



# Maximizing System Utilization via Parallelism Management for Co-Located Parallel Applications

Younghyun Cho  
Seoul National University  
Seoul, South Korea  
younghyun@csap.snu.ac.kr

Camilo A. Celis Guzman  
Seoul National University  
Seoul, South Korea  
camilo@csap.snu.ac.kr

Bernhard Egger  
Seoul National University  
Seoul, South Korea  
bernhard@csap.snu.ac.kr

## ABSTRACT

With an increasing number of cores and memory controllers in multiprocessor platforms, co-location of parallel applications is gaining on importance. Key to achieve good performance is allocating the proper number of threads to co-located applications. This paper presents NuPoCo, a framework for automatically managing parallelism of co-located parallel applications on NUMA multi-socket multi-core systems. NuPoCo maximizes the utilization of CPU cores and memory controllers by dynamically adjusting the number of threads for co-located parallel applications. Evaluated with various scenarios of co-located OpenMP applications on a 64-core AMD and a 72-core Intel machine, NuPoCo achieves a reduction of the total turnaround time by 10-20% compared to the default Linux scheduler and an existing parallelism management policy focusing on CPU utilization only.

## CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*;

## KEYWORDS

Parallelism management, resource utilization, OpenMP

### ACM Reference Format:

Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. 2018. Maximizing System Utilization via Parallelism Management for Co-Located Parallel Applications. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18)*, November 1–4, 2018, Limassol, Cyprus. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3243176.3243199>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
*PACT '18*, November 1–4, 2018, Limassol, Cyprus  
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

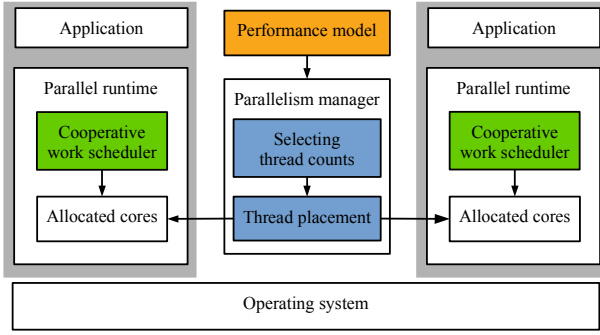
ACM ISBN 978-1-4503-5986-3/18/11...\$15.00  
<https://doi.org/10.1145/3243176.3243199>

## 1 INTRODUCTION

In modern multi-core platforms, parallel applications often share computational and memory resources. Parallel workloads running on these platforms are managed by parallel application runtimes such as OpenMP and Intel TBB. Generally, such workloads are executed with a configurable number of threads [20, 37]. In this context, determining the proper number of threads for co-located parallel applications to optimize application performance [38] or platform throughput [42] has been an important topic of research in the compiler and runtime community. Improving co-location performance is also an important concern in HPC centers to improve both energy efficiency and overall throughput [9, 10].

The focus of this work lies on managing parallelism for co-located parallel applications to fully utilize system resources and therefore to achieve an increased co-location performance (i.e. reduction of the total execution time) for shared-memory multiprocessor systems consisting of multiple CPU sockets and memory controllers with Non-Uniform Memory Accesses (NUMA) latencies. Such multi-socket multi-core architectures are the standard for high-end shared-memory platforms. Existing work typically assigns more worker threads to computation-intensive applications [14, 35, 41, 42]. This can lead to under-utilized memory systems resulting in inefficient tail execution once the computation-intensive applications have finished. In contrast, the method proposed in this paper aims at maximizing the overall utilization of both the CPU cores and the memory system. Several runtime systems manage application parallelism based on machine learning models [16, 27]. While these approaches can also react to varying optimization goals [16], their performance depends on the quality and the amount of trained data. Here, on the other hand, we provide an analytical solution that allows us to understand performance with an analytical model based on queueing theory.

We present NuPoCo, a framework for NUMA multi-core Performance Optimization of CO-located parallel applications. NuPoCo maximizes the overall system utilization by considering the utilization of both CPU cores and memory controllers to determine the proper number of threads for each co-located applications. A dynamic spatial scheduling



**Figure 1: The NuPoCo framework.**

approach is employed that allows only one active thread on each core to reduce the performance interference caused by thread oversubscription [6, 20, 29, 49].

Figure 1 depicts the structure of the NuPoCo framework. The three core components are (1) a performance model, (2) a parallelism manager, and (3) cooperative work schedulers of parallel runtime systems. The performance model predicts the utilization of CPU cores and memory controllers for co-located parallel applications. The parallelism manager periodically performs core allocation (i.e., deciding on the number of threads per application and their location) by leveraging the performance model and monitoring hardware performance counters. The cooperative work schedulers, finally, dynamically adapt their execution to the core allocation dictated by the parallelism manager.

NuPoCo’s parallelism management maximizes the utilization of CPU cores and memory controllers. Section 2 first introduces related research and existing policies to determine thread counts for co-located applications. Section 3 provides an evaluation based on queueing theory that demonstrates the benefits of our strategy compared to existing approaches.

Predicting utilization is based on a queueing system that models memory accesses on multi-socket multi-core systems. The performance model leverages standard hardware performance counters present in commodity AMD and Intel platforms; Section 4 describes this performance model. Based on the performance prediction, the parallelism manager determines the number of threads assigned to individual co-located applications and periodically revisits the placement of the threads. The techniques used in the parallelism manager are explained in Section 5. The cooperative work scheduling is implemented in the dynamic loop scheduler of the GNU OpenMP runtime and allows applications to react to a changing number of worker threads at runtime. Section 6 introduces the cooperative parallel runtime.

Section 7 evaluates NuPoCo on two multi-socket multi-core platforms, a 64-core AMD Opteron [4] and a 72-core Intel Xeon [25] platform. Experimental results for various workload mixes obtained from NPB [5], Parsec [7], and Rodinia [11] show that NuPoCo is able to execute multiple

OpenMP applications in significantly less total execution time compared to the default Linux scheduler and a parallelism management scheme maximizing CPU utilization.

To summarize, the contributions of this paper are

- an analytical analysis demonstrating that maximizing overall utilization of all memory controllers and CPU cores is beneficial for co-located parallel workloads.
- a parallelism management technique that maximizes resource utilization in multi-socket systems with a combination of online performance prediction and parallel runtime system support.
- NuPoCo, a parallelism manager, that improves the average system throughput on commodity multi-socket systems in the order of 10 to 20%.

## 2 RELATED WORK

### 2.1 Parallelism Management

Managing parallelism for parallel applications has been an important issue in the runtime community. To determine the proper thread or core count for parallel applications, SBMP [42], SCAF [14], and Varuna [44] execute a parallel program in several configurations at runtime and perform a regression analysis to estimate the performance scalability. Parcae [38] and C3PO [41] perform hill-climbing to reach an optimal thread count. CRUST [21] manages an application’s working set size (and thread count) in dependence of the available cache size. Emani et al. [16] and ADAPT [27] apply machine learning models to compute the number of threads assigned to applications.

Several policies have been proposed to assign the proper number of threads to co-allocated applications. SCAF [14] maximizes the speedup of all running applications. SBMP [42] minimizes the average normalized turnaround time (the execution time compared to a solo-run) of applications. C3PO [41] maximizes the CPU utilization within a given power budget. All these approaches favor scalable applications; if there is a perfectly scalable application, the majority of core resources is allocated to that application and other applications receive zero or one core. Parcae [38] initially reserves an equal number of cores to all running parallel applications. Applications find the optimal number of threads through hill-climbing. Cho et al. [13] use a simple but inaccurate analytical model to determine the best thread count for each application.

NuPoCo focuses on maximizing overall system utilization of all CPU cores and memory controllers. Thread placement is known to strongly affect performance on multi-socket systems [15, 50]. While previous work does not consider thread placement or uses simplistic linear partitioning [41, 42], NuPoCo considers the architecture’s NUMA properties to determine a good placement of threads to cores.

## 2.2 Cooperative Parallel Runtimes

Another important issue is changing the applications' parallelism in response to a varying core assignment. SBMP [42] and C3PO [41] regulate only the number of assigned cores per application. Worker threads are pinned to the assigned cores without changing the degree of parallelism of the application. On the other hand, several runtime systems [14, 16, 18] assign a varying number of threads to parallel sections. OpenMP runtime systems [1] already provide this feature. Once created, however, the number of worker threads within a parallel section remains constant. To provide dynamic spatial scheduling, several compilers [28, 37, 38] generate flexible code. The basic idea is to divide the total work into composable chunks of work. Varuna's [44] virtual tasks decouple software from hardware threads and require no compiler support. Callisto [20] is a framework for cooperative parallel runtimes. Callisto uses a scheduler activation technique for providing dynamic spatial scheduling while reducing the performance interference. The framework assigns an equal number of threads to each co-located application.

Similarly, NuPoCo provides dynamic spatial scheduling by leveraging a dynamic loop scheduler in the OpenMP runtime system. We discuss the merits of our approach in Section 6.

## 2.3 Thread and Data Placement

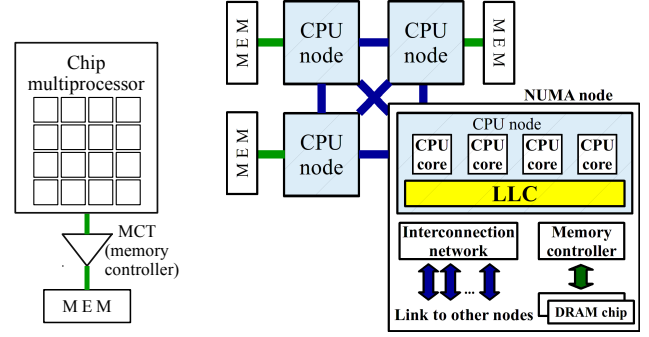
A number of thread and data placement techniques have been presented for multi-threaded applications on multi-socket systems [15, 32, 45, 50]. Threads are placed in order to minimize resource contention while preserving an efficient data placement. Lozi et al. [30] resolved several performance bugs in multi-socket systems by improving the Linux scheduler. Pandia [17] predicts performance of parallel applications for different thread placements based on several profiling runs.

Unlike these approaches, we focus on assigning the proper number of threads for parallel applications at runtime.

# 3 BACKGROUND AND MOTIVATION

## 3.1 Multi-socket Multi-core Systems

Figure 2 provides a simplified view of symmetric multiprocessing (SMP) and multi-socket multi-core systems. Unlike an SMP system that comprises multiple cores and one memory, multi-socket systems contain a number of memory controllers to increase the memory bandwidth in the presence of a large number of cores. In such systems, one node consists of a CPU node, itself composed of a group of CPU cores, and its attached memory node. The individual nodes are connected by an interconnection network such as AMD's HyperTransport [40] or Intel's QPI (Quick Path Interconnect) [36]. These architectures exhibit Non-Uniform Memory Accesses (NUMA) characteristics because of the varying access latencies of the cores to the different memory controllers.



(a) SMP system

(b) Multi-socket multi-core system

**Figure 2: SMP and multi-socket multi-core systems.**

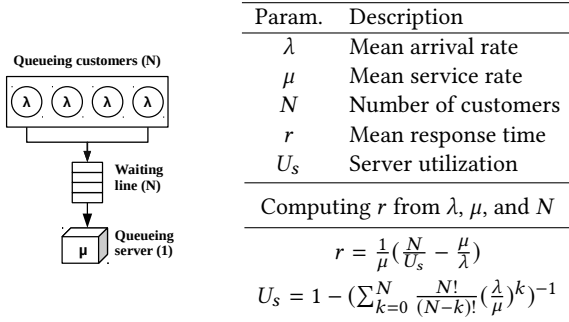
## 3.2 Parallel workloads

NuPoCo maximizes system utilization by controlling the degree of parallelism (DoP) of co-located applications during execution of fork-join-style parallel sections such as the `parallel` construct in OpenMP [8] or Intel TBB [39] workloads. The DoP is controlled using malleable workloads that allow dynamic adjustment of the number of threads for each parallel section. For non-malleable workloads, the threads of a parallel section can be pinned to the assigned hardware cores [41, 42].

The presented parallelism management scheme is based on the resource utilization of the CPU cores and the memory controllers. To predict the resource utilization, a queueing model is employed where the CPU cores are considered queueing customers and the memory controllers are regarded as queueing servers. The queueing model requires the requests from the customers to follow an exponential distribution and assumes that memory requests are blocking, i.e., the issuing core is blocked and does not generate any new requests until the request has been served by the memory controller in a First-In-First-Out (FIFO) order. Several studies have shown that the memory requests of parallel loops follow an exponential distribution on real multiprocessor platforms and that performance can be modeled using queueing models [12, 47, 48]. The spatial scheduling approach of NuPoCo allows us to ignore the effect of interference on performance caused by scheduling two or more threads from different workloads on a single hardware core.

## 3.3 Modeling Performance Metrics

Queueing models are able to compute important performance-related metrics such as the CPU utilization or the memory controller utilization. Let us first consider a simple SMP system with one memory controller (MCT) and 16 cores as shown in Figure 2 (a). Such a system can be modeled using an  $M/M/1/N/N$  queueing system [46] with a finite number of  $N$  customers and 1 server;  $M$  stands for Markovian. Details about queueing models can be found in [26, 46].



**Figure 3: The M/M/1/N/N queueing model.** The mean arrival rate  $\lambda$  and the mean service rate  $\mu$  refer to the number of requests from a customer and the number of requests served by the server, respectively, per time unit in the steady state.

Figure 3 shows the  $M/M/1/N/N$  queueing model and its closed-form expression to compute the mean response time of the server. The  $N$  queueing customers (the cores) each generate requests with a mean arrival rate  $\lambda$  following a Poisson distribution that are served by one queueing server (the memory controller) with a mean service rate  $\mu$  with exponential service times.

Based on this queueing model, the speedup, the per-core utilization, and the MCT utilization in dependence of the number of allocated cores can be derived as follows. The *Speedup* of a program is defined by dividing the execution time on one core, *Total Time(1)*, by the execution time on  $N$  cores, *Total Time(N)*

$$Speedup(N) = Total\ Time(1) / Total\ Time(N) \quad (1)$$

Under the assumption that cores block on outstanding memory requests, *Total Time(N)* is composed of the execution time on  $N$  cores, *CPU Time(N)*, and the total memory response time, *MCT Time(N)*

$$Total\ Time(N) = CPU\ Time(N) + MCT\ Time(N)$$

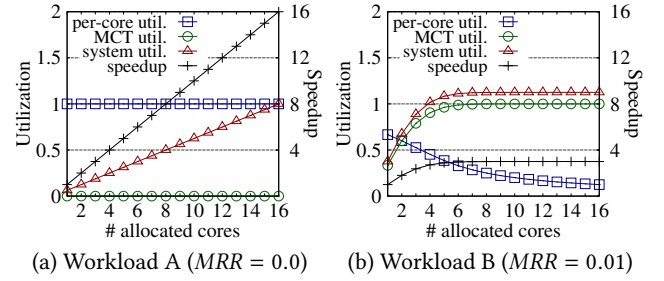
For data-parallel workloads where the total amount of work is constant and balanced, the execution time on  $N$  cores is given by

$$CPU\ Time(N) = CPU\ Time(1) / N$$

The estimated number of generated memory requests for  $N$  cores is the product of the CPU time and the per-core memory request rate, *MRR*. *MCT Time(N)* is obtained by multiplying *MRR* with the mean memory response time for  $N$  cores, *MRT(N)*.

$$MCT\ Time(N) = CPU\ Time(N) \times MRR \times MRT(N) \quad (2)$$

Each application has its own *MRR* value, and the *MRT* for a varying number of cores is regarded as the scaling factor of the parallel application.



**Figure 4: Performance metrics for two workloads with different MRRs at a mean service rate  $\mu$  of 50.**

For a memory controller with a service rate  $\mu$ , the mean memory response time is given by (refer to Figure 3)

$$MRT(N) = \frac{1}{\mu} \left( \frac{N}{MCT\ Util(N)} - \frac{\mu}{MRR} \right) \quad (3)$$

where *MCT Util(N)* denotes the memory controller utilization corresponding to the server utilization  $U_s$  from Figure 3

$$MCT\ Util(N) = 1 - \left( \sum_{k=0}^N \frac{N!}{(N-k)!} \left( \frac{MRR}{\mu} \right)^k \right)^{-1} \quad (4)$$

Finally, the per-core utilization, *CPU Util(N)*, is defined by the ratio of CPU time over the total time

$$CPU\ Util(N) = \frac{CPU\ Time(N)}{Total\ Time(N)} \quad (5)$$

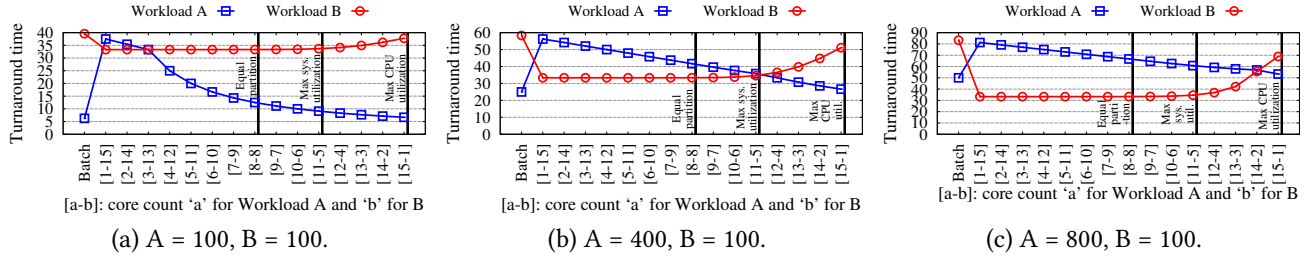
To consider overall utilization of both the CPU and the memory controller, we suggest a new metric, the *system utilization*, defined as the sum of CPU and MCT utilization. For an application using  $N$  of the total  $M$  system cores, the system utilization, *System Util(N)*, is defined as

$$System\ Util(N) = CPU\ Util(N) \times \frac{N}{M} + MCT\ Util(N) \quad (6)$$

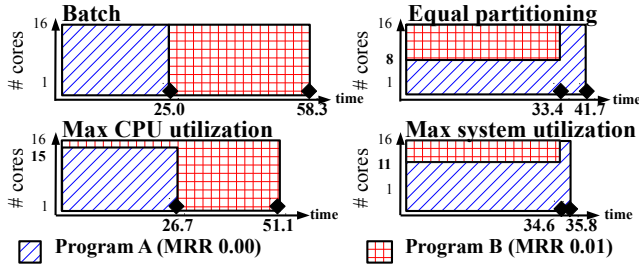
To solve this model for co-located applications, the weighted average of the workloads' *MRR* of all assigned cores is used to compute the mean memory response time and the memory controller utilization (Equations 3 and 4). Based on the computed *MRT* value, we compute the per-application *Speedup* and *CPU Util* using the application-specific *MRR* value.

### 3.4 Performance Analysis

Figure 4 plots the analytical results of the presented model for the four metrics *Speedup*, *MCT Util*, *CPU Util*, and *System Util* for two workloads and a varying number of cores. The results show that the completely CPU-bound workload A is able to fully utilize the given CPU resources, but its *MCT Util* is 0. For workload B with a memory request rate  $MRR = 0.01$ , *CPU Util* decreases with an increasing number of cores while *MCT Util* increases. Looking at *System Util*, the system utilization of workload B is always higher than



**Figure 5: Turnaround times of co-located workloads A and B.** Both workloads are started at the same time and executed with the core allocation given in the X-axis. The vertical bars indicate the core distribution yielding the best performance for the *equal partitioning*, *max system utilization*, and *max CPU utilization* policies, respectively. The line points on the Y-axis indicate the turnaround time of each workload. Subfigures (a)-(c) differ in the amount of work per workload (metric: turnaround time when executed in isolation on a single core).



**Figure 6: Illustration of the performance for the core allocation policies in Figure 5 (b).**

that of workload A. However, *System Util* of workload B is saturated at a relatively small number of cores while the *System Util* of workload A increases linearly. The insight of this analytical result is that co-locating workload A with workload B has the potential to achieve a higher system utilization.

Using the queueing model, we can simulate co-location performance based on the speedup value of each workload. Figure 5 shows the computed total turnaround time of the co-located workloads for different core allocations and a varying amount of work on a 16-core SMP system with one memory controller. Figure 6 visualizes the core allocation over time for three common and the presented allocation policies using the workload distribution from Figure 5 (b).

The first policy, *Batch*, executes the workloads sequentially. *Equal partitioning* executes the two workloads in parallel, assigning the same number of cores to both. The policy *Max CPU utilization* finds the core allocation that maximizes the total CPU utilization. We observe that *Max CPU utilization* allocates 15 cores to the perfectly scalable workload A and only the minimum of one core to workload B. The proposed *Max system utilization* policy, finally, maximizes the *System Util* as defined by Equation 6. *Max system utilization* achieves the

shortest total turnaround time of the four policies, demonstrating that focusing only on CPU utilization may not lead to optimal results.

In Figure 6, the *Max system utilization* policy yields the best turnaround time among all possible core allocations with a 40% of reduction compared to the *Batch* configuration. For *Max CPU utilization*, after workload A has ended, the execution of workload B policy experiences an inefficient tail execution caused by congestion in the memory system. It is also important to note that the optimal partitioning minimizing the total turnaround time depends on the amount of work of the co-located applications. While both workloads end around the same time with *Max system utilization* in Figure 5 (b), yielding the best possible turnaround with a core allocation of 11:5 cores assigned to workload A and B, respectively, this is not the case for Figures 5 (a) and (c). For (a), the best distribution is 6:10 cores, and for (c) it is 14:2. The total turnaround time of the *Max system utilization* policy, however, achieves comparable performance to the best distribution and in all situations performs better than *Max CPU utilization*.

### 3.5 The NuPoCo Policy

The analysis in this section suggests that for co-located parallel applications, maximizing the transient overall system utilization is beneficial if the workloads' size is unknown. Without special provisions, the total execution time of a parallel section is typically not known in advance.

In the NuPoCo framework, we aim to maximize the overall system utilization *NuUtil* of a multi-socket system. Such a NUMA system is a group of SMP systems, as shown in Figure 2 (b). *NuUtil* is therefore defined as the sum of all individual nodes' system utilization:

$$NuUtil = \sum_{i=0}^{num\_nodes} System\ Util_i \quad (7)$$

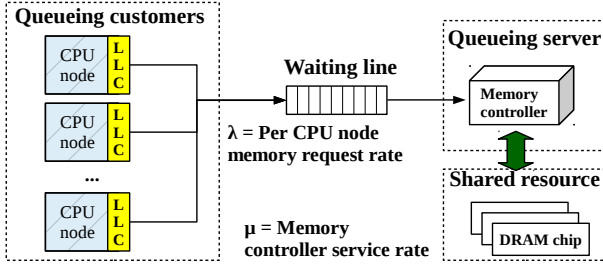


Figure 7: Queueing system for an individual memory controller.

## 4 ONLINE PERFORMANCE MODEL

Multi-socket systems comprise multiple CPU nodes and memory controllers (Figure 2 (b)). Based on a queueing system network for multi-socket multi-core systems in our prior work [12], NuPoCo considers the memory controllers and the interconnection links as separate queueing servers and predicts the mean memory response time.

### 4.1 Memory Controller Utilization

To predict the utilization of individual memory controllers, we model each controller with a queueing system as shown in Figure 7. A memory controller serves the memory requests issued by the last-level caches (LLC) of the individual CPU nodes. For the queueing system of memory controller  $m$ , let  $N_{cpu\_node}$  be the number of CPU nodes and  $MRR_{i,m}^{cpu\_node}$  represent the memory request rate from CPU node  $i$  to memory node  $m$ . The mean request arrival rate at memory controller  $m$ ,  $MRR_{avg,m}^{cpu\_node}$ , is the average of the individual CPU nodes' request rates

$$MRR_{avg,m}^{cpu\_node} = \frac{\sum_{i=0}^{N_{cpu\_node}} MRR_{i,m}^{cpu\_node}}{N_{cpu\_node}}$$

With the average memory request rate  $MRR_{avg,m}^{cpu\_node}$  and the memory service rate  $\mu_m$  for a memory controller  $m$ , we can compute the memory controller utilization  $MCT Util_m$  and the mean response time  $MRT_m$  using Equations 4 and 3 from Section 3.3, respectively. The value of  $MRT_m$  is used to compute the CPU core utilization in the section below.

### 4.2 CPU Core Utilization

To compute the CPU core utilization of a CPU node, we first need to calculate the memory request time to each memory controller. To do so, the queueing system depicted in Figure 8 is employed. This queue models the serialization of outgoing memory requests from the node's LLC to one memory controller. Outgoing memory requests include missed read and write operations and hardware prefetch requests.

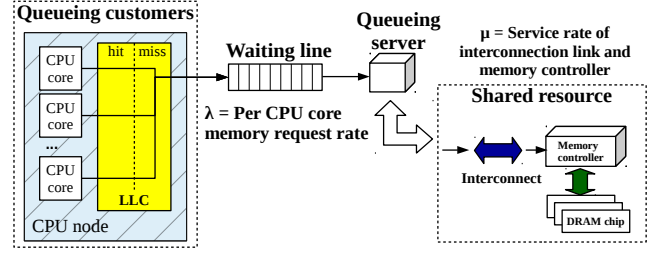


Figure 8: Queueing system for CPU core utilization prediction.

With  $N_{cores\_in\_node}$  representing the number of cores in CPU node  $i$  accessing memory node  $m$ , the average memory request rate  $MRR_{avg,m}^{cpu\_core}$  is estimated as follows

$$MRR_{avg,m}^{cpu\_core} = \frac{\sum_{j=0}^{N_{cores\_in\_node}} MRR_{j,m}^{cpu\_core}}{N_{cores\_in\_node}}$$

The service rate includes the service rate of the interconnection link,  $link_{i,m}$ , from CPU node  $i$  to memory node  $m$ , and the mean response time of the memory node ( $MRT_m$ ) as obtained in Section 4.1. The service rate  $\mu_{i,m}$  is given by

$$\mu_{i,m} = \frac{1}{1/link_{i,m} + MRT_m}$$

With  $MRR_{avg,m}^{cpu\_core}$  and  $\mu_{i,m}$ , the total mean memory response time from CPU node  $i$  to memory node  $m$ ,  $MRT_{i,m}$ , is computed from Equation 3.

The total memory response time for a core is obtained according to Equation 2. While all outgoing memory requests of an LLC affect the mean memory response time, only requests caused by read misses stall a core and thus affect the CPU core utilization. We estimate the rate of outgoing read requests from every core to each memory node using the per-core number of LLC read requests per time<sup>1</sup>. For the cores in CPU node  $i$  with an LLC read request rate to memory node  $m$ ,  $LLC_{i,m}^{read}$ , the total memory response time is computed by

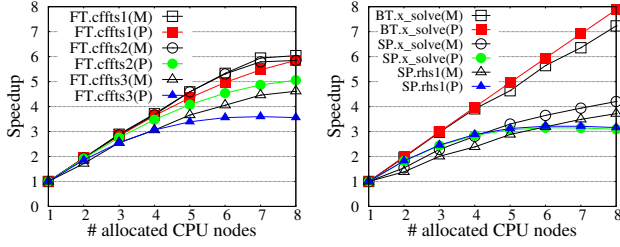
$$MCT Time = \sum_{m \in M} CPU Time \times LLC_{i,m}^{read} \times MRT_{i,m}$$

Based on the ratio between  $CPU Time$  and  $MCT Time$ , we can compute the CPU utilization using Equation 5.

### 4.3 Implementation and Validation

The required inputs for the performance model are obtained from the hardware performance monitoring unit. We measure LLC accesses, LLC misses, all memory requests that affect memory utilization (read, write, prefetch) to all memory controllers, and the total number of CPU cycles. AMD [2, 3] and Intel [22–24] systems support all required counters.

<sup>1</sup>To compute the LLC read request rate per core, we initially allocate only threads of the same application to the cores in a node, then divide the node's LLC read request rate by the number of allocated cores, see Section 5.1.



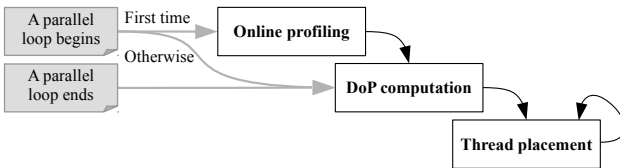
**Figure 9: Speedup predictions (P) and measurements (M) of several parallel loops from FT and SP (NPB) [5] on a 64-core AMD Opteron system.**

The application-specific parameters  $MRR_{i,m}^{cpu\_core}$ ,  $MRR_{i,m}^{cpu\_node}$ , and  $LLC_{i,m}^{read}$  are computed at runtime without depending on offline information. A direct measurement of the per-core LLC accesses and misses is not supported by the hardware. NuPoCo gets around this limitation by initially allocating only threads of one application to the cores in one CPU node, then divide the node's LLC accesses and misses by the number of cores. This happens once for each parallel section during a one-time brief online profiling phase (see Section 5.1). The machine-dependent parameters  $\mu_{mct}$  and  $link_{i,m}$  are determined by executing a synthetic workload from the Stream benchmark [34] that generates memory accesses from one core to specific memory nodes and measures the mean memory service time. This process is required only once for a given hardware platform.

Figure 9 compares the predicted with the actual speedup for several parallel loops from an NPB implementation [43]. The results show that the performance model can capture the trend of the speedup. Since the speedup is computed from the predicted CPU core utilization (Equation 1), this result confirms that predictions of resource utilization are also possible with the presented model. An extensive analysis for other (co-located) NPB parallel loops on a 64-core AMD and a 72-core Intel system shows that the performance model predicts the speedup with moderate absolute percentage errors of 10-15%, similar to the results in the prior work [12].

## 5 MANAGING PARALLELISM

The degree of parallelism and the core assignment of co-located applications is managed at runtime by NuPoCo.



NuPoCo's parallelism manager is activated whenever a parallel loop begins or ends. It performs the following three steps: *online profiling*, *DoP computation*, and *thread placement*. When a parallel loop is executed for the first time, the *online*

### Algorithm 1 DoP computation

```

1: for each cpu_node  $\in$  system do
2:   util_list = [ ]
3:   for each wl  $\in$  running workloads do
4:     cpu_node.allocate(wl)
5:     NuUtil  $\leftarrow$  performanceModeling()
6:     util_list.append(NuUtil)
7:     cpu_node.deallocate(wl)
8:   best_wl  $\leftarrow$  bestExpectedNuUtil(util_list)
9:   for each cpu_core  $\in$  cpu_node do
10:    cpu_core.allocate(best_wl)
11: Communicate core allocation to parallel runtimes

```

*profiling* phase is initiated that profiles the new parallel loop for a short period of time; profiling is skipped for the second and later invocations of the same loop. The *DoP computation* step uses the queueing systems presented in Section 4 to compute a thread allocation that maximizes the overall system utilization. Once the thread count for each co-located application has been determined, the *thread placement* phase begins during which individual threads of an application are relocated if opportunities exist to improve performance.

### 5.1 Online Profiling

During online profiling, all cores of the system are assigned to the new parallel section for a short period of time. This serves two purposes. First, it ensures that the data of an application is distributed in a similar manner as in a standalone execution under a NUMA first-touch allocation policy. Second, it allows NuPoCo to infer the LLC miss rate per core by measuring the node's LLC rate and divide it by the number of cores in the node. This initial profiling period is set to 150ms; long enough to ignore cache warming effects and sufficiently short not to affect other running applications much.

### 5.2 DoP Computation

The goal of this step is to maximize system utilization by allocating the proper thread counts for running parallel applications. Algorithm 1 shows how the parallelism manager determines the degree of parallelism for each application. The number of cores per workload is determined in a greedy manner. The basic allocation unit in this stage is a CPU node. Starting with an empty allocation, each CPU node in the system (line 1) is assigned in turn to the application (lines 9–11) that is expected to yield the best overall system utilization *NuUtil* (Section 3.5) (lines 5–8). The prediction of the system utilization *NuUtil* (line 6) is based on the performance prediction model from Section 4.

The number of CPU nodes is assumed to be larger than the number of co-located applications. At least one CPU node is allocated to each application executing a parallel section. Applications in serial sections are assigned a single core.

**Algorithm 2** Thread placement

---

```

1: Initialize cpu_node_list in descending order of LLC ac-
  cesses since the last invocation
2: repeat
3:   busy_nd  $\leftarrow$  cpu_node_list.pop_front()
4:   idle_nd  $\leftarrow$  cpu_node_list.pop_back()
5:   if  $\frac{\text{busy\_nd.llc\_accesses}()}{\text{idle\_nd.llc\_accesses}()} > \text{threshold}$  then
6:     busy_wl  $\leftarrow$  busy_nd.max_llc_miss_rate()
7:     idle_wl  $\leftarrow$  idle_nd.min_llc_accesses()
8:     SwapCores(busy_wl, idle_wl)
9: until cpu_node_list is empty
10: Communicate core allocation to parallel runtimes

```

---

**5.3 Thread Placement**

The DoP computation assigns all core resources of a CPU node, i.e., cores sharing the same LLC (Section 3.1), to one application. This leaves room for additional performance improvements. Individual threads of memory-intensive applications may require substantial LLC resources. If allocated to the same CPU node, thus sharing the same LLC, this may lead to contention or, even worse, thrashing in the LLC. CPU-bound applications, on the other hand, typically contend less for LLC resources. Co-locating memory-intensive with CPU-bound workloads in the same CPU node thus has the potential to yield an improved overall system utilization.

Algorithm 2 outlines the implementation of this idea. The algorithm is invoked periodically every 50ms after Algorithm 1 has ended. It repeatedly retrieves the CPU nodes that exhibit the highest (*busy\_nd*) and lowest (*idle\_nd*) number of LLC accesses since the last iteration (lines 3–4). If the ratio of LLC accesses exceeds a given threshold (currently set to 2; line 5), we select the workload that observed the highest LLC miss rate from *busy\_nd* (line 6) and the one with the lowest number of LLC accesses from *idle\_node* (line 7), based on the information inferred during online profiling (Section 5.1). The algorithm then swaps the location of a number of cores (NuPoCo exchanges two cores by default) of the two applications (line 8). This process is repeated until the list is empty (line 9). Although this thread placement technique is a hill-climbing method, it quickly reaches a steady state as later demonstrated in Section 7 and Figure 13.

**6 COOPERATIVE OPENMP RUNTIME**

The last core component of the NuPoCo framework are cooperative parallel runtimes that provide dynamic spatial scheduling. We have implemented dynamic spatial scheduling into the GNU OpenMP runtime [1] by adding a new loop scheduling method denoted *cooperative*. The NuPoCo parallelism manager keeps track of the execution of OpenMP applications by intercepting calls that initiate or terminate

**Algorithm 3** Cooperative worker threads

---

```

1: while there is more work do
2:   if own core is not available then
3:     go to sleep
4:   for each thread  $\in$  worker_threads do
5:     if thread's core is available then
6:       wake up thread
7:   work_chunk  $\leftarrow$  get_work_chunk(chunk_size)
8:   if work_chunk received then
9:     work_chunk  $\rightarrow$  execute( )
10:   if id == 0 and elapsed_time < epoch then
11:     chunk_size  $\leftarrow$  (chunk_size  $\times$  2)

```

---

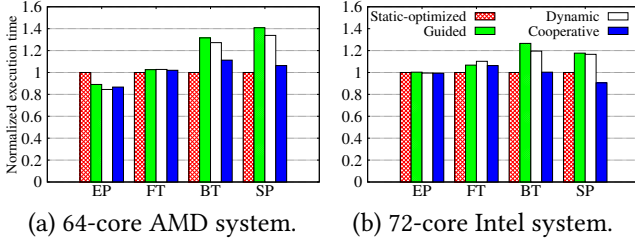
parallel loops. The results of the core allocation are communicated to the OpenMP parallel runtimes through shared memory. The runtimes dynamically change the DoP of cooperative parallel loops by adjusting the number of worker threads and pinning them to the assigned cores.

**6.1 Cooperative Loop Scheduling**

The OpenMP runtime contains three schedulers, *static*, *dynamic*, and *guided*. In dynamic scheduling, all worker threads iteratively acquire and process a chunk of the total work based on a work-sharing model. The implementation of the *cooperative* scheduler leverages the *dynamic* scheduler. Algorithm 3 shows how the worker threads are executed by the *cooperative* scheduler. Before requesting new work, each thread checks the availability of its core. If the core is no longer available, the thread goes to sleep (lines 2–3). Active worker threads review the current core allocation and wake up threads whose core has become available (lines 5–6).

Each thread acquires a chunk of work by calling the *get\_work\_chunk* function on line 7. To decrease the dispatch overhead, the master thread (id 0) dynamically adjusts the work chunk size based on the elapsed execution time of a work chunk (lines 12–13). Furthermore, to preserve data locality optimizations of applications, we partition the work items into multiple regions for each CPU node. Then, a local work queue distributes work to the threads in that node. The regions are equally partitioned according to OpenMP's *static* scheduling policy. Such a partitioning can be effective when neighboring work items exhibit high locality and preserve manual optimizations for static scheduling with a technique such as [33]. Load balancing is also achieved through work stealing from the local queues of other CPU nodes.

A concern is whether, despite its flexibility, the performance of the *cooperative* loop scheduler is on par with the existing schedulers. Figure 10 shows the turnaround times of standalone applications executed with different OpenMP schedulers. *Static-optimized* uses static work partitioning and considers NUMA locality for the data allocation. In the absence of workload imbalance, this approach achieves the



**Figure 10: Normalized execution time of NPB [5] applications under different loop schedulers.**

best possible performance because no scheduling overhead is incurred. We observe that *dynamic* and *guided* scheduling perform worse than static because of scheduling overhead and the unawareness of data locality. The presented *cooperative* work scheduling combines the best of both worlds by respecting data locality, yet being able to react to workload imbalance while also supporting malleable parallelism.

## 7 EVALUATION

### 7.1 Experimental Platforms

We evaluate NuPoCo on a 64-core (8-node) AMD Opteron platform and a 72-core (4-node) Intel Xeon platform. The AMD Opteron [4] processors and Intel Xeon E7-8870 v3 processors [25] run at 2.5GHz and 2.1GHz frequency, respectively. The AMD and Intel platforms are equipped with 128GB and 756GB of DRAM memory, respectively. The DRAM chips operate at 1.6GHz in both system. The Linux kernel versions are 4.4.35 for AMD and 4.4.0 for the Intel platform. We use a modified version of OpenMP v5.4.0 [1] with the *cooperative* loop scheduler (Section 6).

### 7.2 Target Applications

For the co-location scenarios, we utilize target applications from NPB [5], Parsec [7], and Rodinia [11] (Table 1). NPB applications represent HPC workloads that require large amounts of memory and/or lots of computational resources. We selected *BT*, *FT*, *SP*, and *EP* from an OpenMP NPB implementation [43]. *BT*, *FT*, and *SP* are both CPU- and memory-intensive workloads. The data set of *FT* and *SP* is very large. We categorize these three applications as Type-A to represent applications that require a significant amount of system resources. On the other hand, *EP* is an almost perfectly scalable kernel that rarely accesses memory. We classify *EP* as Type-B, a class that extremely under-utilizes the memory system. The four NPB applications use input class D with a large problem size. The number of iteration steps is adjusted to obtain standalone turnaround times that are similar to those of the other applications.

Parsec's *blackscholes* (*BS*) consists of long serial sections and one parallel loop that does not require a lot of system resources compared to Type-A applications. *BS* is executed

App	Resource requirement			
	CPU	Memory	Data size	Type
BT	High	Medium	Medium	A
FT	High	High	Huge	A
SP	High	High	Huge	A
EP	High	Almost none	Almost none	B
KM	Low	Medium	Small	C
BS	Low	Low	Small	C

**Table 1: Target applications.**

with the native input data set. *kmeans* (*KM*) from the Rodinia benchmark is executed with 3,000,000 objects and represents a non-scalable application with frequent synchronization between cores and a long serial section at the beginning of its execution. *KM* under-utilizes CPU resources. These two applications are classified as Type-C, representing applications that under-utilize CPU resources.

### 7.3 Execution Modes

The presented approach is compared with the following execution modes:

- **Batch.** Applications are executed serially. The number of threads is equal to the number of system cores, each thread is pinned to a core <sup>2</sup>.
- **Native.** Applications generate as many threads as there are cores in the system and are co-located by the Linux scheduler. Thread binding is disabled to allow the Linux scheduler to perform thread and data placement.
- **Equal.** This policy assigns the same number of cores to all running parallel sections and a single core to a serial process. The cores are allocated linearly.
- **Scalability.** This core allocator is based on a CPU scalability-based approach. We have implemented the hill-climbing algorithm proposed in C3PO [41]. The algorithm changes the number of assigned cores to the applications based on the measured CPU utilization.
- **NuPoCo Greedy.** To demonstrate the effect of the thread placement technique (Section 5.3), this policy performs only DoP computation (Section 5.2).
- **NuPoCo** Our proposal.

With **Batch** and **Native**, loops are scheduled statically as this yields the best performance among all available OpenMP loop schedulers on our platforms. For **Equal**, **Scalability**, **NuPoCo Greedy**, and **NuPoCo**, we use the cooperative work scheduler presented in Section 6.1 to provide dynamic spatial scheduling. NuPoCo is executed with the following parameters: the initial profiling phase is 150ms (Section 5.1). The thread placement algorithm (Algorithm 2) is invoked every 50ms and uses a threshold value of 2 for core swapping.

<sup>2</sup>On our platforms, thread binding performs better in standalone execution, but worse in co-located executions.

Co-location type	Co-located workloads
Mix of Type-A	(1) BT, FT (2) BT, SP (3) FT, SP (4) BT, FT, SP
Mix of Type-A and B	(5) BT, FT, EP (6) BT, SP, EP (6) BT, SP, EP (7) FT, SP, EP
Mix of Type-A and C	(8) BT, KM (9) FT, KM (10) SP, KM

**Table 2: Co-location scenarios.**

To measure the co-location performance, we consider the total execution time from start to finish of all co-located applications. Each scheduler is evaluated using the normalized total turnaround time (NTT) with regards to **Native**. We also report the speedup relative to the harmonic mean (Hmean) which is known as a speedup metric that also considers the fairness of co-located jobs [31]. All results are obtained by executing each scenario three times and taking the average.

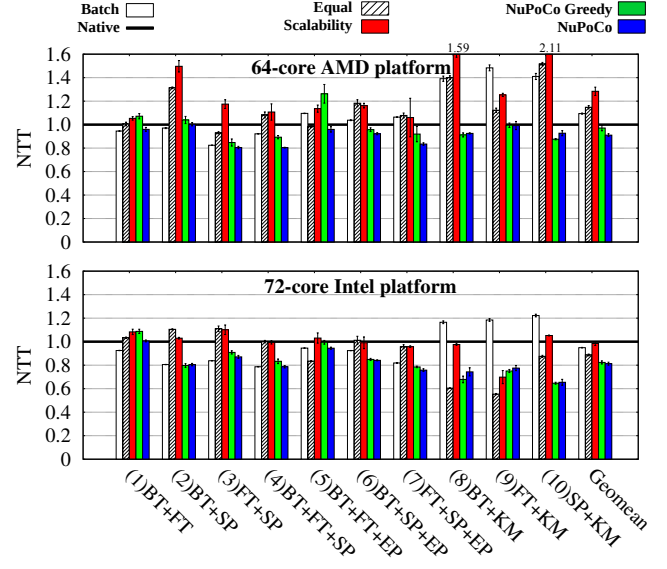
#### 7.4 Co-location Scenarios

We consider various co-location scenarios as follows. First, two Type-A applications that require substantial system resources are co-located. To see the performance behavior for different types of applications, the Type-B (*EP*) and the Type-C (*KM*) application are co-located with several Type-A applications. Details of each scenario are given in Table 2.

Figure 11 shows the NTT of the six execution modes Batch, Native, Equal, Scalability, NuPoCo Greedy, and NuPoCo on the AMD and the Intel system for ten different scenarios. All co-located applications are started at the same time but finish at different points in time.

The results show that, on average, NuPoCo achieves the best system throughput among the six core allocation configurations on both platforms. Under the geometric mean, NuPoCo achieves an NTT of 0.91 (9% improvement) on the AMD system and 0.81 (19% improvement) on the Intel platform over the Linux scheduler. The performance improvement with NuPoCo is up to 20% on the AMD platform (scenario 4) and 35% on the Intel system (scenario 10). NuPoCo also does not report any performance degradation for the ten scenarios. The average job turnaround time of co-located applications with NuPoCo is 10.8% and 12.3% shorter than that of Native for the AMD and the Intel platform, respectively.

Scenarios 1–4 mix two to three Type-A applications causing high competition for platform resources. We observe that the Batch configuration is a suitable choice for scenarios 1–4 as they utilize the platform’s CPU core and memory systems well and show good scalability. Native can not efficiently execute these scenarios (especially scenario 3) compared to Batch or NuPoCo because it suffers from a high resource interference as all of the applications have a high degree of resource demands. For scenarios 1–3, NuPoCo shows almost

**Figure 11: Normalized total turnaround (NTT) to Linux Native for the co-location scenarios.**

the same performance as Batch on the AMD/Intel platforms. For scenario 4, NuPoCo outperforms Batch by 10%. This is because the three Type-A applications contain serial sections during which NuPoCo is able dynamically assign more cores to parallel sections.

To test the effectiveness of the presented methods when different types of applications are co-located with Type-A, we execute *EP*, a (Type-B) application with two applications from Type-A in scenarios 5–7. We observe that performance of Batch decreases compared to scenarios 1–3. Since *EP* puts no pressure on the memory system, the Batch configuration suffers from a low utilization when *EP* is executed standalone. On the other hand, Native is able to increase resource utilization for co-located *EP* and Type-A applications compared to Batch. NuPoCo achieves better performance than the other schedulers thanks to its online performance prediction model and dynamic thread count adjustment.

For the remaining scenarios 8–10, we co-locate *KM* with *BT*, *FT*, and *SP*. *KM* does not require a lot of CPU resources because of its long serial sections and the synchronizations, hence, allocating only a subset of cores to *KM* is beneficial. As expected, Batch experiences a significant performance degradation when executing *KM*. NuPoCo performs well for these scenarios, but we observe that on the Intel platform, the Equal policy performs best for scenarios 8 and 9. A static core allocation scheme can be beneficial for *KM* and its short parallel section that is executed iteratively.

Overall, we observe that the conventional allocation approach to co-location, Scalability, is not beneficial to improve the system’s throughput. Although CPU utilization is maximized, co-located applications suffer from memory

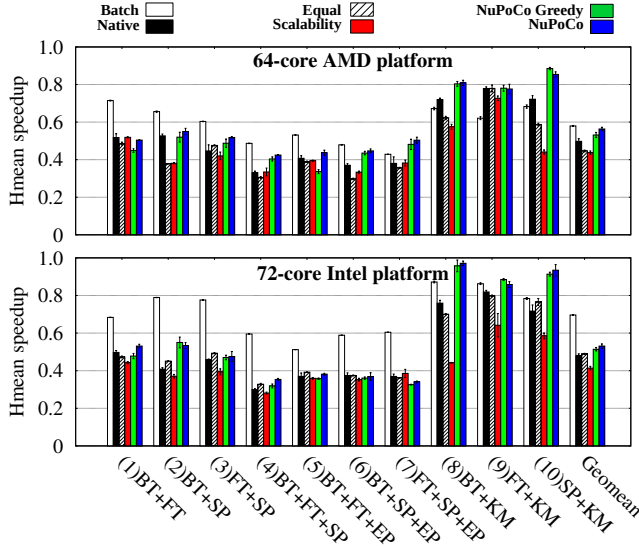


Figure 12: Hmean of speedup relative to standalone execution for the co-location scenarios.

contention and a low CPU utilization when scalable applications finish earlier. Additionally, in scenarios 2, 8, and 10, Scalability suffers from a severe performance degradation. A closer inspection reveals that the hill-climbing algorithm in some cases is oscillating, thus continuously changing the number of assigned cores. The benefit of the *thread placement* (Section 5.3) in NuPoCo is visible in comparison with NuPoCo Greedy. Despite the additional runtime overhead, proper thread placement is beneficial in general.

In terms of the Hmean speedup shown in Figure 12, NuPoCo outperforms Native by 13.2% and 10.8% on the AMD and Intel platforms, respectively. The results show that fairness is not only preserved but improved with NuPoCo. The Linux scheduler often favors specific workloads resulting in a slow performance for other applications. This is also visible in Figure 13. Batch achieves a relatively good Hmean speedup because the first job in Batch is always assigned the optimal value.

To summarize, the results show that NuPoCo performs well for a diverse mix of applications, especially when the co-located applications exhibit different performance characteristics. If the co-located workloads exhibit similar characteristics, NuPoCo consistently provides good performance comparable to the best system configuration (Batch or Native).

## 7.5 Case-Study and Overhead Analysis

To better understand and demonstrate NuPoCo’s operation, three application types, *FT* (Type-A), *EP* (Type-B), and *BS* (Type-C) are co-located. Using the open-source trace visualizer SnuMAP [19], Figure 13 visualizes the core allocations over the course of execution on the 64-core AMD platform.

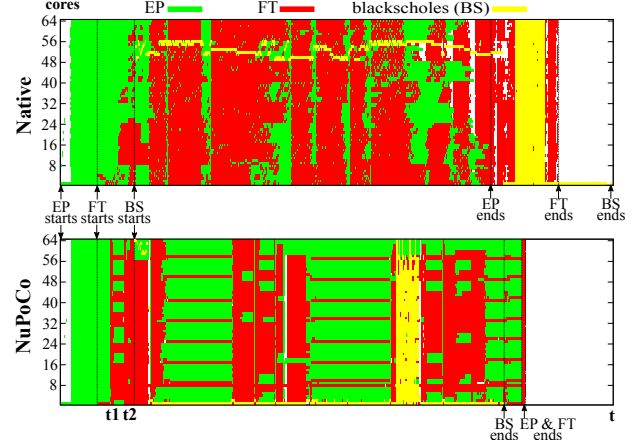


Figure 13: Trace visualization for a co-location under Native and NuPoCo on the AMD machine.

In this scenario, each application starts at a different time. *EP* is started first and monopolizes the core resources. *FT* joins a bit later and starts its first parallel section at  $t_1$  with the clean-profiling phase. NuPoCo then performs the DoP computation followed by the thread placement technique for *EP* and *FT*. We observe that the profiling stage in NuPoCo is almost invisible, and the core allocations quickly converges to the steady state ( $t_1$ – $t_2$ ). The different core allocations indicate that NuPoCo differentiates between multiple parallel sections in *FT*. Once *BS* is started, it uses only one core for its initial long serial section until *BS* reaches the main parallel section. Over the entire execution with a scheduling epoch of 50ms, in total 32 thread count selections and 1,627 thread placements have been executed with an average computational overhead of 1.8ms and 1.5ms. Since NuPoCo runs in parallel to the applications, this overhead is hidden, or rather included in the results. Compared to Native, we observe that thread interference of Native’s time-sharing model causes severe synchronization delays for parallel sections in *FT* and the serial process of *BS*. For this scenario, NuPoCo reports a 19% shorter total execution time over Native.

## 8 CONCLUSION

In this paper, we have presented NuPoCo, a parallelism management framework for co-located parallel workloads on NUMA multi-socket multi-core systems. At-runtime performance prediction of CPU and memory controller utilization is used to determine the degree of parallelism for all running workloads with the goal of maximizing system resource utilization. The evaluations show that the NuPoCo framework executes multiple OpenMP parallel applications with a significantly shorter total turnaround time than the Linux time-sharing model and an existing parallelism management policy maximizing the CPU utilization.

NuPoCo is able to automatically optimize the performance of co-located applications. There is additional room for performance improvements for scheduling a batch of parallel jobs. In future work, we plan to extend NuPoCo to maximize performance under a given power budget and combine NuPoCo with a job scheduler in HPC centers.

The NuPoCo framework and the benchmarks used in this study can be obtained at <https://csap.snu.ac.kr/software/>.

## A ARTIFACT APPENDIX

### A.1 Abstract

An artifact is provided to evaluate the NuPoCo framework and to reproduce the results in Section 7.4.

The artifact consists of three components.

- **Runtime environment (RTE):** the NuPoCo resource manager (i.e. the performance model (Section 4) and the parallelism manager (Section 5)) and a library that communicates between the resource manager and the OpenMP applications.
- **The GNU OpenMP library:** several different versions of the GNU OpenMP (GOMP) library are provided to support the execution modes described in Section 7.3. One of them implements the cooperative work scheduling technique presented in Section 6.
- **Benchmark suite:** it includes the benchmark applications used in Section 7.4 and scripts to automatically conduct the experiments.

### A.2 Artifact check-list (meta-information)

- **Algorithm:** the framework supports all of the execution modes described in Section 7.3.
- **Program:** the RTE is written in C++, and benchmark applications and the GOMP library are based on C. Experiment scripts are written in Bash/Python.
- **Compilation:** gcc/g++, makefile
- **Run-time environment:** Linux with the numactl and lnuma tools installed.
- **Hardware:** commodity AMD/Intel multi-socket multi-core systems.
- **Output:** execution times in CSV and plots in EPS files.
- **Experiments:** reproduce the experiments in Section 7.4.
- **Publicly available?:** yes, the repository is hosted by our webserver.

### A.3 Description

**A.3.1 How delivered.** The artifact is delivered as a downloadable software package at <https://csap.snu.ac.kr/software/>. Since some parts are hardware-dependent, we also provide access to the hardware platforms used for the evaluation. To

get access to the target platforms, please email [bernhard@csap.snu.ac.kr](mailto:bernhard@csap.snu.ac.kr).

**A.3.2 Hardware dependencies.** The framework assumes commodity AMD/Intel NUMA multi-socket multi-core systems that support collecting the number of memory requests at each interconnection link and each memory controller from hardware performance counters.

**A.3.3 Software dependencies.** The framework assumes a Linux environment where the numactl and libnuma tools are installed, and the hardware performance counters are accessible via the perf interface or Intel's PCM tool. Our GOMP libraries are based on gcc-5.4.0.

**A.3.4 Data sets.** The data sets of the benchmark applications are provided with the application code.

### A.4 Installation

After downloading the artifact, install the RTE framework, the GOMP libraries, and the benchmark applications using build scripts in the artifact. The detailed installation steps are provided in the README.md file in the artifact.

### A.5 Experiment workflow

To reproduce the results in Section 7.4, the experiment flow consists of the following steps, (1) selecting the execution mode (Section 7.3) through configuration files (2) executing the run script (3) analyzing the log files through scripts. For the details, refer to the README.md file in the artifact.

### A.6 Evaluation and expected result

The artifact supports experiments in Section 7.4 and generates the plot files that correspond to Figure 11.

### A.7 Experiment customization

Experiment customization such as using different applications and using different GOMP libraries is possible. The README.md file contains information to use the NuPoCo framework for different OpenMP applications.

## ACKNOWLEDGMENTS

We thank our reviewers for the helpful feedback. This work was supported by the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT) (NRF-2015K1A3A1A14021288), BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by NRF (21A20151113068), 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] 2018. GNU libgomp. <http://gcc.gnu.org/onlinedocs/libgomp/>. (2018). [online; accessed July 2018].
- [2] AMD. 2012. BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h-0fh processors. (2012).
- [3] AMD. 2014. Revision Guide for AMD Family 15h Models 00h-0Fh Processors. (2014).
- [4] AMD. 2018. AMD Opteron 6300 Series Processors. <http://www.amd.com/en-us/products/server/opteron/6000/6300>. (2018). [online; accessed July 2018].
- [5] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73.
- [6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 29–44. <https://doi.org/10.1145/1629575.1629579>
- [7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. <https://doi.org/10.1145/1454115.1454128>
- [8] OpenMP Architecture Review Board. 2018. OpenMP. <http://openmp.org>. (2018). [online; accessed July 2018].
- [9] Jens Breitbart, Simon Pickartz, Stefan Lankes, Josef Weidendorfer, and Antonello Monti. 2017. Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 400–409. <https://doi.org/10.1109/CLUSTER.2017.59>
- [10] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. 2015. Case Study on Co-scheduling for HPC Applications. In *2015 44th International Conference on Parallel Processing Workshops*. 277–285. <https://doi.org/10.1109/ICPPW.2015.38>
- [11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [12] Younghyun Cho, Surim Oh, and Bernhard Egger. 2016. Online scalability characterization of data-parallel programs on many cores. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 191–205. <https://doi.org/10.1145/2967938.2967960>
- [13] Younghyun Cho, Surim Oh, and Bernhard Egger. 2017. Adaptive Space-Shared Scheduling for Shared-Memory Parallel Programs. In *Job Scheduling Strategies for Parallel Processing. JSSPP 2015, JSSPP 2016. Lecture Notes in Computer Science, vol. 10353*. Springer International Publishing, Cham, 158–177. [https://doi.org/10.1007/978-3-319-61756-5\\_9](https://doi.org/10.1007/978-3-319-61756-5_9)
- [14] Timothy Creech, Aparna Kotha, and Rajeev Barua. 2013. Efficient multiprogramming for multicores with SCAF. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [15] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. <https://doi.org/10.1145/2451116.2451157>
- [16] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 499–508. <https://doi.org/10.1145/2737924.2737999>
- [17] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 254–269. <https://doi.org/10.1145/3064176.3064177>
- [18] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2011. A Workload-aware Mapping Approach for Data-parallel Programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11)*. ACM, New York, NY, USA, 117–126. <https://doi.org/10.1145/1944862.1944881>
- [19] Camilo A. Celis Guzman, Younghyun Cho, and Bernhard Egger. 2017. SnuMAP: an Open-source Trace Profiler for Manycore Systems. <https://csap.snu.ac.kr/software/snumap/>. (2017). [online; accessed July 2018].
- [20] Tim Harris, Martin Maas, and Virendra J. Marathe. 2014. Callisto: Co-scheduling Parallel Runtime Systems. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 24. <https://doi.org/10.1145/2592798.2592807>
- [21] Wim Heirman, Trevor E. Carlson, Kenzo Van Craeynest, Ibrahim Hur, Aamer Jaleel, and Lieven Eeckhout. 2014. Undersubscribed threading on clustered cache architectures. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 678–689. <https://doi.org/10.1109/HPCA.2014.6835975>
- [22] Intel. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual. (2015).
- [23] Intel. 2015. Intel Xeon Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual. (2015).
- [24] Intel. 2018. Intel Performance Counter Monitor - A better way to measure CPU utilization. <http://www.intel.com/software/pcm>. (2018). [online; accessed July 2018].
- [25] Intel. 2018. Intel Xeon Processor E7-8870 v3. [http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2\\_10-GHz](http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2_10-GHz). (2018). [online; accessed July 2018].
- [26] Henk Jonkers. 1994. Queueing models of parallel applications: the Glamis methodology. In *Computer Performance Evaluation Modelling Techniques and Tools*. Springer, 123–138.
- [27] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2013. ADAPT: A Framework for Coscheduling Multithreaded Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 45 (Jan. 2013), 24 pages. <https://doi.org/10.1145/2400682.2400704>
- [28] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. 2010. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 270–279. <https://doi.org/10.1145/1815961.1815996>
- [29] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiawicz. 2009. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, USA, 10–10. <http://dl.acm.org/citation.cfm?id=1855591.1855601>
- [30] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 1, 16 pages. <https://doi.org/10.1145/2901318.2901326>

- [31] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. 2001. Balancing throughput and fairness in SMT processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 164–171. <https://doi.org/10.1109/ISPASS.2001.990695>
- [32] Zoltan Majo and Thomas R. Gross. 2011. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. *SIGPLAN Not.* 46, 11 (June 2011), 11–20. <https://doi.org/10.1145/2076022.1993481>
- [33] Zoltan Majo and Thomas R. Gross. 2012. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 230–241.
- [34] John D. McCalpin. 1991–2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. <http://www.cs.virginia.edu/stream/> A continually updated technical report. <http://www.cs.virginia.edu/stream/>.
- [35] Ryan W. Moore and Bruce R. Childers. 2012. Using utility prediction models to dynamically choose program thread counts. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 135–144. <https://doi.org/10.1109/ISPASS.2012.6189220>
- [36] Bhyrav Mutnury, Frank Paglia, James Mobley, Girish K. Singh, and Ron Bellomio. 2010. QuickPath Interconnect (QPI) design and analysis in high speed servers. In *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*. 265–268. <https://doi.org/10.1109/EPEPS.2010.5642789>
- [37] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA. <https://doi.org/10.1145/1993498.1993502>
- [38] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 133–144. <https://doi.org/10.1145/2254064.2254082>
- [39] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc.
- [40] Gabriele Sartori. 2001. Hypertransport Technology. *Platform Conference* (2001).
- [41] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. 2013. Coordinated power-performance optimization in manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 51–61. <https://doi.org/10.1109/PACT.2013.6618803>
- [42] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. 2012. Scalability-based Manycore Partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 107–116. <https://doi.org/10.1145/2370816.2370833>
- [43] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 137–148. <https://doi.org/10.1109/IISWC.2011.6114174>
- [44] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 169–180. <https://doi.org/10.1145/2594291.2594292>
- [45] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 529–540. <https://www.usenix.org/conference/atc15/technical-session/presentation/srikanthan>
- [46] János Sztrik. 2011. Basic queueing theory. *University of Debrecen: Faculty of Informatics* (2011).
- [47] Bogdan Marius Tudor and Yong Meng Teo. 2011. A practical approach for performance analysis of shared-memory programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 652–663.
- [48] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. 2011. Understanding off-chip memory contention of parallel programs in multicore systems. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 602–611.
- [49] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 3–14.
- [50] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 129–142. <https://doi.org/10.1145/1736020.1736036>