Cooperative Parallel Runtimes for Multicores

Younghyun Cho, Surim Oh, and Bernhard Egger

Computer Systems and Platforms Laboratory,
Department of Computer Science and Engineering,
Seoul National University, Seoul, Korea
{younghyun,surim,bernhard}@csap.snu.ac.kr

Abstract. In a multiprogrammed environment on multi/many-core architectures, to efficiently execute multiple parallel applications on one machine the platform needs to address two important problems: (1) how to decide the proper amount of core resources for a parallel application, and (2) how to dynamically adapt application parallelism by providing varying number of core resources.
In this paper, we present a multi/many-core scheduling scheme where application parallel runtimes cooperate with each other for a flexible execution. In our scheme, a platform-wide space-sharing scheduler computes the adequate amount of core resources for each application based on an online performance model. Malleable parallel application runtimes dynamically change the active number of threads for a parallel region through application-specific scheduling on the assigned core resources.
We evaluate our scheme for GNU OpenMP runtime on two Linux-based multi-core systems. The experimental results show that the proposed cooperative scheduling outperforms the standard Linux/OpenMP schedulers for various NPB parallel application mixes.

1 Introduction

Multi/many-core systems are widely used in general-purpose servers, network processing, workstations, and computing nodes of high-performance and distributed computing systems. As the number of cores increases in a single platform, it is increasingly important to execute simultaneously running multiple parallel applications efficiently.

Space-sharing core resources is a promising approach to efficiently execute multiple parallel applications [16,10,7,9]. Space-shared scheduling provides a disjoint set of core resources for each application and thus reduces performance interference caused by time-shared scheduling and thread oversubscription. To enable such scheduling we need to address several challenges. First, we need a core resource manager that decides the proper amount of core resources for each parallel application with considerations of varying performance characteristics of an application. In addition, the core allocator needs to be able to allocate cores based on a given scheduler's policy such as maximizing the overall speedup of applications. Second, the parallel application runtimes need to adapt their parallel execution dynamically to react to the varying number of core resources

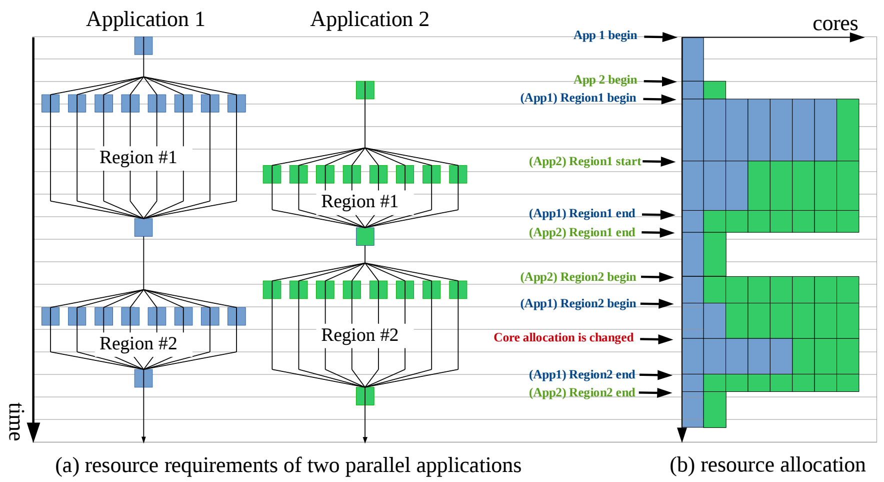


Fig. 1. Operation of the cooperative scheduling framework.

provided for applications. This in order to avoid thread oversubscription while maximizing parallel efficiency.

However, current operating systems and parallel programming models are not adequate for dynamic spatial scheduling. The prominent parallel programming models such as OpenMP [3], TBB [15], Cilk [2], or OpenCL [11] assume that each parallel application can utilize all existing hardware resources without considering the current system workload. At the same time, current Linux schedulers manage all running tasks through time-shared scheduling. In such a disjoint runtime model, simultaneously executing parallel applications suffer not only from a low cache utilization caused by cold misses from context switches caused by an oversubscription of threads to a single core but also from performance interference arising from scheduling threads in a time-shared manner.

In this paper, we present a cooperative many-core scheduling scheme for simultaneously running shared-memory parallel applications. Figure 1 illustrates the main idea of our scheduling strategy. Parallel applications consist of several parallel and sequential regions, and each parallel region may show different performance characteristics. To consider varying applications' performance characteristics, we employ an online scalability prediction model [4] that can estimate the scalability of running parallel code based on a short sampling period. Based on the scalability information, our space-sharing scheduler performs a greedy core allocation that maximizes the summation of speedup of running parallel programs. On the other hand, malleable application runtimes manage application resources such as assigning the application's work chunks to the allocated cores. The application parallelization runtimes dynamically change the active number of threads through adaptive dynamic work scheduling. We evaluate the scheduling scheme on two Linux-based multi-core platforms, a 64-core

AMD Opteron platform and a 36-core Tile-Gx36 processor. We extend the GNU OpenMP runtime to support malleable application execution.

The remainder of this paper is organized as follows. In Section 2, we discuss researches related to this paper. Section 3 describes the techniques for application runtimes to adapt to varying degree of parallelism in a parallel code region. In Section 4, we explain our platform-wide space-sharing scheduler. Sections 5 and 6 provide our experimental environments and performance evaluation for various scheduling scenarios. Finally, we conclude this paper in Section 7.

2 Related Work

To properly manage many-core resources for simultaneously running parallel applications, a number of scheduling techniques that consider applications' performance characteristics have been proposed. Moore and Childers [14] perform offline training to understand a multi-threaded application's scalability. They choose proper thread counts for multi-threaded applications based on the information about an application's performance plateau and performance interference between multi-threaded applications. Sasaki *et al.* [16] partition many-core resources for multiple multi-threaded applications based on information about performance scalability obtained by executing applications on three different core allocations and measure how the application's throughput changes at runtime. However, both works [14,16] consider thread-level parallel programs that do not allow to change the parallelism once parallel threads are created and launched.

Some researchers focus on malleable applications (usually for the OpenMP programming model) where the runtime can choose a thread count when it enters a parallel section. Emani *et al.* [10,9] use machine-learning techniques and compiler-assisted information to predict better thread counts in a multiprogrammed environment. Creech *et al.* [7] introduce SCAF that decides the proper thread count of each OpenMP applications. To determine the scalability of an OpenMP parallel section at runtime, they create and run a serial process concurrently with the parallel section. An online profiler then compares the throughput of the serial process and parallel section and derives a scalability curve. Unlike the work above [10,9,7], however, we focus on (OpenMP) parallel applications that allow a parallel region to change the degree of parallelism with application parallelization runtime support.

3 Malleable Application Runtimes

3.1 Programming Malleability

At every scheduling period our space-sharing scheduler collects performance counters required for performance modeling and computes a core allocation. The scheduling period is decided manually for each multi-core platform [1] The

[1] The scheduling periods used for the experiments are listed in Table 1.

core allocation information is sent to the applications runtimes. Then application runtimes adapt the degree of parallelism within the varying available core resources.

To manage the active threads (cores) at runtime, we need to consider the programming model of the parallel applications. Most parallel programming models [15,2,11,3] provide dynamic scheduling policies that allow specifying the number of worker threads at the entrance to a parallel section but perform no adjustments later. We propose a simple dynamic work scheduling algorithm and communication interface that provide both malleability and satisfactory performance at the same time.

In this work, we focus on OpenMP, the de-facto standard for shared-memory parallel programming. We have extended GNU OpenMP by a new scheduler policy called `adaptive` that communicates with the aforementioned space-sharing scheduler framework and adapts the number of worker cores dynamically based on the number of allocated cores. We limit ourselves to `parallel for` loops in consideration of the fact that such loops constitute the performance dominant part in the majority of OpenMP applications.

3.2 Malleable GNU OpenMP Runtime

In the OpenMP programming model, application programmers can select one of three scheduling disciplines, `static`, `dynamic`, and `guided`, for a parallel loop. Static scheduling divides and assigns the iterations equally to the available threads. The dispatch overhead is small, however the policy may suffer from load imbalances. In dynamic scheduling, a fixed amount of work chunks is assigned to an idle thread one by one. This policy suffers from a high dispatch overhead and a load imbalance if the chunk size is not well chosen. Guided scheduling, finally, assigns several chunks of work to idle threads in order to keep the dispatch overhead low as well as load balanced.

The three scheduling policies are not adequate for changing the degree of parallelism at runtime. Static and guided scheduling are not malleable at all because they assign a comparatively large amount of work in the first assignment. On the other hand, the dynamic scheduler can suffer from a significant dispatch overhead depending on the allocation granularity. To provide malleability while minimizing overhead and maximizing load balance, we use an adaptive dynamic scheduler as illustrated in Algorithm 1. If the processing time of a work chunk is smaller than the global scheduling period of the space-sharing scheduler, we increase the chunk size. To provide sufficient opportunities for load balancing, the maximum chunk size is set to $\lceil W/2N \rceil$ where W represents the remaining iterations, N the number of available cores in the system. This is similar to the guided loop scheduling algorithm for multi-core systems [12].

The OpenMP runtime implements a work sharing approach in which each worker thread shares the data structure containing information about the processed and still unprocessed loop iterations. We designate one worker thread as the delegate thread that is allowed to change the work chunk size. For technical

Algorithm 1 Adaptive dynamic scheduling

```
/* $ws$ represents a work sharing construct in OpenMP */
function GOMP_ITER_ADAPTIVE_NEXT(*pstart, *pend)
    id = gomp_thread()→id // get my thread id
    if $ws$→delegate == id then
        if elapsed_time < schedule_period then
            $ws$→chunk_size = $ws$→chunk_size×2
        if $ws$→chunk_size > $ws$→remain/2N then
            $ws$→chunk_size = $ws$→remain/2N
    if !$ws$→available[id] then
        wait($ws$→cond[id]) // thread is now blocked
    for $i = 0$ to N do // N is # of cores (threads)
        if $ws$→available[i] then
            signal($ws$→cond[i])
    /* assign a chunk which is lock protected*/
    start = $ws$→next
    end = start + $ws$→chunk_size
    $ws$→next = end
    *pstart = start
    *pend = end
```

reasons, we create as many threads as the system cores for every parallel application. However, at any given time, only as many threads are active as cores have been assigned to the runtime.

4 Space-Shared Scheduling

4.1 Core and Memory Management

Here we explain how the platform-wide space-sharing scheduler manages core and memory resources for a multi/many-core platform. For the basic core allocation unit we allocate a *cluster*, that is, a set of computing cores sharing a common last-level cache (LLC). There are two exceptions of this rule, first, when a serial section is scheduled, and, second, when the number of running applications exceeds the number of clusters in the system. If an LLC is shared by the same parallel code, we can obtain higher prediction accuracy by the scalability model since we have stable memory access rates. In addition, we expect to increase the possibility of LLC sharing.

Modern many-core chips feature NUMA (Non-Uniform Memory Access) architectures comprising several of memory nodes. The application schedulers in major parallel programming models distribute work without giving much consideration to the NUMA architectures, and so does our scheduler. The NUMA memory allocation policy is set to interleaving [2] in our scheduling framework in order to eliminate NUMA effects by distributing memory access evenly to all memory nodes.

[2] we used `numactl` to set the NUMA allocation policy

Algorithm 2 Greedy Core Allocation

```
n = # of applications in the system
m = # of clusters in the system
CL[n] ={1, } //# of reserved clusters for each application, initialized to 1

while $sum(CL) < m$ do
    best_app = $-1$
    max_speedup = 0
    for $i = 0$ to $n - 1$ do
        SpeedUp[n] = {0, } // speedup of each application
        TempCL[n] = {0, } // temporary cluster allocation
        for $j = 0$ to $n - 1$ do
            TempCL[j] = $i == j$ ? CL[j]+1 : CL[j]
        /* compute speedup of temporary allocation /*
        for $j = 0$ to $n - 1$ do
            /* parameters: LLC miss rate, NUMA link/node access patterns */
            SpeedUp[j] = $SpeedupModel(TempCL)$
        if $sum$(Speedup) > $max\_speedup$ then
            max_speedup = $sum$(SpeedUp)
            best_app = i
    CL[best_app] += 1
```

4.2 Online Scalability Prediction

To compute a proper resource allocation to the different parallel applications with respect to the present scheduling policy, an accurate online performance model is essential. To characterize an application's scalability at runtime, we employ an online performance model presented by Cho *et al.* [4]. The model considers memory access contention as the major limiting factor of performance scalability for malleable parallel code sections. Based on a short sampling period during which memory access patterns (LLC miss rate, NUMA link and memory controller access patterns) are obtained, the model can estimate the performance scalability of a parallel program code within an acceptable error rate [3] on NUMA architectures. In this work, we employ the scalability model to allocate more cores for scalable parallel codes.

4.3 Greedy Core Allocation

We use a greedy optimal core allocation algorithm which accommodates the specific scheduling policies based on the performance model. The algorithm first reserves at least one allocation unit to each application. Whenever the scheduler allocates a new allocation unit to an application, it chooses the best solution according to the scheduling policy.

In this paper, we use a policy that maximizes the summation of speedup for all running parallel programs. Algorithm 2 computes the proper amount of

[3] 6.8% and 9.7% of mean absolute percentage errors are reported in the paper for parallel kernels in NPB for OpenCL and OpenMP versions, respectively.

core resources for our scheduling policy (max speedup). The complexity of the computation is $O(n^2m)$ where n and m represent the number of applications and allocation units respectively. This is an acceptable overhead because the number of simultaneous parallel applications is usually small. Also, allocating cores to a cluster as the default allocation granularity further reduces the complexity.

After the core resources for all applications have been reserved, we consider core clustering among applications when more than two applications are packed into one cluster. The Tile-Gx36, for example, has 36 tiles in a single chip without a specific LLC. The scheduler allocates the clusters of an application to areas of minimal perimeter in order to minimize the overhead of inter-core communication (i.e., caused by a cache coherence protocol) and task migration.

The space-sharing scheduler periodically wakes up and re-evaluates the current resource allocations to the running applications. One important consideration is to reduce the number of changes in allocations (i.e., re-assignments to a different core) caused by platform-wide rescheduling. In this work, we have implemented a rather simplistic approach in which the scheduler always allocates the cluster/core resources to applications in the same order in order to minimize the number of re-allocations.

5 Experimental Setup

5.1 Target Platforms

The scheduling framework can run on any Linux-based multi-core system. We have evaluated the proposed scheduler on two real-world platforms, a 64-core AMD Opteron server platform [1] and the Tile-Gx36 platform [5]. The main features for performance evaluation of the two architectures are shown in Table 1.

Table 1. Target architecture properties.

Architecture	**AMD64**	**Tile-Gx36**
processor	4 x Opteron6380	Tile-Gx8036
clock frequency	2.5 GHz	1.2 GHz
memory size	128 GB	32 GB
total # of cores	64	36
# of NUMA nodes	8	2
Linux kernel	3.19	2.6.40
scheduling period	30 ms	100 ms

5.2 Scheduling Framework Implementation

The space-sharing scheduler runs as a daemon and directly interacts with the parallel application runtimes. The required communication interfaces and (malleable) work scheduling algorithm are implemented into the GNU OpenMP runtime (AMD system: gcc-4.6.4 / Tile-Gx36: gcc 4.6.2).

Table 2. Target applications.

App.	Description	AMD64		Tile-Gx36	
		Problem size	Iter.	Problem size	Iter.
BT	Block Tri-diagonal solver	408^3	5	64^3	200
CG	Conjugate Gradient	450000	5	75000	75
EP	Embarrassingly Parallel	2^{33}	0	2^{30}	0
FT	Discrete 3D fast Fourier Transform	1024^3	1	512^3	20
SP	Scalar Penta-diagonal solver	408^3	5	408^3	400

The performance indicators required for the performance model are obtained by monitoring the hardware's performance counters. The performance model requires LLC miss events, memory access events (for all NUMA links/nodes), and total cycles. Our implementation employs the system call interface through Linux `perf` which allows us to control the performance event counters.

For the AMD platform [1], the LLC miss event counters are provided on the AMD NorthBridge and we can obtain the count using the "NBPMCx4E1 L3 Cache Misses" event descriptor. "NBPMCx1E0 CPU to DRAM Requests to Target Node" is used to obtain the number of DRAM requests. On the other hand, the Tile-Gx8036 architecture does not have a specific last level cache. Instead, the architecture uses DDC (Dynamic Distributed Cache) techniques [6] in which local cache misses try to fetch their data from distributed caches. For the Tilera architecture, we consider local read cache misses (from local cache to memory) and remote read cache misses (from remote cache to memory) at the same time, and set the LLC misses to the sum of the two.

5.3 Target Applications and Environment

We evaluate the proposed scheduling technique with five OpenMP applications (`BT`, `CG`, `EP`, `FT`, `SP`) from the NPB benchmark suite [17]. Each application exhibits different characteristics: `EP` and `FT` are CPU-intensive benchmarks. `BT` and `SP` generate regular, and `CG` issues irregular memory accesses patterns. The problem size and the number of iterations of the different benchmarks are shown in Table 2. To keep the execution time reasonable, we use smaller working sets on the less powerful Tile-Gx8036 system.

For the experiments, we measure the performance of the default OpenMP runtime scheduling policies `static`, `dynamic`, and `guided` and compare them to the space-sharing scheduler with two policies: `equal partitioning` and `max speedup`. All OpenMP scheduling policy create n threads where n is equal to the number of cores in the system. The `static` policy statically assigns $1/n$ of the total work to each worker thread. In the `dynamic` policy idle threads requesting new work are assigned exactly one, in the `guided` policy a dynamic number of iterations based on the amount of remaining work.

The core allocation for the proposed space-shared scheduling with the `equal partitioning` policy assigns an equal number of cores to each application. For

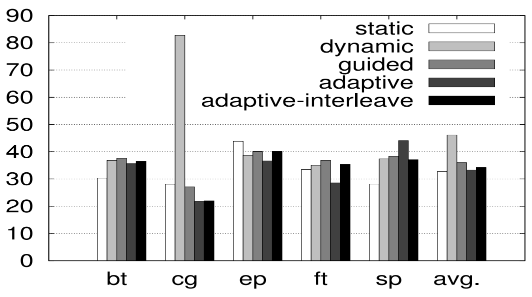


Fig. 2. Standalone application performance.

the `max speedup` policy, the scheduler assigns cores based on the greedy core allocation in Algorithm 2. In all experiments, concurrent applications are started at the same time. The results represent the average of three runs.

6 Evaluation

6.1 Adaptive Dynamic Scheduling

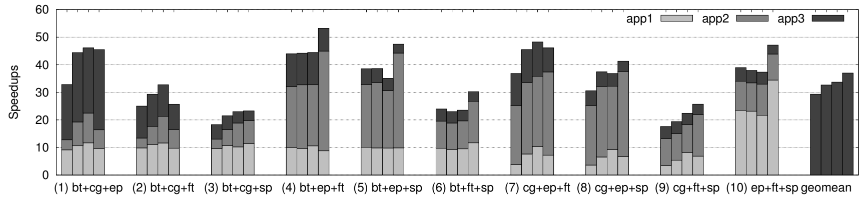
We first compare the performance of the default OpenMP scheduling policies with that of our scheduling framework on the AMD machine for a single application in order to determine the overhead of the different policies and, in particular, our framework.

Figure 2 shows the turnaround time for the three OpenMP policies and the proposed adaptive scheduling technique, both for the NUMA `first-touch` and `interleaved` policy.

Each loop scheduling policy exhibits different characteristics, and there is no policy that consistently outperforms the others. OpenMP `static` scheduling performs best for `BT` and `SP`. The two applications have a relatively small number of loop iterations, generate regular memory accesses and do not suffer from load-imbalance, in other words, these two applications can be efficiently executed with a static work distribution. For the `CG`, `EP`, and `FT` benchmarks, `adaptive` dynamic scheduler performs best among the other dynamic schedulers. This confirms that even in a standalone scenario the proposed technique provides both high performance and malleability and can compete with standard OpenMP policies.

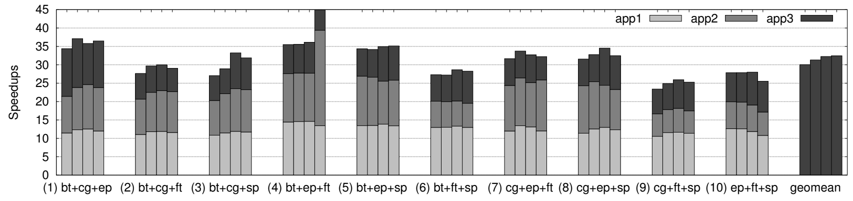
For the multiprogramming scenarios, the NUMA memory allocation policy is set to `first-touch` for the default OpenMP policies and `interleave` for our policies to distribute data to the all memory nodes in the space-shared execution. The nature of the benchmarks generating their own input data using parallel loops in fact favors the `first-touch` policy because the work distribution for the initialization and the actual work loop are likely to match for a given scheduling policy. We argue that real-world applications do not create their input data themselves using parallel loops but rather take it as an external input, and thus the advantage of `first-touch` is artificial. The performance difference between

Sum of Speedup (Normalized Score): 29.33(0.79) / 32.65(0.88) / 33.70(0.91) / 36.99(1.0)



(a) 3-Way Multiprogramming on AMD64

Sum of Speedup (Normalized Score): 30.06(0.93) / 31.32(0.97) / 32.26(0.99) / 32.45(1.0)



(b) 3-Way Multiprogramming on Tile-Gx36

Fig. 3. Speedups of simultaneous OpenMP applications. The four bars represent in the order `OpenMP dynamic`, `OpenMP guided`, `equal partitioning`, and `max speedup`.

`adaptive` and `adaptive-interleave` shows the aforementioned affinity of the benchmarks to the `first-touch` NUMA policy.

6.2 Multiprogramming Performance

Experimental scenarios To evaluate our scheduling scheme in a multiprogramming environment, we execute all possible combinations of three applications (and some two applications) from the five benchmark applications in Table 2 with each available scheduling policy.

We compare our space-sharing scheduler to the two dynamic OpenMP scheduling policies `dynamic` and `guided` that generate as many threads as the number of system cores and do not change the amount of parallelism. Thread scheduling for the default OpenMP policies is performed by Linux's CFS scheduler. We then compare our scheduler to the state-of-the-art space-shared scheduler SCAF [7].

Comparison to OpenMP schedulers First, we compare the OpenMP dynamic schedulers `dynamic` and `guided` to our space-sharing scheduler with the policies `equal partition` and `max speedup` on the AMD and the Tilera system. Figure 3 shows the speedup[4] for three concurrent parallel applications on the two different environments: (a) the 64-core AMD system, and (b) the 36-core Tile-Gx36 platform. The bars represent (left-to-right): OpenMP `dynamic`, OpenMP `guided`, space-sharing with the `equal partitioning`, and space-sharing with

[4] performance against the single-core execution

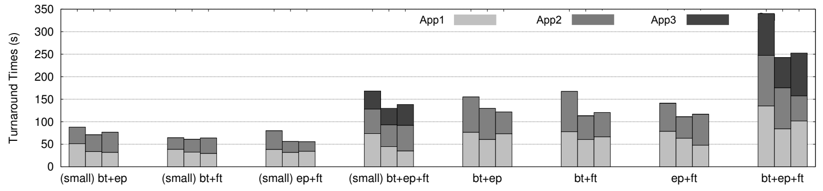


Fig. 4. Comparison to SCAF on AMD64: The three bars represent, in order, `SCAF`, `equal partitioning`, and `max speedup`. The left four benchmarks (small) use smaller problem sizes.

`max speedup` policy. The speedup for each individual application are added up and shown with different levels of gray.

The speedup in Figure 3 shows that both space-sharing approaches outperform standard OpenMP policies. In addition, we observe that the core allocation policy `max speedup` (fourth bar in the series) outperforms the `equal partitioning` in terms of the scheduling goal (maximizing the summation of speedup). This result means that the online speedup prediction model is able to capture the scalability trend of parallel applications well. On the AMD system, the `max speedup` policy achieves a 10% better performance compared to the best dynamic OpenMP scheduling policy.

In the Tile-Gx36 architecture, however, the `max speedup` policy achieves a smaller performance improvement compared to the AMD architecture. The reason is twofold. First, the lower computational power of the Tile-Gx36 platform causes less contention, which means that even memory-intensive applications scale better, i.e., there is less difference between the applications' scalabilities that the policy could exploit. Second, while the overhead caused by the periodic re-allocation of the resources is not an issue on the AMD machine, the effect is noticeable on the slower Tile-Gx36 chip. The `equal partitioning` policy requires re-computation and re-assignment of resources only when applications start or finish and thus suffer from less overhead.

Comparison With SCAF Similar to our work, SCAF [7] is a runtime system that manages OpenMP thread counts to efficiently execute multiple OpenMP applications. The framework estimates performance scalability when a parallel section is the first run by executing the same workload twice, once with a single thread and once with parallel threads. Based on the runtime information of the two executions, SCAF computes an adequate thread count for parallel sections.

The most important difference between the proposed framework and SCAF is that the former can estimate the scalability and adapt resources for parallel code sections already during their very first run and even while the sections are executed with parallel threads. Figure 4 shows the application turnaround time of SCAF (first bar) and our scheduling policies `equal partitioning` and `max speedup` (second and third bar, respectively) on the AMD64 platform for the two and three co-located application scenarios.

From our evaluation of the turnaround time for each concurrent parallel application, the proposed space-shared scheduling outperforms SCAF in all scenarios. The reason is twofold: first, SCAF does not manage thread counts once it has entered a parallel section. In other words, a parallel section cannot increase the number of active threads even though resources may become available. Second, the overhead of generating a serial process executing the same program code concurrently with the parallel section is rather high. SCAF requires not only an additional core but also duplication of the application's input/output data. If a parallel section accesses a large amount of memory, this operation can be too expensive to be performed at runtime. SCAF works well for parallel applications that are executed more than once in which case the profiling overhead is only incurred once. However, even in that case, SCAF cannot adapt to varying input data unlike the proposed technique.

7 Conclusion and Future Work

In this work, we have introduced an efficient core resource management scheme for parallel applications on multi/many-core architectures. Based on an online performance model that uses the memory access patterns of current parallel sections as its main metric, our space-sharing scheduler dynamically recomputes core resource allocations of simultaneously executing parallel applications. Parallel application runtimes execute cooperatively to manage the assigned core resources efficiently.

The evaluation of our implementation for the OpenMP runtime on a 64-core AMD and Tile-Gx36 processor shows that our cooperative scheduling provides efficient execution for concurrent parallel applications, and the space-sharing scheduler can meet the scheduling goal based on an online scalability model.

In this work, we did not consider NUMA features in modern multi/many-core architectures. A proper thread count and data placement can improve application performance significantly [18]. In addition, recent researches have proposed NUMA-aware programming models [13] and parallel runtimes [8]. Our next step is thus to maintain NUMA-efficiency for NUMA-aware parallel workloads while providing an adequate number and placement of core resources for parallel applications in a multiprogrammed environment.

Acknowledgments

We thank the authors of [7] for providing the source code of SCAF which enabled us to perform the comparison for this paper. This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

1. AMD. AMD Opteron 6300 Series Processors. `http://www.amd.com/en-us/products/server/opteron/6000/6300`. [online; accessed October 19, 2016].
2. Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.
3. OpenMP Architecture Review Board. OpenMP. `http://openmp.org`. [online; accessed October 19, 2016].
4. Younghyun Cho, Surim Oh, and Bernhard Egger. Online scalability characterization of data-parallel programs on many cores. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 191–205. ACM, 2016.
5. Tilera Corp. Tile-Gx36 Processor. `http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf`. [online; accessed October 19, 2016].
6. Tilera Corp. UG130:Architecture manual. Tilera Corp.
7. Timothy Creech, Aparna Kotha, and Rajeev Barua. Efficient multiprogramming for multicores with scaf. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 334–345. ACM, 2013.
8. Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 125–137. ACM, 2016.
9. Murali Krishna Emani and Michael O'Boyle. Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments. In *ACM SIGPLAN Notices*, volume 50, pages 499–508. ACM, 2015.
10. Murali Krishna Emani, Zheng Wang, and Michael FP O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.
11. Khronos Group. The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`. [online; accessed October 19, 2016].
12. Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *null*, pages 140–147. IEEE, 1993.
13. Zoltan Majo and Thomas R Gross. *A library for portable and composable data locality optimizations for NUMA systems*, volume 50. ACM, 2015.
14. Ryan W Moore and Bruce R Childers. Using utility prediction models to dynamically choose program thread counts. In *ISPASS*, pages 135–144, 2012.
15. James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.
16. Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 107–116. ACM, 2012.
17. Sangmin Seo, Jungwon Kim, Gangwon Jo, Jun Lee, Jeongho Nah, and Jaejin Lee. SNU NPB Suite. `http://aces.snu.ac.kr/software/snu-npb/`, 2011. [online; accessed October 19, 2016].
18. Wei Wang, Jack W Davidson, and Mary Lou Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431. IEEE, 2016.