LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-769889

Opera: Similarity Analysis on Data Access Patterns of OpenMP Tasks to Optimize Task Affinity

J. Ren, C. Liao, D. Li

March 18, 2019

24TH INTERNATIONAL WORKSHOP ON HIGH-LEVEL PARALLEL PROGRAMMING MODELS AND SUPPORTIVE ENVIRONMENTS
Rio De Janeiro, Brazil
May 20, 2019 through May 24, 2019

Disclaimer

Opera: Similarity Analysis on Data Access Patterns of OpenMP Tasks to Optimize Task Affinity

Jie Ren
University of California, Merced
jren6@ucmerced.edu

Chunhua Liao
Lawrence Livermore National Laboratory
liao6@llnl.gov

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

***Abstract*—OpenMP supports task-based parallelism, but task scheduling is oblivious to data locality, which leads to inconsistent performance. In this paper, we present `Opera`, an OpenMP task scheduler which leverages memory access information profiled offline to guide runtime task scheduling. The evaluation results show that `Opera` improves performance by up to 40% (21.2% on average), comparing with using three schedulers in the Nanos++ runtime library.**

I. Introduction

Nowadays task programming models are widely used in applications [1]. OpenMP task is one of the task programming models that decompose a program into a set of tasks and distribute them among processing elements. The OpenMP tasking model supports parallelization of both irregular and unstructured algorithms [2], which provide expressiveness, flexibility, and huge potential for performance and scalability. With the OpenMP tasking model, users explicitly specify independent tasks by using OpenMP task constructs. Meanwhile, the OpenMP tasking model provides flexibility in task scheduling, by allowing users to specify scheduling point, scheduling policy and whether a task is tied to thread [1]. The OpenMP runtime library takes full responsibility of scheduling the tasks for locality and load balancing.

Similar to other OpenMP constructs, task constructs may involve shared data, either through explicit data-sharing clauses or implicit inheritance rules from parent tasks. In this paper we refer task-to-data affinity as *task affinity*, which hints to execute task as close as possible to the location of data. In addition, modern high-performance computing nodes have complex memory subsystem hierarchies including private and shared caches. For example, Intel Knights Landing (KNL) many-core processors organize CPU cores into tiles. Each tile in KNL consists of two CPU cores sharing 1MB L2 cache [3]. Maximizing data reuse in shared cache among OpenMP tasks is essential for improving data locality and performance.

However, it is not trivial to enable task affinity because of the following reasons. First, collecting data access information and quantifying task affinity at runtime can cause high overhead. Second, maintaining data access information for all tasks at runtime is not memory efficient. Third, the data access information collected offline can be different from that collected at runtime. Hence, using offline analysis results to direct runtime task scheduling can be misleading.

To address the above problems, we propose an approach for the **O**penM**P** tasking model **R**untime scheduler to improve OpenMP task **A**ffinity, `Opera`. Previous efforts [4] improve the task affinity by introducing new clauses. Different from the previous efforts, `Opera` leverages offline profiling to obtain invariant data access patterns of OpenMP tasks, and introduces adaptive runtime task scheduling to optimize thread affinity using the data access patterns. `Opera` does not introduce new clauses. We summarize our contributions in this work as follows:

- We develop a PIN-based profiling tool to collect memory access information for each task. Moreover, we present an approach to analyzing the interaction between tasks and the cache hierarchy.
- We design `Opera`, a runtime task scheduler leveraging data access pattern profiled offline.
- Using the Barcelona OpenMP Tasks Suite, our evaluation results show that `Opera` improves performance by up to 40% (21.2% on average), comparing with using three schedulers in the Nanos++ runtime library.

II. Design and Implementation

To enable affinity-aware runtime OpenMP task scheduling, we introduce `Opera`, a runtime scheduler which contains two stages in its workflow. Figure 1 illustrates the two stages and major components in each stage. The first stage is an offline profiling phase, which collects cache block level memory access information for each task and calculates the similarity for tasks. The result of this stage is a task affinity table. The second state is runtime scheduling. In this stage `Opera` leverages the task affinity table generated in the first stage to map the task with the highest similarity into threads

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Work on offline analysis was supported by the U.S. Department Of Energy, Advanced Scientific Computing Research's SciDAC Program, and the results on runtime scheduling were funded through the LLNL-LDRD Program under Project No. 18-ERD-006. LLNL-CONF-769889.

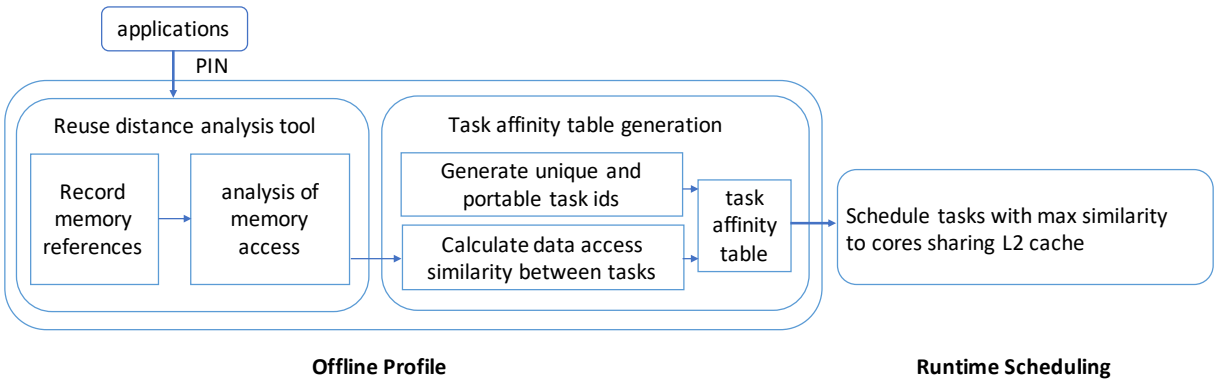

Figure 1. The general workflow of using Opera.

running on cores with shared cache. We describe the details of the design and implementation of Opera as follows.

A. *Offline Profiling*

With the task data access information, tasks can be reasonable scheduled. However, it is difficult to get such information at runtime since fine-grained cache block level data access pattern analysis can cause large overhead. Therefore we propose an offline profiling stage to alleviate runtime overhead.

The process of profiling includes (1) memory access information collection for each task, (2) task ID generation, (3) memory access similarity calculation, and (4) task affinity table generation. We explain each in details as follows.

Memory access information collection for tasks. We developed a PIN-based memory access analysis tool which can be used to record every cache block address a task accessed. The tool provides data access traces for user specified code regions. All tasks' memory access information can be recorded by running the tool once. Based on the information the tool collects, we have the knowledge about which tasks access same cache blocks.

Task ID generation. One challenge of using offline profiling to guide runtime scheduling is that we need portable and consistent task IDs between two stages. The number of tasks generated and the order in which they are generated might be different between profiling and production runs. Therefore, Opera needs to generate task IDs to ensure there is no mismatch between profiling results and runtime scheduling.

Task ID generation has two requirements: (1) The task IDs should be unique and portable; (2) The task ID generation is lightweight with only negligible performance overhead. To achieve above requirements, Opera leverages existing task information provided by Nanos++ runtime and introduces limited processing to generate the task IDs.

An OpenMP task can be uniquely identified by using two pieces of information: the code and the corresponding input data. Nanos++ runtime provides information about the call path of each task and input variables' virtual addresses and sizes. However, since the virtual addresses are not portable across different runs, Nanos++ runtime doesn't provide enough information for generating the unique and portable task IDs. Instead of directly using the virtual addresses of the input variables, Opera maps virtual addresses to variables with offsets.

Table I
DATA NEEDED FOR TASK ID GENERATION

Data	Description
var_id	The order of memory allocation of data objects at runtime
var_addr	The virtual address of the variable
var_size	The size of variable
call_path_info	Task call path information, Provided by Nanos++
input_addr	Input problem virtual address, Provided by Nanos++
input_size	Input problem size, Provided by Nanos++
offset	offset <var_size & input_addr = var_addr + offset

Table I summarizes the information used by Opera for task ID generation. Opera maintains a data object information table and only updates the table when memory allocation happens. The data object information table contains var_id, var_addr and var_size. var_id is generated based on the order of memory allocation of data objects at runtime. For each task Opera consults the OpenMP runtime to obtain $call\_path\_info$, $input\_addr$ and $input\_size$. It then combines the information from the data object to calculates $offset$. Leveraging $call\_path\_info$, var_id and $offset$ of the task, Opera can generate unique and portable IDs for all tasks.

Quantifying the similarity of memory accesses between

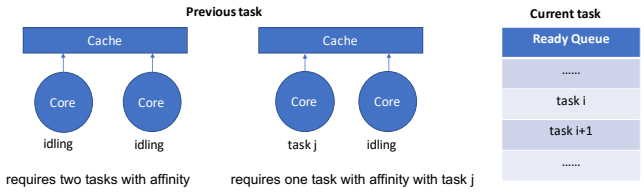

Figure 2. An example of runtime scheduling.

tasks. With the task memory access information we can evaluate the similarity between tasks in terms of memory accesses. In practices, there are several metrics to evaluate the similarity between two sets in different views. We use Jaccard similarity coefficient [5] due to its efficiency and simplicity. Jaccard similarity coefficient can be efficiently calculated as the proximity of the two data sets without the use of data redundancy. Jaccard similarity coefficient is suitable for `Opera` since the task memory footprint is relatively small, which means memory access for each task is limited. The Jaccard similarity coefficient has a floating point value between 0 and 1.0. Jaccard similarity coefficient as 1.0 stands for max similarity while 0 means there is no similarity at all.

Task affinity table generation. `Opera` generates a task affinity table to guide runtime scheduling. The table records the similarity coefficient between two tasks. To speedup the generation of the task affinity table, `Opera` only calculates the similarity coefficient for the tasks which might be executed together. Furthermore, `Opera` only analyzes the tasks' similarity coefficient for leaf tasks without child tasks in recursive applications. The second optimization is based on the observation that the tasks without child task are usually the most computation intensive tasks in recursive applications and the parent tasks are often used to generate child tasks with little memory access.

Note that the task affinity table is precomputed during the offline profiling stage to reduce runtime overhead.

B. Runtime Scheduling

After the task affinity table is generated, `Opera` can schedule tasks with the highest similarity coefficient to be executed together in order to improve the task affinity. The runtime library of OpenMP tasking model manages a task ready queue. When creating a task with no dependency or when a task becomes ready after all its dependences has been fulfilled, the task is placed in this ready queue. The scheduler picks up a task in task ready queue and assigns it to an idle thread for execution. Nanos++ runtime provides the flexibility for a scheduler to use different algorithms to pick up tasks in the task ready queue. `Opera` runtime scheduler is based on the breadth-first scheduling algorithm using a single global ready queue with FIFO scheduling.

An example. Figure 2 illustrates how `Opera` schedules tasks so two threads with good affinity can use the same shared cache. The scheduling algorithm can be easily extended for multiple threads share the same cache. Note that the total number of threads can be an arbitrary value that is greater than two. We refer the *sibling thread* of thread i as the thread which shares cache with thread i. There are two cases when `Opera` tries to assign tasks into threads as shown in figure 2: (1) two threads with share cache are both idle at the scheduling point; (2) only one thread is idle while its sibling thread is busy.

Figure 3 shows the pseudo code of the algorithm used by `Opera` runtime scheduler. `Opera` runtime scheduler periodically checks for an idle thread. We assume the thread i and thread j use the same shared cache. When an idle thread i is found, `Opera` checks if the sibling thread of the thread i (the thread j) is idle (line 20). If the thread j is also idle, `Opera` looks up the task affinity table to find the first pair of ready tasks (t1 and t2) with the max affinity value. It then assigns the two tasks into the threads i and j separately (line 26-32). If the thread j is busy, `Opera` first reads the task (t2) the thread j is working on (line 23). Then `Opera` inquires the task affinity table and gets task (t1) with the max affinity value with t2 (line 26). Finally t1 is assigned to the idle thread i (line 32).

```
1  TaskPool *readQueue;
2
3  /*Look up the task affinity table to find
4  the task with the highest similarity coefficient
5  with task task1 */
6  void  getTask(TASK* task1, TASK* task2)
7  {
8      if(task1 == NULL)
9          task1 = readQueue.getFirstWD();
10     task2= task1.getSimilarityTask();
11 }
12
13 /* Opera calls this function when it finds
14 an idle thread*/
15 void atIdle(Basicthread *thread)
16 {
17     TASK *t1=NULL, *t2=NULL;
18     Basicthread *sibling = thread.getSibling();
19     // checks if the sibling thread is idle
20     if(sibling is not idle)
21     {// find the task t2
22      // which sibling thread is working on
23         t2 = sibling.getWorkingTask();
24     }
25
26     getTask(t2, t1);
27     // assign the task into the thread
28     if(sibling is idle)
29     {
30         sibling.assign(t2);
31     }
32     thread.assign(t1);
33 }
```

Figure 3. `Opera` runtime scheduler algorithm.

C. Implementation

Target Platform. In our prototype implementation of `Opera`, we target the Knights Landing (KNL) many-core processors in Cori, a supercomputer at NERSC [6]. We focus on data reuse in the L2 shared cache in KNL and assign two tasks into threads which use the shared cache. We configure KNL's main memory as quadrant mode. This means the processor is divided into four virtual quadrants and exposed to the OS as a single NUMA domain. Therefore we exclude the NUMA effect. In a parallel region, threads are assigned to hardware threads. To ensure that the threads use the shared L2 cache, we use an OpenMP environment variable to bind to specific processors.

Nanos++ RTL APIs. `Opera` is developed as an alternative scheduler in Nanos++ runtime [7]. Leveraging the task related APIs (e.g. task assigning) provided by Nanos++, `Opera` inquires the task affinity table and assigns tasks into the threads which has better chance to reuse data in the shared cache.

III. Evaluation

A. Benchmarks

We use the OpenMP task kernels in Barcelona OpenMP Tasks Suite (BOTS) [8] to evaluate `Opera`. BOTS contains eight OpenMP task-based kernels covering various domains. Meanwhile, there are kernels with or without nested tasks. The evaluation using BOTS can make a comprehensive understanding about task affinity improvement with `Opera` for the OpenMP tasking model. Both small and medium input data sets are used for the kernels. Their memory footprints are about 1GB and 4GB respectively. Table II shows more details on the kernels we use.

Table II
Benchmark Information

Kernels	Domain	Memory footprint
Alignment	Dynamic programming	1.2GB(small), 4.7GB(medium)
FFT	Spectral method	1.5GB(small), 3GB(medium)
Floorplan	Optimization	7GB(small), 7GB(medium)
Health	Simulation	25MB(small), 4GB(medium)
NQueens	Search	3MB(small), 3MB(medium)
Sort	Integer sorting	500MB(small), 2GB(medium)
SparseLU	Sparse linear algebra	60MB(small), 120MB(medium)
Strassen	Dense linear algebra	1GB(small), 4GB(medium)

B. Experimental Environment

We use a KNL-based compute node on the Cori cluster at NERSC with the default KNL configuration. More KNL configuration details are introduced in section II-C

C. Results

We evaluate the execution time for each kernel with `Opera` scheduler and compare the result with three Nanos++ runtime existing schedulers. We summarized the features of each scheduler's algorithm as follows.

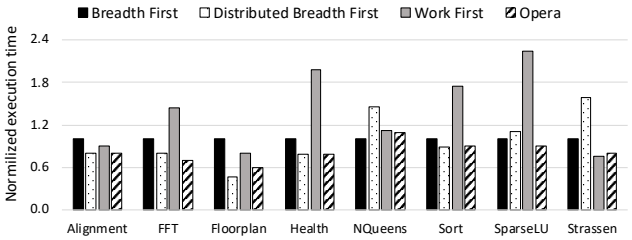

Figure 4. Normalized execution time with small input set.

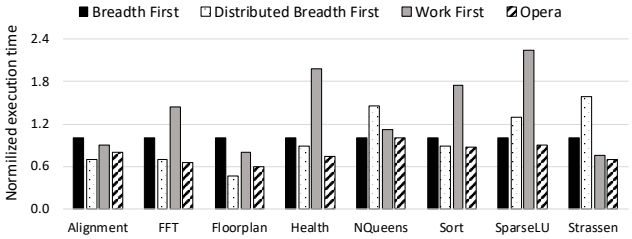

Figure 5. Normalized execution time with medium input set.

- *Breadth First scheduler.* This scheduler implements a single global task ready queue. The task ready queue is ordered following a FIFO algorithm. Breadth First scheduler is the default scheduler.
- *Distributed Breadth First scheduler.* This scheduler is implemented with multiple local thread queues. Each thread inserts its created tasks into its own ready queue following a FIFO algorithm.
- *Work First scheduler.* This scheduler is also implemented with local thread queues. Each thread inserts its created tasks into its own ready queue following a depth first algorithm.
- *`Opera` scheduler.* This scheduler implements a single global task ready queue. The pop out order for tasks in ready queues follows the task affinity relations.

Figures 4 and 5 shows the normalized execution time for each kernel with different schedulers in small and medium input set respectively. The baseline is the Breadth First scheduler. None of the scheduler provided by Nanos++ RTL considers the task affinity during the task scheduling.

The evaluation result shows that `Opera` obtains the best performance in 6 cases over total 8 cases. Compared to the default scheduler (Breath First), `Opera` reduces the average execution time by 17.5% and 21.2% for two different data set sizes, respectively. `Opera` outperforms all other schedulers for the normalized average execution time.

IV. Future Work

The current implementation of `Opera` focuses on proving the idea that considering task affinity during task scheduling can improve the runtime performance. However, the offline profiling can still be the bottleneck of the design. The offline

profiling is expensive by using PIN-based memory access analysis tool. We will explore the method to efficiently profile memory access by predicting large input problems with small input problems and bring the offline solution online.

V. Conclusion

Leveraging the memory access similarity analysis for individual tasks, we propose `Opera`, an approach combining both offline profiling and runtime task scheduling to improve data reuse on the shared cache. Our evaluation results shows that `Opera` improves performance by up to 40% (21.2% on average), comparing with using three schedulers in the Nanos++ runtime library.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distrib. Syst.*, 2009.

[2] OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0. [Online]. Available: http://www.openmp.org/

[3] Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor. [Online]. Available: https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf

[4] C. Terboven, J. Hahnfeld, X. Teruel, S. Mateo, A. Duran, M. Klemm, S. L. Olivier, and B. R. de Supinski, "Approaches for task affinity in openmp," in *OpenMP: Memory, Devices, and Tasks*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds., 2016.

[5] S.-S. Choi, S.-H. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *Journal of Systemics, Cybernetics and Informatics*, 2010.

[6] Cori Intel Xeon Phi (KNL) Nodes. [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/configuration/cori-intel-xeon-phi-nodes/

[7] Barcelona Supercomputing Center, "Nanos ++," https://pm.bsc.es/nanox.

[8] Barcelona OpenMP Tasks Suite. [Online]. Available: https://github.com/bsc-pm/bots