Optimized Execution of Parallel Loops via User-Defined Scheduling Policies

Seonmyeong Bak
Georgia Institute of Technology
sbak5@gatech.edu

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Pavan Balaji
Argonne National Laboratory
balaji@anl.gov

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

ABSTRACT

On-node parallelism continues to increase in importance for high-performance computing and most newly deployed supercomputers have tens of processor cores per node. These higher levels of on-node parallelism exacerbate the impact of load imbalance and locality in parallel computations, and current programming systems notably lack features to enable efficient use of these large numbers of cores or require users to modify codes significantly. Our work is motivated by the need to address application-specific load balance and locality requirements with minimal changes to application codes.

In this paper, we propose a new approach to extend the specification of parallel loops via user functions that specify iteration chunks. We also extend the runtime system to invoke these user functions when determining how to create chunks and schedule them on worker threads. Our runtime system starts with subspaces specified in the user functions, performs load balancing of chunks concurrently, and stores the balanced groups of chunks to reduce load imbalance in future invocations. Our approach can be used to improve load balance and locality in many dynamic iterative applications, including graph and sparse matrix applications. We demonstrate the benefits of this work using MiniMD, a miniapp derived from LAMMPS, and three kernels from the GAP Benchmark Suite: Breadth-First Search, Connected Components, and PageRank, each evaluated with six different graph data sets. Our approach achieves geometric mean speedups of 1.16× to 1.54× over four standard OpenMP schedules and 1.07× over the *static_steal* schedule from recent research.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Parallel programming languages**; **Runtime environments**;

KEYWORDS

OpenMP, Load Balancing, Loop Parallelism, User-Defined Scheduling

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337913

ACM Reference Format:
Seonmyeong Bak, Yanfei Guo, Pavan Balaji, and Vivek Sarkar. 2019. Optimized Execution of Parallel Loops via User-Defined Scheduling Policies. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/3337821.3337913

1 INTRODUCTION

For the past few years, many researchers in high-performance computing have introduced programming models that make use of increasing on-node parallelism in the modern microprocessors. Many of these researchers use hybrid programming models to enable efficient load balancing in the intra- and internode level through the interoperation of distributed- and shared-memory programming models [1, 8, 12, 34].

These programming models have introduced techniques to resolve load imbalance efficiently while maintaining locality. One such technique is chunking. Users create iteration chunks or tasks through language constructs provided by the programming models to express parallelism in their codes, choosing one of the predefined schedules that runtime systems use for scheduling. However, these chunking techniques do not resolve dynamic load imbalance and maintain locality properly on many irregular parallel applications because of variables determined at runtime. To achieve load balancing, researchers often seek a dynamic approach such as work stealing. However, this practice can lead to limited improvement or even degradation in performance.

The main variables causing the unresolved load imbalance are input datasets and parameters. These runtime variables determine the amount of load for each iteration or task. For example, the number of inner loop iterations and conditional statements incurring control divergence for each iteration or task changes depending on the variables. This variability results in load imbalance as in Figure 1, which is handled only by migrating tasks in the previous approaches dynamically or

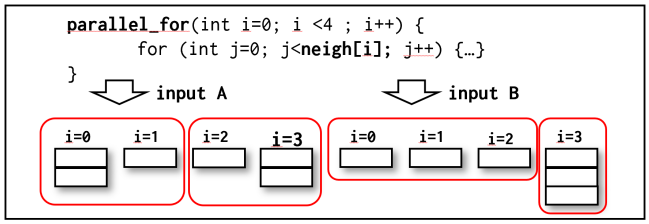

Figure 1: Imbalanced load in each iteration for different input dataset and chunking for balanced load

periodically. The migration of tasks, however, incurs traffic on interconnect as well as loses data locality, which is critical for most memory-intensive parallel applications.

We can solve this input-dependent load imbalance by user provided information because users know how input datasets and parameters determine the amount of load for each loop or task. The in-/out-degree of each vertex in graph applications is approximately correlated with the amount of load for the vertex. A sparse matrix has the same characteristic, which can be exploited to estimate the amount of load for each iteration. With this user information, we can create groups of chunks to have approximate equal amount of load as in Figure 1. The information can be obtained, however, only in runtime through inspection or offline analysis before the execution. Most programming languages do not have features to extract this information at runtime because the overhead for inspection is expected to be too high, outweighing the benefit of the inspection.

Arguably, the overhead can be minimized by reusing the dynamic information during periods where the information does not change. In many parallel applications, the same parallel kernels are used with the same data repeatedly until the applications converge to a certain threshold. Therefore, if we can create chunks that have even load size by inspection using user-provided functions, we can build groups of chunks with better load balance and reuse these groups of chunks for the next invocation of the kernels with minimal overhead. This reuse of balanced groups will improve the performance of irregular applications substantially with better initial load balance and locality from the reduced migration of tasks.

In this paper, we present a set of APIs with which a user can define user functions to inspect each iteration of parallel loops. The user functions estimate the amount of load on each iteration by using data structures or information about each iteration. After threads finish creating chunks by inspection using user functions, one of the threads balances the chunks concurrently while workers execute their created chunks. These balanced groups of chunks are stored and indexed by some unique information of the target loop. We also made the subspace selection from the iteration space of the target loop configurable by another user-provided function, which is critical for performance. Additionally, we adopted work stealing to handle transient load imbalance incurred by variables in runtime, such as Intel Turbo Boost and DVFS. We implemented our approach in OpenMP, a standard shared-memory programming model, and used the LLVM OpenMP runtime as our base runtime system. To evaluate our work, we used the Breadth-First Search (BFS), Connected Components (CC), and PageRank (PR) kernels from the GAP Benchmark Suite with six real graphs as well as MiniMD, a miniapp version of LAMMPS in the Mantevo suite. The contributions of this paper are as follows:

- Introduction of APIs to enable user-defined scheduling with user functions for lightweight inspection
- Efficient implementation for user-defined scheduling with work stealing and runtime profiling
- Evaluation with various applications and real datasets such as MiniMD and with the GAP benchmark kernels: BFS, Connected Components, and PageRank
- Adoption of user-defined scheduling to a graph framework generating OpenMP codes

The remainder of the papers is organized as follows. In Section 2 we present background information about the OpenMP programming model and scheduling policies.In Section 3 we introduce our API to enable user-defined scheduling. In Section 4 we show how we implement the user-defined scheduling in the LLVM OpenMP runtime. In Section 5 we showcase the benefits of our approach with MiniMD from the Mantevo suite and BFS, Connected Components, and PageRank from the GAP suite. In Section 6 we discuss related works, and in Section 7 we conclude with a summary of our work.

2 BACKGROUND

2.1 Overview of OpenMP Programming Model

OpenMP [11] is a de facto standard for shared-memory parallel programming models. It has most forms of parallelism at the intranode level, such as loop, tasking, offloading, work sharing, vectorization, and atomics. OpenMP is a fork-join programming model, meaning that it has an implicit barrier at the end of each parallel region represented by the *omp parallel* directive or work-sharing constructs such as *omp for*. At the beginning of each parallel region, the OpenMP spawns a pool of threads that run the same codes within the parallel region in single program, multiple data fashion. Therefore, tasking in OpenMP is help-first, which means a created task can be stolen by other worker threads while the thread creating the task proceeds with the rest of the codes in the parallel region.

Among many OpenMP constructs, work-sharing constructs are most commonly used because a user can easily parallelize loops or codes with adding just a few pragmas.

2.2 Scheduling Policies in OpenMP

In this section, we discuss scheduling policies for *omp for* constructs in the OpenMP standard and some previous work on scheduling iterations.

2.2.1 Scheduling Policies in the OpenMP Standard. The OpenMP standard has several predefined scheduling policies for scheduling iterations in the *omp for* work-sharing constructs, such as *static*, *dynamic*, *guided*, *runtime*, and *auto*. The *static* construct statically divides the iteration space into chunks the size of which is the number of iterations/the number of threads in an OpenMP team by default or the size the user provides. Each thread gets chunks that are statically assigned by its thread id, so each thread gets the same chunks when the target loop is executed repeatedly. This policy maximizes locality but loses load balancing for dynamic load imbalance. The *dynamic* construct makes all the threads get a chunk from the iteration space whenever they request a new chunk; all the iterations are made available to all the threads in an OpenMP team, thus eradicating the

load imbalance and increasing resource utilization maximally. However, each thread executes chunks in random order and hence loses data locality, resulting in huge degradation. Setting a chunk size can reduce this degradation by exploiting locality within each chunk but still incurs significant degradation. The *guided* construct is a modified version of *dynamic* where a thread requesting a chunk earlier gets a bigger chunk than do threads requesting later. This is a naive heuristic for better load balancing. The chunk size for each request is the number of iterations left divided by the number of threads in the OpenMP team. The *runtime* construct is a scheduling policy that can be determined by an environmental variable or by calling the API in runtime. The *auto* construct determines the scheduling of the loop automatically by the runtime, which is set *static* on most implementations.

2.2.2 Hybrid Scheduling of static and dynamic. A couple of efforts have been made to overcome the limitations of the standard configurations [20] by mixing *static* and *dynamic*. The basic idea is to split the iteration space into subspaces statically and make some portion of the subspace stealable by other threads within the OpenMP team for load balancing. This scheme enables load balancing while keeping locality. We use this as one of the baseline schemes to be compared with our approach. Our base runtime, LLVM OpenMP runtime, has already implemented this approach. In the rest of this paper, we call this hybrid *static steal* scheduling.

3 DESIGN

In this section, we present an overview of our approach for changing the scheduling of a parallel loop with user functions, which are a set of APIs that enable lightweight inspection and user-specified scheduling of parallel loops.

3.1 Overview of User-Defined Scheduling for Parallel Loops

Figure 2 outlines how our runtime implements scheduling of parallel loop iterations with user functions. The left part of the figure represents standard loop scheduling, while the colored steps on the right of the figure shows the steps added for user-defined scheduling. Our extensions on the right also include creating a scheduler based on help-first work stealing [15] rather than work sharing. In our user-defined scheduling on the right, each thread first queries a shared data structure as to whether the current loop is profiled. If so, it retrieves the corresponding stored group of chunks and executes the chunks. If not, it chooses a subspace of the iteration space of the loop as specified by a user function and creates chunks within the subspace with another user function. The user functions enable customized scheduling based on load estimation for each iteration of a parallel loop. Each thread starts stealing chunks from other threads' local subspaces when its local subspace becomes empty. After all iterations are executed, the threads in the OpenMP team for the loop synchronize and terminate the execution of the parallel loop. In Section 3.2, we introduce an example of API details for enabling user-defined scheduling and explain the implementation of this feature in our runtime.

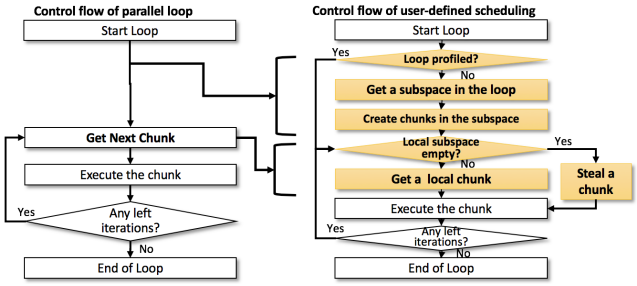

Figure 2: Control flow of parallel loop with user-defined scheduling

3.2 APIs for User-Defined Scheduling and Example

Listing 1 introduces the API to communicate two user-defined functions to the runtime: *inspect_func* and *subspace_select_func*. The first function specifies how each chunk is created. If nothing is specified, it creates chunks of size 1 within each subspace. The second function partitions the iteration space into one subspace per thread. If this function is not specified, the default partitioning is to divide the iteration space evenly among all threads in the OpenMP team as in the *static steal* approach. The third parameter, *user_data*, can optionally point to user-managed data that can be accessed by the two user-defined functions. The next two parameters, *steal_enabled* and *profiling_enabled*, serve as toggles to enable/disable work stealing and profiling in the runtime. Their default values are FALSE. The last parameter, *num_subspaces*, specifies the total number of subspaces in the iteration space of the target loop. The default value is the number of threads in the OpenMP team.

```
// Function to specify user-defined scheduling with user-functions
void ompx_set_usersched_for_loops(
  void (*inspect_func)(int left_start, int left_end,
    int *assigned_start, int *assigned_end, void *user_data),
  // pointer to a user function for chunk creation
  int (*subspace_select_func)(int num_spaces, void *user_data),
  // pointer to a user function for subspace selection
  void *user_data, int steal_enabled,
  int profiling_enabled, int num_subspaces);
/* user_data: pointer to user data accessible within the user
     functions
   steal_enabled, profiling_enabled: toggles to turn on/off
     workstealing and concurrent profiling
   num_subspaces: number of subspaces created by the user function
     for the affected loop */

// Reset the schedule of the upcoming loop to the previous
     schedule set before the user-defined scheduling
void ompx_reset_usersched_for_loops();
```

Listing 1: API for specifying user-defined functions, inspect_func() and subspace_select_func()

Listing 2 shows how the API we proposed can be used to configure user-defined scheduling for iteration chunks in the *omp for* loops. The user can define how each thread selects a subspace from the iteration space and how each iteration chunk is created. In addition, the user can enable work stealing and concurrent load balancing for the created chunks. The function *inspect_func* is called for *omp for* loops with the *schedule(runtime)* clause. Each thread obtains a subspace

```
Graph *g_ptr;
void inspect_func(int left_start, int left_end,
    int *assigned_start, int *assigned_end, void *user_data){
  int weight=0, iter=left_start;
  do {
    weight+=g_ptr->indegree(iter++);
    if (weight>=threshold) break;
    /* Create each chunk when sum of indegrees reaches
       'threshold' */
  } while (curr_idx < left_end);
  *assigned_start=left_start, *assigned_end=curr_idx;
  if (*assigned_end>=left_end) *assigned_end=left_end;
}
int subspace_select_func(int num_subspaces, void *user_data) {
  return omp_get_thread_num();
  //Each thread gets a subspace with its thread id
}
int main (void) {
  g_ptr=&g;
  ompx_set_usersched_for_loops(inspect_func, subspace_select_func,
      NULL, 1, 1, omp_get_num_threads());
#pragma omp parallel for schedule(runtime)
  for (int i=0; i<g.num_nodes(); i++) {
    ...
    for (NodeID v : g.in_neigh(u)) {...}
    ...
  }
}
```

Listing 2: Example of simplified PageRank with user-defined functions applied,

from the iteration space and creates chunks from the iterations within the subspace by the provided *inspect_func*. These chunks are scheduled with or without work stealing or concurrent load balancing, depending on the toggle values provided by the user. In this example, *inspect_func* allocates a maximum of "threshold" indegrees(innerloops) per chunk. The indegree(iter) can be replaced by other load functions for different sparse computations.

4 IMPLEMENTATION

In this section, we describe our implementation of the design from Section 3 in the LLVM OpenMP runtime system. We forked 07/23/2018 commit from the repo[1] in GitHub.

4.1 Overview of Our Implementation

Figure 3 shows an overview of our implementation. We enable chunk creation to be configured by two user-defined functions as described in section 3.2. First, a user can specify a portioning of the loop iteration space into one subspace per worker thread. Next, the OpenMP runtime uses another user-defined function to inspect each iteration and determine how many iterations to be included in the current chunk being created. This inspection enables a variable number of iterations to be included in each chunk, in order to improve load balance as in Listing 2. Each thread pushes the created chunks to a local work-stealing queue. If work stealing is enabled by the API, then other worker threads can steal chunks from busy threads. If a user enables concurrent profiling, each thread copies locally created chunks and stores them in a data structure shared in its OpenMP team. The last thread creating all its chunks starts concurrent load balancing with the stored copy of chunks while others execute chunks in their local work-stealing queue. The balanced groups of chunks after concurrent load balancing are used in future invocations

[1] https://github.com/llvm-mirror/openmp

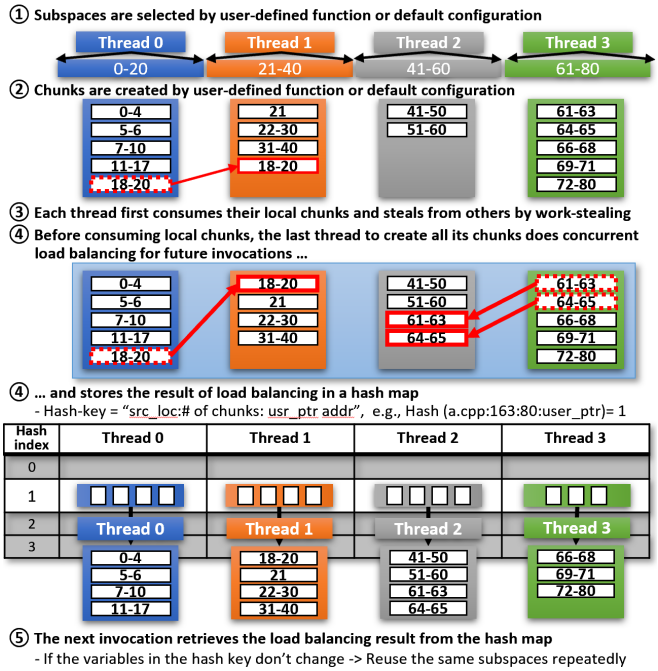

Figure 3: Implementation of user-defined scheduling for iteration chunks, later referred to as "usersched," and "usersched(prof)" with concurrent load balancing in steps 4–5

of this loop. The load balancer thread stores the balanced groups of chunks in a global hash map whose key consists of several identifiers. We describe each step in more detail in the following subsections.

4.2 Runtime Profiling: Concurrent Load Balancing

On steps 4 and 5 of Figure 3, our runtime makes the last thread creating chunks to store balanced groups of chunks in a hash map by concurrent load balancing for the future invocations of the target parallel loops. The load balancer computes the average number of chunks for each subspace in the team and moves chunks across subspaces to make each subspace have the average. The balanced groups of chunks are stored in a hash map indexed by a key, which is a concatenation of three variables: source location info, number of iterations for the target *omp for* loop, and *user_data* pointer address info. If users know when the distribution of chunks of the profiled loop changes by other variables (e.g., communication or load balancing across processes), then they can call *ompx_reset_usersched_loops* just ahead of the loop to initiate load balancing without looking up the hash map. The changes in the variables of each key for the hash map automatically incur load balancing of the loop again after failing to find an entry in the hash map. We use C++ STL unordered_map for the hash map.

4.3 Optimizations to Reduce Runtime Overhead

4.3.1 Selecting a Subspace in the Iteration Space. The way that each thread picks a subspace in the entire iteration space

is important. As mentioned in Section 2, Kale et al. [20] introduced how the hybrid scheduling of *static* and *dynamic* improves load balancing without loss of locality. We adopted this idea for our work and used the subspace selection as our default configuration. In addition, we made this subspace selection capable of being configured by passing a function in the API. For the rest of this paper, we use the default subspace selection and leave exploring the benefit of the configurable subspace selection as our future work. In Section 7 we will briefly discuss applications that can benefit from nonsequential execution of iterations configured by our API.

4.3.2 Work-Stealing Queue Implementation. We adopted the Chase-Lev [10] algorithm, which is an efficient work-stealing algorithm with minimal number of atomic operations incurred. The algorithm is efficient enough for tasks that do not have temporal and spatial locality. However, chunks from consecutive iterations usually have temporal and spatial locality. Therefore, both pop and steal operations of the work-stealing queue should be done in some group of iterations in order to benefit from the locality. Figure 4 shows how we modified the algorithm to enable bulk work stealing in a group of chunks and split the work-stealing queue into two subqueues. When the upcoming loop is **profiled**, each thread computes the size of a group for bulk pop and steal, which is the number of chunks divided by the number of threads in the OpenMP team. Then it pushes chunks in the fixed size of groups of chunks into the first subqueue where pop and steal operations are done in the chunk group size. Then each thread pushes the residual of the division to the second subqueue where pop and steal operations are done in a single chunk. In addition to this modification, we implemented the work-stealing queue in a list of fixed-size arrays in order to increase the size of the queue dynamically, as in the previous work [17]. Profiled loop information also stores created chunks in the same size of fixed arrays, which makes the copy of the profiled information simple.

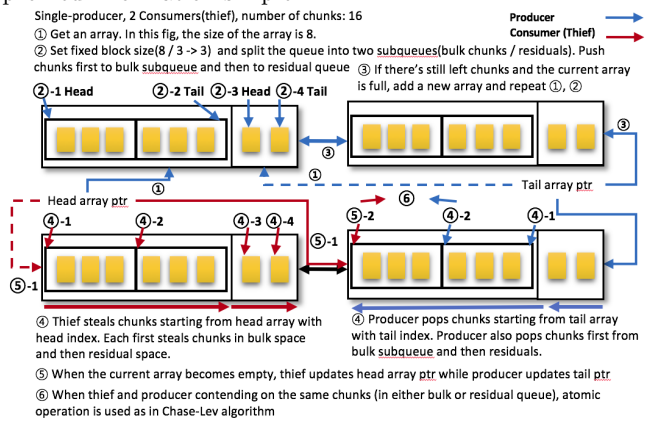

Figure 4: Dynamically increasing work-stealing queue with bulk pop/steal operations

5 APPLICATION STUDY

We use the following four benchmarks to study the performance of our user-defined scheduling: MiniMD from the Mantevo benchmark suite [18] and the BFS, PageRank (PR), and Connected Components (CC) kernels from the GAP benchmark suite [4], each with six different datasets.

We ran all four benchmarks on a single compute node consisting of two Intel Skylake 8180M processors (located at the Joint Laboratory for System Evaluation in Argonne National Laboratory). This processor has 28 cores and 56 hardware threads. All the benchmarks were compiled by Intel Compiler 18.0.1 with *-O3* and executed with *compact* affinity as the selected configuration for OS thread scheduling. We compared the performance of our approach, *usersched* and *usersched(prof)*, with multiple commonly used OpenMP scheduling pragmas (*static*, *dynamic*, *guided*) as well as the *static steal* approach from recent work [20]. The *usersched* label refers to our user-defined scheduling approach, and the *usersched(prof)* label refers to our approach with profiling enabled. We used the same user function shown in Listing 2 with replacing *indegree(i)* with corresponding load function for each application. For MiniMD, the load function is *neigh(i)* and for CC, it is *outdegree(i)*. BFS uses the same function as PageRank.

5.1 MiniMD

MiniMD is a miniapp version of LAMMPS [30] in the Mantevo suite [18], which is one of the most popular molecular dynamics simulations and is developed by Sandia National Laboratories. It has most of the characteristics of LAMMPS and makes it easier to understand molecular dynamics simulations in a few hundreds of lines source code. Regarding the selection of MiniMD over others in the Mantevo suite, we note that the suite has four applications that are written in C/C++ and OpenMP. MiniTri is a duplicate of Triangle Counting, which we study in the GAP suite, and both HPCCG and MiniFE have no need for user-defined scheduling since they already exhibit good load balance with their default OpenMP pragmas. For this reason, we chose MiniMD as the only benchmark to evaluate from the Mantevo suite.

MiniMD can compute forces across neighboring atoms in two ways: embedded atom method and Lennard-Jones (LJ). We optimized the Lennard-Jones force computation kernel with the user function described above. Figure 5 shows the performance improvement in force computation and total execution time of MiniMD with user-defined scheduling. We used 56 threads for the size 10 input and 112 threads for the size 20 input. In the LJ force computation, we achieve 14.99% improvement compared with the best-performing standard configuration and 13.81% improvement compared with the *static steal* scheduling configuration from recent work [20], which leads to 11.38% and 10.17% improvement in total execution time respectively with size 10 input. With size 20 input, the improvement in force computation is 24.0% and 17.5% compared with *static* and *static steal*, which results in 15.83% and 11.54% improvements in total execution time. For MiniMD, *static* works with the default chunk size better than *dynamic* and *guided* do, which means the application has relatively minor load imbalance compared with graph applications. Thus, other dynamic schedules such as *dynamic*

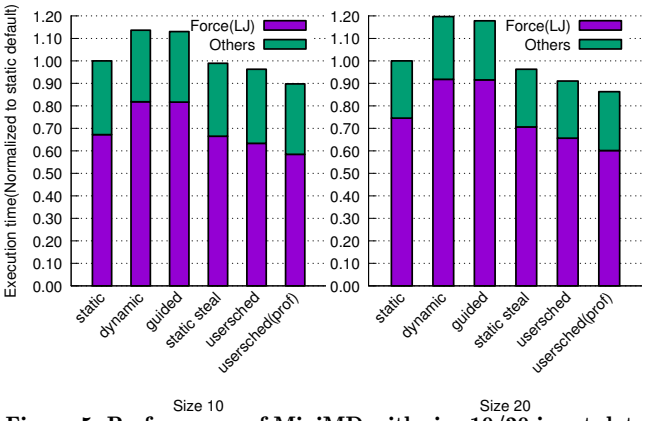

Figure 5: Performance of MiniMD with size 10/20 input data

and *guided* make the performance worse by the loss of data locality. The *static steal* configuration improves the performance marginally for MiniMD, but it cannot achieve an improvement in load imbalance by adjusting iterations in each subspace because each subspace has the same number of iterations in *static steal*. The *usersched* and *usersched(prof)* configuration achieves better load balance than the other configuration deos, by leveraging user-defined functions to create chunks that process close to an equal number of neighbors. With concurrent profiling in *usersched(prof)*, each thread starts with a better initial load balance, which improves the performance by further reducing load imbalance while maintaining locality.

5.2 GAP Benchmark Suite: BFS, CC, and PR

The GAP Benchmark Suite [4] consists of six graph computation kernels written in C++ with OpenMP constructs. It extends prior work for graph evaluation with more diverse input datasets and, like MiniMD, provides a robust and credible baseline for our performance evaluations. Of the six kernels, we focused our evaluation on the three kernels BFS, CC, and PR that are implemented by using the pull-based approach, since pull-based implementations of graph kernels are more amenable to our user-defined approach than are push-based implementations. The Between Centrality (BC) and Single Source Shortest Path (SSSP) kernels use the push-based approach, which keeps updating the active set of vertices, thereby making it unsuitable for our user-defined scheduling approach. Triangle Counting (TC) has extreme control divergence in three levels of nested loops with a conditional statement in each level, thereby making it less amenable for our user-defined scheduling approach because of the difficulty in predicting the load for each iteration of a parallel loop. We ran the three kernels with six real datasets, as shown in Table 1.

The user function for BFS, CC and PR is the same as in Listing 2 with the corresponding load function as described in the beginning of Section 5. CC and PR contain one main parallel region for the computation, whereas BFS uses a hybrid combination of bottom-up (BU) and top-down (TD) steps [3]. We optimized only the bottom-up step using our approach because the top-down step repeatedly changes the

Category	**Wikipedia**	**Internet Topo**	**Patents Citation**
Graph	**Wiki-2007** [13]	**Skitter** [23]	**Patents** [16]
# of Vertices	3.57M	1.70M	3.77M
# of Edges	45.01M	22.19M	16.52M
Category	**Social Network**	**USA Road**	**Web Crawl**
Graph	**LiveJournal** [39]	**Road** [9]	**Web** [6, 7]
# of Vertices	4.00M	23.95M	50.64M
# of Edges	69.36M	57.71M	1.93B

Table 1: Graph datasets used for GAP Benchmark Suite

number of chunks in the parallel loop, thereby making it less amenable to our approach.

Table 2 shows the performance (speedup) of each schedule normalized to the default *static* schedule, for the BFS, CC, and PR kernels, each evaluated with six graphs. For each schedule on each application, we determined the best geometric mean chunk size across all six inputs and used it for all. The chunk sizes we used are also represented in Table 2. For *usersched* and *usersched(prof)*, the chunk size means the numbers of indegree/outdegree for each chunk. So, each chunk may have a different number of outer iterations, as depicted in Figure 3. For all experiments on GAP, we used 112 threads, the scale limit of all the applications on our machine. For BFS, the performance improvement of user-defined scheduling is marginal or worse than the best-performing standard policy because we optimized only the BU step of the BFS algorithm in GAP. The BFS algorithm in GAP switches search direction between BU and TD by comparing the outdegree of the source vertex with heuristic value. Thus, user-defined scheduling shows marginal improvement, 4.4% compared with *static steal* in the Web dataset which have relatively larger number of outdegrees than the heuristic value so incurs BU steps many times. For other graphs, user-defined scheduling works close to the best-performing policy. CC and PR show a huge improvement in performance with *usersched* and *usersched(prof)* compared with all the other

BFS	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	1024	0.998	**1.151**	1.000	0.987	1.001	1.178
dynamic	2048	0.988	1.036	0.976	0.981	0.979	1.124
guided	4096	0.943	0.963	1.025	0.897	0.988	0.872
static_steal	256	**1.009**	0.912	**1.051**	1.044	0.986	1.322
usersched	8192	0.991	1.018	1.000	**1.053**	0.978	1.331
usersched(prof)	8192	0.959	0.892	1.025	0.953	**1.003**	**1.381**
CC	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	1024	1.690	1.872	1.078	2.111	**1.089**	1.151
dynamic	512	1.625	2.454	1.058	2.021	0.985	0.669
guided	512	1.250	1.499	0.993	1.368	1.069	0.713
static_steal	256	1.787	2.193	1.055	**2.299**	0.992	1.237
usersched	8192	1.796	2.568	1.048	2.152	1.074	1.144
usersched(prof)	8192	**1.855**	**3.012**	**1.080**	2.281	1.043	**1.282**
PR	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	256	5.099	2.402	1.167	2.512	0.681	1.059
dynamic	512	4.083	2.240	1.175	2.538	0.663	0.819
guided	1024	1.288	1.345	1.150	1.463	0.766	0.806
static_steal	64	7.688	2.507	1.085	2.901	1.093	1.332
usersched	8192	9.985	2.645	**1.335**	2.651	1.004	1.407
usersched(prof)	8192	**11.326**	**2.845**	1.289	**3.156**	**1.147**	**1.410**

Table 2: Performance (speedup) of BFS, CC, and PR with 6 different graphs (normalized to static default)

schedules on various graphs. Both apps run one parallel main loop repeatedly to reach termination condition. Therefore, creating chunks having an equal amount of load by inner loop info and profiling the groups of chunks after concurrent load balancing make the loop run with better load balance multiple times. For this characteristic, CC works best or closest to best across all input graphs with *usersched(prof)*, improving the performance by 37.3% and 22.7% on Skitter compared with *static steal* and the best standard schedule, *dynamic*. For other graphs, *usersched(prof)* works better than *static steal* by 3∼5%, while *static* shows the best performance on Road compared with others, but the performance difference is marginal. PR, with *usersched(prof)*, shows 47.3%, 13.5%, 18.9%, 8.8%, 4.9%, and 5.8% compared with *static steal* on corresponding graphs in Table 2. For Patents, *static steal* works worst among all the policies. Thus, compared with the best-performing standard policies, our approach achieves 9.7% improvement

For all the graphs and applications we tested, *usersched(prof)* works best or near to the best-performing policies we compared. Even though some graphs and applications work better with different schedules, our schedule with concurrent profiling performs close to the best without significant degradation. In addition, *usersched* shows the best geometric mean performance with the same chunk size, while others require different chunk sizes on each application. This consistency reduces tuning efforts. In the following section, we analyze the benefit of our approach in terms of performance variability, load imbalance, and cache performance. Following this analysis, we will show the applicability of our approach to graph domain-specfic languages (DSLs) by showing a performance improvement of GraphIt PR with user-defined scheduling.

5.2.1 Performance Variability by Chunk Size. Figure 6 shows the performance variance of PR with different chunk size and schedules on Wiki-2007 and LiveJournal graphs. On Wiki-2007, all the schedules show the best performance with 64 chunk size. On LiveJournal, however, 512, 1024, and 2048 chunk sizes are optimal for *static steal*, *static*, and *dynamic*, respectively. The *guided* schedule does not show much variance because the chunk size determines only the minimum chunk size created by the scheduling, which does not affect most of the chunks created other than last few chunks. The *usersched* schedule shows a consistent performance trend and

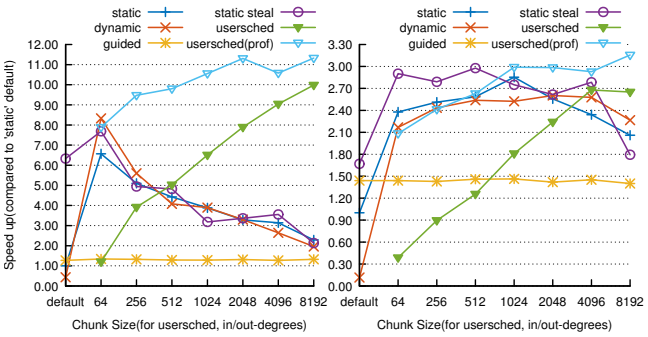

Figure 6: Performance variance of PR with Wiki-2007 and LiveJournal (normalized to static default)

performs best with the biggest chunk size we tested on both graphs. The reason is that our chunk size is the indegree/outdegree of each chunk, which is proportional to the amount of load for each chunk. This makes all the chunks have an approximately equal amount of load, and runtime requires only a certain size large enough to alleviate overhead from the extremely fine-grained size of the chunk. This characteristic makes *usersched* perform best on various input graphs with less tuning effort.

5.2.2 Load Imbalance and Cache Performance Analysis. We measured the amount of load imbalance with Equation 1 [28]. *maxL* and *medL* represent maximum and median load of threads in a OpenMP team for the parallelized loops, respectively. We used median instead of average to measure the imbalance more accurately.

$$\lambda = \left(\frac{maxL}{medL} - 1\right) \times 100\% \quad (1)$$

Table 3 shows the improvement of PR in the load imbalance factor with all the input graphs. We included the result of *static default* to show how much load imbalance exists for each graph on PR. In Table 3, Wiki-2007 has the highest load imbalance. Skitter, LiveJournal, and Web also have considerable load imbalance, while Patents and Road have relatively small imbalance. *dynamic* shows the greatest reduction in load imbalance on most graphs over all the schedules. The *usersched(prof)* schedule achieves a remarkable reduction in load imbalance compared with *static steal*. This metric, however, measures the difference in the median and maximum load of threads in the same OpenMP team. Thus, the smaller value can lead to worse performance due to loss of data locality by excessive migration of data. Hence, while reducing load imbalance for high performance, one also must keep in mind locality.

schedule	Wiki-2007	livejournal	Skitter	Patents	Web	Road
static_default	1242.310	245.320	327.263	56.938	111.484	26.136
static	127.309	7.364	22.727	18.140	32.882	6.054
dynamic	2.941	1.045	2.308	8.162	1.048	1.834
guided	106.489	18.182	21.552	9.026	1.004	0.477
static_steal	4.878	1.271	10.702	28.717	37.324	2.559
usersched	4.331	1.674	16.190	7.272	17.506	2.518
usersched(prof)	2.155	1.843	6.762	13.157	11.813	2.712

Table 3: Load imbalance factor of PR (%)

To clarify how much locality is maintained with our approach, we collected hardware counters by PAPI using native events on PR with the Wiki-2007 graph. We chose three counters to measure total, stall, and cache miss cycles: CPU_CLK_UNHALTED, CYCLE_ACTIVITY:STALLS_TOTAL, and CYCLE_ACTIVITY:STALLS_L1D_MISS (all cycles from L1D to LLC miss) [25]. Table 4 shows *cache miss* cycles, *other stall* cycles (total stall - cache miss), and *productive* cycles (total - (other stall+cache miss)). The difference in the total number of cycles may not correspond to the performance improvement in the earlier figures because of Turbo Boost, which makes the measurement of cycles inaccurate. However, we can see the approximate trend of cache misses and others considering the inaccuracy. In Table 4,

	Cache miss	Productive	Other stall	Total
static_default	**0.0439**	**0.1630**	**0.7931**	**1.0000**
static	0.0586	0.0354	0.1147	0.2087
dynamic	0.0683	0.0372	0.1251	0.2306
guided	0.0599	0.1216	0.5722	0.7537
static_steal	0.0566	0.0225	0.0451	0.1242
usersched	0.0652	0.0164	0.0073	0.0889
usersched(prof)	**0.0554**	**0.0152**	**0.0037**	**0.0743**

Table 4: Performance counter results (average of per-thread cycles) of PR with Wiki-2007 dataset (normalized to static default)

dynamic notably increases cache misses, whereas other schedules have a modest increase in cache misses. Our approach shows minimum cache miss cycles among all the schedules other than *static default*, but it reduces nonmemory stall cycles and productive cycles by removing load imbalance remarkably. From the results in Tables 3 and 4, we can see that our approach successfully maintains locality while improving load imbalance.

5.2.3 Applicability to Graph DSL (GraphIt). Graph is the most popular domain in DSLs. These DSLs generate task-/loop-level parallel codes using Cilk or OpenMP. Among popular graph DSLs, GraphIt [40] is the most recent work for graphs; it provides a separate interface for algorithms and schedules and performs well compared with previous DSLs. GraphIt generates OpenMP codes from high-level DSLs, which use *omp parallel for* for loop-level parallelism. We chose PR with the *dense parallel pull* schedule and optimized the generated code by user-defined scheduling with the similar function we had used for GAP PR. Exploiting GraphIt's schedule-tuning feature, we also ran GraphIt PR with *dense pull and segmentation*, which performs better than the PR with only *dense pull*. For comparison of GAP and GraphIt, we also normalized the performance of GAP PR to the GraphIt PR Pull with *static default*. We made both GraphIt and GAP PR run the same number of iterations for each experiment.

Table 5 shows the normalized performance (speedup) of GraphIt and GAP PR with Wiki-2007 and LiveJournal. The *segmentation* schedule improves the performance of PR Pull on both graphs. However, it also requires parameter tuning to find the optimal number of segments for each dataset. With our manual tuning, *static*, *dynamic*, and *static steal* perform better than *segmentation*. The *usersched(prof)* schedule improved the GraphIt PR much further. We achieved 48.97% and 58.5% speedup compared with the GraphIt PR with *segmentation* on Wiki-2007 and LiveJournal. Compared with *static steal*, we improved 29.3% and 7.3% on both graphs. For the standard schedules, we used different chunk sizes for GraphIt and GAP PRs, while *static steal* and *usersched(prof)* used the same chunk size. The results show that our approach can substantially improve the generated OpenMP codes by graph DSLs with less parameter tuning. Our improvement in GraphIt PR on LiveJournal is smaller than in GAP PR because GraphIt generates several fine-grained parallel loops, whereas GAP PR runs in a single big parallel loop, which has more room for improvement by load balancing. This coalesced parallel loop in GAP also makes its base performance better than that of GraphIt.

Wiki-2007	**GraphIt**	**GAP**	**LiveJournal**	**GraphIt**	**GAP**
segmentation	4.840		**segmentation**	1.639	
static default	1.000	1.096	**static default**	1.000	1.316
static	5.127	5.590	**static**	2.130	3.307
dynamic	5.576	4.476	**dynamic**	2.018	3.341
guided	1.271	1.412	**guided**	1.320	1.926
static_steal	4.802	8.428	**static_steal**	2.419	3.819
usersched	7.193	10.947	**usersched**	2.357	3.490
usersched(prof)	**7.210**	**12.416**	**usersched(prof)**	**2.598**	**4.154**

Table 5: Performance (speedup) of GraphIt PR Pull with different schedules and graphs (normalized to GraphIt static default)

5.3 Overhead Analysis

Since our approach changes runtime flow with user functions, significant overhead may be incurred. To measure the runtime overhead accurately, we used a simple flat parallel loop and user functions that create a certain size of chunks specified by the user. Each iteration in this simple flat loop executes 10 integer additions into a local variable to remove any cache-related variables. To minimize load imbalance, we set the number of iterations for the loop to be divisible by the number of threads of the machine, which is 1792^2 (1792/112=16). We executed this example 300 times for each run with variable chunk sizes from 1 to 1024. We used the LLVM OpenMP internal statistics module (LIBOMP_STATS=1) to measure per-thread runtime overhead and the median of the per-thread value.

Figure 7 shows the normalized overhead of the simple flat loop with variable chunk sizes, where each value is computed by $\frac{Tsched,chunk}{Tstatic,1}$. The *dynamic* schedule always incurs more overhead than do *static steal* and *usersched* because of huge contention on the shared index by atomic operations; *static* shows minimum overhead as expected; *guided* does not change much because the chunk size determines only the minimum size of chunk; and *usersched* shows slightly more overhead than does *static steal* because of the user functions called. The overhead incurred by user functions is removed by concurrent profiling, which makes *usersched(prof)* almost similar to *static steal*. Work stealing incurs some overhead

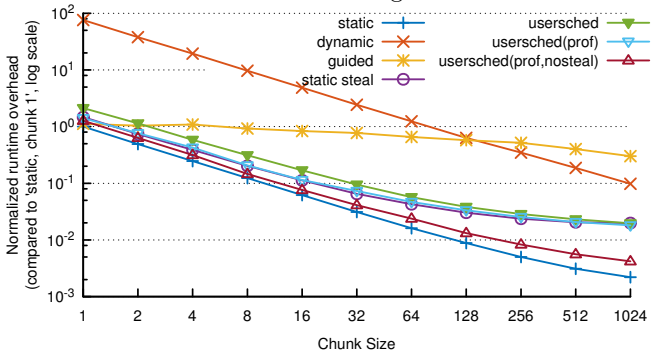

Figure 7: Runtime overhead time of simple flat loop with different chunk sizes (normalized to static, chunk 1)

in this load-balanced example because atomic operations are called waiting for others. This overhead is incurred regardless of the size and number of chunks, thus making the overhead more obvious between *usersched* and *static* with bigger chunk sizes. With work stealing off, our approach shows the closest overhead to that of *static*. This experiment shows that our approach is efficiently implemented with more flexibility to OpenMP compared with other dynamic schedules.

5.4 Applicability of Our Approach

In the previous sections, our approach improved irregular parallel loops where each iteration has variable amount of load. User functions enable more optimal chunking and scheduling of iterations. However, our approach still has limitations in its applicability. First, if the affected loop is not used repeatedly, it cannot benefit from the concurrent profiling. For example, BFS is improved only when their algorithm selects pull-based approach running same loop repeatedly. In addition, our approach is not amenable to improve parallel loops with extreme control divergence in multiple nested parallelism which is hard to predict as described in Section 5.2 with Triangle Counting.

6 RELATED WORK

Many previous works have developed locality-aware load balancing and scheduling policies in task-level and distributed parallel programming models. Loop scheduling in particular has been studied for decades, and most of the well-known previous works [31, 37, 38] have been implemented in shared-memory programming models. Hybrid scheduling of static and dynamic has also been proposed and adopted to improve load balancing with locality maintained [20]. In addition to the hybrid algorithm, task-level parallelism such as Cilk [5], TBB [29], OpenMP 3.0 [27], OmpSs [8], HPX [19], and Habanero [2] have adopted work stealing for load balancing with optimizations to reduce migration cost and maintain locality. This locality-aware load balancing has also been studied in distributed programming models through hybrid programming models and hierarchical load balancing by restricted load balancing in a region of PEs [21, 24].

The previous approaches based on load information of PEs incur unnecessary migration and inefficient scheduling. To overcome this inefficiency, there have been efforts to use information on application codes. The inspector-executor model is one of the most well-known approaches in this direction [22, 32], in which user codes are inspected and parallelism is extracted from the codes by checking conflicts on data structures. This model enables efficient scheduling but it cannot handle performance variance and dynamic load imbalance efficiently. Our approach also uses inspection but in a different way, namely, using user functins to look through user codes for load balancing.

In addition to the efforts in parallel programming models, domain-specific languages(DSL) have been introduced that resolve load imbalance and achieve locality. Graph is a popular domain in the DSLs. The graph DSLs generate codes in parallel programming languages or lower-level runtime codes [14, 26, 33, 35, 36, 40, 41]. They provide reasonable performance and programmability by high-level API but lose an opportunity to optimize generated codes further when the applications are written directly in the target parallel programming languages. Our work shows opportunities to improve the DSLs with configurable loop parallelism by user-functions in Section 5.2.3.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a set of APIs and implementations to enable user-defined scheduling on parallel loops, handling load imbalance and performance variance while maintaining locality. Our proposal uses user functions to inspect each iteration and store distribution of loads for the target loop dynamically in runtime after concurrent load balancing for the future invocations of the loop, reusing the information to schedule them with better initial load balance for each invocation of the loop. This reuse of the stored information helps the performance of irregular applications that have different configurations for optimal performance depending on input datasets. Through evaluations with the GAP Benchmark Suite and MiniMD, we show that our approach helps resolve performance variance and load imbalance on graph applications as well as scientific applications. Without profiling enabled (*usersched*), our approach achieves geometric mean speedups of 1.11× to 1.48× over four standard OpenMP schedules and 1.03× over the *static_steal* schedule. With profiling enabled (*usersched(prof)*), our approach achieves higher geometric mean speedups of 1.16× to 1.54×, and 1.07×, respectively. Furthermore, compared with *static steal*, we achieve 17.5% improvement in the LJ force computation of MiniMD and 47.3% in PR, 37.3% in CC, and 4.4% in BFS from the GAP suite. In addition, we achieve 49.0% and 58.5% improvements relative to GraphIt's best-performing settings for PR on two graphs.

An interesting direction for future research is addressing the high level of control divergence in applications such as Triangle Counting. In addition, with configurable subspace selection, nonsequential selection of subspaces may be beneficial for some irregular applications, relative to our current static decomposition of subspaces.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

REFERENCES

[1] S. Bak, H. Menon, S. White, M. Diener, and L. V. Kalé. 2018. Multi-Level Load Balancing with an Integrated Runtime Approach. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. 31–40.

[2] R. Barik, Z. Budimlic, V. Cave, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 735–736.

[3] S. Beamer, K. Asanović, and D. Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 12, 10 pages.

[4] S. Beamer, K. Asanovic, and D. A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*. Santa Barbara, California, 207–216. MIT.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 587–596.

[7] P. Boldi and S. Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[8] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. 2011. Productive Cluster Programming with ompss. In *Euro-Par 2011 Parallel Processing*. Springer, 555–566.

[9] A. G. Camil Demetrescu and D. Johnson. 2006. 9th DIMACS Implementation Challenge - Shortest Paths. (2006). Retrieved April 8, 2019 from http://users.diag.uniroma1.it/challenge9/

[10] D. Chase and Y. Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 21–28.

[11] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[12] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. 2010. Hybrid parallel programming with MPI and unified parallel C. In *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 177–186.

[13] D. Gleich. 2007. Wiki-20070206. (2007). Retrieved April 08, 2019 from https://sparse.tamu.edu/Gleich/wikipedia-20070206

[14] S. Grossman, H. Litz, and C. Kozyrakis. 2018. Making Pull-based Graph Processing Performant. *SIGPLAN Not.* 53, 1 (Feb. 2018), 246–260.

[15] Y. Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12.

[16] B. H. Hall, A. B. Jaffe, and M. Trajtenberg. 2001. *The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools*. Working Paper 8498. National Bureau of Economic Research.

[17] D. Hendler, Y. Lev, M. Moir, and N. Shavit. 2006. A Dynamic-Sized Nonblocking Work Stealing Deque. *Distributed Computing* 18, 3 (01 Feb 2006), 189–207.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report. Sandia National Laboratories.

[19] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages.

[20] V. Kale, A. Randles, and W. D. Gropp. 2014. Locality-Optimized Mixed Static/Dynamic Scheduling for Improving Load Balancing on SMPs. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM, New York, NY, USA, Article 115, 2 pages.

[21] V. Karamcheti and A. A. Chien. 1998. A Hierarchical Load-Balancing Framework for Dynamic Multithreaded Computations. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 6–6.

[22] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 211–222.

[23] J. Leskovec, J. Kleinberg, and C. Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. ACM, New York, NY, USA, 177–187.

[24] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. 2012. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*. 137–148.

[25] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel. 2017. Detecting Memory-Boundedness with Hardware Performance Counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 27–38.

[26] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471.

[27] OpenMP ARB. 2008. OpenMP Application Program Interface Version 3.0. In *The OpenMP Forum, Tech. Rep.*

[28] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. 2012. Quantifying the effectiveness of load balance algorithms. In *26th ACM international conference on Supercomputing (ICS '12)*. 185–194.

[29] C. Pheatt. 2008. Intel® Threading Building Blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[30] S. Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (1995), 1 – 19.

[31] C. D. Polychronopoulos and D. J. Kuck. 1987. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.* C-36, 12 (Dec 1987), 1425–1439.

[32] J. H. Saltz, R. Mirchandaney, and K. Crowley. 1991. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.* 40, 5 (May 1991), 603–612.

[33] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 135–146.

[34] L. Smith and M. Bull. 2001. Development of Mixed Mode MPI / OpenMP Applications. *Scientific Programming* 9, 2,3 (Aug. 2001), 83–98.

[35] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 16, 10 pages.

[36] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225.

[37] P. Tang and P. Yew. 1986. Processor Self-Scheduling for Multiple-Nested Parallel Loops.. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 528–535.

[38] T. H. Tzen and L. M. Ni. 1993. Trapezoid Welf-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems* 4, 1 (Jan 1993), 87–98.

[39] J. Yang and J. Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *2012 IEEE 12th International Conference on Data Mining*. 745–754.

[40] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages.

[41] X. Zhu, W. Chen, W. Zheng, and X. Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316.