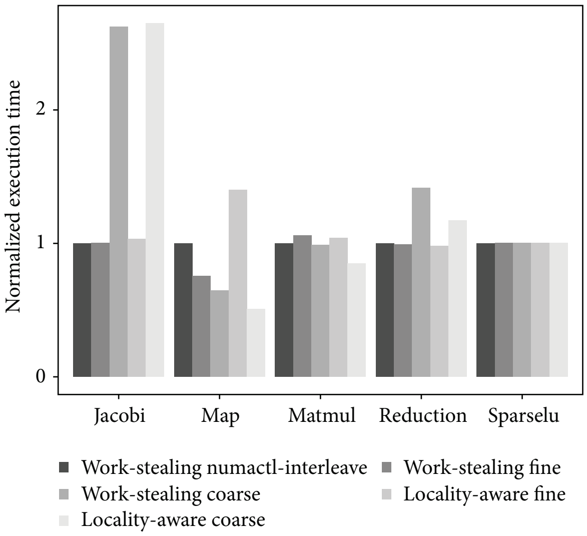


Figure 7: Performance of data distribution combined with work-stealing and locality-aware scheduling on eight-node Opteron system. Execution time is normalized to performance of work-stealing with memory page interleaving using `numactl` for each benchmark. Inputs to Map: 48 floating-point vectors, 1 MB each; Jacobi: 16384 × 16384 floating-point matrix and block size = 512; Matmul: 4096 × 4096 floating-point matrix and block size = 128; SparseLU: 8192 × 8192 floating-point matrix and block size = 256; Reduction: 256 MB floating-point array and depth = 10. Combination of numactl page-wise interleaving and locality-aware scheduling is excluded since the locality-aware scheduler does not currently support querying numactl for page locality information. Locality-aware scheduling, in combination with heuristic-guided data distribution, improves or maintains performance compared to work-stealing.

6. Results

We show performance of evaluation benchmarks for combinations of data distribution and scheduling policy for the eight-node Opteron system in Figure 7. The fine distribution is a feasible replacement for numactl since execution times with the work-stealing scheduler are comparable to page-wise interleaving using numactl. Performance degrades when distribution policies violate the guidelines in Table 3 for both work-stealing and locality-aware schedulers. For example, performance of Matmul degrades when the fine distribution policy is used. The locality-aware scheduler coupled with proper data distribution improves or maintains performance compared to the work-stealing scheduler for each benchmark.

We use thread timelines for Map and Matmul in Figure 8 to explain that reduced memory page access time is the main reason behind the difference in task execution times of the work-stealing and locality-aware scheduler.

The thread timeline indicates time spent by threads in different states and state transition events. Threads are shown on the y-axis, time is shown on the x-axis, and memory access latencies are shown on the z-axis. The z-axis is represented using a linear green to blue gradient which encodes memory

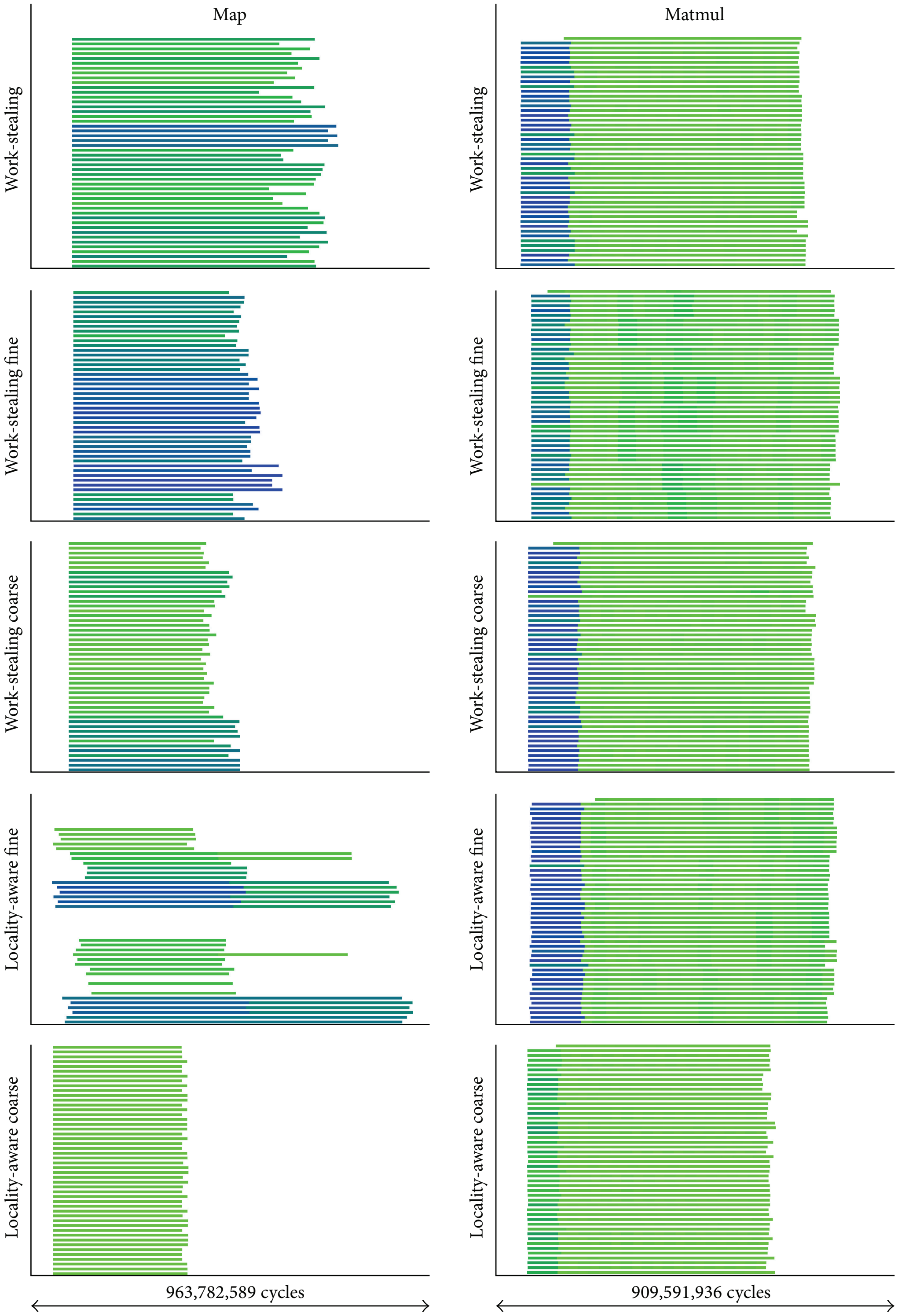


FIGURE 8: Thread timelines showing task execution on the eight-node Opteron system. Threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Tasks stall for fewer cycles under locality-aware scheduling combined with heuristic-guided data distribution.

access latencies measured at state transition boundaries. Green represents lower memory access latencies and blue represents higher ones. We filter out all thread states except task execution. Timelines of a benchmark are time aligned (same x-axis span) and gradient aligned (same z-axis span). Timelines are additionally zoomed-in to focus on task execution and omit runtime system initialization activity.

Understanding benchmark structure is also necessary to explain the performance difference. Each task in the Map benchmark scales a separate vector in a list. Coarse distribution places all memory pages of a given vector in a single node whereas fine distribution spreads the pages uniformly across all nodes.

The locality-aware scheduler combined with coarse distribution minimizes node access latency by ensuring that each task accesses its separate vector from the local node. The behavior can be confirmed by low memory access latencies seen in Figure 8 (light green). The work-stealing scheduler with coarse distribution loses performance due to increased remote memory access latencies as indicated by the relatively higher memory access latencies (dark green and blue).

We can also explain performance of cases that violate the guidelines by using timelines. The locality-aware scheduler with fine distribution detects that pages are uniformly distributed across nodes and places all tasks in the same local queue. The imbalance can not be completely recovered from since steals are restricted. The work-stealing scheduler with fine distribution balances loads more effectively in comparison.

Each task in the Matmul benchmark updates a block in the output matrix using a chain of blocks from two input matrices. Coarse distribution places all memory pages of a given block in a single node whereas fine distribution spreads the pages uniformly across all nodes. The memory pages touched by a task are located on different nodes for both coarse and fine distribution. The locality-aware scheduler with fine distribution detects that data is evenly distributed and falls back to work-stealing by queuing tasks in local queues. Task execute for a longer time with both schedulers as indicated by similar memory access latency (similar intensity of green and blue). However, the locality-aware scheduler with coarse distribution exploits locality arising from distributing blocks in round-robin as indicated by the relatively lower memory access latency (lighter intensity of green and blue) in comparison to the work-stealing scheduler.

We show the performance of evaluation benchmarks for combinations of data distribution and scheduling policy for the TILEPro64 processor in Figure 9. Results are similar to those on the eight-node Opteron system. Performance degrades when distribution policies are chosen against heuristic guidelines in Table 3 for both work-stealing and locality-aware schedulers. The locality-aware scheduler coupled with proper data distribution improves or maintains performance compared to the work-stealing scheduler for each benchmark. Locality-aware scheduler performance is also sensitive to vicinity sizes.

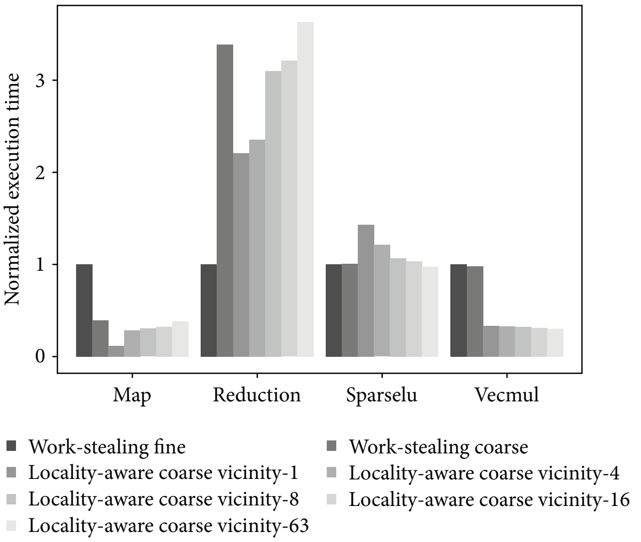


Figure 9: Performance of data distribution combined with work-stealing and locality-aware scheduling on TILEPro64. Execution time is normalized to performance of work-stealing scheduling with fine distribution for each benchmark. Inputs to Map: 63 integer vectors, 32 kB each; Reduction: 700 kB integer array and depth = 6; Vecmul: 128 integer vectors, 28 kB each; SparseLU: 1152 × 1152 integer matrix, block size = 36, and intensity heuristic enabled for tasks executing the *bmod* function. Locality-aware scheduling, in combination with heuristic-guided data distribution, improves or maintains performance compared to work-stealing.

SparseLU is a counter-example whose performance degrades with heuristic-guided coarse distribution and work-stealing scheduling. Performance is also maintained with both coarse and fine distribution on NUMA systems. SparseLU tasks have complex data access patterns which require a data distribution scheme more advanced than fine and coarse.

Reduction allocates memory using a single `malloc` call. Coarse distribution is a bad choice since all cache lines are allocated in a single home cache. Locality-aware scheduling serializes execution by scheduling tasks on the core associated with the single home cache. Stealing from larger vicinities balances load to win back performance.

Thread timelines for Map and Vecmul in Figure 10 confirm that reduced cache line access time is the main reason behind the reduction in task execution times. The work-stealing scheduler loses performance by being oblivious to locality despite balancing the load evenly.

We can explain vicinity sensitivity using timelines for Map and Vecmul benchmarks in Figure 11. Increasing vicinity sizes for Map increases the risk of tasks being stolen by threads far from the home cache. Stolen tasks experience large and nonuniform cache line access latencies as shown by long blue bars. Threads fast enough to pick tasks from their own queue finish execution faster. Larger vicinity sizes promote better load-balancing and improve performance in Vecmul.

The locality-aware scheduler can safely be used as the default scheduler for all workloads without performance degradation. There is a performance benefit in using the locality-aware scheduler for workloads which provide strong locality with data distribution. The locality-aware scheduler

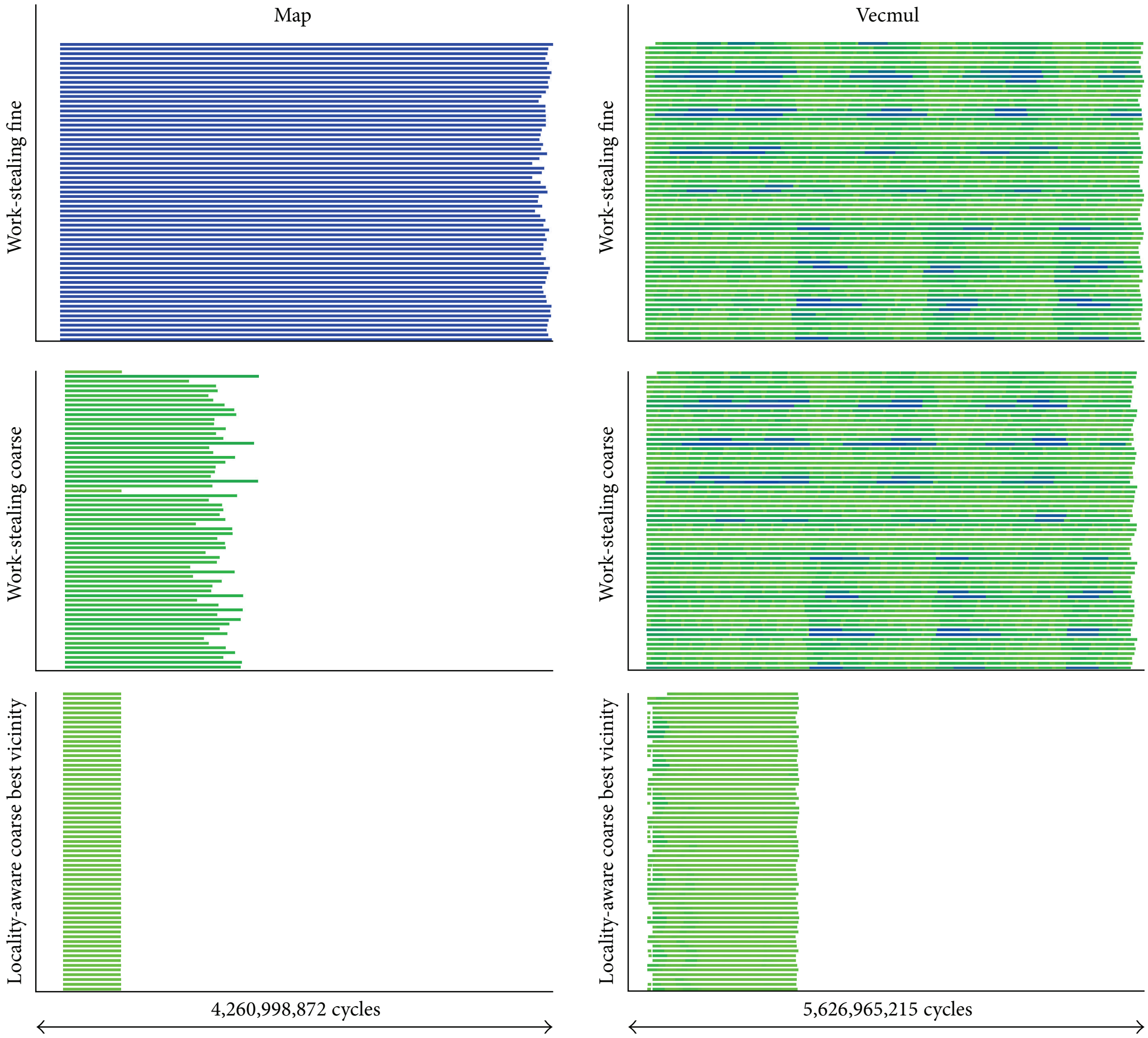


Figure 10: Thread timelines showing task execution on the TILEPro64. Threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Tasks access memory faster under locality-aware scheduling combined with heuristic-guided data distribution.

falls back to load-balancing similar to work-stealing scheduler for workloads which do not improve locality with data distribution.

7. Related Work

Numerous ways of how to distribute data programmatically on NUMA systems have been proposed in the literature. We discuss the proposals that are closest to our approach.

Huang et al. [17] propose extensions to OpenMP to distribute data over an abstract notion of locations. The primary distribution scheme is a block-wise distribution which is similar to our coarse distribution scheme. The scheme allows precise control of data distribution but relies on compiler support and additionally requires changes to the OpenMP specification. Locations provide fine-grained control over data distribution at the expense of programming effort.

The Minas framework [4] incorporates a sophisticated data distribution API which gives precise control on where memory pages end up. The API is intended to be used by an automatic code transformation in Minas that uses profiling information for finding the best distribution for a given program. The precise control is powerful but requires expert programmers who are capable of writing code that will decide on the distribution required.

Majo and Gross [18] use fine-grained data distribution API to distribute memory pages. Execution profiling is used to get data access patterns of loops and used for both guiding code transformation and data distribution. Data distribution is performed in between loop iterations which guarantee that each loop iteration accesses memory pages locally.

Runtime tracing techniques that provide automatic page migration based on hardware monitoring through performance counters have the same end goal as we do: to provide good performance with low programming effort. Nikolopoulos et al. [19] pioneered the idea of page migration with user-level framework. Page access is traced in the background and *hot* pages are migrated closer to the accessing node.

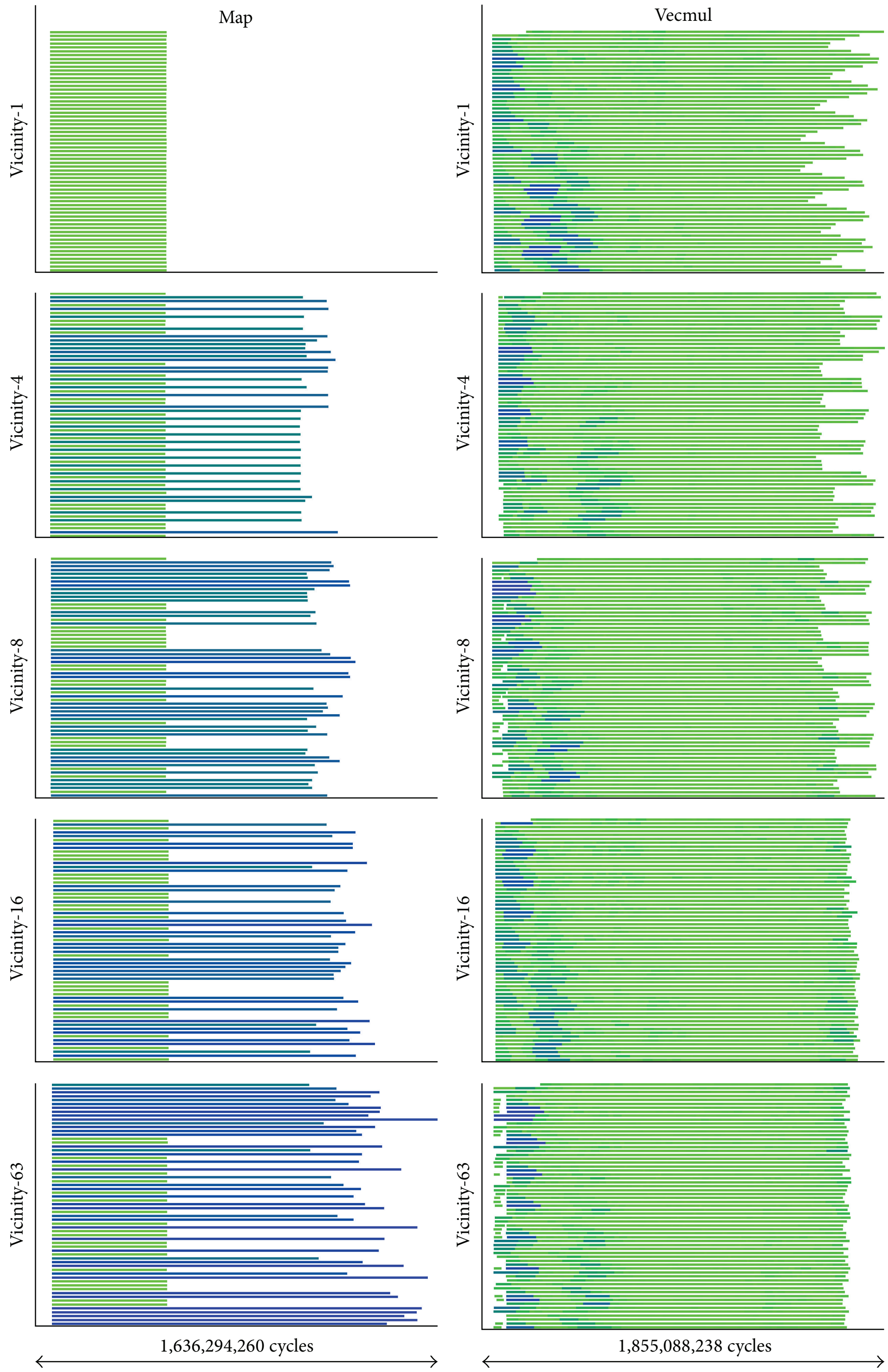


FIGURE 11: Vicinity size sensitivity on the TILEPro64. On each thread timeline, threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Increasing vicinity size improves load balance but adversely affects memory access time.

Terboven et al. [20] presented a next-touch dynamic page migration implementation on Linux. An alternative approach to page migration, which is expensive, is to move threads instead, an idea exploited by Broquedis et al. [21] in a framework where decisions to migrate threads and data are based on information about thread idleness, available node memory, and hardware performance counters. Carrefour is a modification of the Linux kernel that targets traffic congestion for NUMA systems through traffic management by page replication and page migration [22]. One advantage of the approach is that performance will improve without having to modify applications.

Dynamic page migration requires no effort from the programmer, which is a double edged sword. The benefit of getting good performance without any effort is obvious, but when the programmer experiences bad performance it is difficult to analyze the root cause of the problem. Performance can also be affected by input changes. Attempts at reducing the cost of page migration by providing native kernel support give promising results for matrix multiplication on large matrices [23].

Locality-aware scheduling for OpenMP has been studied extensively. We focus on other task-based approaches since our approach is based on tasks.

Locality domains where programmers manually place tasks in abstract bins have been proposed [1, 24]. Tasks are scheduled within their locality domain to reduce remote memory access. MTS [25] is a scheduling policy structured on the socket hierarchy of the machine. MTS uses one task queue per socket which is similar to our task queue per NUMA node. Only one idle core per socket is allowed to steal bulk work from other sockets. Charm++ uses NUMA topology information and task communication information to reduce communication costs between tasks [26]. Chen et al. [27] reduce performance degradation from cache pollution and stealing tasks across sockets in multisocket systems by memory access aware task-graph partitioning.

Memphis uses hardware monitoring techniques and provides methods to fix NUMA problems on general class of OpenMP computations [7]. Monitoring crossbar (QPI) related and LLC cache miss related performance counters is used to measure network activity. Memphis provides diagnostics to the programmer for when to pin threads, distribute memory, and keep computation in a consistent shape throughout the execution. Their recommendations have inspired the design of our locality-aware scheduler and our evaluation methodology.

Liu and Mellor-Crummey [28] add detailed NUMA performance measurement and data distribution guidance capability to HPCToolkit. They report several case studies where coarse (block-wise) distribution improves performance over default policies. Their multiarchitecture tool is a good starting-point for implementing advanced data distribution policies.

Schmidl et al. propose the keywords scatter and compact for guiding thread placement using SLIT-like distance matrices [29]. Our names for data distribution, fine and coarse, are directly inspired by their work.

Task and data affinity mechanisms discussed in our work are greatly inspired by the large body of research on NUMA optimizations for OpenMP runtime systems. The implicit memory allocation and architectural locality based scheduling mechanisms we implemented in the runtime system are inspired by a similar work on NUMA systems by Broquedis et al. [30].

Few works have tackled data distribution and locality-aware scheduling on manycore processors.

Yoo et al. [31] provide an in-depth quantitative analysis of locality-aware scheduling for data-parallel programs on manycore processors. They conclude that work-stealing scheduling cannot capture locality present in data-parallel programs which we also demonstrate through scheduling results for the map program. They propose a sophisticated locality-aware scheduling and stealing technique that maximizes the probability of finding the combined memory footprint of a task group in the lowest level cache that can accommodate the footprint. The technique however requires task grouping and ordering information obtained by profiling read-write sets of tasks and off-line graph analysis.

Vikranth et al. [32] propose to restrict stealing to groups of cores based on processor topology similar to our vicinity-based stealing approach.

Tousimojarad and Vanderbauwhede [33] cleverly reduce access latencies to uniformly distributed data by using copies whose home cache is local to the access thread on the TILEPro64 processor. Zhou and Demsky [2] build a NUMA-aware adaptive garbage collector that migrate objects to improve locality on manycore processors. We target standard OpenMP programs written in C which makes it difficult to migrate objects.

Techniques to minimize cache access latency by capturing access patterns and laying out data both at compile-time and runtime have been proposed for manycore processors. Lu et al. [34] rearrange affine for-loops during compilation to minimize access latency to data distributed uniformly on banked shared caches of manycore processors. Marongiu and Benini [35] extend OpenMP with interfaces to partition arrays which are then distributed by their compiler backend based on profiled access patterns. The motivation for their work is enabling data distribution on MPSoCs without hardware support for memory management. Li et al. [36, 37] use compilation-time information to guide the runtime system in data placement. R-NUCA automatically migrates shared memory pages to shared cache memory using OS support reducing hardware costs for cache coherence [38].

8. Conclusions

We have presented a data distribution and memory page/cache line locality-aware scheduling technique that gives good performance in our tests on NUMA systems and manycore processors. The major benefit is usage simplicity which allows ordinary programmers to reduce their suffering from NUMA effects which hurt performance. Our technique is easy to adopt since it is built using standard components provided by the OS. The locality-aware scheduler can be used as the default scheduler since it will fall back to behaving like a work-stealing scheduler when locality is missing, something also indicated from our measurements.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The work was partially funded by European FP7 project ENCORE Project Grant Agreement no. 248647 and Artemis PaPP Project no. 295440.

References

[1] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *Proceedings of the 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–12, Salt Lake City, Utah, USA, November 2012.

[2] J. Zhou and B. Demsky, "Memory management for many-core processors with software configurable locality policies," *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 3–14, 2012.

[3] F. Broquedis, J. Clet-Ortega, S. Moreaud et al., "Hwloc: a generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '10)*, pp. 180–186, February 2010.

[4] C. P. Ribeiro, M. Castro, J.-F. Méhaut, and A. Carissimi, "Improving memory affinity of geophysics applications on NUMA platforms using minas," in *High Performance Computing for Computational Science—VECPAR 2010*, vol. 6449 of *Lecture Notes in Computer Science*, pp. 279–292, Springer, Berlin, Germany, 2011.

[5] A. Kleen, *A NUMA API for Linux*, Novel, Kirkland, Wash, USA, 2005.

[6] C. Terboven, D. Schmidl, T. Cramer, and D. An Mey, "Assessing OpenMP tasking implementations on NUMA architectures," in *OpenMP in a Heterogeneous World*, vol. 7312 of *Lecture Notes in Computer Science*, pp. 182–195, Springer, Berlin, Germany, 2012.

[7] C. McCurdy and J. S. Vetter, "Memphis: finding and fixing NUMA-related performance problems on multi-core platforms," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '10)*, pp. 87–96, March 2010.

[8] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, vol. 8122 of *Lecture Notes in Computer Science*, pp. 156–170, Springer, Berlin, Germany, 2013.

[9] A. Muddukrishna, A. Podobas, M. Brorsson, and V. Vlassov, "Task scheduling on manycore processors with home caches," in *Euro-Par 2012: Parallel Processing Workshops*, Lecture Notes in Computer Science, pp. 357–367, Springer, Berlin, Germany, 2013.

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proceedings of the International Conference on Parallel Processing (ICPP '09)*, pp. 124–131, Vienna, Austria, September 2009.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[12] D. Molka, R. Schöne, D. Hackenberg, and M. Müller, "Memory performance and SPEC OpenMP scalability on quad-socket x86_64 systems," in *Algorithms and Architectures for Parallel Processing*, vol. 7016 of *Lecture Notes in Computer Science*, pp. 170–181, Springer, Berlin, Germany, 2011.

[13] AMD, BIOS and kernel developer's guide for AMD family 10h processors, 2010.

[14] Tilera, *Tile Processor User Architecture Manual*, 2012, http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf.

[15] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, Elsevier, 2012.

[16] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*, pp. 21–28, ACM, Las Vegas, Nev, USA, July 2005.

[17] L. Huang, H. Jin, L. Yi, and B. Chapman, "Enabling locality-aware computations in OpenMP," *Scientific Programming*, vol. 18, no. 3-4, pp. 169–181, 2010.

[18] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for NUMA systems," in *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO '12)*, pp. 230–241, April 2012.

[19] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade, "Is data distribution necessary in OpenMP?" in *Proceedings of the ACM/IEEE Conference on Supercomputing (CDROM '07)*, p. 47, November 2000.

[20] C. Terboven, D. Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in OpenMP programs," in *Proceedings of the Workshop on Memory Access on Future Processors: A Solved Problem? (MAW '08)*, pp. 377–384, May 2008.

[21] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568 of *Lecture Notes in Computer Science*, pp. 79–92, Springer, Berlin, Germany, 2009.

[22] M. Dashti, A. Fedorova, J. Funston et al., "Traffic management: a holistic approach to memory placement on NUMA systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 381–394, ACM, March 2013.

[23] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on linux," in *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS '09)*, pp. 1–9, May 2009.

[24] M. Wittmann and G. Hager, "Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems," Computing Research Repository, http://arxiv.org/abs/1101.0093.

[25] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, 2012.

[26] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, and J.-F. Méhaut, "Charm++ on NUMA platforms: the impact of SMP optimizations and a NUMA-aware load balancer," in *Proceedings of the 4th Workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing*, Urbana, Ill, USA, 2010.

[27] Q. Chen, M. Guo, and Z. Huang, "Adaptive cache aware bitier work-stealing in multisocket multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2334–2343, 2013.

[28] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pp. 259–271, ACM, Orlando, Fla, USA, February 2014.

[29] D. Schmidl, C. Terboven, and D. an Mey, "Towards NUMA support with distance information," in *OpenMP in the Petascale Era*, vol. 6665 of *Lecture Notes in Computer Science*, pp. 69–79, Springer, Berlin, Germany, 2011.

[30] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P. Wacrenier, "Dynamic task and data placement over numa architectures: an openmp runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568 of *Lecture Notes in Computer Science*, pp. 79–92, Springer, Berlin, Germany, 2009.

[31] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: a quantitative limit study," in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*, pp. 315–325, ACM, Portland, Ore, USA, July 2013.

[32] B. Vikranth, R. Wankar, and C. R. Rao, "Topology aware task stealing for on-chip NUMA multi-core processors," *Procedia Computer Science*, vol. 18, pp. 379–388, 2013.

[33] A. Tousimojarad and W. Vanderbauwhede, "A parallel task-based approach to linear algebra," in *Proceedings of the IEEE 13th International Symposium on Parallel and Distributed Computing (ISPDC '14)*, pp. 59–66, IEEE, 2014.

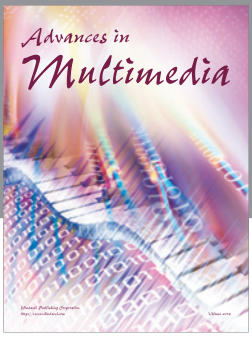
[34] Q. Lu, C. Alias, U. Bondhugula et al., "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*, pp. 348–357, IEEE, September 2009.

[35] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 222–236, 2012.

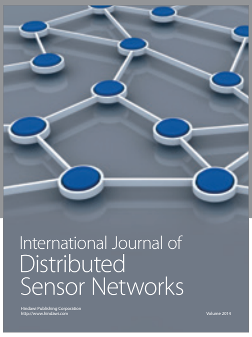
[36] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pp. 501–512, ACM, September 2010.

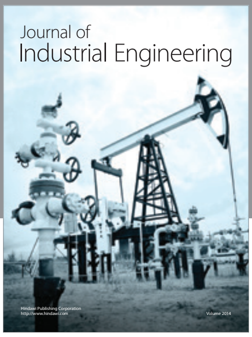
[37] Y. Li, R. Melhem, and A. K. Jones, "Practically private: enabling high performance CMPs through compiler-assisted data classification," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, pp. 231–240, ACM, September 2012.

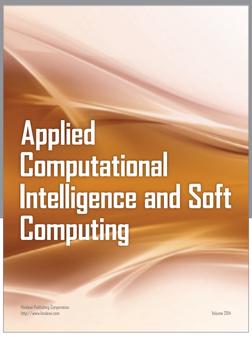
[38] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 184–195, 2009.

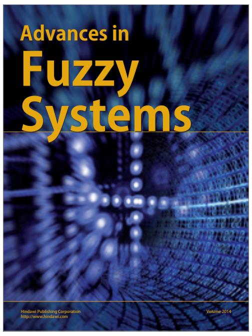

Advances in
Multimedia

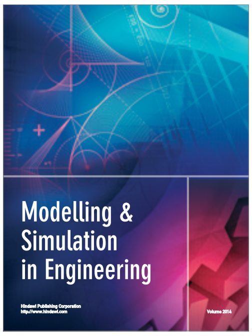
The Scientific
World Journal

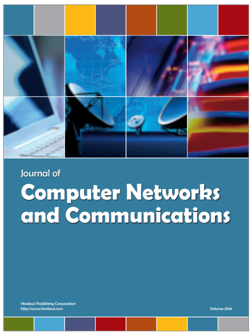

International Journal of
Distributed
Sensor Networks

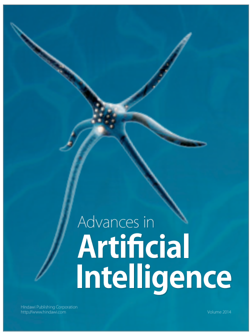

Journal of
Industrial Engineering

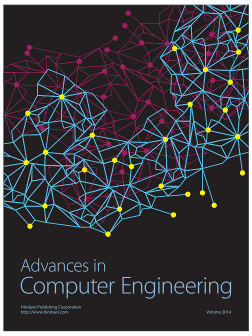

Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

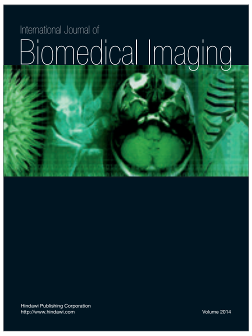

Modelling &
Simulation
in Engineering

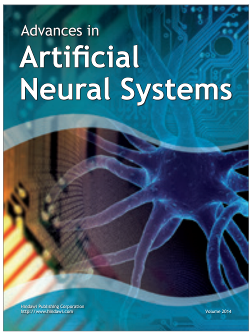

Journal of
Computer Networks
and Communications

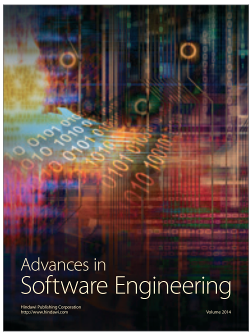

Advances in
Artificial
Intelligence


Advances in
Computer Engineering

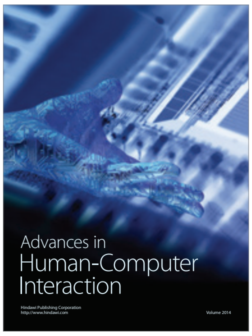
International Journal of
Computer Games
Technology

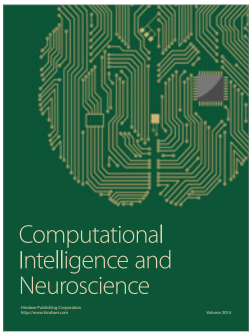

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering