Merchandiser: Data Placement on Persistent Memory-based Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness

Zhen Xie
zhen.xie@anl.gov
University of California, Merced
Argonne National Laboratory

Dong Li
dli35@ucmerced.edu
University of California, Merced

1 INTRODUCTION

Many high-performance computing (HPC) applications are becoming memory-consuming. For example, the density matrix renormalization group (DMRG), a numerical algorithm to obtain the low-energy physics of quantum many-body systems, can consume 1.271 TB memory in a single machine when solving the Hubbard 2D model at the scale of 320×320. To meet memory requirements of those applications, the big memory system is emerging. An example of such a system is the Intel Optane persistent memory module (PMM)-based machine where there can be up to 12 TB memory in a single two-socket machine. Another example of such a system is the Amazon EC2 High Memory Instance built upon eight NUMA nodes and providing up to 12 TB memory. The big memory system is often heterogeneous, which means multiple memory components with different latency and bandwidth form the main memory. In the example of Intel PMM, DRAM and persistent memory (PM) form heterogeneous memory (HM). DRAM is faster (in terms of bandwidth and latency) but costly and smaller, while PM is slower but cheaper and larger. HM provides a cost-effective solution to significantly increase memory capacity.

HM raises a data placement problem. Because of small capacity of fast memory and relatively worse performance of slow memory, memory pages have to be allocated and migrated between fast and slow memories, such that most of memory accesses can happen in fast memory for high performance. It has been shown that some HPC applications can suffer from up to 5.7× performance loss (compared with using a fast memory-only solution) with suboptimal data placement on HM.

Many solutions to address the data placement problem on HM uses a profiling-guided optimization (PGO) approach. These solutions identify frequently accessed memory pages ("hot pages") by periodically sampling memory pages and tracking memory accesses to them. Hot pages are then migrated to fast memory for better performance. These solutions are application-agnostic, meaning that they do not need application knowledge or change applications. Success of these solutions is based on an implicit assumption that placing hot pages in fast memory always leads to better performance. However, we find that it is not true for many task-parallel HPC applications.

Task-parallel programs are common in HPC. A task-parallel program can be MPI-based, and each MPI process performs a task. It can be OpenMP-based, and each OpenMP thread performs a task. There is synchronization among tasks where tasks must reach the synchronization point before they move on to the rest of computation. Because of synchronization among tasks, finishing *all* tasks fast instead of finish individual tasks fast is a key for high performance.

The PGO on HM cannot work well for task-parallel applications. They lack a view of "finishing all tasks fast" for high performance. They migrate and place hot pages into fast memory, but do not consider which task accesses those memory pages. As a result, the existing efforts could introduce load imbalance: a task unnecessarily reaches the synchronization point earlier than the others and waits for other tasks to finish, because many pages of this task are resident in fast memory, leading to its shorter execution time.

To reveal the load imbalance problem on HM, we study five HPC applications on an Optane-based HM. This HM consists of 192GB DRAM and 1.5TB Optane. We study two representative solutions: an industry-quality, software solution (Intel MemoryOptimizer) and a hardware solution (Memory Mode of Optane). We have two observations.

- Compared with running on homogeneous memory, running on HM increases performance difference among tasks: on average, the performance difference among tasks is increased by 17% and 16% (when MemoryOptimizer and Memory Mode are used respectively), which indicates more load imbalance after using MemoryOptimizer and Memory Mode on HM.
- Performance improvement is minimal after using MemoryOptimizer and Memory Mode. The performance improvement is only 4.32% and 3.71% respectively (compared with using Optane only), because the overall performance is hindered by the slowest task.

There are two fundamental reasons accounting for the above performance problem. *First*, the PGO solutions (such as MemoryOptimizer) are not aware of task parallelism. There is a lack of coordination among tasks to share the limited fast memory space. That space is allocated to tasks based on opportunistic detection of hot pages from tasks, not based on performance analysis on potential performance benefit of using fast memory for tasks. It may unfairly place too many pages from one task into fast memory, causing load imbalance. *Second*, the PGO solutions use random page sampling-based memory profiling. Random sampling is effective to avoid large overhead of profiling all memory pages in a big memory system. However, it may collect many memory accesses from one task, which leads to too many pages of that task migrating to fast memory, causing load imbalance.

We introduce a load balance-aware data placement system for HM, named *Merchandiser*, to address the problem. Merchandiser introduces task semantics during memory profiling. This means

★The original version [1] of this paper was accepted in the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23).

Merchandiser associates memory accesses with tasks during profiling, instead of being application-agnostic. Using limited task semantics, Merchandiser effectively sets up coordination among tasks on the usage of HM. Furthermore, Merchandiser uses historical, fine-grained profiling results of the task to guide data placement for the subsequent executions of the same task with new inputs.

However, to realize Merchandiser we face two challenges. *First*, the input problem to a task during program execution can vary, and the historical profiling results collected from one input cannot be directly used to predict performance for another input, because of the difference in the number of memory accesses. *Second*, how to partition the fast memory space among tasks is challenging. Unless all tasks have the same memory access patterns and data object sizes, evenly sharing fast memory among tasks cannot work. We must decide for each task with a new input, which objects should be placed in fast memory without priori knowledge on the number of memory accesses to the objects. We must also predict execution time of tasks after migration, such that the effectiveness of load balance can be quantified and estimated.

To address the first challenge on handling new input problems, Merchandiser classifies data objects in terms of their memory access patterns, based on which Merchandiser analytically derives the number of main memory accesses for a new input problem. The memory access patterns are mostly invariant across input problems for a given task in many HPC applications, providing a reliable indication on the number of memory accesses. We also recognize the difference in the impacts of memory access patterns, and estimate the number of memory accesses differently for different patterns.

Based on the estimated memory accesses, Merchandiser introduces performance modeling to predict execution time of the task when a certain portion of memory accesses happens in fast memory while the remaining memory accesses happen in slow memory. The novelty of our performance modeling lies in the modeling of performance correlation between different data placements of the task. In particular, performance modeling takes the performance of a data placement as input, and then predicts the performance of another data placement. The performance modeling sets up a correlation between the above two performances. The task characteristics are represented and quantified using a few performance events collected from only one execution of a specific data placement.

To address the second challenge on deciding which pages should be migrated to fast memory for parallel tasks, we introduce a greedy heuristic algorithm to decide how to allocate the fast memory space among tasks to maximize performance benefit of all tasks (not an individual task). The algorithm varies the portion of fast-memory accesses based on the performance modeling to find a load-balance solution.

2 OVERVIEW OF MD-HM

Merchandiser uses performance modeling to guide data placement in HM. The performance modeling uses task information as input, which includes the execution time of basic blocks in the task program and runtime performance events critical to decide the performance sensitivity of the task to data placement. The task information is collected in the first instance of the task using an input problem and used by the performance modeling to predict the performance of the same task for a new input under various data placement on HM. The performance modeling is integrated into a runtime system to decide if data migration can introduce load imbalance among tasks.

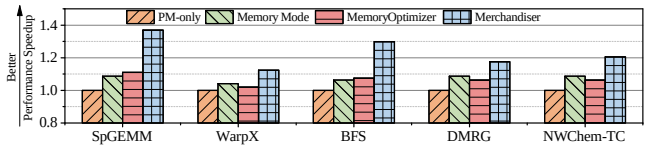

Figure 1: Performance of Memory Mode, MemoryOptimizer, and Merchandiser, compared to the PM-only execution.

To accurately predict the execution time of a task with a new input, our performance modeling first estimates the number of memory accesses to data objects with the new input. Merchandiser performs analysis on memory access patterns at the data object level through static analysis; then the estimation is made according to data object sizes, the number of memory accesses collected from the base input, and memory access patterns. Merchandiser has a runtime system using performance modeling to decide if data migration should happen or not with load-balance awareness. Before task execution, the runtime first employs a heuristic algorithm to decide how many fast memory accesses should happen for each task based on the performance modeling. Then, utilizing memory profiling mechanisms in existing solutions, Merchandiser determines if the pages corresponding to each task should be migrated from slow memory (Optane) to fast memory (DRAM).

3 EVALUATION

Platform. We evaluate Merchandiser on a two-socket server with two Intel Xeon Gold 6252N 24-core processors running Linux 5.17.0. Each socket has 12 DIMM slots: six for 16-GB DDR4 DRAM modules, and six for 128-GB Optane PMM. In total, the system has 192 GB DRAM and 1.5 TB PM. We use Memkind to manage the page placement and migration on HM.

Input problems. We use five task-parallel HPC applications: SpGEMM and BFS are derived from high-performance math libraries. WarpX is a production code for plasma simulation. DMRG comes from Itensor and simulates quantum many-body systems. NWChem-TC is the tensor contraction component in NWChem.

Figure 1 shows overall performance normalized to PM-only (Optane-only). Merchandiser introduces 23.6%, 17.1%, and 15.4% performance improvement on average (up to 37.8%, 26.0%, and 23.2%) over PM-only, Memory Mode, and MemoryOptimizer respectively. Compared with Memory Mode and MemoryOptimizer, Merchandiser reduces A.C.V (average coefficient of variation of execution time across threads/processes) by 51.6% and 42.7% on average respectively. The performance of PM only shows the load imbalance from the applications themselves. We notice that using Merchandiser, A.C.V is reduced by 39.1% and 21.4% for SpGEMM and BFS, compared with using PM-only. This indicates that Merchandiser can even remove load imbalance in applications themselves.

REFERENCES

[1] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. 2023. Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness. In *Proceedings of the 28th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*