Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory

HAN ZHAO, QUAN CHEN, and YUXIAN QIU, Shanghai Jiao Tong University, China
MING WU, Microsoft Research Asia, China
YAO SHEN, JINGWEN LENG, CHAO LI, and MINYI GUO, Shanghai Jiao Tong University, China

Parallel computers now start to adopt Bandwidth-Asymmetric Memory architecture that consists of traditional DRAM memory and new High Bandwidth Memory (HBM) for high memory bandwidth. However, existing task schedulers suffer from low bandwidth usage and poor data locality problems in bandwidth-asymmetric memory architectures. To solve the two problems, we propose BATS, a task scheduling system that consists of an HBM-aware data allocator, a bandwidth-aware traffic balancer, and a hierarchical task-stealing scheduler. Leveraging compile-time code transformation and run-time data distribution, the data allocator enables HBM usage automatically without user interference. According to data access hotness, the traffic balancer migrates data to balance memory traffic across memory nodes proportional to their bandwidth. The hierarchical scheduler improves data locality at runtime without priori program knowledge. Experiments on an Intel Knights Landing server that adopts bandwidth-asymmetric memory show that BATS reduces the execution time of memory-bound programs up to 83.5% compared with traditional task-stealing schedulers.

CCS Concepts: • **General and reference** → *Performance*; • **Computing methodologies** → *Parallel programming languages*;

Additional Key Words and Phrases: Task-Stealing, Bandwidth, Data Locality, Runtime Scheduling

ACM Reference Format:
Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. 2018. Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Trans. Arch. Code Optim.* 1, 1, Article 10 (October 2018), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In the multicore and manycore era, hardware manufacturers integrate more and more cores into a single computer for fulfilling the ever-growing demands on computational capacity. The increased core number raises a daunting challenge in satisfying memory bandwidth requirement, because all the cores need to access data from main memory concurrently. To this end, recent manycore computers (such as Intel Knights Landing [18], aka. KNL) start to integrate *Multi-Channel DRAM* (MCDRAM) or *Hybird Memory Cube* (HMC) that provides higher bandwidth than traditional DRAM

Authors' addresses: Han Zhao; Quan Chen; Yuxian Qiu, Shanghai Jiao Tong University, Department of Computer Science and Engineering, Shanghai Institute for Advanced Communication and Data Science, Shanghai, 200240, China; Ming Wu, Microsoft Research Asia, Beijing, China; Yao Shen; Jingwen Leng; Chao Li; Minyi Guo, Shanghai Jiao Tong University, Department of Computer Science and Engineering, Shanghai, China, 200240, zhaohan_miven@sjtu.edu.cn;chen-quan@cs.sjtu.edu.cn;qiuyuxian@sjtu.edu.cn,miw@microsoft.com;yshen@cs.sjtu.edu.cn;leng-jw@cs.sjtu.edu.cn;lichao@cs.sjtu.edu.cn;guo-my@cs.sjtu.edu.cn.

XXXX-XXXX/2018/10-ART10 $15.00
https://doi.org/0000001.0000001

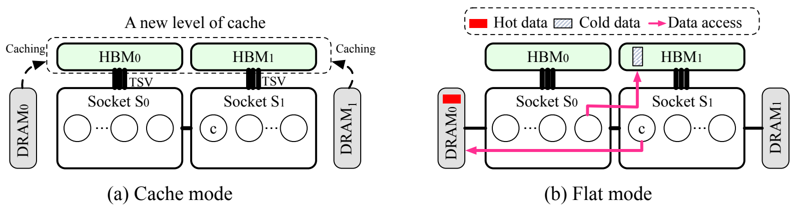


Fig. 1. Two configuration modes of bandwidth-asymmetric memory in manycore architectures.

memory, by stacking multiple DRAM dies on top of CPU chips with *through-silicon vias* (TSV). For easing of description, both MCDRAM and HMC are referred as High Bandwidth Memory (HBM).

Due to the limitations in industrial manufacture (e.g., thermal dissipation), the capacity of the stacked HBM is often smaller than 16GB. HBM cannot directly replace traditional DRAM (hundreds of gigabytes) due to the small capacity. Emerging manycore computers, such as KNL, use HBM along with traditional DRAM to leverage both the high bandwidth of HBM and the large capacity of DRAM memory (this memory architecture is referred as *Bandwidth-Asymmetric Memory architecture*). Note that, unlike non-volatile memory (NVM) that has more than 2X/10X longer read/write latency than traditional DRAM, the access latency of HBM is only 10%-15% longer than the latency of DRAM. Benefit from efficient data prefetching, the slightly longer access latency of HBM does not degrade the performance of an application (proved in Section 3 and prior work [35]).

As shown in Figure 1, bandwidth-asymmetric memory can be configured in two modes: *cache mode* and *flat mode*. In cache mode, HBM dies (called HBM nodes) stacked on all the CPUs as a whole is managed as a new level of cache. Intuitively, in cache mode, the bandwidth of DRAM nodes is wasted because the cores do not directly read/write data from them. In addition, it is possible that a core's data is cached into a remote HBM node. For instance, the data used by a core c in S_1 may be cached in HBM_0. In this case, c has to read data from the remote HBM_0 with much longer latency due to the *Non-Uniform Memory Access* (NUMA) effect. On the other hand, HBM nodes are addressable in flat mode. A parallel program is able to explicitly access data from HBM nodes and DRAM nodes simultaneously. In this case, if the program is memory-bound and the data is distributed properly, it can use higher memory bandwidth in flat mode than in cache mode.

From software aspect, parallel programs are often scheduled with dynamic scheduling policies that can automatically balance its workload across a large number of cores in manycore architectures. In dynamic scheduling policies, the execution of a parallel program is divided into a large amount of fine-grained tasks and is expressed by a task graph (aka. Directed Acyclic Graph or DAG). Each node in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption. Task-sharing and task-stealing (aka. work-stealing) [5] are two widely-used dynamic task scheduling policies. In task-sharing, all the workers (cores) share a central task pool. To guarantee that task-sharing functions properly, all the workers need to acquire a unique "lock" on the central task pool before pushing and popping tasks. For instance, early version of X10 [4] implements a task-sharing scheduler. On the other hand, in task-stealing, each worker has an individual pool that stores its own tasks and only when a worker's pool is empty does it try to "steal" tasks from other workers with locking. Task-stealing performs better than task-sharing due to the reduced competition over locks [5]. For instance, Intel TBB [36], XKaapi [16], LLVM [20] implementation of OpenMP, Cilk++ [22], X10 [21], LAWS [9] and RELWS [25] use task-stealing.

However, task-sharing and task-stealing fail to efficiently utilize the high bandwidth of bandwidth-asymmetric memory in both cache mode and flat mode, resulting in the poor performance of memory-bound programs. In cache mode, the bandwidth of DRAM nodes is wasted and it could incur severe remote memory accesses as stated before. In flat mode, as shown in Figure 1(b), a task-stealing program suffers from both **low bandwidth usage** and **poor data locality** problems.

First of all, using "numactl" interface in Linux to explicitly distribute the data of a program to both DRAM nodes and HBM nodes is the most straightforward way to utilize HBM nodes (all the data is stored in DRAM nodes by default in flat mode). However, it is possible that hot data is stored in DRAM nodes while cold data is stored in HBM nodes. Under this improper distribution, the bandwidth of HBM is under-utilized, resulting in the low bandwidth usage. Meanwhile, random task stealing results in serious remote memory accesses that further degrades a program's performance (pink arrows in Figure 1(b)). Lacking a runtime system that can distribute data based on access hotness and schedule tasks to the sockets where they can access data from local memory nodes, the performance of a memory-bound program could be worse when the bandwidth-asymmetric memory is configured in flat mode than in cache mode, although the available bandwidth in flat mode is higher (to be discussed in detail in Section 3).

To solve the above problems, we configure bandwidth-asymmetric memory in flat mode and propose **BATS**, a task scheduling system that consists of an *HBM-aware data allocator*, a *bandwidth-aware traffic balancer*, and a *hierarchical task-stealing scheduler*. BATS targets for iterative memory-bound programs in which the structures of the task graph are the same in different iterations, and data chunks of a program may have different hotness. Leverage compile-time code transformation and runtime placement, the data allocator automatically distributes the data set of a program to DRAM and HBM nodes without user interference. The traffic balancer monitors the hotness of different data chunks and the access traffic of each memory node. Based on online-collected statistics, the traffic balancer migrates data accordingly to balance the traffic to all the memory nodes proportionally to their bandwidths. The hierarchical task-stealing scheduler identifies appropriate socket for each task so that the task can access its data from either local DRAM node or local HBM node. To the best of our knowledge, **BATS is the first task-stealing runtime system that improves bandwidth utilization of asymmetric memory and enhances data locality automatically without any program modification, hardware modification, or user interference**.

Although Intel recently announced that it will discontinue KNL, other vendors (e.g., Micron and AMD) plan to adopt bandwidth-asymmetric memory in their products to mitigate the low memory bandwidth. Investigation [37] shows that HBM market is continuously growing. Because the design of BATS does not rely on any specific features of KNL, it is also applicable for other computers that adopt bandwidth-asymmetric memory. The main contributions of this paper are as follows.

- **The design of a weighted data distribution mechanism for bandwidth-asymmetric memory**. Benefitting from this mechanism, task-stealing programs can take advantage of the high bandwidth of HBM nodes without user interference.
- **The design of a low-overhead traffic balancing mechanism**. We propose a mechanism that automatically balances the memory traffic across all the memory nodes through sample-based hotness monitoring and hot-first data migration.
- **The design of a hierarchical task-stealing policy**. We design and implement a novel task-stealing policy for improving the data locality without priori program knowledge.

We implement BATS based on MIT Cilk [6] and evaluate it on an Intel KNL server that uses bandwidth-asymmetric memory. Our evaluation shows that BATS significantly reduces the execution time of memory-bound programs compared with traditional task-stealing schedulers.

2 RELATED WORK

Targeting scheduling systems for task-based programs, a large amount of prior work aims to improve energy-efficiency [38, 41], to improve data locality [9, 10], or to reduce scheduling overhead [17, 29]. On the other hand, with the increasing bandwidth requirements of computing tasks, many papers have also conducted related research for efficient bandwidth usage.

Table 1. Comparing BATS with related prior work.

	Improve locality	Optimize data alloc.	Bandwidth aware	No user interfer.	No HW Modifi.	HBM aware
CATS [10]	✓			✓	✓	
HWS [34]	✓				✓	
DFA [15]	✓	✓			✓	
LAWS [9]	✓	✓		✓	✓	
RELWS [25]	✓			✓	✓	
HPT [52]	✓	✓			✓	
Jenga [45]		✓	✓	✓		
BATS	✓	✓	✓	✓	✓	✓

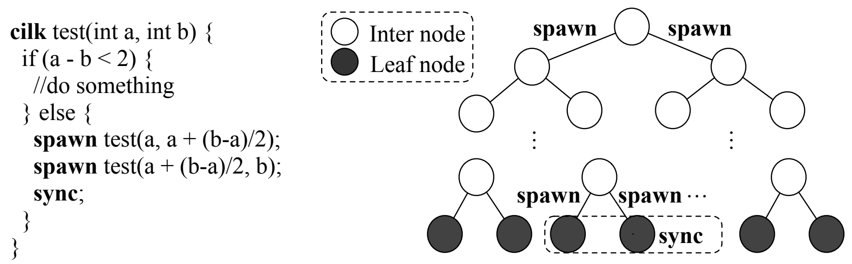
Many task-stealing schedulers have been proposed to improve data locality by reducing shared cache misses [10, 11] and increasing local memory accesses [9, 25, 40, 46]. Based on Charm++ [19], NUMALB [32] is proposed to balance the workload while avoiding unnecessary migrations and reducing cross-core communication. Yoo et al. [53] proposed an offline graph-based locality analysis framework based on HPT [52] designed an interface for programmers to rewrite their programs in a locality-aware manner. CAB [11] and HWS [34] used a rigid boundary level to divide tasks into global tasks and local tasks. By scheduling local tasks within the same socket, the shared cache misses can be reduced. However, users have to give the level manually in HWS or provide a number of command line arguments for the scheduler to calculate the boundary level in CAB. To relieve the above burden, CATS [10] was proposed to divide task graph based on the online information, without extra user-provided information. These techniques assume that the data accessed by a task is known by analyzing the task graph, which is not always true in real-system applications.

Paudel et. al [31] proposed task-stealing schedulers that schedule locality-flexible tasks across nodes and locality-sensitive tasks to where their data is stored. The scheduler relies on programmer-specified locality hints to identify locality-flexible tasks. Drebes et. al [15] proposed data placement techniques to distribute the data set of a program to NUMA nodes based on inter-task data dependencies. Virouleau et. al [47] proposed heuristics to control data placement according to architecture topology and task-data dependencies. Lifflander et al. [25] proposed RELWS that records previous good schedule using steal tree [24] and reuses the good schedule for later iterations.

There is also prior work on the memory bandwidth optimization. Yoon et al proposed a dynamic granularity memory system to save bandwidth for unused data [54], which is an architecture-level solution based on DRAM memory. Carrefour [14] optimized the performance of memory-bound applications by avoiding congestion on memory controllers and interconnects in multi-socket computers. Carrefour neither consider bandwidth asymmetry nor improve data locality. BATMAN [12] aimed at maximizing the bandwidth utilization of bandwidth-asymmetric memory through explicit data movement. However, BATMAN required hardware and software modifications, and did not consider data locality as BATS does. Other work optimized bandwidth utilization when multiple applications run simultaneously: Jenga [45] proposed a module in an HBM-based cache to improve bandwidth usage; Xu et.al [51] designed a bandwidth-aware scheduling method to mitigate memory bandwidth contention between processes on OS; Lin et.al [26] proposed a bandwidth-aware divisible task scheduling algorithm for cloud computing. Our work differs in the aspect that targets bandwidth optimization for a task-based parallel program.

Besides data access performance optimization, prior work also optimizes task-stealing for asymmetric multicore architectures [7, 8, 44]. AAWS [44] used work-pacing, work-sprinting, and work-mugging to improve the CPU-bound applications' performance on both static and dynamic asymmetric multi-core architectures. Li et al. [23] proposed techniques to guarantee the Quality-of-Service of latency-sensitive applications. Those efforts are orthogonal to BATS.

There is also prior work on improving the performance of other schedulers, such as OpenMP. Olivier et. al [30] proposed a hierarchical scheduling strategy that uses one thread to steal work on behalf of all of the threads in a chip. This strategy provides good cache performance and load



```
cilk test(int a, int b) {
  if (a - b < 2) {
    //do something
  } else {
    spawn test(a, a + (b-a)/2);
    spawn test(a + (b-a)/2, b);
    sync;
  }
}
```

(a) A Cilk program (b) The corresponding task graph

Fig. 2. DAG of a task-stealing program.

balance. BubbleSched [43] provided a framework for users to customize thread scheduler and tracing tool. Clet-Ortega et. al [13] evaluated several scheduling strategies and proposed a configurable scheduler to select the task list granularity and choose the most convenient work-stealing strategy for OpenMP. StarPU [2] provided simple tasking APIs so that programmers can easily create task-based programs for heterogeneous hardware, and scheduled tasks based on data-flow dependencies. Planas et. al [33] extended the StarSS syntax so that programmers can adjust task hierarchy by knowing the complexity of the hardware. They are orthogonal to BATS.

Prior work on task-stealing did not consider the bandwidth-asymmetric memory as BATS. Table 1 compares BATS and related work.

3 BACKGROUND AND MOTIVATION

In this section, we first give a general overview of MIT Cilk, a task-stealing programming environment. Then, we present the architecture of Intel KNL server that adopts bandwidth-asymmetric memory, and discuss the hardware configurations. After that, we show existing problems of emerging task-stealing schedulers on computers that use bandwidth-asymmetric memory.

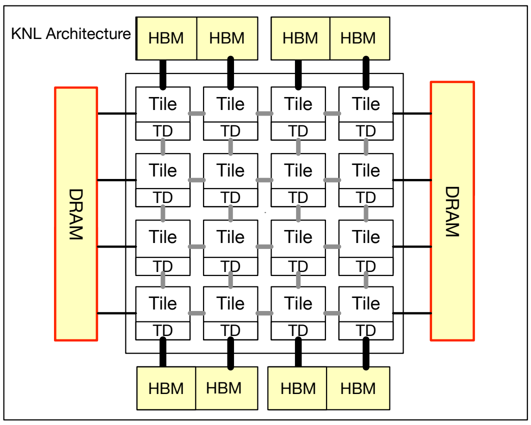
3.1 Task-stealing Program Background

Our work targets task-based parallel program. Figure 2(a) shows a task-based parallel program written in MIT Cilk [42] (denoted by "Cilk" for short), a task-stealing programming language based on C language. Cilk extends the C language with three keywords: *cilk*, *spawn* and *sync*. The *cilk* keyword identifies that a function can be executed in parallel, the *spawn* keyword specifies the parallel function invocation, and the *sync* keyword guarantees that no statement after it can be executed until all preceding tasks complete.

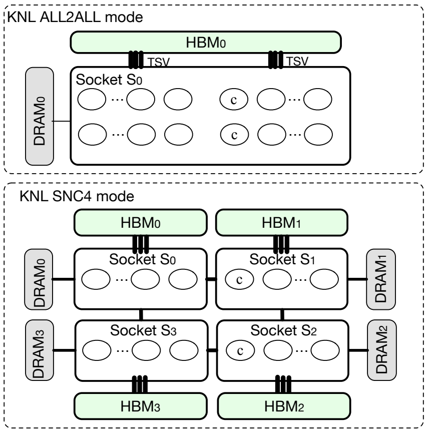
Figure 2(b) shows the task graph of the the Cilk code in Figure 2(a). In the task graph, each node represents a task, each edge represents the "spawn" relationship between two tasks. Observed from this figure, we can find that only leaf tasks actually perform the computation, while the other nodes divide the workload into smaller pieces. When the program in Figure 2(a) runs in a multicore/manycore architecture, Cilk runtime system adopts a random task-stealing policy to schedule all its tasks. It is worth noting that the task graph of a task-based program is not known before it is executed, because the tasks are dynamically generated at runtime. It is not applicable to analyze the task graph of a task-stealing program offline to find it optimal scheduling.

3.2 Hardware Configuration

We use an Intel KNL server that uses bandwidth-asymmetric memory as the experimental platform. Figure 3(a) shows hardware architecture of the KNL server. Observed from the figure, eight HBM nodes are attached with the KNL processor, and the cores are connected with a mesh interconnect. The interconnect can be configured in five clustering modes: ALL2ALL, QUADRANT, HEMISPHERE,



(a) KNL physics architecture



(b) KNL logical architecture

Fig. 3. KNL architectures and its interconnect modes.

SNC2, and SNC4. In ALL2ALL mode (Figure 3(b)), all the cores can route each other and can access all memory directly (similar to single-socket multi-core architecture). In SNC4/SNC2 mode (Figure 3(b)), the cores are divided into four/two parts, while exposing HBM nodes as NUMA nodes (similar to multi-socket architecture with NUMA memory). In QUADRANT/HEMISPHERE mode, all the cores are divided into four/two parts as well, but the cores can directly access all the memory nodes.

By default, the interconnect of a KNL server is configured in ALL2ALL mode, while the memory is configured in flat mode. With this configuration, the performance of a memory-bound program p is never optimal, because there is no interface to balance its data to the physically disjoint HBM/DRAM nodes and p's tasks have to access data from remote HBM/DRAM nodes.

To control data placement based on a program's data access pattern, similar to traditional multi-socket computers with NUMA memory, we configure the bandwidth-asymmetric memory in flat mode and the interconnect in SNC4 clustering mode (recommended in prior work [48] for memory-bound applications). Our experiment in Section 9.5 shows that memory-bound programs achieve the best performance when the KNL server is in SNC4+flat mode and the tasks are scheduled with BATS, compared with all the other scheduler-mode combinations.

When the KNL server is configured in SNC4+flat mode, memory-bound programs do not achieve good performance automatically. By default, a legacy program would only use DRAM in flat mode. The easiest way to use HBM is explicitly allocated data to HBM using "numactl" interface in Linux. Using "numactl", there are three policies to distribute the data of a program without modifying its source code: *all-DRAM policy*, *even allocation policy*, and *all-HBM policy*. In all-DRAM policy (the default policy), all the data is stored in DRAM nodes. In even allocation policy, we can use "numactl –interleave" to evenly distributes the data of a parallel program to all the memory nodes in a round-robin manner. There is no interface to precisely control the percentage of data allocated to a memory node. In all-HBM policy, we can force the program to store all its data in HBM nodes. Obviously, the all-DRAM policy wastes the bandwidth of HBM nodes, the all-HBM policy wastes the bandwidth of DRAM nodes, the even allocation policy can better take advantage the bandwidth of all the memory nodes when the bandwidth-asymmetric memory is configured in flat mode.

Figure 4(a) and Figure 4(b) show the bandwidth and access latency of an HBM node normalized to the counterparts of a DRAM node measured with Intel MLC tool [1]. Observed from Figure 4, the bandwidth of an HBM node is 4.2X of the bandwidth of a DRAM node, while the latency of

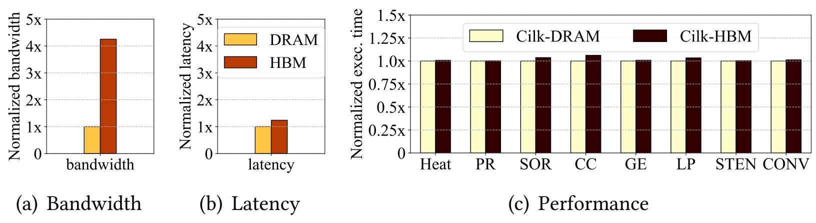


Fig. 4. Bandwidth and latency of DRAM and HBM, and the impact of latency on application performance.

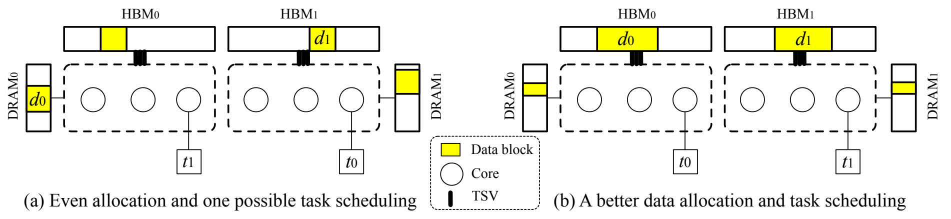

HBM0
HBM1
DRAM0
DRAM1
d0
d1
t0
t1
Data block
Core
TSV
(a) Even allocation and one possible task scheduling
(b) A better data allocation and task scheduling

Fig. 5. Two schedulings of program p's tasks on a computer that uses bandwidth-asymmetric memory.

an HBM node is 1.1X of the latency of a DRAM node. To show the impact of the slightly longer access latency of HBM nodes on application performance, we run all the benchmarks in Table 2 with one thread so that the required memory bandwidth is smaller than the bandwidth of a DRAM node. Figure 4(c) shows their performance in Cilk-DRAM and Cilk-HBM that store data in a DRAM node and an HBM node respectively. Observed from Figure 4(c), except *CC*, the slightly longer latency of HBM nodes does not degrade application performance. This finding is consistent with the observation in prior work [35]. We do not consider the impact of access latency in BATS.

3.3 Problems in Random Task-stealing

However, memory-bound task-based programs still suffer from poor performance when their data is allocated with the even allocation policy. To better explain the problem, Figure 5 shows two data allocations and two possible schedulings of tasks t_0 and t_1 in a memory-bound program p. In the figure, d_0 and d_1 are the data chunks that contain the data used by t_0 and t_1 respectively. Observed from Figure 5(a), even data allocation and random task-stealing incur **low bandwidth usage** and **poor data locality** problems. The two problems together result in the poor performance of p on computers with bandwidth-asymmetric memory.

The low bandwidth usage problem originates from two sources. First of all, the bandwidth of a HBM node is much higher than the bandwidth of a DRAM node but the dataset of program p is often evenly allocated to all the memory nodes as shown in Figure 5(a). In this case, even if all the data has similar hotness, it is highly possible that DRAM nodes are overloaded while the bandwidth of HBM nodes is still under-utilized. In addition, p's dataset may have uneven access hotness. For instance, it is possible that all the hot data of program p is stored in a single memory node $DRAM_0$, while all the other memory nodes store the cold data. In this case, most memory traffic is served by $DRAM_0$, while the bandwidth of $DRAM_1$, HBM_0 and HBM_1 is under-utilized. Note that, with the all-HBM policy or all-DRAM policy, if different parts of p's dataset have different hotnesses, it is also possible that most hot data is allocated to one of the HBM or DRAM nodes.

As for the poor data locality problem, tasks of p are randomly scheduled to different cores with random task-stealing. It is highly possible that tasks t_0 and t_1 are scheduled as shown in Figure 5(a). In this case, t_0 and t_1 have to access their data d_0 and d_1 from $DRAM_0$ and HBM_1 remotely. The

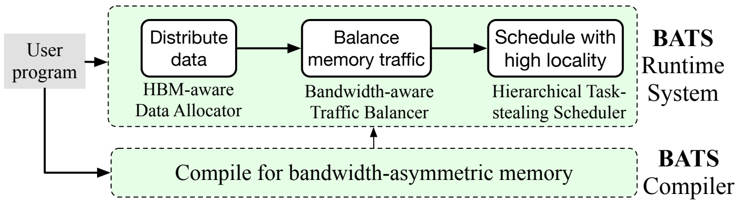

User program
Distribute data
HBM-aware Data Allocator
Balance memory traffic
Bandwidth-aware Traffic Balancer
Schedule with high locality
Hierarchical Task-stealing Scheduler
BATS Runtime System
Compile for bandwidth-asymmetric memory
BATS Compiler

Fig. 6. The design of BATS.

serious remote memory accesses further degrade the performance of p. Random task-stealing also results in the poor data locality problem even if p's dataset is allocated in all-DRAM policy or all-HBM policy. According to our measurement in Section 9, more than 70% of memory accesses are from/to remote nodes with traditional random task-stealing on our KNL server.

According to the above analysis, the coarse-grained even allocation policy with "numactl" results in low bandwidth usage problem and the random task-stealing policy results in poor data locality problem. For traditional computers that only use DRAM nodes, prior locality-aware schedulers (e.g., LAWS [9] and RELWS [25]) partially solve the second problem under the assumption that a task's data set is known according to task graph topology. However, this assumption is not valid for programs that have uneven data access patterns. In addition, they are not applicable for computers that employ bandwidth-asymmetric memory (evaluated in Section 9).

If the dataset of p is allocated in the way that memory traffic is balanced across all the memory nodes proportionally to their bandwidth, while t_0 and t_1 are scheduled to the cores where they can access their data d_0 and d_1 from local memory nodes as shown in Figure 5(b), the performance of p can be improved. However, there is no interface to precisely control how the dataset is split and distributed without modifying the source code of a legacy program. Made more challenging, because different parts of a program's data set may have different access hotness and the hotness is not known before program execution, a static data distribution could easily result in unbalanced memory traffics. Furthermore, which piece of data will a task accesses is not known before the task actual runs, invalidating prior locality-aware task-stealing techniques.

4 THE BATS METHODOLOGY

Figure 6 presents the design of **BATS** that is comprised of a source-to-source compiler and a runtime system to address the above challenges. The runtime system has three components: an *HBM-aware data allocator*, a *bandwidth-aware traffic balancer*, and a *hierarchical task-stealing scheduler*.

BATS targets iterative programs where the task graph structures of different iterations are identical. Before scheduling such an iterative program p with BATS, we first compile p using BATS compiler. The compiler automatically transforms p's source code and generates the binary so that BATS runtime system can specify the memory node in which a task's data set is stored. When p starts to run, the HBM-aware data allocator splits and distributes p's dataset to all the memory nodes proportional to their bandwidth in the initialization iteration (Section 5).

In the first iteration, the traffic balancer monitors and balances the traffic of all the memory nodes. To achieve this purpose, BATS migrates hot data from overloaded memory nodes to under-utilized memory nodes. Because BATS only migrates data in the first iteration, the migration overhead can be easily amortized in later iterations (Section 6).

In the following several iterations, for every task t, BATS identifies the socket in which t can access most of its data from local memory nodes. To reduce overhead, BATS divides the task graph of an iteration into *intra-socket subgraphs*. The tasks in the same intra-socket subgraph are profiled and scheduled in a whole between the sockets. On a computer having M socket, BATS schedules each intra-socket subgraph SG to a different socket in M iterations, and identifies the socket in

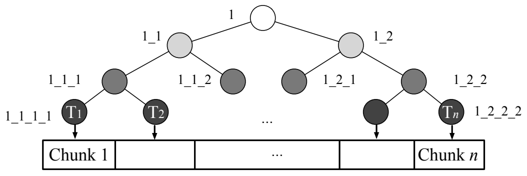


Fig. 7. An example task graph of initializing a program's data set in parallel.

which *SG* achieves the highest local data accesses (Section 6.1). This high locality mapping is recorded for guiding subgraph allocation in future iterations.

Starting from the $(M+1)$-*th* iterations, BATS adopts a hierarchical task-stealing policy to schedule tasks in a bandwidth and locality-aware manner (Section 7). Each intra-socket subgraph is directly allocated to the socket where it can find most of its data from local memory nodes. Meanwhile, if a socket completes all its intra-socket subgraphs, it tries to steal a subgraph from a randomly chosen victim socket in order to solve the possible slight load imbalance.

5 HBM-AWARE DATA ALLOCATION

Most existing task-stealing programs (e.g., examples in MIT Cilk package) adopt parallelized data initialization, in which the whole dataset of a program is initialized by multiple tasks. By scheduling these data initializing tasks to different sockets, the dataset of the program is split and distributed to multiple memory nodes (*first-touch policy*). In this case, workers can access data from multiple memory nodes concurrently, increasing the available bandwidth. Same to state-of-the-art locality-aware task-stealing schedulers, such as LAWS [9] and RELWS [25], BATS targets for programs that use parallelized data initialization.

Figure 7 shows an example task graph of the parallelized data initializer of a task-stealing program p. In the figure, p's data set is split into n equal-size chunks, and each leaf task allocates memory space for a chunk. In this paper, we assume that the data initializer adopts divide-and-conquer pattern to initialize data chunks in parallel using "malloc", and all the data of the program is initialized by the initializer. Program p can also directly spawn n tasks to perform the initialization, and the generated task graph equivalents to a two-level task graph where the root task has n branches. BATS assumes that applications adopt the above parallelized data allocation mode.

5.1 Weighted Data Distribution

Without prior knowledge of p's data access pattern, it is not possible to identify the optimal data distribution for p in the initialization iteration. As a reasonable start point, we assume all the initialized data has the same hotness and rely on the bandwidth-aware traffic balancer to adjust data distribution according to p's real data access pattern later.

Specifically, we design a weighted data distribution scheme that satisfies two constraints. First, if the bandwidth of memory node N_a is n times of the bandwidth of memory node N_b, n times more data should be stored in N_a. Second, adjacent data should be stored in the same memory node to better utilize data prefetching and spatial locality. Traditional task schedulers do not satisfy both the two constraints, lacking the ability to control the data placement.

Suppose p runs on a computer that has k memory nodes: N_1, ..., N_k. Let B_1, ..., B_k represent their bandwidth respectively, and D represent p's dataset size. Obeying the first constraint, Equation 1 calculates the size of data that should be stored in N_i (denoted by S_i).

$$S_i = D \times \frac{B_i}{\sum_{j=1}^{k} B_j} \tag{1}$$

To achieve the data distribution that satisfies Equation 1, BATS distributes data based on the topology of the initializer's task graph. As shown in Figure 7, suppose there are n tasks (denoted by $T_1, ..., T_n$) in the leaf level of the initializer's task graph. In this case, the data initialized by tasks from T_s to T_e (included) is stored in memory node N_i ($s < e$), where s and e are calculated in Equation 2.

$$s = \lfloor n \times \frac{\sum_{j=1}^{i-1} B_j}{\sum_{j=1}^{k} B_j} \rfloor ; e = \lfloor n \times \frac{\sum_{j=1}^{i} B_j}{\sum_{j=1}^{k} B_j} \rfloor - 1 \tag{2}$$

Distributing data based on Equation 2 satisfies both the two constraints. First of all, $e - s$ tasks allocate memory pages for their data in memory node N_i, and the size of data stored in N_i is $(e - s) \times \frac{D}{n} = S_i$, satisfying Equation 1 (the first constraint). In addition, the data initialized by tasks $T_s, ..., T_e$ are adjacent parts of p's dataset, satisfying the second constraint.

In Equation 2, B_i can be found in the machine design document. It is also easy to profile them using bandwidth profiler such as the Intel latency checker [1]. Furthermore, to actually distribute data according to Equation 2, BATS needs to know the number of leaf tasks and the position of each leaf task in the leaf level. In our current implementation, each task is automatically given a unique identifier (a string) when it is generated. As shown in Figure 7, if a task's identifier is S, then its ith sub-task's identifier is S_i. The string by side of each task in Figure 7 is its identifier. By sorting the tasks according to their identifiers, the position of each task can be identified.

5.2 Compiling Support

The weighted distribution scheme requires that BATS has the ability to explicitly specify the memory node in which each individual initialization task stores its corresponding data. However, the numactl interface is too coarse-grained to achieve this purpose, while the *first-touch policy* itself is not able to "touch" data to HBM nodes.

The task identifier is also used to identify the tasks in the same position of task graphs in different iterations (tasks that have the same identifiers) and specify memory nodes for them. During the program execution, if the scheduler finds that an identifier ID appears again for the first time, the task is the root of an iteration's task graph. Therefore, whenever a task that has identifier ID is generated, the program enters a new iteration.

To precisely control data placement in the absence of user interference, BATS adopts a compile-time and run-time joint solution. In this solution, BATS runtime system declares a globally visible parameter `mem_id`, and BATS compiler transforms the data allocation instructions at compile-time as follows. The instruction "`malloc (space_size)`" in the data initializer is transformed into "`numa_alloc_on_node (space_size, mem_id)`", which allocates pages of *space_size* on the memory node numbered with `mem_id`, no matter it is a DRAM node or an HBM node.

By analyzing task identifiers, BATS can find the number of tasks in each level of the task graph and their positions. Let (T_s, T_e) represent the range of tasks that should store their data chunks on memory node N_i as calculated in Equation 2. When a leaf task T_j is generated, if $s \leq j < e$, the `mem_id` of T_j equals to i. The calculated `mem_id` is saved in the struct `CilkStackFrame` that stores T_j's meta-data. The instruction "`numa_alloc_on_node (space_size, mem_id)`" in T_j reads *mem_id* from its *CilkStackFrame*.

Readers may think that it is easy to manually declare `mem_id` in user program and change instruction "`malloc (space_size)`" into "`numa_alloc_on_node (space_size, mem_id)`". However, this program-side solution is not working in BATS, because parameters declared in a program is not visible in runtime system. In this case, the scheduler is not able to calculate the value of `mem_id` for each task accordingly.

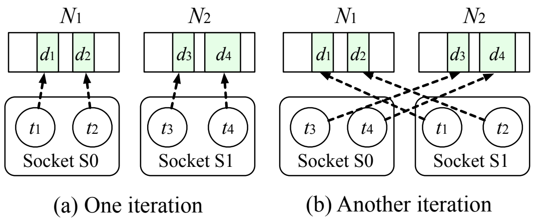


Fig. 8. The memory accesses in two iterations.

6 BANDWIDTH-AWARE TRAFFIC BALANCING

Adopting the weighted data distribution scheme, memory nodes may still suffer from unbalanced memory traffic when different parts of p's dataset have different access hotness.

BATS identifies whether the memory traffic is balanced in the first iteration, and mitigates the imbalance by migrating data. We adopt this method because later iterations would also suffer from traffic imbalance if the imbalance exists in the first iteration. Figure 8 shows the data access behavior of four tasks (t_1, ..., t_4) in different iterations of an iterative program in a dual-socket architecture. As shown in this figure, the traffic of memory node N_1 (and N_2) are the same in the two iterations although the tasks are scheduled differently. For instance, data block d_1 stored in N_1 is always accessed by task t_1, no matter t_1 is scheduled to socket $S0$ or $S1$.

6.1 Identifying the Traffic Imbalance

To identify whether the data accesses of program p are balanced across all the k memory nodes N_1, ..., N_k, BATS needs to collect the traffic to each memory node. A naive method is to directly collect the traffic of each memory node at runtime. However, it is not applicable in real system, because there are no such hardware performance counters. In emerging real-system hardware, for a memory node N_i, we can only collect the number of local memory accesses but cannot find the number of accesses to N_i from remote sockets.

To solve this problem, we propose to monitor the number of accesses to each memory page by modifying operating system and calculate the traffic to each memory node according to the location of each memory page. This method is applicable when the dataset size of p is small. However, if program p accesses too many pages, the overhead of monitoring the number of accesses to every page is too large to be directly used in the real system. According to our measurement, sweeping through one million pages (4GB) to check whether each page is accessed consumes 3 seconds.

To reduce the overhead of identifying the imbalance, we leverage the virtual memory area (VMA) notion in Linux kernel. Each VMA is a contiguous range of virtual addresses. In our scenario, the virtual pages allocated by each leaf task in the parallel data initializer form an individual VMA, and the VMAs have the same number of virtual pages because leaf tasks allocate memory for the data of the same size. BATS treats the virtual pages in a VMA as a whole, monitors the access hotness and migrates data in the granularity of a VMA. According to the virtual address, we can obtain the memory node corresponding to each VMA. By counting the access number of every VMA of program p, we can obtain the overall access number to all the memory nodes and the number of accesses to each node (denoted by A_1, ..., A_k) in the first iteration. Equation 3 calculates the number of memory accesses to memory node N_i, in which m is the number of VMAs in N_i, and Hot_j is hotness of the j-th VMA in N_i.

$$A_i = \sum_{j=1}^{m} Hot_j \tag{3}$$

ALGORITHM 1: Bandwidth-aware traffic balancing algorithm

```
Input: $N_1, ..., N_m$ (Overloaded memory nodes)
Input: $N_{m+1}, ..., N_k$ (Under-utilized memory nodes)
Input: $OPT_1, ..., OPT_k$ (Optimal traffics to the $k$ nodes)
for ($und = k; und > m + 1; und - -$) do        // Migrate data to the nodes with the lowest util. first
    REQ = $OPT_{und} - A_{und}$ ;
    if $REQ > 0$ then // Still under-utilized
        for ($ol = 1; ol \leq m; ol + +$) do        // Offload data from the busy nodes first
            HR = $A_{ol} - OPT_{ol}$ ;
            while $HR > 0$ do
                Identify the hottest VMA $v$ in $N_{ol}$ whose hotness ($H_v$) < HR and REQ ;
                if $v == NULL$ then
                    break ;
                else
                    Migrate $v$ to $N_{und}$ ;
                    HR $- =$ $H_v$; REQ $- =$ $H_v$ ;
```

In more detail, we randomly choose a small number of virtual pages (saying b pages) from every VMA and collect the numbers of memory accesses to the b pages. Based on the number of accesses to the sampled pages, we accumulate the number of memory accesses to all the VMAs in a memory node N_i to be the hotness of N_i. Equation 4 calculates the hotness of VMA v, in which n_p is the number of virtual pages in v and a_i is the number of accesses to the i-th sampled page in v.

$$Hot = \frac{n_p}{b} \times \sum_{i=1}^{b} a_i \tag{4}$$

Equation 4 assumes that the pages in a VMA tend to have similar access hotness. Although Equation 4 is only an approximation of the number of accesses to VMA v, our experiment shows that it is enough to identify the traffic imbalance between multiple memory nodes.

We identify whether the memory traffic is balanced as follows. The memory traffic is proportionally balanced to the k memory nodes according to their bandwidth when Equation 5 is satisfied. In the equation, B_i is the bandwidth of N_i as defined in Section 5.1.

$$B_1 : B_2 : ... : B_k = A_1 : A_2 : ... : A_k \tag{5}$$

Deduced from Equation 5, Equation 6 calculates the optimal memory traffic to N_i (denoted by OPT_i) when the traffic is balanced. For N_i, if $A_i < OPT_i$, its current memory traffic is lower than its optimal traffic thus it is under-utilized. Otherwise, it is overloaded as N_i's memory traffic is higher than its optimal traffic.

$$OPT_i = \sum_{j=1}^{k}(A_j) \times \frac{B_j}{\sum_{j=1}^{k} B_j} \tag{6}$$

Based on Equation 6 and online-collected information, BATS migrates data from overloaded memory nodes to under-utilized memory nodes to mitigate the traffic imbalance.

6.2 Mitigating the Traffic Imbalance

Based on the hotness of every VMA, we design a hot-first migration scheme following three principles. First, if both N_i and N_j are overloaded but N_i is more congested than N_j, we offload data from N_i to the under-utilized memory nodes before N_j. Second, if both N_i and N_j are under-utilized but the utilization of N_i is lower than the utilization of N_j, the data is migrated to N_i before N_j. Third, we migrate hot VMA before cold VMA to minimize the size of data to be migrated for balancing the traffic.

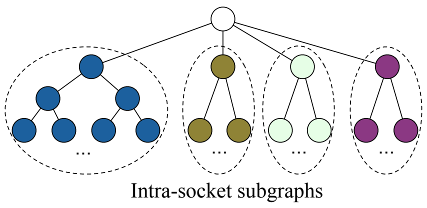


Fig. 9. Group tasks into intra-socket subgraphs.

Algorithm 1 lists the designed traffic balancing algorithm according to the hot-first migration scheme. In the algorithm, the k memory nodes N_1, ..., N_k are sorted in the descending order of $A_i - OPT_i$ $(1 \leq i \leq k)$. The m memory nodes N_1, ..., N_m are overloaded and the later N_{m+1}, ..., N_k are under-utilized. Memory node N_i is more congested than N_j if $i < j$.

7 HIERARCHICAL TASK-STEALING

After the memory traffic is balanced, tasks may still suffer from serious remote memory access in later iterations. To improve data locality, BATS adopts a hierarchical task-stealing policy.

7.1 Online Profiling

BATS groups tasks into *intra-socket subgraphs* and identifies optimal socket for each of them. Figure 9 shows the way to group tasks into intra-socket subgraphs. In the figure, the subgraph in each ellipse is an intra-socket subgraph. Tasks in the same subgraph have the same optimal socket in which they achieve the best locality because neighbor tasks often process adjacent data stored in the same memory node. Another reason we choose this method is that it is too space-consuming to record the mapping of each individual task to its socket and too time-consuming to obtain the mapping online for a task. We evaluate the space overhead of online profiling in Section 9.7.

The intra-socket subgraphs are created satisfying three constraints. First, an intra-socket subgraph should have a single root node for easy scheduling. Second, the number of tasks in an intra-socket subgraph is at least 10X of the number of cores in a socket. In terms of the second constraint, as stated in prior work [5], when the task number is 10X of the core number, the workload can be balanced across the cores with task-stealing. Third, an intra-socket subgraph should have the smallest number of tasks that fulfill the aforementioned two constraints at the same time.

When the first iteration of p completes, BATS knows the task graph topology and groups tasks according to the above constraints. On an M-socket computer, in the following M iterations, BATS schedules each subgraph SG to each of the M sockets and collects its memory access statistics. Let L_{dram}, R_{dram}, L_{hbm}, and R_{hbm} represent the local DRAM accesses, remote DRAM accesses, local HBM accesses, and remote HBM accesses caused by SG when it is scheduled to socket S.

If $L_{dram} + L_{hbm} \geq R_{dram} + R_{hbm}$, SG can access most of the data from either the local DRAM node or HBM node attached with S. In this case, SG is directly allocated to socket S in all the later iterations, and SG is called a *high-localized subgraph* of S. Otherwise, if the local memory accesses (local DRAM accesses + local HBM accesses) of SG on all the M sockets are smaller than the corresponding remote memory accesses, BATS assigns SG to the socket S_i in which SG has the largest local memory accesses, and SG is called a *low-localized subgraph* of S_i in this case.

7.2 Scheduling Policies

Figure 10 gives the structure of BATS runtime system on an M-socket computer with bandwidth-asymmetric memory and illustrates our hierarchical task-stealing design. On each core, BATS launches a worker. For easing of description, we use “core” to represent “worker”.

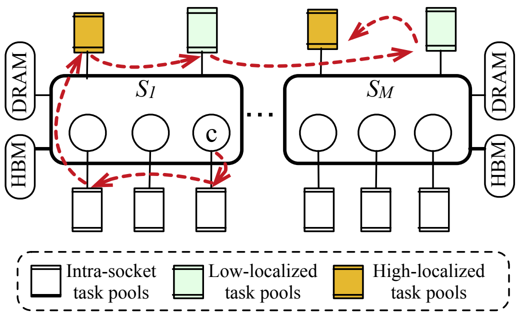


Fig. 10. Structure of BATS runtime system on an M-socket computer with bandwidth-asymmetric memory.

BATS creates a *high-localized task pool* and a *low-localized task pool* for each socket S, while the high-localized task pool stores the root tasks of its high-localized subgraphs and the low-localized task pool stores the root tasks of its low-localized subgraphs. For each core, BATS creates an *intra-socket task pool* to store the tasks within an intra-socket subgraph. Suppose a core c in socket S is free, in different iterations, it obtains new tasks in different ways.

In the initialization iteration, all the tasks are first pushed into a single task pool, saying the high-localized task pool of socket S_1 in Figure 10. Once all the tasks are generated, BATS schedules the tasks according to the weighted data distribution scheme. After that, BATS adopts random task-stealing to schedule the tasks. Note that, when the leaf tasks are distributed to the cores, the memory node in which a task will store its data set is already known according to Equation 2.

In the first compute iteration, BATS adopts random task-stealing to schedule all the tasks because the data is supposed to be migrated later and the task scheduling does not affect the memory traffic to each memory node. At the end of the first iteration, BATS migrates data to balance the traffic across all the memory nodes.

In the following M iterations, each intra-socket subgraph is directly allocated to the target socket for collecting the numbers of local memory accesses and remote memory accesses. Meanwhile, core c is not allowed to steal tasks from other sockets and cores in the same socket are not allowed to execute tasks in multiple intra-socket subgraphs concurrently.

After at most M iterations, intra-socket subgraphs are stabilized to different sockets. In this stage, adopting hierarchical task-stealing, core c can steal a new task from five levels: intra-socket task pools of other cores in its socket S, high-localized task pool of S, low-localized task pool of S, low-localized task pool of other sockets, and high-localized task pools of other sockets.

BATS allows a socket to help other sockets execute their intra-socket subgraphs. After all the tasks in both the high-localized and low-localized task pools of S complete, c tries to steal a task from low-localized task pools of other sockets. If the low-localized task pools of all the other sockets are empty, c tries to steal a task from high-localized task pools of other sockets. Core c first tries to steal a task from low-localized task pools because the victim sockets execute the high-localized subgraphs more efficiently due to better data locality. In addition, although S needs longer time to process the stolen subgraphs, the workload is balanced and the performance of memory-bound programs is highly possible to be improved.

8 IMPLEMENTATION OF BATS

We implement BATS in MIT Cilk that consists of a compiler and a random task-stealing scheduler. We modify the compiler to transform programs as described in Section 5.2; and modify the task-stealing scheduler to support bandwidth-aware traffic balancing and hierarchical task-stealing.

8.1 Implementation of Hotness Monitoring

In BATS, we implement a module (enhanced from Memos [27] and SysMon [49]) to collect the access hotness of VMAs in kernel space. The two frameworks have been proved to be able to precisely collect the hotness of pages [27, 49]. The module passes hotness data to BATS runtime system through "/proc" virtual file system in Linux operating system.

Basically, the module first obtains `process descriptor` of the target program from its PID, and obtains VMA structs of the program. For each VMA, the module finds the page table entries in it according to page directory conversion in Linux. The module then selects the sampled pages according to the policy in Section 6.1 and monitors their hotness. For a page *pg*, the `_access_bit` in its page table entry represents whether it has been accessed. We do not use `_dirty_bit` here because it is set only when the page is modified, ignoring read operations. The module periodically checks whether *pg* is accessed in the first iteration. In each period, the module uses system call `pte_young()` to check `_access_bit` of *pg* and then clear the bit using `pte_mkold()`. In this way, the module obtains the number of times that *pg* is accessed in the first iteration.

We minimize the overhead of hotness monitoring by ignoring the VMAs that are related to the data set of the program. In Linux, the VMA structs of a program contain not only data segment, but also text segment, Block Started by Symbol (BSS) segment, etc. When collecting access hotness of VMAs, the module skips file-mapping VMAs, stacks for every thread, unreadable/unwritable/executable VMAs, and VMAs that contain data for the scheduler itself.

Another problem we need to solve is that Linux tends to merge VMAs if their virtual addresses are continuous, but BATS assumes that the pages allocated by each leaf task in the data initializer form an individual VMA. To solve this problem, BATS calculates the overall number of pages allocated by the program (denoted by N_p), and obtains the number of leaf tasks in the data initializer (denoted by N_l). By dividing N_p by N_l, we can get the actual number of pages allocated by a leaf task in the data initializer (N_{vma}). The merged VMAs are then divided into "logic" VMAs of size N_{vma}. The hotness monitoring and data migration are performed in the granularity of "logic" VMA.

In our current implementation, the module uses four threads to collect hotness of VMAs concurrently. In the first iteration, BATS scheduler disables four cores for the module to ensure the performance of the hotness detection. After getting the hotness data, the module stores the addresses of VMAs and their hotness in "/proc" virtual file system. BATS identifies the memory node that each VMA belongs to using move_pages() function, and applies Algorithm 1 to identify the VMAs to migrate. BATS uses numa_move_pages() to perform the page migration.

8.2 Implementation of Online Profiling

During online-profiling, BATS needs to collect local/remote DRAM access, local/remote HBM access for every intra-socket subgraph. However, a core in KNL can only collect two events because a tile (consists of two hardware cores, eight virtual cores) only has two performance counter registers.

To solve this problem, we modify the online profiling algorithm slightly for KNL. When profiling an intra-socket subgraph *SG* on a socket, the two counters first record the numbers of DRAM accesses (OFFCORE_RESPONSE_0: DDR) and HBM accesses (OFFCORE_RESPONSE_0: MCDRAM) respectively. If *SG* accesses more data from DRAM/HBM, the two counters record the numbers of its local DRAM/HBM accesses (OFFCORE_RESPONSE_0: DDR_NEAR / HBM_NEAR) and its remote DRAM/HBM accesses (OFFCORE_RESPONSE_0: DDR_REMOTE / HBM_REMOTE) in later online-profiling iterations respectively. BATS uses local DRAM/HBM accesses and remote DRAM/HBM accesses to approximate the overall local and remote memory accesses in Section 6.1.

Because local DRAM/HBM accesses and remote DRAM/HBM accesses are approximations, our current implementation of BATS for KNL performs worse than its implementation if each core has four performance counter registers in future. Our evaluation shows that the current implementation of BATS for KNL is already able to significantly improve the performance of memory-bound programs.

9 EVALUATION

In this section, we first detail our experimental setup and evaluate the performance advantages of BATS over existing task-stealing schedulers such as Cilk and LAWS. Then, we show the scalability of BATS, and demonstrate how the three components in BATS, i.e., data allocator, traffic balancer, and scheduler, contribute to the overall effectiveness. Lastly, we compare BATS with OpenMP that adopts static scheduling and analyze the overhead of BATS.

9.1 Experimental Configuration

We use an Intel Knights Landing (KNL) server that uses bandwidth-asymmetric memory as the experimental platform to evaluate the performance of BATS. We configure the memory of KNL in *flat mode* so that the schedulers can explicitly manage all the memory nodes. We also configure the interconnect of KNL in *SNC4 mode* so that the cores are divided into four sockets. As a result, each socket has 64 virtual cores, a 32GB DRAM node and a 4GB HBM node. The peak bandwidth of an HBM node (384/4=96GB/s) is around 4× of the peak bandwidth of a DRAM node (90/4=22.5GB/s).

We compare BATS with state-of-the-art task-stealing schedulers: MIT Cilk (version 5.4.6) [6] and LAWS [9]. LAWS evenly distributes the dataset of a program to all the memory nodes and schedules tasks to the sockets where the memory nodes store their data for applications with regular data access patterns. However, LAWS considers neither bandwidth-asymmetric memory nor traffic imbalance as BATS does. Once the data is stored in a memory node in LAWS, it would not be migrated to other memory nodes even if the memory traffic is not balanced. All the tested schedulers use 256 workers to fully utilize all the cores in KNL.

While both Cilk and LAWS store data in DRAM nodes by default, for the sake of comprehensive comparison, we also evaluate the cases of using HBM as the last-level cache for Cilk and LAWS. We achieve that by configuring KNL in the ALL2ALL-cache mode. We call them as Cilk-C and LAWS-C, respectively. We skip the comparison for SNC4-cache mode because our results show that its performance is worse than ALL2ALL-cache mode and we give the comparison of different modes on KNL in Section 9.5. We also evaluate two other cases by forcing Cilk and LAWS to use HBM nodes only, which we call Cilk-H and LAWS-H, respectively. This two cases are achieved by using “`numactl --membind=4-7`” as the prefix to launch the benchmark (memory nodes 0-3 are DRAM nodes, and 4-7 are HBM nodes), Note that, if we force Cilk and LAWS to use both DRAM and HBM at the same time (i.e., use “`numactl --interleave=0-7`”), their performance is worse than Cilk-H and LAWS-H according to our measurement.

We implement all the schedulers mentioned above by modifying the original Cilk scheduler while Cilk programs run without any modification. In the traffic balancer of BATS, we randomly select 4 pages from each VMA to calculate its hotness with low overhead.

We use memory-bound benchmarks in Table 2 to evaluate the performance of BATS. Besides the benchmarks that have regular grid-based access pattern, we further implement *GE*, *PR*, *LP* and *CC* that have trajectory-based and graph-based data access patterns to evaluate the performance of BATS for applications with uneven data accesses. All the benchmarks have 200 iterations and are compiled with “GCC-5.4.0" and “-O3” option, which involves auto-vectorization.

9.2 Performance of BATS

Figure 11(a) shows the performance of all the benchmarks in Cilk-C/H, LAWS, LAWS-C/H, and BATS, normalized to their performance in Cilk. In the experiment, for *HEAT*, *SOR*, *CONV* and *STEN*,

Table 2. Benchmarks used in the experiments

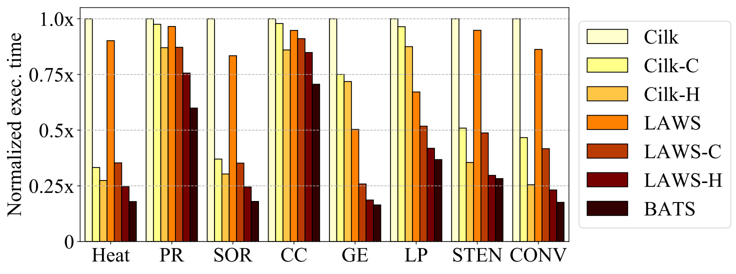
Benchmark	Access pattern	Description	Benchmark's source
SOR	Grid-based	Successive over relaxation [28]	LAWS [9]
HEAT	Grid-based	2D heat distribution	Cilk's example [42]
CONV	Grid-based	Convolution filter	LAWS [9]
STEN	Grid-based	2D 9-point stencil computing	LAWS [9]
GE	Trajectory	Gaussian elimination algorithm [39]	LAWS [9]
PR	Graph-based	Page rank algorithm	Ported from Spark GraphX [50]
LP	Graph-based	Label propagation algorithm in ML	Ported from Spark GraphX [50]
CC	Graph-based	Identifying connected components	Ported from Spark GraphX [50]

we use a $224K \times 4K$ matrix as their input data and the input size is approximately 16GB. For *GE*, the input data used is a $48K \times 48K$ matrix due to the algorithm constraint which has about 10GB. For *PR*, the input graph has 16 million nodes and the edge count of different nodes ranges from 1 to 10. For both *LP* and *CC*, the input graph has 8 million nodes and the edge count of different nodes range from 2 to 20. For these graph algorithms, the input size is approximately 16GB. Since *numactl* cannot allocate memory larger than HBM capacity, we choose 16GB as input size in this subsection, and we will discuss the performance of benchmarks with larger input size in Section 9.3.

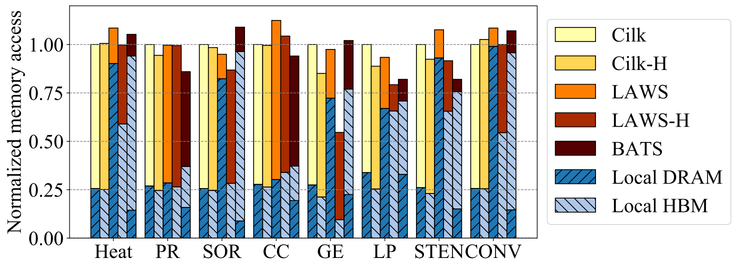
As Figure 11(a) shows, BATS significantly improves the performance of all benchmarks compared to Cilk, Cilk-C, Cilk-H, LAWS, LAWS-C, and LAWS-H. Because HBM node has higher bandwidth than DRAM node, using HBM as last level cache (Cilk-C and LAWS-C) or main memory (Cilk-H and LAWS-H) always perform better than their counterparts that only use DRAM nodes. Plus, we observe that Cilk-H performs better than Cilk-C for some applications and this phenomenon also exists for LAWS-C and LAWS-H. Such observations suggest that there are performance benefits of explicitly managing the HBM nodes but existing schedulers fail to leverage. BATS successfully takes advantage of such optimization potential with principled design and outperforms both Cilk-H and LAWS-H. In summary, BATS reduces execution time of the benchmarks ranging from 29.3% to 83.5% compared with Cilk, from 27.8% to 78% compared with Cilk-C, from 20.3% to 57.9% compared with Cilk-H, from 25.5% to 80.1% compared with LAWS, from 22.4% to 57.7% compared with LAWS-C, and from 8.6% to 27.2% compared with LAWS-H.

We then explain why BATS outperforms all the other schedulers by measuring each benchmark's local memory access ratio (including DRAM and HBM nodes). Figure 11(b) shows the results. In the figure, the bars show the overall memory accesses when scheduling the benchmarks with Cilk, Cilk-H, LAWS, LAWS-H, and BATS normalized to the number of their memory accesses with Cilk. In each bar, the slash-filled parts represent local DRAM accesses and local HBM accesses respectively while the rest part (i.e., the unfilled part) represents the remote accesses that include remote DRAM accesses and HBM accesses. We only compare BATS with Cilk, Cilk-H, LAWS, and LAWS-H in Figure 11(b), because the interconnect is configured in ALL2ALL mode to support Cilk-C and LAWS-C. In ALL2ALL mode, there is only one NUMA node, thus no remote memory accesses will be reported.

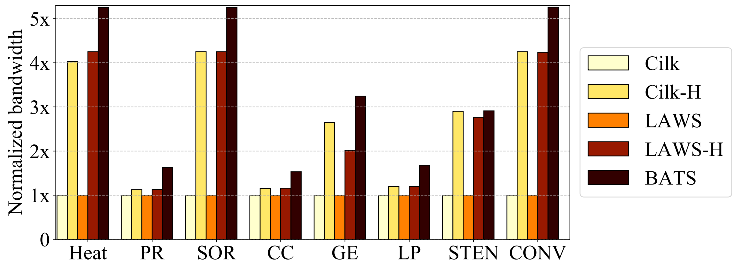
Observed from Figure 11(b), both Cilk and Cilk-H involve a significant portion of remote memory accesses. Although LAWS and LAWS-H reduce remote memory accesses, they suffer from imbalanced access traffic. For example, LAWS directs most of the local accesses to DRAM nodes while LAWS-H directs most of the local accesses to HBM nodes. On the contrary, BATS not only reduces remote accesses, but also balances the access traffic between DRAM and HBM nodes. For instance, with BATS, more than 90% memory accesses in *HEAT* are local accesses, and the number of accesses to HBM nodes is around 4× of the counterpart to DRAM nodes. The access traffic is



(a) Normalized execution time (smaller is better)



(b) Normalized overall memory accesses, local DRAM accesses, local HBM accesses



(c) Normalized memory bandwidth usage

Fig. 11. The normalized performance, the corresponding memory accesses and memory bandwidth usage.

balanced across memory nodes proportional to their bandwidth. Other benchmarks (except *PR* and *CC*) have similar effects. *PR* and *CC* have relatively large number of remote accesses because most tasks in all the sockets need to access the hot data from remote nodes (BATS still improves data locality for *PR* and *CC* compared with all the other schedulers).

For programs with regular blocked data accesses, BATS is able to improve data locality because it identifies the very socket for each intra-socket subgraph in which it can find most of the required data in the local DRAM or HBM nodes through online profiling, and schedules intra-socket subgraphs accordingly. For programs that have uneven data access hotness, BATS improves their data locality because it properly distributes hot data and schedules tasks to the sockets where they can access most data from local DRAM or HBM nodes.

Figure 11(c) shows the normalized memory bandwidth usage of different schedulers normalized to the Cilk case. Because Cilk-C and LAWS-C use HBM as the last level cache, we cannot measure their real bandwidth usage and therefore do not include them in this comparison. From the figure, we observe that BATS always brings the highest memory bandwidth usage. On the contrary, Cilk, Cilk-H, LAWS, and LAWS-H use either DRAM nodes or HBM nodes, failing to simultaneously leverage the bandwidth of both nodes. The high bandwidth usage and the improved data locality together result in the good performance of BATS.

From the above observations, we summarize why BATS performs the best among all schedulers. First, BATS reduces remote memory accesses by improving the data locality. BATS uses online profiling to identify the very socket for each intra-socket subgraph in which it can find most of the required data in the local DRAM or HBM nodes, and then schedules the intra-socket subgraphs accordingly. In the mean time, BATS identifies hot data and performs proper migration to balance access traffic. Lacking of online profiling, LAWS and LAWS-H cannot improve the data locality as much as BATS does. In addition, they also suffer from low bandwidth usage resulted from the bandwidth-asymmetric-unaware data distribution.

BATS is efficient when it is able to precisely monitor the access hotness of different VMAs. To show the accuracy of monitoring, for each benchmark, Table 3 further presents its dataset size, the number of data pages, the number of "logic" VMAs, the number of actual VMAs, the number of sampled pages, and the number of test loops performed in the first iteration. Observed from the figure, BATS identifies all the data pages. For instance, for *HEAT*, the pages are able to store data of size $4.49M \times 4K = 17.96GB$, which is larger than its dataset size (16GB). In addition, the number of VMAs is small, because multiple adjacent "logic" VMAs are merged as described in Section 8. Furthermore, BATS monitors more than 4 pages in each "logic" VMA (the number of sampled pages is much larger than the number of "logic" VMAs). Also, for all the benchmarks, BATS tests whether the sampled pages are accessed by more than 10 times (except *SOR*) in the first iteration, thus is able to tell the hotness of the VMAs.

Table 3. Statistics when performing hotness detection in BATS for all the benchmarks.

Benchmark	Dataset	Pages	"Logic" VMAs	Actual VMAs	Samples	Loops
HEAT	16GB	4.49M	8769	1368	35100	17
PR	16GB	4.23M	8261	1542	33205	32
SOR	16GB	4.89M	9562	1269	38353	5
CC	16GB	4.18M	8166	1125	32678	30
GE	10GB	2.81M	5498	1206	21969	32
LP	16GB	4.35M	8498	1119	33983	10
STEN	16GB	4.69M	8507	1364	36715	17
CONV	16GB	4.53M	8853	1316	35484	16

In the following experiments, due to the space limit, we only present the result for Cilk-H and LAWS-H because they always perform better than the default Cilk/LAWS.

9.3 Scalability of BATS

To evaluate the scalability of BATS, we compare the performance of the benchmarks with different input data sizes in Cilk-C/H, LAWS-C/H and BATS. For the benchmarks using an $x \times y$ 2D matrix as the input (*HEAT*, *SOR*, *CONV*, *STEN*), we fix $y = 4K$ for all the input 2D matrices and only adjust the x of the matrices in the experiment. For *PR* and *CC*, we adjust the number of nodes in the input graph without changing the number of edges of each node. In this way, we can measure the scalability of BATS without the impact of the leaf task granularity. We use *HEAT* and *PR* as the representative benchmarks with both even and uneven data access patterns. These two benchmarks' data size is up to approximately 22GB that is above the HBM capacity.

Figure 12 shows the performance of *HEAT* and *PR* with different input data sizes in Cilk-H, LAWS-H and BATS. In the figure, the x-axis represents dataset size of *HEAT* and *PR*. When the input data is small (e.g., 2GB), BATS reduces the execution time of *HEAT* by 42.9%-47.3% compared to

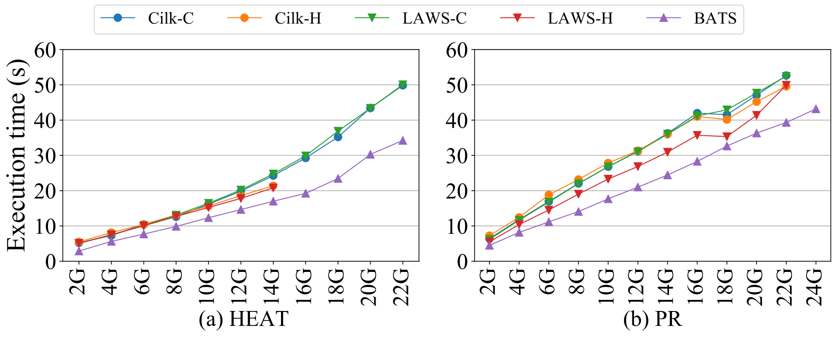


Fig. 12. The performance of *HEAT* and *PR* with different input data sizes.

Cilk-C, Cilk-H, LAWS-C and LAWS-H. When the input size is close to the capacity of HBM memory (i.e., 16GB), BATS reduces its execution time by 18.2%-29.9%. When the dataset size exceeds the capacity of HBM memory, BATS reduces its execution time by 30.3%-34.2%. Note that, Cilk-H and LAWS-H fail to run in this case owing to the HBM capacity limitation. On the contrary, if the size of data to be allocated to HBM memory is larger than HBM capacity, BATS re-allocates the extra data to DRAM. When the input size is small (e.g., 2GB), BATS reduces *PR*'s execution time by 18.2%-38.1% compared to other schedulers. When the input size is large (e.g., 24GB), BATS reduces *PR*'s execution time by 20.7-25.3%.

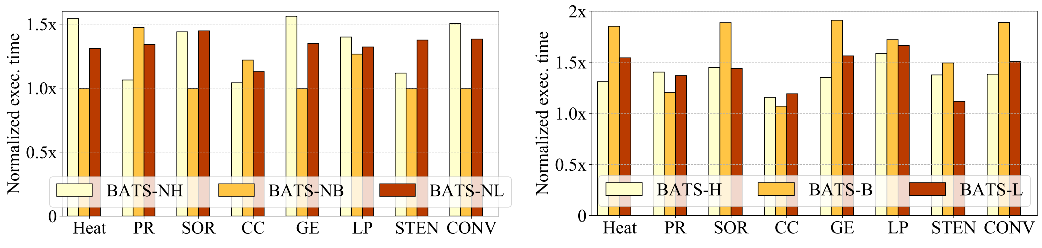
Observed from Figure 12, the execution time of the benchmarks in Cilk-H, Cilk-C, LAWS-H, LAWS-C and BATS increases linearly with the increasing of their input data sizes. For all the input data sizes, BATS can always reduce the execution time of memory-bound applications. Especially, for *PR* that has uneven data access hotness, its execution time increases much slower in BATS than other systems. The larger the input data is, the better BATS performs.

In summary, BATS is scalable in scheduling memory-bound program no matter its dataset has even or uneven access hotness.

9.4 Effectiveness of the Components

To evaluate the effectiveness of BATS components including the HBM-aware data allocator, the bandwidth-aware traffic balancer, and the hierarchical task-stealing scheduler, we implemented six BATS variants, *BATS-NH*, *BATS-NB*, *BATS-NL*, which disable one component accordingly, and *BATS-H*, *BATS-B*, *BATS-L*, which enable one component accordingly. In BATS-NH, the dataset is evenly distributed to all the nodes while the traffic balancer and hierarchical task-stealing scheduler are still active. In BATS-NB, the data is not migrated according to the memory traffics of different memory nodes. In BATS-NL, tasks are scheduled with the random task-stealing policy. In BATS-H, the data set is distributed to all the memory nodes according to their bandwidths while bandwidth-aware traffic balancer and hierarchical task-stealing scheduler are disabled. In BATS-B, only bandwidth-aware traffic balancer is active while dataset is evenly distributed and no locality improvement. In BATS-L, hierarchical task-stealing scheduler is enabled while the HBM-aware data allocator and the bandwidth-aware traffic balancer are disabled. Figure 13 shows the performance of all the benchmarks with the six variants normalized to their performance in BATS.

Figure 13(a) shows that BATS-NH performs much worse than BATS. In BATS-NH, because the data is evenly distributed, the memory bandwidth utilization of DRAM nodes and HBM nodes are not balanced. As a result, DRAM nodes are overloaded while the HBM nodes are under-utilized. Even if bandwidth-aware traffic balancer mitigates the imbalance later through migrating data, the migration overhead still results in the low performance. Observed from Figure 13(b), BATS-H performs better when executing benchmarks that have even data access hotness (e.g., *HEAT*) than



(a) Variants that disable one component (b) Variants that enable one component

Fig. 13. The normalized execution time of all the benchmarks in six BATS variants.

benchmarks that have uneven data access hotness (e.g., *PR*). This is mainly because the HBM-aware data allocation in BATS-H has already balance the data access traffic across all the memory nodes for benchmarks that have even data access hotness. Therefore, the HBM-aware data allocator is effective in improving memory bandwidth usage.

For benchmarks that have even data access hotness (e.g., *HEAT*), BATS-NB performs similarly to BATS. This is because the weighted data distribution already balances the memory traffics for these benchmarks. On the contrary, for the benchmarks in which different parts of the dataset have different hotness (e.g., *PR*), BATS-NB performs much worse than BATS. For these benchmarks, the memory traffic is not balanced only with the weighted data distribution, and the traffic imbalance results in the poor performance of BATS-NB. Meanwhile, BATS-B performs poor for all the benchmarks. It means that we still need the hierarchical task-stealing to distribute tasks to improve data locality, even if the memory traffic is balanced. The traffic balancer is effective in balancing the memory traffics across memory nodes with different bandwidths.

In addition, BATS-NL performs worse than BATS for all the benchmarks, because the benchmarks suffer from serious remote memory accesses due to the random task-stealing in BATS-NL. The large number of remote memory accesses results in severe performance degradation. BATS-L also performs much worse than BATS because it suffers from poor bandwidth utilization. Therefore, the hierarchical task-stealing scheduler is effective in scheduling the tasks to the sockets where they can find most of their data from the local memory nodes.

In summary, all the three components: the HBM-aware data allocator, the bandwidth-aware traffic balancer, and the hierarchical task-stealing scheduler are necessary and effective for BATS to achieve good performance in computers with bandwidth-asymmetric memories.

9.5 Comparison of Configuration Modes

To evaluate the performance of traditional task-stealing when the hardware runs in other modes, we show the performance of memory-bound benchmarks when the KNL server is configured in ALL2ALL-cache mode, ALL2ALL-flat mode, QUADRANT-cache mode, QUADRANT-flat mode, SNC4-cache mode and SNC4-flat mode. "*X*-*Y*" mode means that the interconnect is configured in *X* mode while the bandwidth-asymmetric memory is configured in *Y* mode.

In this experiment, if the memory is in flat mode, we use "numactl" to allocate all the data of benchmarks in HBM nodes. if the memory is in cache mode, KNL could directly use the high-bandwidth memory as the last-level cache. In this experiment, all the results are normalized to the performance of Cilk in ALL2ALL-cache mode (i.e., the performance of Cilk-C in Section 9.2).

Observed from Figure 14, ALL2ALL-cache mode and QUADRANT-cache mode have similar performance, ALL2ALL-flat and QUADRANT-flat have similar performance. The reason is that

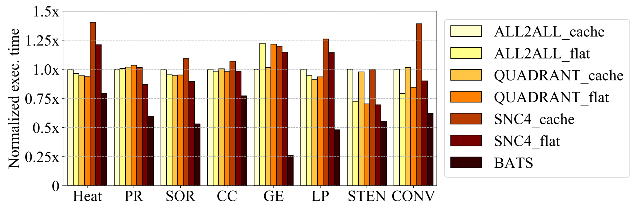


Fig. 14. The performance of the benchmarks in different hardware configurations with Cilk.

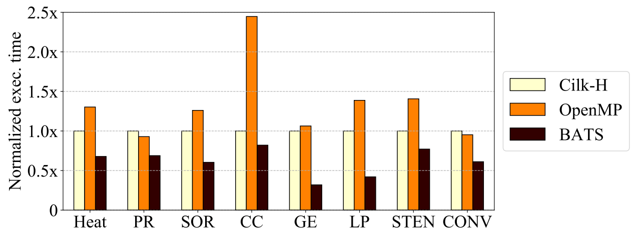


Fig. 15. The performance of all the benchmarks in Cilk-H, OpenMP and BATS.

the two modes have the same architecture from the system level which is single-socket multi-core architecture. Although they have slight difference inside the memory configuration, the performance difference of most memory-bound benchmarks on these configurations is negligible, such as Heat and Pagerank. Besides, SNC4 mode has more fluctuant performance compared with other configurations. We can also find that the benchmarks perform slightly better in SNC4-flat mode than in SNC-cache mode. That is because SNC4 has four-socket multi-core architecture and it has the whole HBM as the last level cache, which is opaque for the task. As we discussed before, it caused serious remote accesses.

Most importantly, by configuring hardware in SNC4-flat mode and using BATS to schedule the tasks, BATS reduces the execution time of memory-bound benchmarks ranging from 20% to 70% compared with all the other traditional task-stealing policy and hardware configuration combinations. According to Section 9.2, higher bandwidth usage and better memory locality are the two main reasons that BATS has better performance than other systems. This experiment further illustrates the effectiveness of BATS.

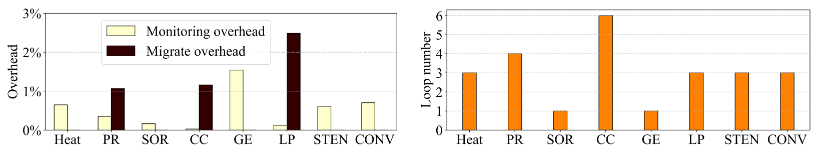
9.6 Comparing with OpenMP

We compare BATS with OpenMP [3] in executing memory-bound task-based programs. To achieve this purpose, we port all the benchmarks in Table 2 to OpenMP (Version 4.0) by simply using OpenMP loop function. In each OpenMP benchmark, similar to Cilk-H, we force that all the data is stored in HBM nodes to utilize the high memory bandwidth. OpenMP adopts static scheduling to balance workload and relies on first touch page allocation policy to enhance data locality.

Figure 15 shows the performance of all the benchmarks in Cilk-H, OpenMP, and BATS. OpenMP performs slightly worse than Cilk-H for most benchmarks, and performs much worse than BATS for all the benchmarks. OpenMP performs bad because it is not able to fully utilize the bandwidth of all the memory nodes and it does not perceive the hotness of different data chunks.

9.7 Overhead Analysis

As we described in Section 6 and Section 8, the hotness detection and the page migration when the imbalance is identified incur extra runtime overhead. The hotness detection incurs extra overhead in the first iteration because four virtual cores are used to perform the detection instead of task



(a) Overhead (b) Compensation

Fig. 16. Overhead of BATS and the number of iterations needed to compensate the overhead.

processing. Figure 16(a) shows the percentage of execution time of the benchmarks consumed by the hotness detection and page migration. Figure 16(b) shows the number of required iterations to compensate the extra overhead caused by the hotness detection and page migration.

Observed from the figure, for all the benchmarks, less than 1.5% of the execution time is used on the hotness monitoring and less than 2.5% of the overall execution time is incurred by the page migration. In order to compensate the extra overhead introduced by BATS, the benchmarks need to run for at most 6 iterations, while scientific applications often run for thousands of iterations.

The hierarchical task-stealing scheduler in BATS needs extra memory space to store hotness related data of every intra-socket subgraph. The data includes local memory access number, remote memory access number, and four flags. The first flag designates the task mainly accesses DRMA or HBM. The second flag determines whether the task already find the best-locality position. The third flag indicates the number that the task has been placed on the different socket. The fourth flag shows the best-locality position that the task has in previous attempts. The extra memory space needed by each intra-socket subgraph is only $8 + 8 + 4 \times 2 = 24$ bytes. For instance, *HEAT* has 2048 intra-socket subgraphs thus requires 48KB extra memory space, while the size of its dataset is 16GB. Therefore, the extra spatial overhead caused by BATS is negligible.

10 CONCLUSION AND FUTURE WORK

Task-based programs scheduled with traditional task-stealing schedulers are not able to utilize the HBM nodes efficiently and suffer from poor data locality in bandwidth-asymmetric memory architectures. To solve the two problems, we have proposed BATS, which consists of an HBM-aware data allocator, a bandwidth-aware traffic balancer, and a hierarchical task-stealing scheduler. The data allocator automatically distributes the data set of a task-based program to the DRAM nodes and HBM nodes according to their bandwidths. The traffic balancer migrates hot data to balance the memory traffics across different memory nodes. The hierarchical task-stealing scheduler schedules tasks to the socket where they can find their data from local memory nodes. Our experiment demonstrates that BATS can achieve up to 83.5% execution time reduction for memory-bound programs compared with traditional task-stealing schedulers.

In future, while we have new counters like the memory access number of every page, BATS could avoid the hotness monitoring module. We can utilize new counters and the memory addresses transformation to collect the memory access data which is more efficient and involves less overhead. Besides, we will extend BATS to improve the performance of memory-bound applications on computers that adopt heterogeneous memory architecture (such as DRAM + Non-Volatile Memory). Because Non-Volatile Memory has much higher latency than DRAM, BATS should consider the impact of data access latency in advance.

ACKNOWLEDGMENTS

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004802), the National Natural Science Foundation of China (NSFC) (61602301, 61632017).

REFERENCES

[1] 2017. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE* 23, 2 (2011), 187–198.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. 2009. The Design of OpenMP tasks. *IEEE TPDS* 20, 3 (2009), 404–418.

[4] Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra, Igor Peshansky, and Vivek Sarkar. 2006. Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities. In *Workshop on Programming Models for Ubiquitous Parallelism*. Citeseer.

[5] Robert D. Blumofe. 1995. *Executing Multithreaded Programs Efficiently*. Ph.D. Dissertation. MIT.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *JPDC* 37, 1 (1996), 55–69.

[7] Quan Chen, Yawen Chen, Zhiyi Huang, and Minyi Guo. 2012. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures. In *IPDPS*. 249–260.

[8] Quan Chen and Minyi Guo. 2014. Adaptive Workload Aware Task Scheduling for Single-ISA Multi-core Architectures. *ACM TACO* 11, 1 (2014).

[9] Quan Chen, Minyi Guo, and Haibing Guan. 2014. LAWS: Locality-Aware Work-Stealing for Multi-socket Multi-core Architectures. In *ICS*. 3–12.

[10] Quan Chen, Minyi Guo, and Zhiyi Huang. 2012. CATS: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures. In *ICS*. 163–172.

[11] Quan Chen, Zhiyi Huang, Minyi Guo, and Jingyu Zhou. 2011. CAB: Cache-Aware Bi-tier Task-stealing in Multi-socket Multi-core Architecture. In *ICPP*. 722–732.

[12] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *MemSys*. ACM, 268–280.

[13] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. 2014. Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures. In *EuroPar*. Springer, 596–607.

[14] 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 *ASPLOS*. ACM, 381–394.

[15] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *PACT*. ACM, 125–137.

[16] Thierry Gautier, Joao V.F. Lima, Nicolas Maillard, and Bruno Raffin. 2013. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *IPDPS*. 1299–1308.

[17] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A Scalable Locality-Aware Adaptive Work-Stealing Scheduler. In *IPDPS*. 1–12.

[18] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[19] Laxmikant V Kale and Sanjeev Krishnan. 1993. *CHARM++: A Portable Concurrent Object Oriented System based on C++*. ACM.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, 75.

[21] J.K. Lee and J. Palsberg. 2010. Featherweight X10: A Core Calculus for Async-finish Parallelism. In *PPoPP*. 25–36.

[22] C.E. Leiserson. 2009. The Cilk++ Concurrency Platform. In *DAC*. 522–527.

[23] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I Lee, Chenyang Lu, Kathryn S McKinley, et al. 2016. Work Stealing for Interactive Services to Meet Target Latency. In *PPoPP*. ACM, 14.

[24] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2013. Steal Tree: Low-overhead Tracing of Work Stealing Schedulers. In *PLDI*. 507–518.

[25] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. 2014. Optimizing Data Locality for Fork/Join Programs using Constrained Work Stealing. In *SC*. 857–868.

[26] Weiwei Lin, Chen Liang, James Z Wang, and Rajkumar Buyya. 2014. Bandwidth-Aware Divisible Task Scheduling for Cloud Computing. *Software: Practice and Experience* 44, 2 (2014), 163–174.

[27] Lei Liu, Hao Yang, Yong Li, Mengyao Xie, Lian Li, and Chenggang Wu. 2016. Memos: A Full Hierarchy Hybrid Memory Management Framework. In *ICCD*. IEEE, 368–371.

[28] Olvi L Mangasarian and David R Musicant. 1999. Successive Over-Relaxation for Support Vector Machines. *IEEE Transactions on Neural Networks* 10, 5 (1999), 1032–1037.

[29] Adam Morrison and Yehuda Afek. 2014. Fence-free Work Stealing on Bounded TSO processors. In *ASPLOS*. 413–426.

[30] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *International Journal of High Performance Computing Applications* 26, 2 (2012), 110–124.

[31] Jeeva Paudel and José Nelson Amaral. 2015. Hybrid Parallel Task Placement in Irregular Applications. *JPDC* 76 (2015), 94–105.

[32] Laércio L Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Abhinav Bhatele, Philippe OA Navaux, Jean-François Méhaut, Laxmikant V Kalé, et al. 2011. Improving Parallel System Performance with a NUMA-aware Load Balancer. *TR-JLPC-11-02* (2011).

[33] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical Task-Based Programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3 (2009), 284–299.

[34] Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical Work-Stealing. In *EuroPar*. 217–229.

[35] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-study with Xeon Phi KNL. In *IPDPS*. IEEE, 297–306.

[36] J. Reinders. 2007. *Intel Threading Building Blocks*. Intel.

[37] Research and markets. [n. d.]. Hybrid Memory Cube (HMC) and High-bandwidth Memory (HBM) Market by Memory Type (HMC and HBM), Product type (GPU, CPU, APU, FPGA, ASIC), Application, and Geography-Global Forecast to 2023.

[38] Haris Ribic and David Yu. 2014. Energy-Efficient Work-Stealing Language Runtimes. In *ASPLOS*. 513–528.

[39] Robert Schreiber. 1982. A New Implementation of Sparse Gaussian Elimination. *ACM Trans. Math. Software* 8, 3 (1982), 256–276.

[40] Mohammed Shaheen and Robert Strzodka. 2012. NUMA Aware Iterative Stencil Computations on Many-Core Systems. In *IPDPS*. 461–473.

[41] Srinath Sridharan, Gagan Gupta, and Gurindar S Sohi. 2013. Holistic Run-time Parallelism Management for Time and Energy Efficiency. In *ICS*. 337–348.

[42] Supercomputing Technologies Group, MIT 2001. *Cilk 5.4.6 Reference Manual*. Supercomputing Technologies Group, MIT. http://supertech.lcs.mit.edu/cilk/manual-5.4.6.pdf

[43] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2007. Building Portable Thread Schedulers for Hierarchical Multiprocessors: The Bubblesched Framework. In *EuroPar*. Springer, 42–51.

[44] Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-Aware Work-Stealing Runtimes. In *ISCA*. 40–52.

[45] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined Cache Hierarchies. In *ISCA*. ACM, 652–665.

[46] B Vikranth, Rajeev Wankar, and C Raghavendra Rao. 2013. Topology Aware Task stealing for on-Chip NUMA Multi-Core Processors. *Procedia Computer Science* 18 (2013), 379–388.

[47] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. 2016. Using Data Dependencies to Improve Task-based Scheduling Strategies on NUMA Architectures. In *EuroPar*. Springer, 531–544.

[48] Andrey Vladimirov and Ryo Asai. 2016. *Clustering Modes in Knights Landing Processors: Developer's Guide*. Technical Report. Colfax International.

[49] Mengyao Xie, Lei Liu, Hao Yang, Chenggang Wu, and Hongna Geng. 2017. SysMon: Monitoring Memory Behaviors via OS Approach. In *International Workshop on Advanced Parallel Processing Technologies*. Springer, 51–63.

[50] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

[51] Di Xu, Chenggang Wu, and Pen-Chung Yew. 2010. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. In *PACT*. ACM, 237–248.

[52] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 172–187.

[53] Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *SPAA*. 315–325.

[54] Doe Hyun Yoon, Min Kyu Jeong, Michael Sullivan, and Mattan Erez. 2012. The Dynamic Granularity Memory System. In *ISCA*. 548–559.

Received June 2018; revised Sep. 2018; accepted Oct. 2018