

# Composing Low-Overhead Scheduling Strategies for Improving Performance of Scientific Applications

Vivek Kale<sup>(✉)</sup> and William D. Gropp

University of Illinois at Urbana-Champaign, Urbana, IL 61822, USA  
vivek@illinois.edu

**Abstract.** Many different sources of overheads impact the efficiency of a scheduling strategy applied to a parallel loop within a scientific application. In prior work, we handled these overheads using multiple loop scheduling strategies, with each scheduling strategy focusing on mitigating a subset of the overheads. However, mitigating the impact of one source of overhead can lead to an increase in the impact of another source of overhead, and vice versa. In this work, we show that in order to improve efficiency of loop scheduling strategies, one must adapt the loop scheduling strategies so as to handle all overheads simultaneously. To show this, we describe a composition of our existing loop scheduling strategies, and experiment with the composed scheduling strategy on standard benchmarks and application codes. Applying the composed scheduling strategy to three MPI+OpenMP scientific codes run on a cluster of SMPs improves performance an average of 31 % over standard OpenMP static scheduling.

## 1 Introduction

Performance of scientific application code can be impacted by how efficiently iterations of a parallel loop are scheduled to cores. Many different sources of performance loss impact the efficiency of a scheduling strategy applied to a parallel loop, as we will show in Sect. 2. In prior work, we developed multiple loop scheduling strategies, with each scheduling strategy focusing on mitigating a subset of the overheads. However, mitigating the impact of one source of performance loss can lead to an increase in the impact of another source of performance loss, and vice versa. In this work, we show that in order to schedule loops efficiently, we need to compose loop scheduling strategies so as to handle multiple sources of performance loss simultaneously.

Our contribution, in addition to a specific composite scheduling strategy, is a guide to combining scheduling strategies to handle multiple sources of the overhead together, to handle the circumstances and challenges posed by an application and architecture. Such a scheduling strategy can be beneficial to improve performance of scientific applications on clusters of multi-cores, and can be beneficial in the context of next-generation, e.g., exascale, clusters of SMPs.

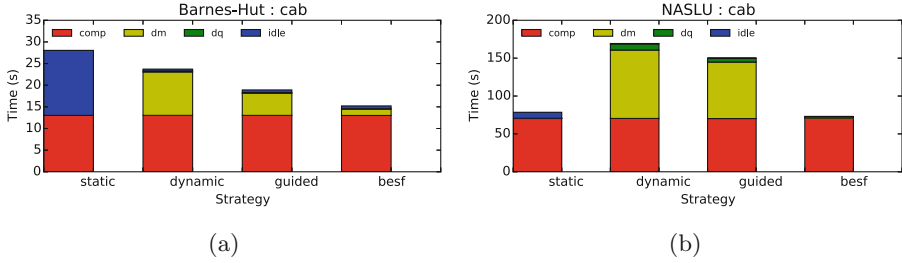
In the sections that follow, we discuss implementation of a scheduling strategy composition containing many different scheduling techniques implemented up to this point. We show results for different scientific application codes, i.e., two CORAL benchmarks and one particle simulation application code, using different types of scheduling strategies. Finally, we conclude the paper through a discussion of scheduling techniques and the scheduling strategy composition in the context of running applications on next-generation architectures.

## 2 Scheduling Strategies

Consider a common structure of scientific applications: outer iterations, e.g., timesteps, which enclose inner iterations that typically loop over data arrays. For these codes, load balancing of the computational work across cores of a node is necessary for obtaining high performance. Load balancing can be attained through the use of OpenMP loop scheduling strategies. However, there are multiple sources of performance loss in parallel scientific applications, and different schedulers affect these sources differently.

Figures 1 and 2 show the sources of performance loss through a breakdown of execution times for widely used loop scheduling strategies applied to two different application codes: a Barnes-Hut code (left) with non-uniform iteration times, i.e., load imbalance across iterations, and a NAS LU code (right) with uniform iteration times. NAS LU can still benefit from dynamic load balancing within a node because such load balancing can deal with imbalances caused by noise, which are *amplified* in synchronous MPI codes [7]. The performance data are for a node of a cluster of Intel Xeon 16-core processors. The execution time breakdown is shown as a stacked bar graph in Fig. 1. Thread idle time is labeled as ‘idle’ and cost of synchronization is labeled as ‘dq’. We measure the time each thread waits at the barrier, and use the average over threads as the cost of thread idle time. We estimate the cost of synchronization by using hpcToolkit to obtain the time spent in the `omp_lock()` function. The computation time, labeled ‘comp’, is calculated by dividing the sequential execution time by the number of threads. The remaining execution time is attributed to data movement and labeled as ‘dm’. Note that this breakdown may not be exact, but it gives us an adequate estimate to understand the impact of overheads to the efficiency of the scheduling strategies. For obtaining the cache misses, we used PAPI counters `PAPI_L2.TCM` and `PAPI_L3.TCM` for the L2 and L3 cache misses, respectively. We measured cache misses for each OpenMP parallel loop for thread 0. In Fig. 2, the L3 cache misses are shown.

Using static scheduling for these codes makes data movement small and eliminates synchronization overhead, but does not mitigate load imbalance. For Barnes-Hut, the thread idle time is 21 % of the total execution time. For NAS LU, idle time shown is small, but not negligible, at 2.8 %. Using dynamic scheduling improves load balance almost completely, but dynamic scheduling still causes data movement and synchronization overhead. Also, the synchronization overhead is still noticeable at 5.9 % when dynamic scheduling is applied to the two codes.



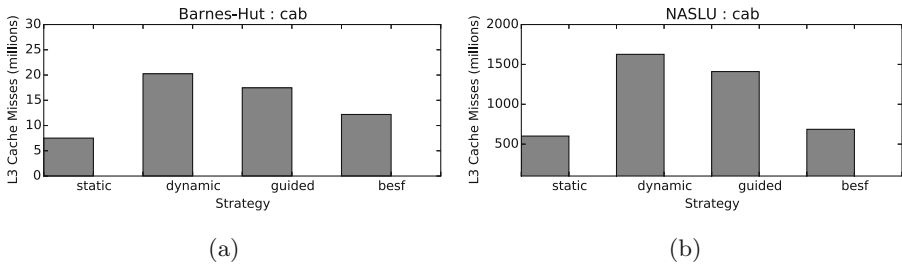
**Fig. 1.** Breakdown of execution time for NAS LU and n-body code.

Finally, guided scheduling can reduce synchronization overhead. However, guided scheduling still incurs data movement across cores, as is seen by the large number of cache misses for Barnes-Hut and NAS LU in Fig. 2.

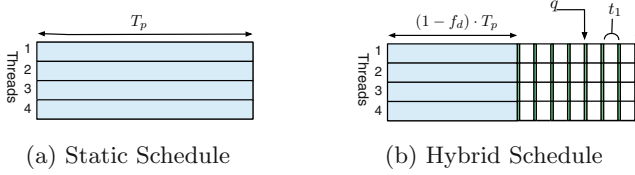
We have identified three challenges to obtaining good performance using dynamic load balancing within a node: (1) cost of load imbalance due to load imbalances from the application or system noise, (2) data movement overhead, and (3) synchronization overheads from runtimes. None of the scheduling strategies examined was able to handle all sources of performance loss. This challenge provides motivation for developing a new set of scheduling strategies.

To handle all 3 challenges, one could intelligently blend static and dynamic scheduling strategies, where the first  $k$  loop iterations are scheduled statically across threads, and the remaining  $n-k$  loop iterations are scheduled dynamically across threads [4]. The parameter  $k$  is experimentally tuned. We define  $\frac{n-k}{n}$  as the dynamic fraction  $f_d$ . Correspondingly, the static fraction  $f_s = 1 - f_d$ . We refer to this scheduling strategy as hybrid static/dynamic scheduling. Figure 3a shows loop iterations scheduled statically across 4 cores during one invocation of a threaded computation region. Figure 3b shows the corresponding diagram for the hybrid static/dynamic scheduling strategy.

The 4<sup>th</sup> bars from the left in Fig. 1a and b show the execution time for NAS LU and Barnes-Hut when the hybrid static/dynamic scheduling strategy, labeled *besf*, is used. The hybrid static/dynamic scheduling strategy is the best



**Fig. 2.** L3 cache misses for different OpenMP scheduling strategies.



**Fig. 3.** Diagram of threaded computation region with different schedules applied to it.

performing of the four scheduling strategies shown. The reason is that data movement overhead is reduced significantly compared to the dynamic scheduling scheme, but the scheduling scheme does enough dynamic scheduling to handle the cost of load imbalance. Using hybrid static/dynamic scheduling for NAS LU does not improve performance significantly over OpenMP static scheduling, but it does not degrade performance either. The hybrid static/dynamic scheduling strategy reduces thread idle time for NAS LU, rather than increasing it. Although NAS LU seems efficient with static scheduling, consider the situation when it is running on a machine with significant OS noise, i.e., the interference created by OS daemons. In this situation, amplification of noise across MPI processes can cause large performance degradation, and dynamic scheduling of loop iterations can potentially mitigate this impact of noise [7].

As we will see in the next section, different circumstances, including architectural/OS and application characteristics, require different scheduling techniques to modify the above basic hybrid static/dynamic scheduling strategy. We next show what those techniques are and how to compose the techniques into a single effective scheduler.

### 3 Techniques for Composing Scheduling Strategies

In the context of the problem listed in the previous section, we design a scheduling strategy that can handle the many different sources of performance loss and the inefficiencies of the scheduling strategies. We first give a description of each of the elemental scheduling strategies, which are based on existing scheduling strategies from prior work; the existing scheduling strategies are adapted from the perspective of composing the scheduling strategies together. We then show a composition of the scheduling strategies described.

#### 3.1 uSched

This scheduling strategy is designed to mitigate the impact of transient events such as OS noise as well as application-induced imbalances. *uSched* first measures its parameters such as iteration time and noise duration [7]. It then uses a model-guided determination of the dynamic fraction (considering both application imbalance and imbalance due to noise) to determine a reasonable baseline value of the static fraction  $f_s$ , as described in [7]. After this, we conduct an

exhaustive search in a small neighborhood around  $f_s$ . We try different static fractions in the range  $[f_s - 0.05, f_s + 0.05]$ . This increment can be adjusted by the application programmer and requires knowledge of iteration granularity. The resulting static fraction is  $f_{stuned}$ , which is the static fraction used for *uSched*. This is the static fraction used for all nodes.

### 3.2 slackSched

This scheduling strategy is an optimization over *uSched*, as described in prior work [7]. It uses a distinct static fraction for each node based on MPI slack. MPI slack is the deadline that each process has to finish its work, before this process extends the applications critical path thereby increasing the cost of application execution. Because of the way collective calls are implemented in MPI, the slack is different on different processors. We use the call-path method [7, 12] for predicting the slack for each collective call. In the context of the scheduling strategy composition that we want to do, the scheduling strategy is put together and works as follows:

1. On each process, start with the static fraction  $f_s$  obtained in *uSched*.
  - (a) On each process, retrieve that process's invocation of the last MPI collective, where the invocation of the last MPI collective is retrieved through the callsite slack-prediction method, as shown in prior work [7].
  - (b) Given the identifier of the last the MPI collective call invoked, estimate that collective call's slack value from the history of slack values stored by the slack-conscious runtime. The slack estimate is based on the slack value recorded in the previous MPI collective invocation, as is done in prior work [7].
2. On each process, adjust its dynamic fraction based on the slack value. This adjustment is done using a performance model and theoretical analysis described in prior work [7].

### 3.3 vSched

This scheduling strategy is based on prior work [8]. The motivation of this scheduling strategy is to improve the spatial locality in the dynamically scheduled iterations. In the above schedulers, the dynamically allocated iterations are grouped at the end of the iteration space. Here, we stagger them, so as to keep the iterations executed by a thread contiguous as much as possible. Let  $p$  denote the number of cores on one multi-core node. Let  $t$  denote the thread with thread ID  $t$ . Let  $n$  be the number of loop iterations of an OpenMP loop. The static iterations assigned to each thread are from  $\lfloor \frac{n \cdot t}{p} \rfloor$  to  $\lfloor \frac{n \cdot (t + f_s)}{p} \rfloor$ , while dynamic iterations associated with thread  $t$  are from  $\lfloor \frac{n \cdot (t + f_s)}{p} \rfloor + 1$  to  $\lfloor \frac{n \cdot (t + 1)}{p} \rfloor - 1$ . In the context of the scheduling strategy composition that we want to do, we implement this scheduling strategy by starting with the hybrid static/dynamic scheduling strategy, and then apply the staggering of iterations to this hybrid static/dynamic scheduling strategy.

### 3.4 ComboSched

The comboSched scheduling strategy is vSched, i.e., locality-optimized scheduling, with slackSched, i.e., slack-conscious scheduling, added into it. In other words, one optimization over uSched, slackSched, is composed with another optimization over uSched, vSched, to form the comboSched scheduling strategy. The comboSched scheduling strategy is put together and works as follows:

1. Stagger the iterations, as specified in vSched.
2. Start with the static fraction obtained from the uSched scheduling strategy.
3. Specify the queue to steal from in the vSched scheduling strategy.
4. On each process, adjust its dynamic fraction based on the slack value, as described in slackSched.

In summary, we described a series of scheduling strategies, and showed the design of a scheduling strategy composition using the features of these scheduling strategies. We next show code transformation needed to use the scheduling strategy composition and assess performance of the application of these scheduling strategies and the scheduling strategy composition to three application codes.

## 4 Code Transformation

Below, we show the changes to a simple MPI+OpenMP code needed to use our scheduling strategy. Figure 4 shows an application program containing a basic OpenMP loop. Figure 5 shows the same application code containing the OpenMP loop transformed to use our composed scheduler. The macro functions used for invoking our library’s loop scheduling strategies are defined at lines 5–7 of Fig. 5, and the parameter value ‘strat’ of the macro function indicates the scheduling strategy to be used from our library. The `sds` parameter value in the macro functions’ invocations at lines 24 and 27 specifies the staggered static/dynamic scheduling strategy of our library, i.e., the *vSched* strategy described in Sect. 3. The implementation changes needed for the composition are done within our macro-invoked scheduler. The `record` struct variable is used to store information about previous invocations of the threaded computation region in lines 25 and 26, and necessary for the slack-conscious scheduling strategy, i.e., the *slackSched* strategy described in Sect. 3. Our scheduling strategy could equivalently be implemented in an OpenMP runtime and offered as an OpenMP loop schedule.

## 5 Results

With the above composition of schedulers, the question we ask is: does our composition of the schedulers and adjustment of the scheduler parameters help provide further performance improvement than each of the schedulers in isolation?

To answer the above, we experimented with three different MPI+OpenMP application codes. The first application code is Rebound [11], an MPI+OpenMP

```

1  #include "mpi.h"
2  #include <omp.h>
3  int main(int argc, char* argv[])
4  {
5      // ...
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
8      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9      // ...
10     while(timestep < 1000){
11         #pragma omp parallel for
12         for(int i=0; i<n; i++){
13             c[i] += a[i]*b[i];
14             MPI_Allreduce(&sum,&global_sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
15             timestep++;
16         }
17     }
18     MPI_Finalize();
19 }

```

Fig. 4. Code with OpenMP loop.

```

1  #include "mpi.h"
2  #include <omp.h>
3  #include "vSched.h"
4  // ...
5  // In the macros below, strat specifies the sched strategy.
6  #define FORALL_BEGIN(strat, s, e, start, end, tid, numThds) loop_start_ ## strat (s, e, &start,
7      &end, tid, numThds); do {
8  #define FORALL_END(strat, start, end, tid) } while(loop_next_ ## strat (&start, &end, tid));
9  int main(int argc, char* argv[]){
10     // ...
11     int tid, numThrs, start, end = 0;
12     double fd, fs;
13     static LoopTimeRecord *record = NULL;
14     MPI_Init(&argc,&argv);
15     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
16     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
17     vSched_init(numThrs);
18     // ...
19     while(timestep < 1000) {
20         fd = predict_dynamic_fraction(&record); fs = 1.0 - fd;
21         #pragma omp parallel
22         {
23             tid = omp_get_thread_num();
24             numThrs = omp_get_num_threads();
25             FORALL_BEGIN(sds,tid,numThrs,0,n,start,end,fs)
26             for (int i=start;i<end;i++)
27                 c[i] += a[i]*b[i];
28             FORALL_END(sds,tid,start,end)
29         }
30         end_timing(&record, n);
31         MPI_Allreduce(&sum,&global_sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
32         timestep++;
33     }
34     endLoop(&lr, (int) (n*fd));
35     vSched_finalize(numThrs);
36     MPI_Finalize();
37 }

```

Fig. 5. Code transformed to use composed scheduling strategy.

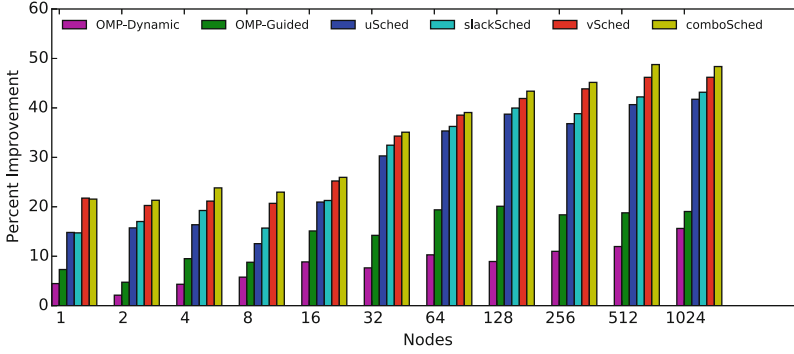
n-body simulation that simulates bio-molecular interactions. The second application code is the CORAL SNAP code [13], regular mesh code which has computation used in the context of heat diffusion. The third application code is the CORAL miniFE code [6], an MPI+OpenMP finite element code involving computation on an unstructured mesh used in the context of earthquake simulations.

We performed the experiments on Cab, an Intel Xeon cluster with 16 cores per node, 2.66 GHz clock speed, a 32 KB L1 data cache, a 256 KB L2 cache, 24 MB shared L3 cache, the TOSS operating system, an InfiniBand interconnect with a fat-tree network topology. We ran each application code with 1 MPI process per node and 16 OpenMP threads per MPI process.

Figure 6 shows the results for the MPI+OpenMP n-body code Rebound [11] run on Cab, with different schedulers applied to this code. In this code, every particle loops through its neighborhood of particles to calculate forces applied to it, identifying the position in the next application timestep; there is geometric locality in this application. This geometric locality is reflected by the order in which the particles are organized in the tree. For example, nearby particles tend to interact with the same sets of particles with a few exceptions. Therefore, the *vSched* strategy of keeping nearby iterations on the same thread in the dynamic section provides performance benefits. The *slackSched* benefits are the generic benefits of reducing the dynamic fraction and its associated overheads. The benefits are not as large for other applications because of its relatively large grain size of each iteration. For Rebound at 1024 nodes, the *comboSched* improves performance 45 % over OpenMP static scheduling. The percent gains of each of the scheduling strategies are significant even at low node counts. Specifically, at 2 nodes, performance improves 35 % over OpenMP static scheduling when we apply only *uSched* to the Rebound code. Using *slackSched* on Rebound gets limited gains of 5.6 % over the *uSched* scheduling strategy. Using *vSched*, performance improves 8.5 % over *uSched*. This is likely because *vSched* can take advantage of the geometric locality in this application. Using the *comboSched* strategy, which combines *slackSched* and *vSched*, the Rebound code gets an overall 44 % performance gain over the OpenMP static scheduled version of Rebound.

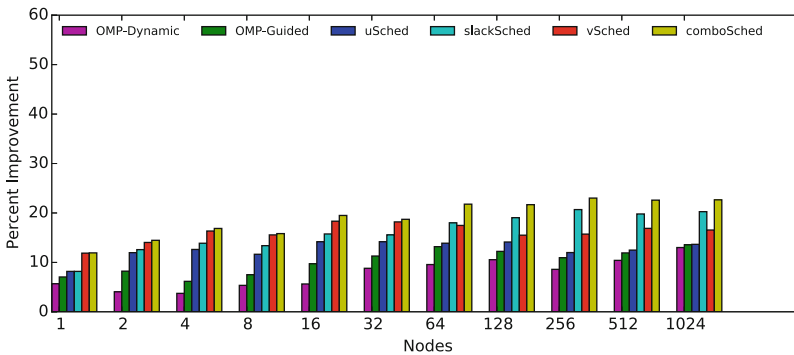
Figure 7 shows the results for miniFE [6] run on Cab, with different schedulers applied to miniFE. Here, iteration-to-iteration spatial locality is relatively low because of indirect access caused by the unstructured mesh; for unstructured meshes, the spatial locality across iterations is not as strong as looping over a 1-D array. However, with reasonable variable ordering of mesh elements, there is still a significant amount of spatial locality that *vSched* exploits. Because of imperfect data partitioning of the problem across nodes, moderate load imbalances across nodes exist. Due to the law of large numbers, the imbalances across cores are larger at larger number of nodes. Thus, dynamic or guided scheduling by itself should be able to provide significant performance gains. Consider the results for miniFE running at 1024 nodes of Cab. The *vSched* scheduling strategy gets 15 % performance improvement over OpenMP static scheduling, while the *slackSched* gets 19 % performance gain over OpenMP static scheduling. The *comboStrat* gets 23 % performance improvement over OpenMP static scheduling, and also gets 9.0 % performance improvement over OpenMP guided scheduling. By putting together *vSched* and *slackSched*, we are able to improve performance further, to make our scheduling methodology perform better than *guided*. The benefits of *vSched* and *slackSched* are not completely additive. Composing the scheduling strategies along with tuning of parameters could increase performance benefits, and could yield better performance for the *comboSched*.



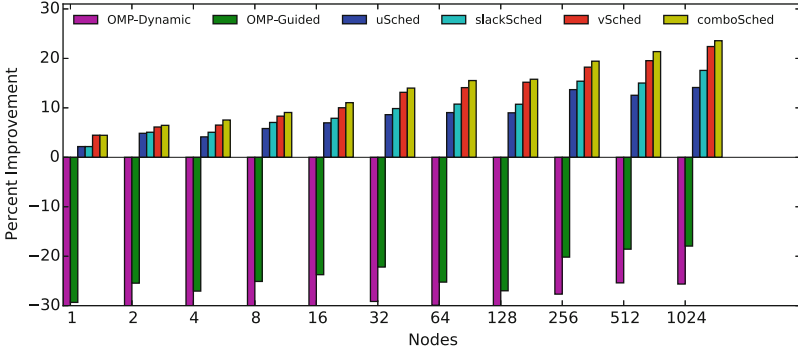


**Fig. 6.** Rebound (n-body): Performance improvement obtained over OpenMP static scheduling.

Figure 8 shows the results for the regular mesh code SNAP [13] run on Cab, with different schedulers applied to the SNAP code. The regular mesh computation has no application load imbalance; the only load imbalance during application execution is that due to noise. Note that the regular mesh computation has inherent spatial locality (because the computation’s sweep operation works on contiguous array elements). At 1024 nodes of Cab, performance improves 10 % over OpenMP static with *slackSched*, and we get a reasonable performance gain of 16 % over static scheduling with *vSched*. The *comboSched* scheduler gets 19 % performance improvement over OpenMP static scheduling. This result of *comboSched* specifically helps to show that the optimizations of *vSched* and *slackSched* composed in *comboSched* do not cancel out each other’s performance benefits.



**Fig. 7.** miniFE (finite element): Performance improvement obtained over OpenMP static scheduling.



**Fig. 8.** SNAP (regular mesh): Performance improvement obtained over OpenMP static scheduling.

## 6 Related Work

The work in [1, 9, 10] attends to outer iteration locality by dynamically scheduling the loop iterations so that each core tends to get the same inner loop iterations over successive outer iterations. In contrast, our strategy sets aside iterations that are scheduled statically without the locking overhead to maintain outer iteration locality. The problem of amplification and the phenomenon of MPI slack arise only in the context of a cluster of multiprocessors, and are absent from older work which was focused on shared memory machines. Also, hybrid programming with MPI and pthreads/OpenMP did not exist at the time of the work. Zhang and Voss [14] present scheduling techniques based on runtime measurements, but the techniques are designed specifically for the problems arising out of simultaneous multi-threading (hyperthreads). For example, the techniques involve runtime decisions about whether the number of threads should be equal to the number of cores, or equal to the number of hyperthreads.

Loop iterations are a form of independent tasks. Several programming models support creation of independent tasks directly. One of the primary shortcomings of work-stealing [5] is that work-stealing incurs overhead due to the cost of coherence cache misses, which depend on the number of cores and the shared memory interconnect of the node architecture [2]. In contrast, our work focuses on reducing coherence cache misses. Scalable work-stealing [3] can be beneficial in a distributed memory context, but it mainly focuses on steals across a large number of nodes. Our work is focused on within-node scheduling, and to that extent is orthogonal to scalable work stealing.

## 7 Conclusions

In this work, we identified a number of scheduling strategies, each with different features. We expect many of the features of the scheduling strategies to be relevant for running parallel applications on current and future clusters of SMPs.

We then provided a guide for composing these scheduling strategies together. Our results showed on average 31 % performance improvements over static scheduling for three scientific applications.

Many unknown circumstances will likely exist when running applications on next-generation supercomputers, e.g., exascale machines. The composition of existing scheduling strategies, as well as the invention of new scheduling strategies inspired by specific circumstances of current and future clusters of SMPs, could help ensure that the approach remains viable for these next-generation supercomputers.

**Acknowledgements.** This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-13ER26138/DE-SC0010049.

## References

1. Bull, J.M.: Feedback guided dynamic loop scheduling: algorithms and experiments. In: Pritchard, D., Reeve, J.S. (eds.) Euro-Par 1998. LNCS, vol. 1470, p. 377. Springer, Heidelberg (1998)
2. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105, Lund, Sweden (1999)
3. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 53:1–53:11, Portland, OR, USA. ACM (2009)
4. Donfack, S., Grigori, L., Gropp, W.D., Kale, V.: Hybrid static/dynamic scheduling for already optimized dense matrix factorizations. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China (2012)
5. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. SIGPLAN Not. **33**(5), 212–223 (1998)
6. Heroux, M.: MiniFE documentation. <http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/minife/>
7. Kale, V., Gamblin, T., Hoeffler, T., de Supinski, B.R., Gropp, W.D.: Abstract: Slack-Conscious Lightweight Loop Scheduling for Improving Scalability of Bulk-synchronous MPI Applications, November 2012
8. Kale, V., Randles, A.P., Kale, V., Gropp, W.D.: Locality-optimized scheduling for improved load balancing on SMPs. In: Proceedings of the 21st European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface, vol. 0, pp. 1063–1074. Association for Computing Machinery (2014)
9. Markatos, E.P., LeBlanc, T.J.: Using processor affinity in loop scheduling on shared-memory multiprocessors. In: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Supercomputing 1992, pp. 104–113, Los Alamitos, CA, USA. IEEE Computer Society Press (1992)

10. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12, Salt Lake City, UT, USA. IEEE Computer Society Press (2012)
11. Rein, H., Liu, S.F.: REBOUND: an open-source multi-purpose N-body code for collisional dynamics. *Astron. Astrophys.* **537**, A128 (2012)
12. Rountree, B., Lowenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.: Adagio: making DVS practical for complex HPC applications. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009, pp. 460–469, Yorktown Heights, NY, USA. ACM (2009)
13. Talamo, A.: Numerical solution of the time dependent neutron transport equation by the method of the characteristics. *J. Comput. Phys.* **240**, 248–267 (2013)
14. Zhang, Y., Voss, M.: Runtime empirical selection of loop schedulers on hyper-threaded SMPs. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), vol. 01, pp. 44.2, Washington, DC, USA. IEEE Computer Society (2005)

OpenMP: Heterogenous Execution and Data  
Movements

11th International Workshop on OpenMP, IWOMP 2015,  
Aachen, Germany, October 1-2, 2015, Proceedings  
Terboven, C.; de Supinski, B.R.; Reble, P.; Chapman,  
B.M.; Müller, M.S. (Eds.)  
2015, XI, 274 p. 146 illus. in color., Softcover  
ISBN: 978-3-319-24594-2